# TD ASM - 2

## 1 - Le registre de status

Pour l'instant le code est linéaire : il est lu depuis le début et est exécuté instruction après instruction, sans revenir en arrière. Pour changer cela, il faut pouvoir faire un test, et ce test se fera sur les valeurs contenues dans le registre de status (celui qui est tout en bas à droite).

Le registre de status va garder quatre informations, quatre « flags » qui seront à 1 si la condition est vrai et à 0 si elle est fausse.
Ces 4 conditions sont :
- N : négatif
- Z : zéro
- C : retenu (carry en anglais)
- V : débordement (oVerflow en anglais)

Ces informations sont mises à jour en fonction des instructions exécutées.
L'instruction CMP permet notamment de comparer les valeurs de 2 registres, en soustrayant un registre à un autre. Si les 2 valeurs sont égales, alors le flag « Z » sera à 1 (car le résultat de la soustraction sera 0).
Il y a d'autres instructions de comparaison (CMN, TST, TEQ). A vous de lire la documentation pour voir les différences.

Les instructions normales (ADD, SUB, …) peuvent aussi modifier le registre de status, mais ceci uniquement si le programmeur le demande. Pour cela, il suffit d'ajouter un « S » à la fin du nom de l'instruction.
Par exemple, SUB fait une soustraction entre 2 registres, mais n'affecte pas le registre de status. SUBS, par contre, fait la soustraction et met à jour le registre de status si besoin.

Une fois qu'il y a une condition mise à jour, il faut pouvoir changer l'exécution du code en fonction de cette condition.

Cela peut se faire de 2 manières.
Tout d'abord, quasiment chaque instruction « normale » peut avoir une condition d'exécution. Cela signifie que l'instruction ne s'exécute que si la condition est réalisé. Cela s'indique au niveau du code en ajoutant le suffixe de la condition au nom de l'instruction.
Par exemple, l'instruction ADDEQ ne s'exécutera que si le flag Z est à 1 dans le registre de status. Si la condition n'est pas réalisé, alors l'instruction n'est pas exécutée.
Une seconde manière est d'utiliser l'instruction « B » (pour « branch »). Dans ce cas, l'instruction B est suivi du nom d'un label (un label est simplement un nom mis devant une ligne de code, qui permet de référencer cette ligne). Comme les autres instructions, B peut avoir une condition attachée, ce qui en fait un saut conditionnel. Sans condition, le saut aura toujours lieu.

Muni de ces informations, faites un premier programme qui teste 2 valeurs entrées dans des registres, et qui met un 1 dans r4 si les 2 valeurs sont égales, et un 1 dans r5 si elles sont différentes.

# 2 - Les décalages

Comme vous avez pu le voir, il n'y a pas d'instruction de multiplications. C'est une limitation de l'émulateur, il y a bien des instructions de multiplication sur ARM. Cependant, ces instructions sont couteuses et sont le plus souvent remplacées par des opérations de décalage. En effet, vous pouvez toujours faire des multiplications (et des divisions) par des puissances de 2 grâce aux décalages.

Les décalages peuvent se faire par la gauche (multiplication) ou par la droite (division), et peuvent (ou non) garder le bit de poids fort (ou faible), grâce aux instructions LSL, LSR, ASR, ROR et RRX.

En utilisant ces instructions, modifiez votre calculatrice pour ajouter une multiplication de 10 par 35.

Les instructions ARM ont une particularité : il est possible de faire le décalage directement dans une instruction « normale ». Pour cela, il suffit de faire suivre l'instruction par une virgule ',' puis par le décalage voulu. Cela n'est pas simplement deux instructions qui se suivent, mais bien une et une seule instruction qui fait les 2 choses en même temps (1 top d'horloge).

Modifiez votre programme pour qu'il utilise cette fonctionnalité.

En ajoutant ces décalages aux opérations d'additions et de soustractions, et en enchainant ces opérations, vous êtes en fait capable d'exécuter n'importe quelle multiplication par n'importe quel nombre 32 bits, en un nombre faible d'instruction (et donc un faible temps d'horloge). C'est là la puissance de l'assembleur ARM. Attention, on ne parle ici que de multiplication « statique », dont les nombres sont connus au départ.

# 3 - Les données immédiates

Dans le programme précédent, vous avez peut-etre eu des soucis pour entrer certaines valeurs.

