IUT GRAND OUEST NORMANDIE

iNFO

R 2.04

2023 - 2024

Communication et fonctionnement bas niveau

TP n°1
Simulation CPU

ANNE Jean-François
D'après plusieurs auteurs

Simulation CPU

A. Instructions machine

Un CPU ne gère que des grandeurs booléennes : les instructions exécutées au niveau du CPU sont donc codées en binaire. L'ensemble des instructions exécutables directement par le microprocesseur constitue ce que l'on appelle le **langage machine**.

Les langages de programmation « évolués » (Python, C++, …), destinés à être utilisés par des humains, se composent d'instructions complexes, opérant sur des types de données beaucoup plus complexes que des booléens.

Il faudra donc passer par une étape de « conversion » du langage évolué vers le langage machine, chaque instruction du langage « évolué » donnant lieu à un grand nombre d'instructions « élémentaires » du langage machine. On distinguera l'opération de compilation (conversion de tout le code évolué en langage machine) de l'opération d'interprétation (la conversion est réalisée au fur et à mesure du déroulement du code).

Une instruction machine est une chaîne binaire composée principalement de 2 parties :

- Le champ « **code opération** » qui indique au processeur le type de traitement à réaliser.
    - Par exemple le code « 0010 0110 » donne l'ordre au CPU d'effectuer une multiplication.
- Le champ « **opérandes** » indique la nature des données sur lesquelles l'opération désignée par le « code opération » doit être effectuée.

Les instructions machines sont relativement basiques (on parle d'instructions de bas niveau), voici quelques exemples :

- Les instructions arithmétiques (addition, soustraction, multiplication…) ou de décalage/comparaisons de valeurs.
    - Par exemple, on peut avoir une instruction qui ressemble à « additionne la valeur contenue dans le registre R1 et le nombre 789 et range le résultat dans le registre R0 ».
- Les instructions de transfert de données qui permettent de transférer une donnée d'un registre du CPU vers la mémoire vive (`STR`) et vice versa de lecture en mémoire vers un registre (`LDR`). Pour préciser une adresse, on utilisera les crochets. Par exemple `LDR R2, [R5]` place dans le registre R2 la valeur à l'adresse précisée dans le registre R5.
    - Par exemple, on peut avoir une instruction qui ressemble à « prendre la valeur située à l'adresse mémoire 487 et la placer dans le registre R2 » ou encore « prendre la valeur située dans le registre R1 et la placer à l'adresse mémoire 512 ».
- Les instructions de rupture de séquence : les instructions machines sont situées en mémoire vive dans l'ordre :
    - Par exemple : si l'instruction n°1 est située à l'adresse mémoire 343, l'instruction n°2 sera située à l'adresse mémoire 344, l'instruction n°3 sera située à l'adresse mémoire 345…
    - Au cours de l'exécution d'un programme, le CPU passe d'une instruction à une autre en passant d'une adresse mémoire à l'adresse mémoire immédiatement supérieure :
    - Par exemple, après avoir exécuté l'instruction n°2 (situé à l'adresse mémoire 344), le CPU « va chercher » l'instruction suivante à l'adresse mémoire 344+1=345.
    - Les instructions de rupture de séquence d'exécution encore appelées instructions de

saut ou de branchement permettent d'interrompre l'ordre initial sous certaines conditions en passant à une instruction située une adresse mémoire donnée.

- Par exemple, dans le cas où à l'adresse mémoire 354 nous avons l'instruction suivante : « si la valeur contenue dans le registre R1 est strictement supérieure à 0 alors exécuter l'instruction située à l'adresse mémoire 4521 ».
- Si la valeur contenue dans le registre R1 est strictement supérieure à 0 alors la prochaine instruction à exécuter est l'adresse mémoire 4521, dans le contraire, la prochaine instruction à exécuter est à l'adresse mémoire 355.

- Des opérations d'entrée-sortie : `INP R1, 2` pour lire un entier signé (type 2) dans le registre R1 ou `OUT R1, 4` pour en afficher le contenu.
- On peut utiliser des constantes numériques avec # pour faire un calcul ou faire une affectation avec l'opérateur MOV, par exemple MOV R2, #5

Comme déjà dit, les **opérandes** désignent les données sur lesquelles le **code opération** de l'instruction doit être réalisée. Un opérande peut être de 3 natures différentes :

1. L'opérande est une **valeur immédiate** : l'opération est effectuée directement sur la valeur donnée dans l'opérande
2. L'opérande est un **registre du CPU** : l'opération est effectuée sur la valeur située dans un des registres (R0, R1, R2, …), l'opérande indique de quel registre il s'agit
3. L'opérande est une donnée située en **mémoire vive** : l'opération est effectuée sur la valeur située en mémoire vive à l'adresse XXXXX. Cette adresse est indiquée dans l'opérande.

Exemples :

- Quand on considère l'instruction machine :
  - « Additionne le nombre 125 et la valeur située dans le registre R2, range le résultat dans le registre R1 »
  - Nous avons 2 valeurs : « le nombre 125 » (qui est une valeur immédiate : cas n°1) et « la valeur située dans le registre R2 » (cas n°2)
- Quand on considère l'instruction machine :
  - « Prendre la valeur située dans le registre R1 et la placer à l'adresse mémoire 512 »
  - Nous avons 2 valeurs : « à l'adresse mémoire 512 » (cas n°3) et « la valeur située dans le registre R1 » (cas n°2)

B. Assembleur

Le microprocesseur étant incapable d'interpréter la phrase « additionne le nombre 125 et la valeur située dans le registre R2, range le résultat dans le registre R1 », il faut coder cette instruction sous forme binaire :

« Additionne le nombre 125 et la valeur située dans le registre R2, range le résultat dans le registre R1 »

⇓

« 11100010100000100001000001111101 »

Afin de faciliter la lecture et l'écriture d'instructions machine par les informaticiens, on remplace les codes binaires par des symboles mnémoniques, en utilisant la syntaxe. La syntaxe concerne le signifiant, soit ce qu'est l'énoncé, du langage appelé ***assembleur***.

« Additionne le nombre 125 et la valeur située dans le registre R2, range le résultat dans le registre R1 »

⇓

« ADD R1, R2, #125 »

⇓

« 11100010 10000010 00010000 01111101 »

Remarque :

Chaque élément de l'instruction occupe une quantité de mémoire bien définie. Par exemple, en architecture ARM :

- L'instruction ADD occupe les 12 bits de poids fort
- Les adresses des registres R1 et R2 occupent chacune 4 bits
- L'opérande #125 occupe les 12 bits de poids faible

Pour en savoir plus : Présentation Architecture et jeu d'instructions ARM

Exercice :

Expliquer brièvement par une phrase, les instructions suivantes :

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

Exercice :

Écrire les instructions en assembleur correspondant aux phrases suivantes :

- « 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 »

C. Labels

En réalité, les instructions assembleur B, BEQ, BNE, BGT et BLT n'utilisent pas directement l'adresse mémoire de la prochaine instruction à exécuter, mais des **labels**.

Un label correspond à une adresse en mémoire vive (c'est l'assembleur qui fera la traduction « label » → « adresse mémoire »). L'utilisation d'un label évite donc d'avoir à manipuler des adresses mémoires.

