TD ASM - 1

1 - VisUAL 2

Comme indiqué pendant le cours, nous allons apprendre l'assembleur pour le processeur ARM. Pour cela, nous allons utiliser un émulateur, qui prendra le code assembleur et l'exécutera en montrant les résultats dans les registres comme dans la mémoire.

Commencez, si ce n'est déjà fait, par télécharger l'émulateur VisUAL 2, en suivant les liens donnés sur Arche. Préférez l'installation de l'émulateur sur votre machine hôte, et non pas dans la VM Linux fournie par l'école.

Attention, on utilise la version 2 de VisUAL (il y a une version 1 plus ancienne mais avec plus de bugs), et en particulier VisUAL 2 version 1.06.09 ou 1.06.10.

Une fois installé, lancez l'émulateur et vérifiez qu'il fonctionne bien en allant dans le menu « help » et en chargeant le code d'exemple donné (« load complex demo code »).

L'appui sur le bouton « Run » doit lancer le programme sans erreur.
Il doit juste y avoir écrit en vert : « Exécution Complete »Ω

Les pages d'aides de l'émulateur se trouvent ici :
https://tomcl.github.io/visual2.github.io/guide.html

En particulier la page des instructions :
https://tomcl.github.io/visual2.github.io/list.html#instructions

La page des instructions de VisUAL 1 est parfois plus lisible :
https://salmanarif.bitbucket.io/visual/supported_instructions.html

L'émulateur a une interface simple : à gauche se trouve l'éditeur de texte, à droite les registres (avec notamment en bas le registre de status), la mémoire et les symboles . L'éditeur de texte peut avoir plusieurs onglets ouverts (mais un seul programme qui tourne à la fois).
La barre de boutons en haut permet de gérer l'exécution du programme (à gauche) et le format d'affichage des zones mémoires (à droite) :

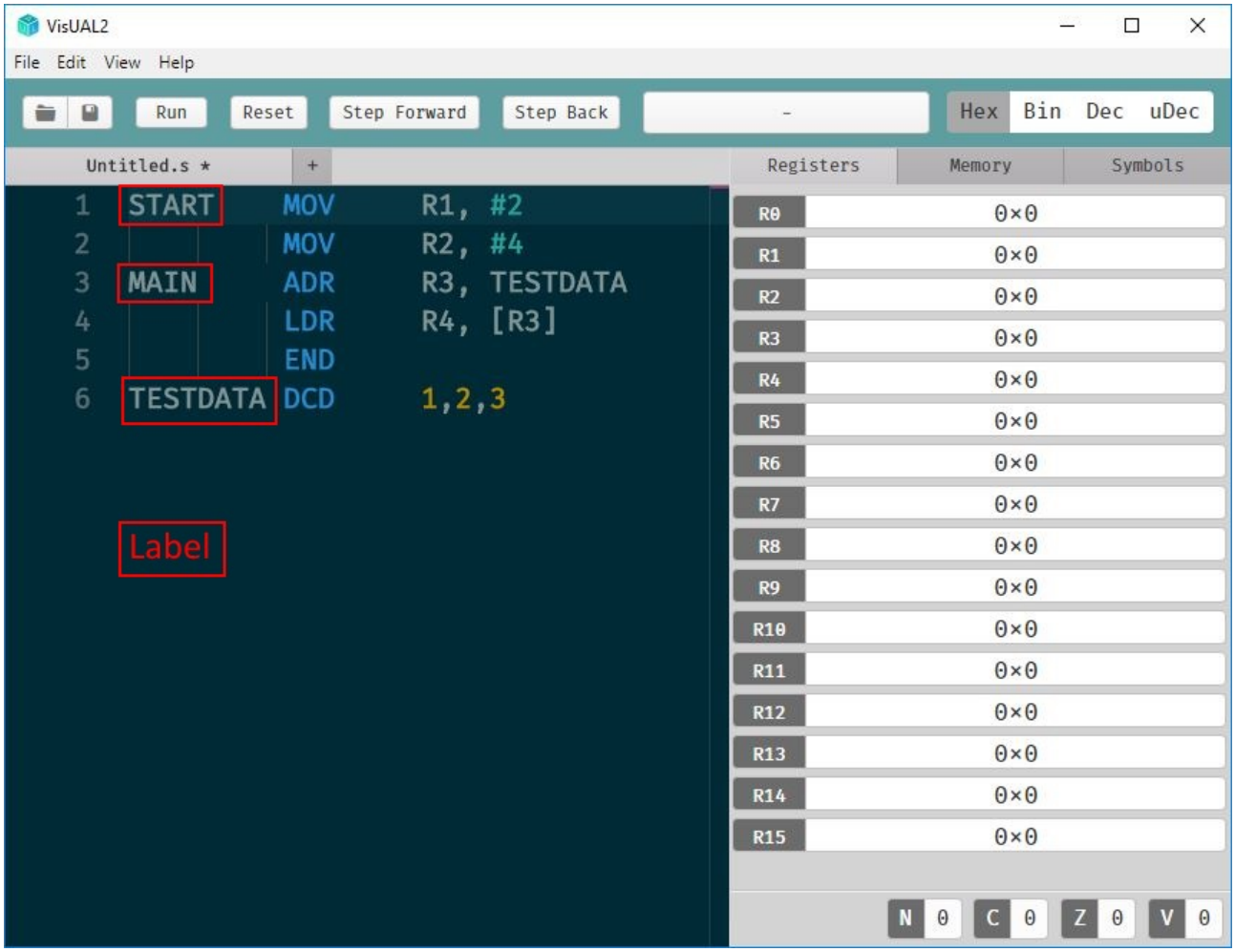
Le code assembleur est tapé dans l'éditeur de code à gauche, puis est exécuté en appuyant sur le bouton « Run » au dessus.

