Les programmes du TD5 ne faisaient appel qu'aux registres, et d'autre part, la mémoire centrale (RAM) ne contenait que des codes d'instructions. Dans ce TD, nous allons voir comment utiliser la mémoire pour stocker des données et aussi comment effectuer des calculs arithmétiques plus élaborés.

👉 Ouvrez l'URL https://perso.univ-rennes1.fr/pierre.nerzic/IntroArchi/CimPU/.

Agrandissez la fenêtre (ou zoomez) pour travailler confortablement.

1. Accès mémoire dans une instruction

La mémoire centrale peut être lue ou modifiée par l'unité centrale lors de l'exécution d'une instruction. Dans CimPU, la syntaxe est simple : quand un opérande d'instruction est écrit sous la forme `[nombre]`, alors il s'agit d'une cellule dans la mémoire centrale, et ce nombre est son adresse. Exemple :

```
        LD  R0, [56]    ; R0 = contenu de la cellule adresse 56
        ADD R0, [57]    ; ajouter à R0 la cellule d'adresse 57
        ST  R0, [58]    ; stocker R0 dans la cellule 58
```

Ces instructions utilisent les cellules mémoire d'adresse 56, 57 et 58, les deux premières en consultation (lecture mémoire) et la dernière en modification (écriture mémoire). Pourquoi ces adresses-là ? En fait, n'importe quelle adresse convient, du moment qu'elle n'est pas dans le programme. Donc vous pouvez changer pour d'autres adresses si vous voulez.

Attention à ne pas confondre `LD R0,56` avec `LD R0,[56]`. La première met la valeur constante 56 dans `R0` ; la seconde va chercher l'octet qui se trouve dans la mémoire case 56 et le met dans `R0`. Un nombre entre crochets `[nb]` est comme un indice de tableau en langage C : `T[2]` désigne la case d'indice 2. Dans CimPU, c'est comme si on écrivait `LD R0, mémoire[56]`.

1.1. Vocabulaire (pour information)

Vous connaissez maintenant plusieurs sortes d'opérandes : registre, constante et cellule mémoire. On les appelle « modes d'adressage ».

- Lorsque l'opérande est un registre, on parle d'« adressage registre ».
- Lorsqu'on fournit une constante, on parle d'« adressage immédiat ». C'est parce que l'opérande se trouve dans le programme, dans l'octet qui suit immédiatement le code d'instruction.
- Lorsqu'on fournit une adresse mémoire, on parle d'« adressage direct ». L'adresse se trouve aussi dans l'octet qui suit le code d'instruction, mais ensuite, il y a un accès mémoire supplémentaire pour lire la cellule mémoire voulue.

Dans les prochains TD, nous verrons d'autres modes d'adressage permettant de travailler avec les tableaux et aussi les paramètres et variables locales des fonctions.

Certaines instructions, `LD`, `ST` et les arithmétiques et logiques acceptent tous ces modes d'adressages. D'autres instructions comme `INC` et `DEC` n'ont que le mode d'adressage registre. C'est parce qu'il faudrait trop de codes différents pour représenter toutes les possibilités. Dans un x86_64 ou amd64, il peut y avoir jusqu'à 15 octets par instruction ; dans CimPU, seulement 2.

1.2. Nommage des adresses : *labels*

Il est très recommandé de nommer les cellules mémoire, avec des « labels ». Un label est un nom pour une case mémoire. La syntaxe est :

```
label:  instruction ou donnée
```

Le label est défini avec l'adresse courante, celle de l'instruction ou de la donnée.

Voici le même programme que le précédent, mais en beaucoup plus compréhensible : 

```
calcul_total:
        ; calcule total = prix + taxe
        LD  R0, [prix]    ; R0 = prix
        ADD R0, [taxe]    ; ajouter la taxe à R0
        ST  R0, [total]   ; prix total = prix + taxe
        HLT

        ORG 56            ; données à partir de l'adresse 56 (en base 10)
prix:   DB 134
taxe:   DB 18
total:  DB 0
```

La directive `DB nombre` (*data byte*) place un nombre en mémoire. Ce nombre est une donnée, pas une instruction. Il n'y a aucune différence entre elles, car ce sont des octets. Donc il faut clairement séparer les deux, programmes et données, et s'assurer que jamais on n'exécutera des données en tant qu'instructions. Il faut **toujours** prévoir une instruction `HLT` (*halt*) à la fin d'un programme.