Voici un exemple qui montre comment utiliser un label :

```
CMP R4, #18
```

```
BGT monLabel
MOV R0, #14
HALT
monLabel:
MOV R0, #18
HALT
```

Dans l'exemple ci-dessus, nous avons choisi monLabel comme nom de label. La ligne MOV R0, #18 a pour label « monLabel » car elle est située juste après la ligne monLabel : .

Concrètement, voici ce qui se passe avec ce programme : si la valeur stockée dans le registre R4 est supérieure à 18 on place le nombre 18 dans le registre R0 sinon on place le nombre 14 dans le registre R0.

ATTENTION :

La présence du HALT juste après la ligne MOV R0, #14 est indispensable, car sinon, la ligne MOV R0,#18 sera aussi exécutée (même si la valeur stockée dans le registre R4 est inférieure à 18).

Exemple :

Si l'on voulait exécuter le programme suivant écrit en pseudo-code :

```
Lire la valeur de A
Lire la valeur de B
Si A > B alors afficher A
Sinon afficher B
Fin
```

On utilisera les registres R0 et R1 pour lire A et B et le code assembleur sera le suivant :

```
INP R0, 2
INP R1, 2
// on compare R0 et R1
CMP R0, R1
// si R0 > R1 on fait
// le saut à Plusgrand
BGT Plusgrand
// saut non fait
// donc R0 <= R1
// on affiche R1
OUT R1, 4
// on saute à la fin
B fin
Plusgrand:
// branche R0 > R1
// on affiche R0
OUT R0, 4
```

```
fin:
// fin du programme
HALT
```

Autre Exemple :

Le code ci-dessous place une séquence d'entiers (exprimés en hexadécimal) au début de la mémoire (à partir de l'adresse 1). Cette séquence se termine par la valeur particulière 0. La première ligne fait un saut à la branche "code" pour ne pas interpréter les données qui suivent comme des instructions. Le programme ensuite fait la somme des entiers de la séquence :

```
// données
// début = @1
B code
dat 0x12
dat 0x20
dat 0x3
dat 0x11
dat 0x0
code:
// R0 : adresse à lire, initialisée à 1
// R1 : valeur courante
// R2 : somme
MOV R0, #1
MOV R2, #0
Boucle:
LDR R1, [R0]
ADD R2, R2, R1
ADD R0, R0, #1
CMP R1, #0
BNE boucle
OUT R2, 4
HALT
```

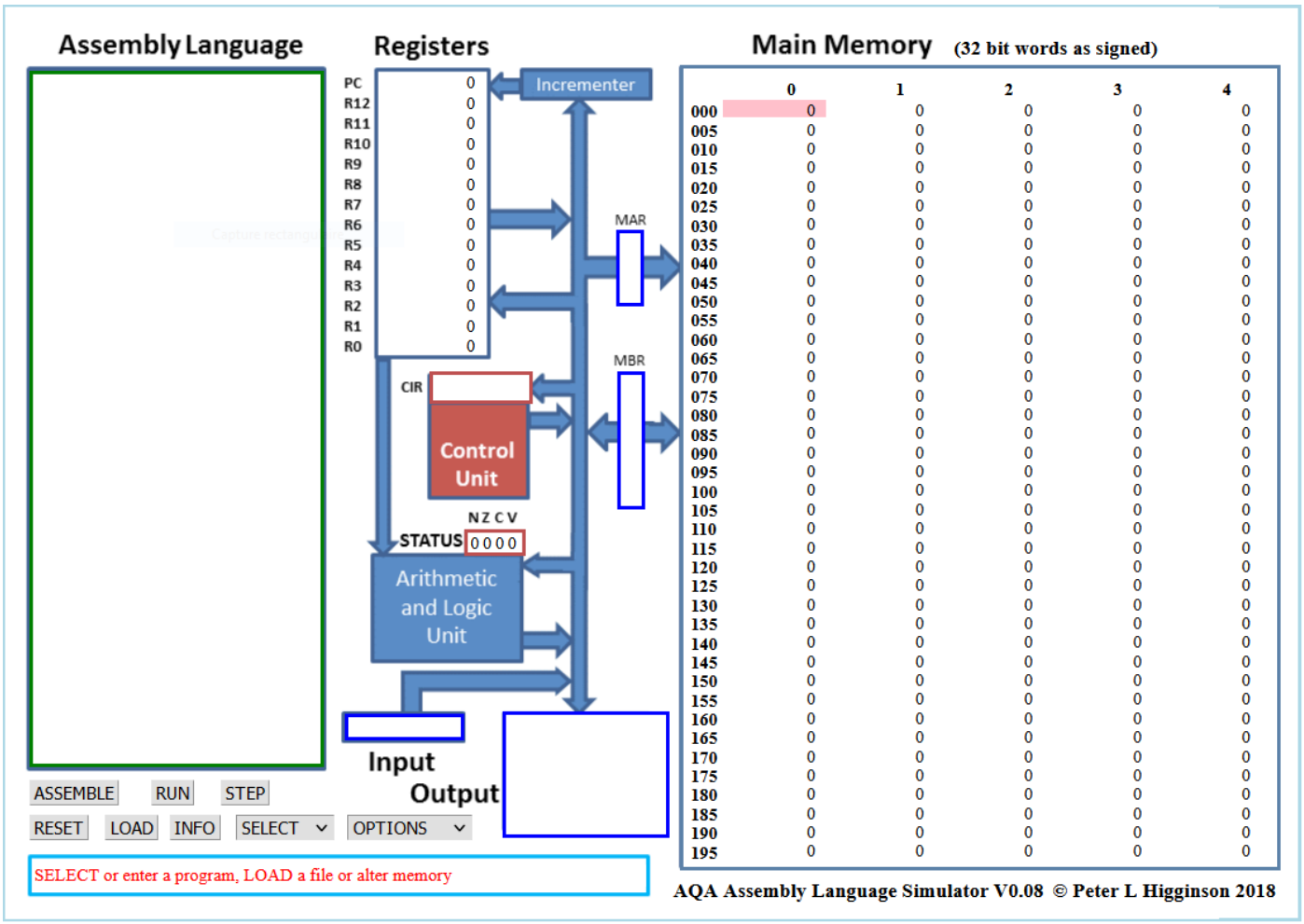
D. *Simulateur de CPU*

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/

Voici ce que vous devriez obtenir en vous rendant à l'adresse indiquée ci-dessus :