Afin de tester que votre installation est correcte, tapez le code suivant dans l'éditeur de texte :

```
        MOV     r0, #1
```

N'oubliez pas l'indentation à gauche. Pour l'instant ce n'est pas utile mais c'est une bonne pratique à prendre, cela servira lorsque nous aurons des labels.

Si tout se passe bien, vous devriez voir, à droite, le registre r0 changer de 0 à 1.



Prise en main

Afin de tester les autres possibilités de VisUAL, tapez le code suivant (nous verrons le sens après) et éxécutez-le :

```
VAL1        DCD         10      ; definition d'une valeur
VAL2        DCD         20
RES         FILL        4       ; reservation d'un espace mémoire

            MOV         r2, #0x20

            ADR         r0, VAL1
            ADR         r1, VAL2
            ADR         r2, RES

            LDR         r3, [r0]
            LDR         r4, [r1]

JUMP        ADD         r5, r3, r4

            STR         r5, [r2]

            END
```

L'instruction « END » à la fin n'est pas nécessaire (le programme s'arrête sans) mais là encore c'est une bonne pratique.

Mode Debug

Appuyez sur « Step > » au lieu de « Run ». Le code se lance alors en mode « pas à pas » (instruction après instruction), dans lequel vous pouvez suivre l'action de chaque instruction. Pour avancer d'une instruction, appuyez sur « Step > », pour revenir en arrière appuyez sur « < Step ».

La fenêtre des symboles affichent tous les symboles reconnus dans le programme, avec le type de leur emplacement (data pour une variable, code pour un label), leur nom et leur valeur.
La fenêtre de mémoire affiche la mémoire correspondante aux pointeurs trouvés dans le code, avec pour chaque ligne son pointeur, son adresse mémoire et sa valeur, qui peut etre affichée sous la forme des 4 octets de la ligne ou de la valeur totale du mot (valeur sur 32 bits).

2 - Premières instructions : une calculatrice

Les premières instructions que nous allons voir vont vous permettre de créer une calculatrice simple (addition et soustraction) :

- MOV permet de mettre une valeur dans un registre. Cela peut être une valeur immédiate (avec # devant), que ce soit en décimal (#15) ou hexadécimal (#0xF), ou bien la valeur d'un autre registre.
- ADD permet d'additionner 2 valeurs de registres.
- SUB permet de soustraire 2 valeurs de registres.