Utiliser des labels rend le programme plus solide parce qu'on peut déplacer les données en mémoire, ça ne cassera pas le programme (à condition de le ré-assembler). Dans un programme C, toutes les variables et toutes les fonctions deviennent des labels.

On n'est pas obligé d'indenter les instructions (mettre 8 espaces), mais ça améliore la lisibilité.

👉 Assemblez ce programme et allez voir la mémoire, à l'adresse hexadécimale `38`, c'est 56 en base 10, là où on a mis les données. Vous devez trouver les nombres $(86)_{16} = 134$, $(12)_{16} = 18$, suivi de 0. Ce sont les données initialisées par `DB`.

👉 Mettez la mémoire centrale en mode `Contexte d'exécution`. Ça donne une vue très claire du programme et des cellules mémoire associées à des labels. Il y a la liste des variables globales et la traductions des octets du programme en instructions, ce qu'on appelle le « désassemblage ».

👉 Lancez le programme précédent par `Reset` puis `RUN` et examinez la mémoire. L'adresse $58 = (3A)_{16}$ doit avoir changé. Passez la souris au dessus pour voir la valeur en base 10.

1.3. Exercice

👉 Compléter ce programme pour qu'il additionne `prix1`, `prix2` et `prix3` dans `total` : 

```
        ; addition de trois cases mémoire dans une quatrième
        ; ...à vous de le faire...
        HLT
```

```
        ORG 30          ; données
prix1:  DB  %1010       ; %NNN : en base 2, ici (1010)2 = 10
prix2:  DB  $64         ; $NN : en base 16, ici (64)16 = 100
prix3:  DB  1           ; NNN : en base 10
total:  DB  0
```

👉 Assemblez ce programme en effaçant la mémoire.

👉 Lancez le programme (`Reset` puis `RUN`). Vous devriez voir le résultat de l'addition dans la cellule d'adresse 33 (en base 10, c'est à dire 21 en base 16).

1.4. Notation des constantes (pour information)

Dans CimPU et d'autres assembleurs, les constantes peuvent être écrites de plusieurs manières :

- en base 10, exemple `LD R0, 23` met 23 dans `R0`
- en base 16, exemple `LD R0, $A7` affecte `R0` avec $(A7)_{16} = 167$,
- en base 2, exemple `LD R0, %10110101` affecte $(10110101)_2 = 181$ à `R0`,
- en ASCII, exemple `LD R0, 'a'`, ça affecte le code de `a` c'est à dire 97 au registre `R0`.

Il existe encore une possibilité, c'est de définir un label avec une valeur précise, et d'utiliser cette valeur dans une instruction :

```
        LD R0, nombre

nombre: EQU 23
```

La directive `EQU` (*equals*) donne une valeur au label, sans occuper d'octet en mémoire. C'est comme écrire `#define nombre 23` en langage C.

2. Additions et soustractions

On va utiliser quelques instructions de calcul. Pour l'instant, ça n'a pas encore d'utilité.

Voici un programme : 

```
        ; calcule V1 + V2
        LD  R0, [V1]    ; R0 = V1
        ADD R0, [V2]    ; R0 = R0 + V2
        OUT R0, 10      ; afficher R0
        HLT             ; stop

        ORG 32
V1:     DB  134
V2:     DB  18
```

Ce programme écrit le résultat du calcul sur le DSKY en mode 10. Dans ce mode, quand on fait un `OUT reg,10`, ça écrit le registre dans différentes bases et conventions.

👉 Assemblez ce programme en effaçant la mémoire.

☛ Mettez le DSKY en mode 10 et lancez le programme (`Rst+Run`). Vous devriez voir le résultat de $134 + 18$.

☛ Modifiez le programme pour qu'il calcule $V1 - V2$. L'instruction s'appelle `SUB`, à mettre à la place de `ADD`.