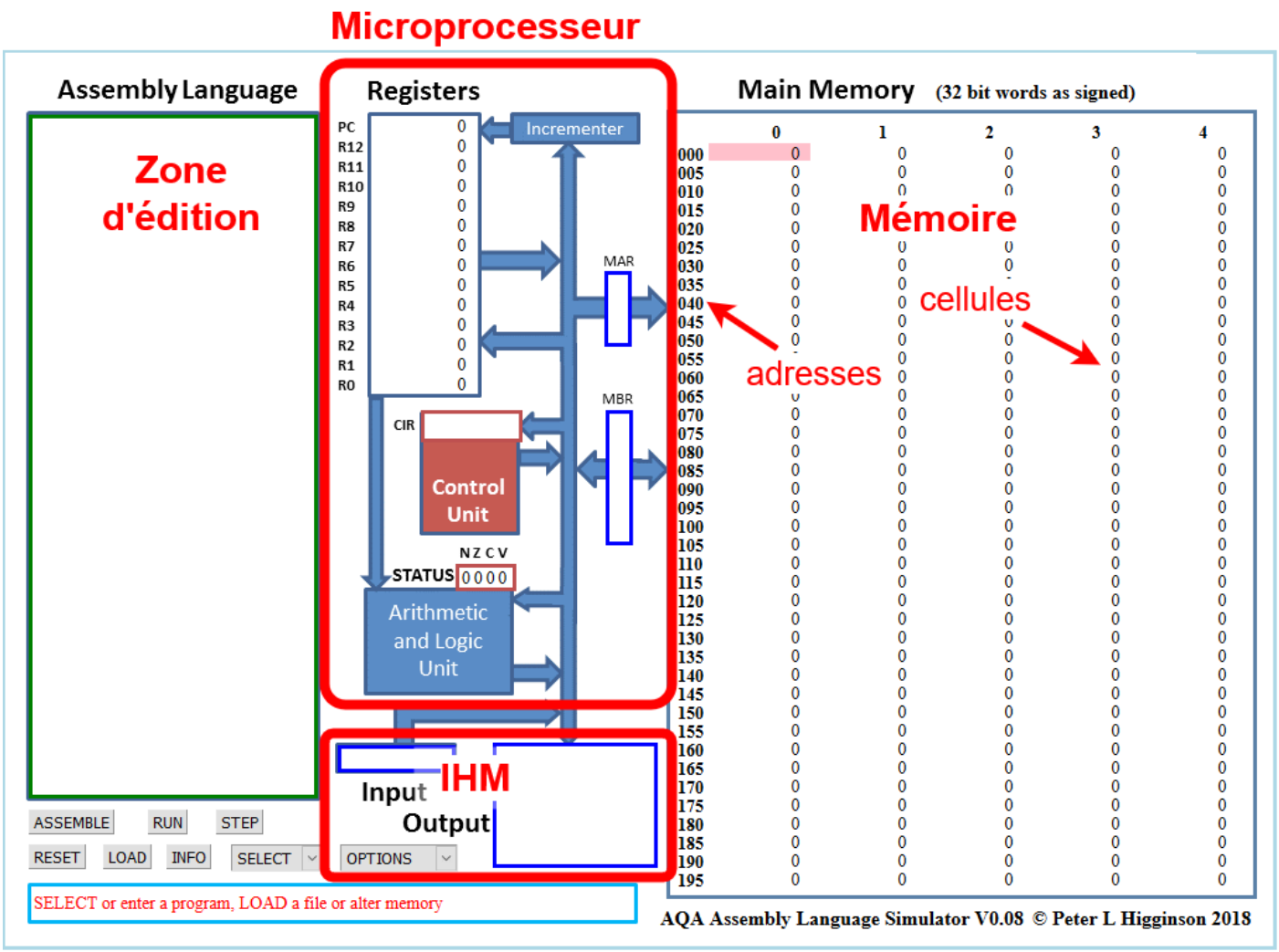


Source : https://www.peterhigginson.co.uk/AQA/

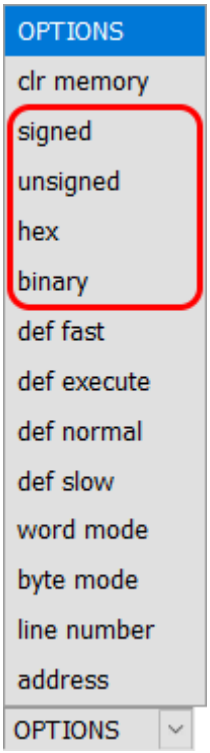
1°) ***Présentation***

Il est relativement facile de distinguer les différentes parties du simulateur :

- à droite, la **mémoire vive** ou **RAM** (***Main Memory***) ;
- au centre, le **microprocesseur** (avec ses différentes composantes : ALU, CU, registres, …) ;
- à gauche, la **zone d'édition** (***Assembly Language***), permettant de saisir des programmes en assembleur.



a) *La RAM*



Le contenu des différentes cellules de la mémoire peut être affiché dans différents formats :

- Base 10 (entier signé, signed) – par défaut
- Base 10 (entier non-signé, unsigned),
- Base 16 (hex),
- Base 2 (binary).

On peut modifier le format d'affichage de la mémoire à l'aide du bouton **OPTIONS** situé en bas dans la partie gauche du simulateur.

À l'aide du bouton **OPTIONS**, passer 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")

Chaque cellule de la mémoire est accessible par son **adresse**. Il existe deux formats d'adressage des cellules de la mémoire :

- **32 bits** – format **mot** (option *word mode*) – *par défaut*

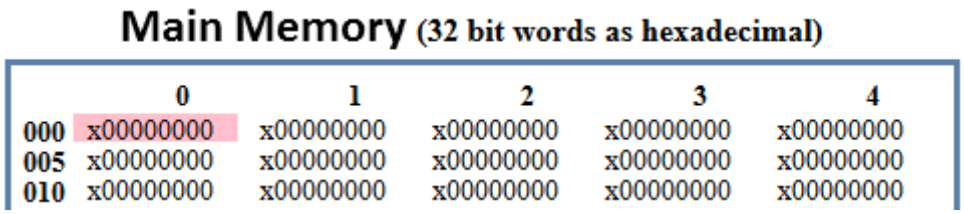
  Les adresses vont de **000** à **199** (codée en base 10).

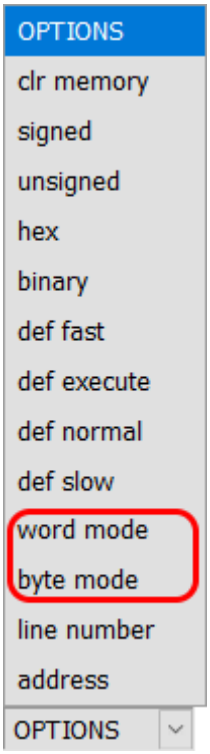
- **8 bits** – format **octet** (option *byte mode*)

  Les adresses vont de **000** à **799** (codée en base 10).

On peut modifier le format d'adressage de la mémoire à l'aide du bouton **OPTIONS**.

Régler la mémoire de sorte d'avoir un affichage **hexadécimal**, avec des cellules au format **32 bits**.