Avec ces 2 méthodes, faites un programme qui met une valeur dans r0 (par exemple 10), une valeur dans r1 (par exemple 20), qui additionne les 2 valeurs et qui met le résultat dans r2. Faites bien attention à l'ordre des paramètres. Aidez-vous pour cela de la page des instructions : https://tomcl.github.io/visual2.github.io/data-processing.html

Faites la même chose avec une soustraction, d'abord en soustrayant 20 à 10, puis dans le sens inverse. Regardez les valeurs obtenues en décimal puis en hexa. Que constatez-vous ?
Est-ce normal ? Si vous pensez que ce n'est pas normal, voici un peu de lecture :
https://fr.wikipedia.org/wiki/Complément_à_deux

Bonne pratique

Comme dans n'importe quel langage, on évite de garder des valeurs « importantes » perdues en plein milieu du code, on préfère toujours avoir des constantes déclarées en début de programme, afin de pouvoir les modifier facilement.
Pour cela on va utiliser la pseudo-instruction « EQU » (on parle de « pseudo-instruction », car elle ne sert qu'à l'assembleur, elle n'est pas traduite en code binaire dans le programme).
En vous aidant de la page des instructions, déclarez vos deux valeurs comme des symboles, et utilisez ces symboles dans votre code.
Vous pouvez vérifier que vos symboles sont bien déclarés dans la fenêtre des symboles.

3 - Accès à la mémoire

Accéder aux registres est une première chose, mais il serait plus pratique de mettre les valeurs quelque part en mémoire, et d'y accéder ensuite.

Pour accéder à la mémoire, il nous faut 3 instructions :
- ADR qui va charger une adresse mémoire dans un registre

- LDR qui va charger une valeur dans un registre, possiblement depuis la mémoire
- STR qui va remettre une valeur d'un registre, possiblement en mémoire

Les adresses mémoires sont, comme les registres, limitées ici à 32 bits / 4 octets (ce qui donne donc un espace d'au maximum 4Go de mémoire).

Avoir une adresse mémoire dans un registre n'est utile que si l'on peut manipuler non pas l'adresse elle-même mais surtout son contenu. Comme ceci est une fonctionnalité de base du CPU (accéder et manipuler la mémoire), il y a plusieurs manière de le faire. On parle alors de « mode d'adressage ».
Celui que nous allons voir maintenant est le plus simple : le mode « register offset », noté avec des crochets ([ ]). Il permet tout simplement d'utiliser, en écriture ou en lecture, non pas la valeur du registre, mais la valeur pointée par cette valeur. La valeur du registre est donc vue comme une adresse mémoire. Il faut donc commencer par charger dans le registre une adresse mémoire.

La pseudo-directive DCD permet de créer dans la zone mémoire « data » une zone avec une valeur. Nous n'avons pas encore parlé de l'organisation de la mémoire (zone de code et zone data), nous verrons cela dans le prochain cours. Sachez seulement que le code est stocké en mémoire, comme les données, et que pour l'instant, le code est mis « avant » (adresses plus basses) les données. DCD permet donc de réserver un « mot » (32 bits - 4 octets) dans la zone data, de lui donner un nom, et d'y mettre une valeur.
Ce nom est ensuite référençable pour pouvoir y accéder.

Pour faire la même chose mais sans mettre de valeur dedans, il faut utiliser la pseudo-instruction FILL. Attention cette instruction prend en paramètre un nombre d'octets (pas de mots). Il faut donc obligatoirement que cela soit un multiple de 4.

Utilisez donc toutes ces informations pour mettre les valeurs voulues en mémoire, puis pour stocker le résultat en mémoire

Visualisation

Lorsqu'on utilise des adresses mémoires, il peut être intéressant de voir directement dans le code les références et la mémoire.
VisUAL permet de faire ça facilement. Lorsque vous utilisez certaines instructions, comme LDR et STR, un bouton de visualisation « Pointer » apparait à coté de l'instruction. Il ouvre un popup vous donnant toutes les informations utiles concernant la référence à la mémoire. Je vous encourage donc à les utiliser.