C'est normal.

Rappelez-vous qu'une instruction RISC a une taille fixe (ici, 32 bits). Cela signifie notamment que tous les paramètres de l'instruction doivent se trouver dans ces 32 bits. Or, il y a déjà un certain nombre de bits qui sont pris pour l'instruction elle-même. Il n'est donc pas possible de rentrer n'importe quelle valeur comme paramètre d'une instruction.

Pour contourner cela, les concepteurs d'ARM ont eu une idée ingénieuse : ils ont regardés les statistiques d'utilisation de leur code assembleur, et ont remarqué que les valeurs données avaient souvent toujours la même forme : quelques 1 regroupés ensembles, mais qui pouvaient être n'importe où dans les 32 bits.

Du coup, ils ont inventé le stratagème suivant pour coder une valeur immédiate dans une instruction : 8 bits sont réservés à la valeur elle-même, et 4 bits sont réservé à la valeur de décalage, qui permet de positionner ces 8 bits dans les 32 bits.

Il est ainsi possible de stocker n'importe quelle puissance de deux, ou n'importe quel nombre dont la valeur binaire possède au maximum 8 « 1 », et tous regroupés dans la même suite de 8 bits.
Ainsi, la valeur 0b0000101110010000 peut être stockée sans soucis, mais pas 0b1000101110010000. Pour une telle valeur, il faut alors soit combiner les opérations, soit mettre la valeur en dur dans le code et la référencer.
Cela est fait automatiquement par l'assembleur, avec la pseudo-instruction « LDR rx, =0x…. ».

Par exemple :

```
MOV r0, #0xFF00FF00
```

est interdit par l'assembleur, alors que :

```
LDR r0, =0xFF00FF00
```

est permis.

Vérifiez que c'est bien le cas.

# 4 - un vrai programme

Grace à ces instructions, et en utilisant les conditions, faites un petit programme qui permet de compter le nombre de 1 d'une valeur en mémoire, et qui met le résultat en mémoire.

Pensez bien à appliquer toutes les bonnes pratiques décrites dans ce TD. En particulier, la valeur à tester ainsi que le résultat (nombre de 1) devront obligatoirement etre définis par des instructions DCD dans la mémoire.

# 5 - Endians

Il y a 2 manières de représenter la mémoire : soit comme un tableau, avec la case 0 en haut à gauche, avec les adresses qui vont vers la droite et en bas :

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

Soit comme un immeuble, avec les adresses hautes en haut et les adresses basses en bas. Du coup, les adresses se lisent de droit à gauche :

| | | | |
|---|---|---|---|
| 15 | 14 | 13 | 12 |
| 11 | 10 | 9 | 8 |
| 7 | 6 | 5 | 4 |
| 3 | 2 | 1 | 0 |

Cela ne change rien à la mémoire, ce n'est qu'une manière de la représenter.
Par contre, cela va changer du coup l'ordre de lecture des octets, donc la manière dont on va stocker une information qui dépasse un octet, par exemple 0x12345678.
Cette valeur est composée de 4 octets : 0x12, 0x34, 0x56, 0x78. 0x12 est dit l'octet de « poid fort », et 0x78 l'octet de « poid faible ».

Dans la première vue (tableau), si l'on veut que la valeur soit lisible directement, il faut mettre l'octet de poid fort en premier (0x12 en case 0), puis continuer et l'octet de poid faible à la fin (0x78 en case 3) :

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 |

Inversement, dans la seconde vue (immeuble), si l'on veut que la valeur soit lisible directement, il faut mettre l'octet de poid faible en premier (0x78 en case 0), puis continuer et l'octet de poid fort à la fin (0x12 en case 3) :

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 |

Cette différentiation s'appelle le « boutisme » (endian en anglais). Chaque processeur respecte l'une ou l'autre de ces manières de stocker les infos en mémoire. Ceux qui respectent la première manière sont appelé « Big Endian » (car il mette le poid fort en tete), les autres « Little Endian » (poid faible en tête).

Déterminez si le processeur ARM est « Big Endian » ou « Little Endian ».

Pour plus d'information :
https://fr.wikipedia.org/wiki/Boutisme

A savoir : en fait le processeur ARM est « bytesexual », c'est à dire qu'il est possible de changer son mode en l'un ou l'autre des deux « endians ». Mais il a un mode « natif » si l'on ne demande pas de changement, et c'est ce mode que l'on vous demande de trouver.