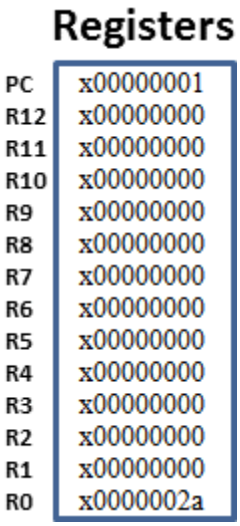


b) Le CPU

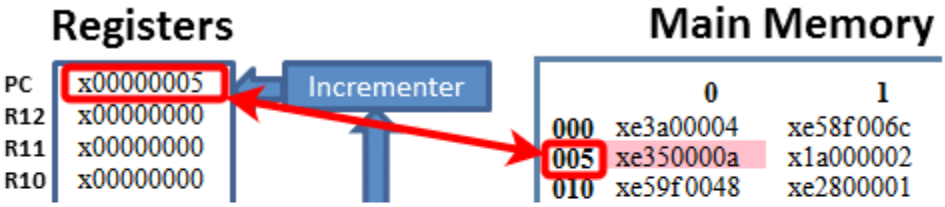
Dans la partie centrale du simulateur, on trouve les différents composants du microprocesseur :

- 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")

Les ***registres*** (**Registers**) : 13 registres (***R0*** à ***R12***)



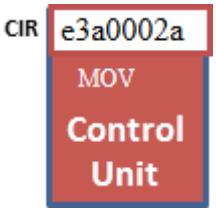
- 1 registre spécial (**PC**) qui contient l'adresse mémoire de l'instruction en cours d'exécution ;



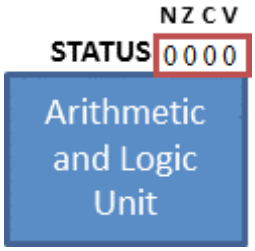
La cellule mémoire contenant l'instruction machine **en cours d'exécution** :

- est surlignée en rouge dans la zone de mémoire
- a son adresse inscrite dans le registre PC

- *L'unité **de commande** (**Control Unit**) qui contient l'instruction machine en cours d'exécution (au format hexadécimal)*



- *L'unité **arithmétique et logique** (**Arithmetic and Logic Unit**)*



2°) ***Programmer en assembleur***

Comme déjà dit plus haut, la partie de gauche permet de saisir des programmes en assembleur. L'assembleur du simulateur correspond exactement à l'assembleur que nous avons étudié dans le cours.

Cette **zone d'édition** (***Assembly Language***) permet de saisir des programmes en assembleur.

Exercice :

Dans la partie "éditeur" (***Assembly Language***) saisissez les lignes de codes suivantes :

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

Une fois la saisie terminée, cliquez sur le bouton "***submit***".

- Adresse **000** : MOV R0, #42
- Adresse **001** : STR R0,150
- Adresse **002** : HALT

Vous devriez voir apparaitre des nombres "étranges" dans les cellules mémoires d'adresses ***000, 001 et 002*** :

	0	1	2	3	4
000	xe3a0002a	xe58f024c	xef000000	x00000000	x00000000
005	x00000000	x00000000	x00000000	x00000000	x00000000

L'assembleur a fait son travail, il a converti les 3 lignes de notre programme en ***instructions machines*** :

- La première instruction machine « xe3a0002a » est stockée à l'adresse mémoire 000 (elle correspond à "MOV R0, #42" en assembleur),
- La deuxième instruction machine « xe58f024c » est stockée à l'adresse mémoire 001 (elle correspond à "STR R0,150" en assembleur) et
- La troisième instruction machine « xef000000 » est stockée à l'adresse mémoire 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 :

Pour avoir une idée des véritables instructions machines, repasser à un affichage en binaire (**OPTIONS** / **binary**)) pour obtenir ceci :

	0	1	2
000	11100011 10100000 00000000 00101010	11100101 10001111 00000010 01001100	11101111 00000000 00000000 00000000

On constate que chaque **instruction machine** occupe un *mot* (32 bits) et correspond à une ligne de code en assembleur :

- 11100011 10100000 00000000 00101010 correspond au code assembleur MOV R0,#42
- 11100101 10001111 00000010 01001100 correspond au code assembleur STR R0,150
- 11101111 00000000 00000000 00000000 correspond au code assembleur HALT

Question :

Identifiez la valeur **42** dans la mémoire ?

On peut remarquer pour la première instruction, que l'octet le plus à droite, 00101010_2, est bien égale à 42_{10} si on le découpe en 5+3 bits !

Repasser à un affichage en *hexadécimal* afin de faciliter la lecture des données présentes en mémoire.

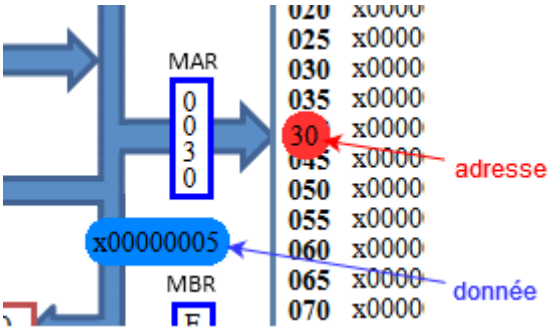
a) ***Simulation***

Pour exécuter un programme, il suffit de cliquer sur le bouton **RUN** (exécution en continu) ou **STEP** (exécution pas à pas). 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 >> apparaissant à côté du bouton **STOP.** 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. Ce qui permet de mettre l'exécution en *pause* :

ATTENTION : pour relancer la simulation, il est nécessaire d'appuyer sur le bouton **RESET** ou **ASSEMBLE** afin de remettre les registres R0 à R12 à zéro, et en particulier le registre PC permettant à l'*unité de commande* de pointer 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.

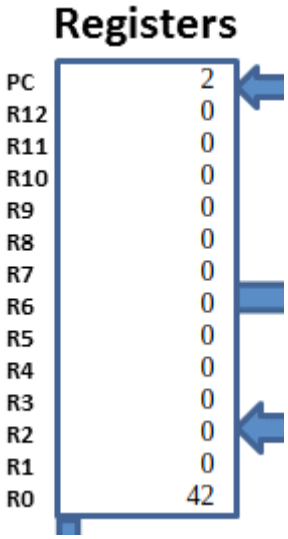
Tester l'exécution du code, en ralentissant suffisamment la vitesse afin de bien comprendre toutes les étapes de cette exécution.



On constate que deux types de valeurs circulent au sein du système :

- Des **données** (valeurs lues/écrites de/vers la mémoire/les registres/l'unité de commande)
- Des **adresses** des cellules de mémoire

Une fois la simulation terminée, on peut constater que la cellule mémoire d'adresse **150**, contient bien le nombre **42** (en base 10). Il en est de même pour le registre **R0**.



Pour remettre la mémoire à 0, il faut cliquer sur le bouton **OPTIONS** et choisir **clr memory**.

Exercice :

Modifier 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**. Tester les modifications en exécutant la simulation.

b) Les labels

Le simulateur prend en charge les **labels**.

Exercice :

Voici un programme Python très simple :

```
x = 4
y = 8
if x == 10:
    y = 9
```

```
else:
   x = x+1
z = 6
```

et voici maintenant voici son équivalent en assembleur, Saisir et tester le programme suivant :

```
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, établir une correspondance entre les lignes du programme en Python et les lignes du programme en assembleur.

À quoi sert la ligne B endif ?

À quoi correspondent les adresses mémoires 23, 75 et 30 ?

c) Les entrées-sorties

Si l'on voulait exécuter le programme suivant écrit en pseudo-code :

```
lire la valeur de A
lire la valeur de B
si A > B alors afficher A
sinon afficher B
fin
```

On utilisera les registres R0 et R1 pour lire A et B et le code assembleur sera le suivant :