☛ Remaniez le programme pour qu'il calcule $V2 - V1$ et cette fois, le résultat sera négatif. C'est à dire qu'il faudra l'interpréter en convention $C2^8$.

3. Décalages

Voici figure 1, deux opérations un peu bizarres à première vue, les décalages à gauche et à droite.

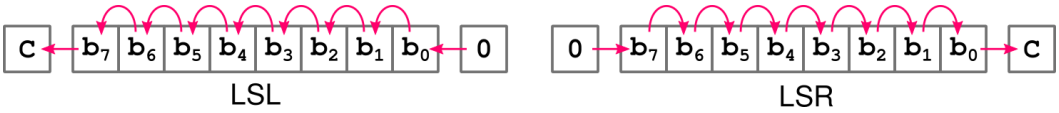

Figure 1: Décalages logiques (`Logic Shift Left/Right`)

Ces deux opérateurs décalent les chiffres d'un nombre écrit en base 2. Que pensez-vous d'une opération similaire en base 10 ? Par exemple, si on prend 1234 et qu'on le décale vers la gauche en ajoutant un 0, on obtient 12340. Et 1234 décalé vers la droite devient 123. Quel est le rapport entre ces nombres ?

On les utilise ainsi : `LSL reg` ou `LSR reg`, donc `LSL R0`, `LSR R1`...

Il y a une case `C` dessinée à gauche et à droite. On verra ce qu'elle est au prochain TD. Elle a le même rôle qu'une retenue dans une addition.

3.1. Moyenne de deux entiers

Comment calcule-t-on la moyenne de deux entiers ? C'est la somme divisée par 2. Comment divise-t-on un entier par 2 ?

☛ Complétez et testez le programme suivant qui calcule la moyenne des deux entiers :

```
        ; calcule la moyenne de V1 et V2
        ; ...à vous de le faire...
        OUT R0, 10      ; afficher le résultat = moyenne de V1 et V2
        HLT             ; stop

        ORG 32
V1:     DB  72
V2:     DB  24
```

Si vous essayez avec des nombres quelconques, il y aura des erreurs de calcul :

- des « débordements » avec des entiers trop grands, la somme dépasse 255 et devient fausse, et elle le reste ensuite quand elle est divisée par 2.
- des problèmes de signe quand vous souhaitez que les nombres soient signés. Par exemple, la moyenne de -88 et -36 n'est pas négative, et en plus elle est fausse. Alors, au lieu d'utiliser

l'instruction `LSL` ou `LSR`, utilisez `ASL` ou `ASR`. Elles sont encore plus bizarres, figure 2, sauf si vous vous rappelez où est le signe d'un entier codé en convention $C2^8$, et alors voyez que b_7 est inchangé par un décalage.

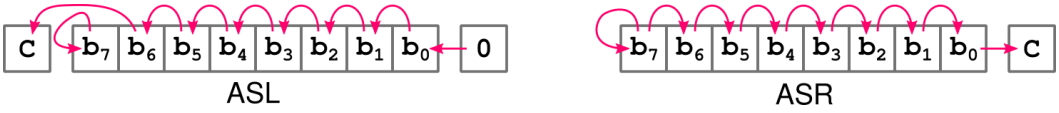

Figure 2: Décalages arithmétiques

3.2. Multiplication par une constante

Un dernier exercice : comment multiplier un nombre par 5 ? Il suffit d'additionner ce nombre à lui-même multiplié par 4. On vient de voir comment multiplier un nombre par 2...

☛ Complétez et testez le programme suivant qui multiplie le nombre V par 5 :

```
        ; multiplie V par 5
        ; ...à vous de le faire...
        OUT R0, 10      ; afficher le résultat = V * 5
        HLT             ; stop

        ORG 32
V:      DB  12          ; un nombre pas trop grand
```

Le nombre V ne doit pas être trop grand car on se heurte tout de suite aux limites de représentation.

☛ Remaniez ce programme pour qu'il multiplie V par 6 (deux instructions à permuter).

Ce programme doit être reconstruit si on veut multiplier par un autre nombre, par exemple par 11. On verra dans un prochain TD comment permettre une multiplication par n'importe quel nombre (pas trop grand).