# 6 - Adressage indirect indexé

Pour l'instant, nous avons vu lors des accès mémoires la possibilité d'accéder à un mot (4 octets) en mettant son adresse dans un registre et en utilisant la notation [rx] pour accéder au contenu de la case.

Cette facon d'accéder à la mémoire est simple et efficace pour accéder à une case mémoire (une variable). Mais elle l'est beaucoup moins si l'on veut accéder à un tableau.
En effet, dans ce cas, il faudrait pour chaque case modifier l'adresse mémoire pour pointer sur la bonne case. Si c'est un accès linéaire, c'est encore gérable, mais si c'est un accès à une case random…

Heureusement, ce cas a été prévu par les concepteurs de l'ARM. Il faut alors utiliser l'adressage indexé :

```
LDR rx, [ry, OFFSET]
```

OFFSET peut être une valeur immédiate, un registre ou un registre avec décallage (comme vu dans le TD dernier).

Pour tester cela, écrivez un programme avec ce tableau :

```
TABLE DCD   0xDEADFEED, 0xC0DEC0DE, 0xCAFEBABE, 0xB105F00D, 0xBAAAAAAD
```

Et écrivez de 3 manières différentes l'accès à la 2eme case (0xC0DEC0DE) :
- une fois avec un offset en valeur immédiate
- Une fois avec un offset dans un registre
- Une fois avec un offset dans un registre décalé

ATTENTION : vous accédez non pas à un octet mais à un mot, c'est à dire une zone de 4 octets. Cette zone doit forcément être alignée sur une adresse multiple de 4 (on se déplace de mots en mots, pas d'octet en octet).
Il est possible de se déplacer d'octet en octet, simplement en ajoutant la lettre « B » après l'instruction de chargement (LDR ou STR).

# 7 - Adressage indirect indexé auto-incrémenté

Quand on se déplace linéairement dans un tableau, par exemple pour un parcours exhaustif ou une recherche, il faut incrémenter l'offset d'une même constante pour chaque case.
Il est possible bien sur d'utiliser une instruction ADD séparée pour cela. Mais comme il s'agit d'un cas fréquent, les concepteurs de l'ARM ont inclu, dans le mode d'adressage, la possibilité d'avoir une auto-incrémentation de l'offset directement dans l'instruction.

Cet auto-incrémentation peut avoir 2 formes : une pré-incrémentation, où la valeur est incrémentée avant d'être utilisée, et une post-incrémentation, ou la valeur est utilisée puis incrémentée. Cela permet de gérer les 2 cas standards : celui où le registre pointe déjà vers la première case du tableau (post-incrémentation) et celui où le registre pointe sur la case avant le début du tableau (pré-incrémentation).

Pour faire une pré-incrémentation, il suffit de rajouter un « ! » à la fin du crochet fermant :

```
LDR rx, [ry, OFFSET]!
```

Ainsi, l'instruction suivante :

```
LDR r2, [r0, #4]!
```

change la valeur de r0 en l'incrémentant de l'offset #4. r0 pointe donc désormais sur la 2eme case du tableau. r2 recoit la valeur de la 2eme case du tableau.

Pour faire une post-incrémentation, il suffit mettre l'offset non pas entre les crochets mais après les crochets :

```
LDR rx, [ry], OFFSET
```

Ainsi, l'instruction suivante :

```
LDR r2, [r0], #4
```

place dans r2 la valeur de la première case du tableau (pointé par r0), puis change la valeur de r0 en l'incrémentant de l'offset #4. r0 pointe donc désormais sur la 2eme case du tableau.
Une très bonne documentation sur ces modes d'adressages se trouve ici : https://web.archive.org/web/20200728032926/https://heyrick.eu/aw/index.php?title=LDR

Avec le même tableau que précédemment, faites un programme qui place successivement les 4 valeurs du tableau dans les registres r2, r3, r4 et r5.
Essayez de l'écrire avec le moins de lignes possibles.

# 8 - Exercices

Faire un programme qui recopie en mémoire le contenu du tableau donné précédemment.
Dans un premier temps, mettez en dur la taille du tableau dans une constante, et servez-vous en pour controler votre boucle.
Dans un second temps, ajoutez la valeur « 0x00 » à la fin du tableau, et servez-vous en pour controler votre boucle.