```
INP R0, 2
INP R1, 2
// on compare R0 et R1
```

```
CMP R0, R1
// si R0 > R1 on fait
// le saut à plusgrand
BGT plusgrand
// saut non fait
// donc R0 <= R1
// on affiche R1
OUT R1, 4
// on saute à la fin
B fin
plusgrand:
// branche R0 > R1
// on affiche R0
OUT R0, 4
fin:
// fin du programme
HALT
```

d) Les données en mémoire

Par défaut, les instructions sont codées à partir de l'adresse 0. Pour placer des données traitées par un programme, le plus simple est de les placer au début de la mémoire avec l'instruction `DAT`.

Le code ci-dessous place une séquence d'entiers (exprimés en hexadécimal) au début de la mémoire (à partir de l'adresse 1). Cette séquence se termine par la valeur particulière 0. La première ligne fait un saut à la branche "code" pour ne pas interpréter les données qui suivent comme des instructions. Le programme ensuite fait la somme des entiers de la séquence :

```
// donnees
// debut = @1
B code
dat 0x12
dat 0x20
dat 0x3
dat 0x11
dat 0x0
code:
// R0: adresse à lire, initialisée à 1
// R1: valeur courante
// R2: sommme
MOV R0, #1
MOV R2, #0
```

```
boucle:
LDR R1, [R0]
ADD R2, R2, R1
ADD R0, R0, #1
CMP R1, #0
BNE boucle
OUT R2, 4
HALT
```

Exercice de conversion :

Voici un programme Python :

```
x = 0
while x<3
   x = x+1
   print(x)
```

Écrire et tester un programme en assembleur équivalent au programme ci-dessus.

E. ***Travail à réaliser***

- Un programme qui demande un nombre en entrée et calcule la somme des nombres de ce nombre à 1(ex pour 5 : 5 + 4 + 3 + 2 + 1).
- Un programme qui affiche la valeur maximum d'une série d'entiers placés en mémoire.
- Un programme qui devine un nombre que vous avez entré en mémoire, avec des indications : plus, moins, gagné.

F. ***Pour aller plus loin :***

- ***https://qkzk.xyz/docs/nsi/cours_premiere/architecture/6_armlite/exos_armlite/***

ANNEXES

1°) ***Exemples d'instructions en assembleur***

Instructions	Description
`LDR R1, 78`	Place la valeur stockée à l'adresse mémoire 78 dans le registre R1
`STR R3, 125`	Place la valeur stockée dans le registre R3 en mémoire vive à l'adresse 125
`INP R0, 2`	Lit la valeur d'entrée et la place dans le registre R0
`OUT R1, 4`	Place dans la sortie la valeur du registre R1
`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
`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
`B 45`	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, R1`	Compare la valeur stockée dans le registre R0 et la valeur stockée dans le registre R1.
`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`	Arrête l'exécution du programme

2°) AIDE : L'ensemble d'instructions AQA

Voici une traduction de la documentation proposée à partir du bouton INFO

Les registres sont numérotés de `R0` à `R12`.

Les opérations peuvent être effectuées sur deux types de valeurs (notées `<opérande2>` dans la suite) :

- Une constante notée `#nnn`. Ex : `#12` : la valeur décimale 12.
- Le contenu du registre n `Rn`. Ex : `R1` : la valeur contenue dans le registre 1.

On peut ajouter des étiquettes dans le programme (notées `<label>`) ; il suffit de noter le nom de l'étiquette de son choix et de le faire suivre de deux points.

Exemple :

```
asm test:
```

Toute référence à cette étiquette fera "sauter" le programme à la case mémoire de l'étiquette et exécutera donc les instructions qui suivent.

Les commentaires commencent par au moins un `/`.

- **Affectations**
  - **`LDR Rd, <adresse mémoire>`**

Charge la valeur stockée dans l'emplacement de mémoire spécifié par `<adresse mémoire>` dans le registre d.

  - **`STR Rd, <adresse mémoire>`**

Stocke la valeur qui se trouve dans le registre d dans l'emplacement de mémoire spécifié par `<adresse mémoire>`.

  - **`MOV Rd, <opérande2>`**

Copiez la valeur spécifiée par `<opérande 2>`dans le registre d.

- **Opérations**
  - **`ADD Rd, Rn, <opérande2>`**

Ajouter la valeur spécifiée dans `<opérande 2>`à la valeur du registre n et stocker le résultat dans le registre d.

  - **`SUB Rd, Rn, <opérande2>`**

Soustrayez la valeur spécifiée par `<opérande 2>`de la valeur du registre n et stockez le résultat dans le registre d.

- **Comparaisons**
  - **`CMP Rn, <opérande2>`**

Comparez la valeur stockée dans le registre n avec la valeur spécifiée par `<opérande 2>`

  - **`B <condition> <label>`**

Connectez conditionnellement l'instruction à la position `<label>` dans le programme si la dernière comparaison a répondu aux critères spécifiés par la `<condition>`. Les valeurs possibles pour `<condition>` et leur signification sont les suivantes :

- **EQ : égal à,**
- **NE : différent de,**
- **GT : supérieur à,**
- **LT : inférieur à.**

- **Sauts (JUMP)**
  - **B <label>**

Toujours relier l'instruction à la position `<label>` dans le programme.

- **Fin du programme**
  - **HALT: Arrête l'exécution du programme.**
- **Non utilisées dans ce TD**
  - **AND Rd, Rn, <opérande2> :**

Effectue une opération ET logique au niveau du bit entre la valeur du registre n et la valeur spécifiée par `<opérande 2>` et stocke le résultat dans le registre d.

- **ORR Rd, Rn, <opérande2> :**

Effectue une opération OU logique au niveau des bits entre la valeur du registre n et la valeur spécifiée par `<opérande 2>`et stocke le résultat dans le registre d.

- **EOR Rd, Rn, <opérande2> :**

Effectue une opération logique ou binaire exclusive au sens des bits entre la valeur du registre n et la valeur spécifiée par `<opérande 2>`et stocke le résultat dans le registre d.

- **MVN Rd, <opérande2> :**

Exécutez une opération NOT logique au niveau des bits sur la valeur spécifiée par `<opérande 2>`et stockez le résultat dans le registre d.

- **LSL Rd, Rn, <opérande2> :**

Décalez logiquement vers la gauche la valeur stockée dans le registre n du nombre de bits spécifié par `<opérande 2>` et stockez le résultat dans le registre d.

- **LSR Rd, Rn, <opérande2> :**

Décalez logiquement à droite la valeur stockée dans le registre n du nombre de bits spécifié par `<opérande 2>` et stockez le résultat dans le registre d.

3°) Documentation

L'objectif de ce simulateur est de permettre aux étudiants de se familiariser avec le langage assembleur spécifié par AQA pour une utilisation informatique de niveau AS et A. La mise en œuvre a pris mon simulateur LMC précédent comme base et toute personne familière avec cela trouvera cela facile à utiliser. Vous pouvez écrire un programme dans la zone Langage d'assemblage, modifier le contenu de la mémoire ou du PC et afficher la mémoire en nombre signé (par défaut), non signé, hexadécimal ou binaire. Vous pouvez ajouter des espaces et des commentaires dans des endroits sensibles.

a) Project

Il existe un projet pour la simulation AQA écrit par Richard Pawson Écrivez un jeu snake complet en langage d'assemblage AQA.

b) Hypothèses

AQA ne spécifie pas leur langage suffisamment en détail pour créer un assembleur, de sorte que certaines hypothèses doivent être faites.

Taille du mot. : AQA ne spécifie pas de taille de mot, dans une épreuve d'examen, ils dessinent un registre comme 8 bits et dans un autre donnent la réponse à une question comme 860 (ce qui est plus de 8 bits). Dans une question de niveau A de 2017, ils disent « Chaque emplacement de mémoire principal et registre peut stocker une valeur de 16 bits » mais c'est dans la question et non dans la spécification générale. Comme ils ont treize registres, les instructions les plus complexes (comme ADD R2, R3, # 100) ne peuvent pas être codées en moins de 20 bits environ. Donc, une taille de mot de 16 bits est impossible et 32 bits est ce que j'ai implémenté.

Codage des instructions. J'ai décidé d'utiliser exactement l'encodage d'instructions ARM 32 bits (et de ne pas l'étendre ou le simplifier). Ce sera un problème pour tous ceux qui regardent les instructions (car c'est complexe) et un avantage pour quiconque utilise le Raspberry Pi ou MicroBit à un niveau bas.

Adressage. Les ordinateurs ARM (et la plupart des ordinateurs modernes) adressent la mémoire en termes d'octets. Les exemples de questions de l'AQA (avec les adresses 100,101,102, 103) suggèrent que c'est peut-être la façon dont l'AQA pensait, mais ils utilisent ensuite l'instruction LDR plutôt que LDRB et parlent des numéros de ligne dans le programme. À l'origine, j'ai implémenté l'adressage d'octets (vous pouvez sélectionner « mode octet » pour voir cela), mais j'ai décidé que l'adressage de mots était beaucoup plus facile pour l'affichage visuel. Dans une note récente, AQA indique que les étiquettes doivent être utilisées pour l'adressage de la mémoire, de sorte que le choix d'adressage 8/32 bits n'est qu'un choix d'affichage.

Écran PC et MAR. Ceux-ci changent si le mode d'adressage change. ARM utilise le mode octet et le PC contient l'adresse octet de l'instruction suivante. Le simulateur passe par défaut en mode mot et le PC est affiché divisé par 4 - c'est beaucoup plus facile à comprendre. De même, l'extraction d'octets est normalement une fonction CPU et donc le MAR est correct en mode word mais probablement incorrect en mode octet. (C'est au-delà du programme.)

Comparer. Dans ARM, BGT et BLT sont les instructions de comparaison complémentaires de 2, c'est donc ainsi qu'ils fonctionnent dans ce simulateur. Si AQA voulait une comparaison non signée, les instructions sont BLO et BHI qu'ils n'ont pas (mais je pourrais implémenter comme une extension si je reçois suffisamment de demandes). Notez qu'avec 32 bits, la plupart des calculs simples ne verront pas de problème.

ARRÊTER. Halt n'est pas une instruction ARM. La simulation utilise l'instruction ARM « software interrupt » pour cela. C'est ainsi que les appels à un système d'exploitation seraient normalement traités.

Taille du champ immédiat. AQA ne spécifie pas cela mais je n'ai trouvé aucune valeur supérieure à 255. Donc, 8 bits pourraient être une hypothèse raisonnable. Après avoir implémenté cela, j'ai constaté que ne pas pouvoir faire des choses comme AND R2, R2, #0x8000 tester des bits uniques (bien sûr, vous avez vraiment besoin d'ANDS) ou avoir à faire MOV, LSL, ADD pour obtenir des constantes supérieures à 255 (alors que 2 instructions feraient l'affaire) était fastidieux. Ainsi, le simulateur implémente maintenant les mêmes règles que l'ARM, c'est-à-dire que tout modèle de 8 bits a tourné un nombre pair d'endroits (par exemple, 512, 0xC8000003, 260, 4096 et bien d'autres sont valides).

Étiquettes. AQA a adopté un format où l'étiquette est terminée par un deux-points (:) et généralement sur une ligne qui lui est propre. Étant donné que tous les assembleurs que j'ai utilisés ont permis à l'étiquette d'être sur la même ligne que l'instruction à cette adresse, le simulateur acceptera les deux formats et j'ai modifié quelques exemples pour correspondre à la méthode « ligne seule ». Étant donné que les étiquettes n'occupent pas la mémoire, le fait de ne pas avoir d'instruction entraîne une divergence entre les numéros de ligne du programme et les adresses mémoire. J'ai ajouté l'option « numéro de ligne » pour afficher les numéros de ligne dans la zone Langue de l'assemblage et l'option « adresse » pour afficher les adresses (en ne numérotant pas les définitions d'étiquettes sur une ligne qui leur est propre). La valeur par défaut est l'adresse et cela semble le plus utile. (Un commentaire sur une ligne seule n'est jamais numéroté par le simulateur - mais AQA n'a pas de commentaires de toute façon.)

c) The AQA Instruction Set.

LDR Rd, <memory ref>	Charge la valeur stockée dans l'emplacement mémoire spécifié par <memory ref> dans le registre d..
STR Rd, <memory ref>	Stocke la valeur qui se trouve dans le registre d dans l'emplacement de mémoire spécifié par <memory ref>.
ADD Rd, Rn, <operand2>	Ajoute la valeur spécifiée dans <operand2>à la valeur du registre n et stockez le résultat dans le registre d.
SUB Rd, Rn, <operand2>	Soustrait la valeur spécifiée par <operand2> de la valeur du registre n et stockez le résultat dans le registre d.
MOV Rd, <operand2>	Copie la valeur spécifiée par <operand2>into register d.
CMP Rn, <operand2>	Compare la valeur stockée dans le registre n avec la valeur spécifiée par<operand2>.
B <label>	Toujours se brancher à l'instruction à la position <label>dans le programme.
B<condition> <label>	Branchement conditionnel à l'instruction de la position <label> dans le programme si la dernière comparaison répondait aux critères spécifiés par la <condition>. Les valeurs possibles pour <condition> et leur signification sont : EQ : Equal to, NE : Not equal to, GT : Greater than, LT : Less than.
AND Rd, Rn, <operand2>	Effectue une opération logique ET binaire entre la valeur du registre n et la valeur spécifiée par <operand2> et stocke le résultat dans le registre d.
ORR Rd, Rn, <operand2>	Effectue une opération logique OU binaire entre la valeur du registre n et la valeur spécifiée par <operand2> et stocke le résultat dans le registre d.
EOR Rd, Rn, <operand2>	Effectue une opération logique OU exclusive binaire ou (XOR) entre la valeur du registre n et la valeur spécifiée par <operand2> et stocke le résultat dans le registre d.

MVN Rd, <operand2>	Effectue une opération logique NON binaire sur la valeur spécifiée par <operand2> et stocke le résultat dans le registre d.
LSL Rd, Rn, <operand2>	Décale logiquement à gauche la valeur stockée dans le registre n du nombre de bits spécifié par <operand2> et stocke le résultat dans le registre d.
LSR Rd, Rn, <operand2>	Décale logiquement à droite la valeur stockée dans le registre n du nombre de bits spécifié par <operand2> et stocke le résultat dans le registre d.
HALT	Arrêter l'exécution du programme.
<operand2>	Peut être #nnn ou Rm pour utiliser une constante ou le contenu du registre Rm.
Registres	sont de R0 à R12.

d) Extension - data

La pseudo-instruction DAT vous permet de placer un nombre dans une cellule mémoire à l'aide de l'assembleur. Vous pouvez laisser de côté le fichier DAT et simplement mettre le numéro. Une étiquette en tant que données fonctionnera également.

e) Extensions - INP and OUT.

Ceux-ci fonctionnent de la même manière que sur le LMC. INP Rd,2 lit un nombre dans le registre d et OUT Rd,4 sort le numéro du registre d. Pour OUT, le périphérique 4 est traité comme signé, mais vous pouvez sortir non signé (périphérique 5), hexadécimal (périphérique 6) ou caractère (périphérique 7). Vous pouvez entrer hexadécimal comme 0xnnn partout où un nombre est attendu. Comme HALT, INP et OUT utilisent l'instruction Software Interrupt - ceci est similaire à la façon dont les programmes appellent le système d'exploitation ou le BIOS pour effectuer des E/S.

Dans le document AS 2017, AQA utilise une adresse mémoire pour contrôler un moteur et de nombreux ordinateurs modernes effectuent des E/S via des adresses plutôt que des instructions spéciales. Cependant, sur tous les systèmes que je connais, les adresses d'E/S ne sont pas proches des adresses mémoire et le STR R0, 17 (utilisé dans la question) n'a pu être assemblé à l'aide d'aucune partie du jeu d'instructions ARM. (Le code correct est MOV R1, #17 ; STR R0, [R1] mais AQA n'ont pas non plus d'adressage indirect.)

f) Extension - Indirect Addressing.

Vous pouvez, comme les gens l'ont fait avec le LMC, adresser une liste (ou un tableau) en créant votre propre instruction, puis en l'exécutant. Étant donné que AQA ne spécifie pas le format d'instruction, vous ne pouvez pas le faire avec leurs instructions et il n'y a aucun moyen de savoir quel numéro est associé à une étiquette ou à une référence de mémoire (dans la spécification AQA). La plupart des ordinateurs modernes ne vous permettent pas de modifier vos propres instructions (ou dans certains, le résultat peut être imprévisible). ARM n'a pas d'instructions de mémoire directes - si vous regardez comment LDR et STR sont codés, vous constaterez que les adresses sont formées en ajoutant ou en soustrayant des nombres du PC (appelé adressage relatif).

Donc, pour LDR et STR, j'ai implémenté ce qu'ARM appelle « register offset addressing » où vous écrivez [Rn + <label>] et l'adresse utilisée est formée en ajoutant le contenu du registre à l'étiquette. Dans le mode de mot par défaut, le registre est incrémenté d'un pour accéder au mot suivant. En mode octet, vous devez ajouter 4 (ce qui est la même chose qu'ARM).

Si vous dites simplement [Rn], l'adresse utilisée est le contenu du registre. Il s'agit d'un véritable « adressage indirect », mais vous n'avez aucun moyen dans le jeu d'instructions AQA de connaître les adresses mémoire réelles. Dans le simulateur, cela peut être fait.

Il est également possible d'utiliser [Rn+xxx] où xxx est un nombre positif et le nombre de mots à ajouter au contenu du registre (pas d'octets). Notez que la sélection du mode octet/mot affecte l'utilisation du contenu du registre pendant l'exécution, mais pas l'interprétation du décalage xxx par l'assembleur.

g) Extension - Comments.

Les commentaires sont utiles, donc tout ce qui suit un '/' est pris comme un commentaire. Dans les exemples, j'ai utilisé '//' pour suivre le style de nombreuses langues de niveau supérieur.

h) Self-Modifying Code

Construire vos propres instructions ARM n'est pas facile et avec les ordinateurs modernes, le code auto-modifiable est considéré comme une mauvaise pratique. Cependant, dans ce simulateur et avec quelques instructions, vous pouvez apporter des modifications simples. Une chose importante à retenir est que ARM n'a pas d'adressage direct et donc toutes les instructions directes AQA sont en fait implémentées comme un décalage de « PC + 8 » (c'est-à-dire l'instruction après la suivante).

Pour les instructions de branchement, vous pouvez ajouter ou soustraire le nombre de mots pour modifier la destination de la branche (c'est-à-dire octets/4).

Pour les instructions LDR et STR, vous pouvez ajouter ou soustraire un certain nombre d'octets (c'est-à-dire des mots*4) à une instruction de référencement directe à condition que les adresses d'origine et modifiées soient supérieures à PC+8 et que l'adresse finale se trouve dans la plage du simulateur. (D'autres cas sont plus complexes.)

Pour les instructions #immediate non décalées, si la valeur immédiate est comprise entre 0 et 255, vous pouvez ajouter ou soustraire de l'instruction à condition que le résultat immédiat soit également compris entre 0 et 255.

Ce n'est pas parce que c'est possible que vous devez le faire. Veuillez utiliser l'extension d'adressage indirect à la place.

i) MAR and MBR

Comme certains d'entre vous le savent peut-être, je ne suis pas d'accord avec MAR et MBR car il est très difficile de trouver quoi que ce soit qui les implémente exactement dans n'importe quelle conception de processeur moderne que je n'ai jamais vue. Cependant, ils sont dans votre programme, alors j'ai pensé que je céderais et que je les mettrais là où ils pourraient être. Pour LDR/STR, vous remarquerez peut-être un léger bégaiement dans certains modes tandis que le simulateur vous montre les instructions récupérées en MAR/MBR suivies des valeurs utilisées par le LDR ou le STR. Dans tous les autres cas (si vous effectuez une seule étape), la récupération des instructions est la seule référence mémoire.

Le MAR affiche toujours l'adresse en déniaire et le MBR affiche toujours les données en hexadécimal. (Je sais que ce n'est pas particulièrement cohérent, mais un MBR signé aurait besoin de trois chiffres de plus et binaire d'un 24 chiffres de plus.)

j) SELECT

« SELECT » vous permet de choisir l'un des quelques exemples de programmes à assembler et, si vous le souhaitez, à modifier et/ou à exécuter.

k) OPTIONS

Il y a quelques options sous la liste déroulante « OPTIONS ». « Clr memory » efface la mémoire et les quatre suivants (« signé », « non signé », « hexadécimal » et « binaire ») sélectionnent le mode pour afficher le contenu de la mémoire et les registres. Signé signifie le complément à deux signé et hexadécimal est l'abréviation d'hexadécimal (base 16 avec a = 10, b = 11, c = 12, d = 13, e = 14 et f = 15). Binaire affiche uniquement la mémoire en binaire. Les blobs de données suivent normalement le mode d'affichage (sauf binaire qui serait trop gros) mais les blobs d'adressage (rouge) sont normalement en denary.

Les quatre options suivantes contrôlent la vitesse d'exécution. « def normal » est la vitesse par défaut (7.5) montrant les blobs de données en mouvement. Une fois en cours d'exécution, vous pouvez changer la vitesse de 0 (def slow) à 12,5 et toujours voir les blobs en mouvement. « def execute » est la vitesse la plus lente (15) à exécuter sans le retard de déplacement des blobs et « def fast » (24) est la plus rapide que le simulateur ira et affichera toujours tous les progrès à l'écran. Vous pouvez augmenter la vitesse jusqu'à 24,9 tout en fonctionnant au détriment de ne pas rafraîchir l'écran après chaque instruction.

Les deux options suivantes contrôlent le mode d'adressage de la mémoire (octet/mot). Les deux derniers contrôlent si la fenêtre Langage d'assemblage affiche le numéro de ligne dans le programme ou l'adresse du mot en mémoire.

l) STATUS

Le registre d'état contient le résultat de la dernière opération CMP - négatif (N), zéro (Z), carry (C) et débordement (V). Overflow est pour un complément à 2 comparer tandis que Carry signifie pas de débordement pour la comparaison considérée comme une opération non signée.

L'indicateur Z est testé par les instructions BEQ et BNE. Tests BLT (N set et V clear) OU (N clear et V set) et tests BGT Z clear AND (N et V tous deux réglés ou les deux clairs). AQA ne vous donne pas d'instructions pour utiliser ou tester C. (C est important pour faire de l'arithmétique étendue et / ou non signée - par exemple 64 bits ou plus sur un ordinateur 32 bits ou 32 bits comme non signé.)

Dans le jeu d'instructions AQA, CMP est la seule instruction qui définit les bits d'état.

1. Cycle de récupération/exécution

Ceci est animé par des blobs en mouvement. Comme je m'y attendais, la taille de certains blobs de données avec des nombres de 32 bits est parfois un problème et il semble y avoir des navigateurs qui ne mettent pas à l'échelle le texte et les graphiques au même rythme.

m) Version V0.07

Cela a été produit en coopération avec Richard Pawson qui a produit un ensemble de notes de cours culminant dans un jeu de serpent que vous pouvez jouer sur le simulateur. V0.07 dispose d'un certain nombre de fonctionnalités supplémentaires conçues pour rendre le fonctionnement du jeu possible et la tâche de programmation légèrement plus simple. L'option SELECT « Nouvelles E/S » est un exemple d'utilisation des nouvelles fonctionnalités.

Mémoire vidéo. La zone de sortie peut également être traitée sous la forme d'un tableau de 32 colonnes par 24 lignes, commençant à l'adresse 256 en haut à gauche et se terminant à l'adresse 1023 en bas à droite. L'instruction STR est utilisée pour écrire un code couleur HTML 24 bits à l'adresse du pixel. (Ainsi, par exemple, 0 est noir et 0xff0000 est rouge.) Notez que les 8 premiers bits du registre sont ignorés et que le modèle 24 bits peut être lu à l'aide de LDR.

Codes de périphérique d'entrée supplémentaires. INP Rd,4 lit le code de touche de la dernière touche enfoncée sur le clavier dans le registre d. Si aucune touche n'a été enfoncée, un zéro est renvoyé. (Le code de périphérique 5 efface le caractère lors de la lecture afin que vous puissiez savoir si une

nouvelle touche a été enfoncée.) Le code clé pour a (ou A) est 65 et pour 1 est 49. Méfiez-vous que l'utilisation de n'importe quelle touche avec une fonction de navigateur peut se comporter de manière imprévisible.

Code de périphérique d'entrée supplémentaire. INP Rd,8 lit un modèle aléatoire de 32 bits dans le registre d.

Code de périphérique de sortie supplémentaire. OUT Rd,8 utilise le contenu du registre d comme adresse d'une chaîne ASCII terminée par null en mémoire qui est écrite dans la zone de sortie. Notez que les caractères doivent être dans un ordre peu finien.

Référence numérique <mémoire>. Dans les versions précédentes de ce simulateur, <memory ref> ne peut être qu'une étiquette parce que ma compréhension initiale était que l'adressage de la mémoire était caché (en particulier à cause du conflit octets/mots). Cependant, AQA a défini des questions qui utilisent des <mémoires> et c'est donc maintenant autorisé. (Notez que cela ne s'applique pas à la branche qui nécessite toujours une étiquette.)

Plage immédiate étendue pour MOV. AQA ne spécifie pas la plage autorisée d'opérandes immédiats. Le simulateur implémente la plage autorisée par ARM - dont une version simplifiée est un octet dans l'une des quatre positions d'un mot. Il est difficile d'expliquer aux étudiants pourquoi, par exemple, MOV R1, #768 est valide mais MOV R1, #767 n'est pas autorisé. Pour éviter cette complication inutile et rendre les codes couleur HTML utilisables comme valeurs d'opérande immédiates, j'ai implémenté une plage immédiate étendue pour l'instruction MOV. Toute valeur positive de 26 bits peut être utilisée (uniquement avec MOV). AFAIK c'est le seul écart par rapport aux op-codes ARM valides. (En supposant qu'un gestionnaire pour SPI soit utilisé pour HALT, INP et OUT. Le MOV étendu utilise d'autres codes d'opération SPI et coprocesseur et il est donc peu probable qu'il soit affecté par une extension du jeu d'instructions AQA.)

n) Version V0.08

Je suis reconnaissant à Daniel Stone de l'école de lecture d'avoir souligné qu'avec Chrome ou Safari sur MacOS, la mémoire et les cellules de registre ne s'affichent pas correctement. La cause première en est que, dans ces configurations,   est considérablement plus large que l'espace occupé par un nombre conduisant à un remplissage poussant les numéros au-delà du champ assigné. (Firefox sur MacOS est meilleur mais pas parfait.)

La meilleure solution que je puisse trouver (étant donné la façon dont le simulateur est structuré) est d'utiliser   pour le rembourrage. Dans tous les tests que j'ai vus, cela ne fait aucune différence pour les affichages de n'importe quel navigateur sur Windows OS (qui s'affichent tous correctement), mais corrige Chrome et Safari sur MacOS. Firefox sur MacOS semble avoir   plus grand et   plus petit que l'espace numérique correct - donc je pense que plus petit est plus sûr; c'est cependant perceptible. Il n'y a pas d'autres changements entre V0.07 et V0.08.

o) The Simulator

Je débogue cela à l'aide de Chrome, donc si vous rencontrez des problèmes, vous m'aideriez si vous pouviez vérifier s'ils apparaissent également dans Chrome. N'hésitez pas à m'envoyer quelque commentaire que ce soit.

Le simulateur LMC est à www.peterhigginson.co.uk/LMC et le simulateur RISC est à www.peterhigginson.co.uk/RISC.

Clause de non-responsabilité - Je ne suis en aucun cas associé à AQA et ce simulateur est basé sur leur documentation publique.

G. Webographie

- https://info.blaisepascal.fr/nsi-simulateur-de-cpu#Simulation
- http://www.peterhigginson.co.uk/AQA/
- https://pixees.fr/informatiquelycee/n_site/nsi_prem_sim_cpu.html
- https://pixees.fr/informatiquelycee/n_site/nsi_prem_von_neu.html
- https://qkzk.xyz/docs/nsi/cours_premiere/architecture/assembleur_aqa_intro/
- http://nsinfo.yo.fr/nsi_prem_sim_cpu.html
- http://peterhigginson.co.uk/AQA/info.html
- https://ecariou.perso.univ-pau.fr/cours/archi/tp-AQA.html
- https://archives.lyceum.fr/2019-2020/1g/nsi/6-architectures-materielles-et-systemes-dexploitation/2-jeu-dinstructions-du-processeur/exo
- https://peterhigginson.co.uk/ARMlite/Programming%20reference%20manual_v1_2.pdf
- https://www.gladir.com/CODER/ASM1802/reference.htm
- http://iamjmm.ovh/NSI/assembleur/site/assembleur.html
- http://www.mathartung.xyz/nsi/exo_architectures_assembleur.html