UNIVERSITE Joseph FOURIER
Polytech' Grenoble

ALM Soft : Travail pratique No 1
Retenues et débordements, Mise au point avec "débogueur"

Ce premier TP permet de se familiariser avec l'environnement de compilation, exécution, observation et mise au point de programmes écrits en Assembleur ARM. Il permet également de mieux comprendre la signification des codes conditions arithmétiques mis à jour dans le registre d'état du processeur après exécution de certaines actions.

Pour effectuer ce TP vous devez au préalable recopier dans votre répertoire de travail les fichiers `add.s`, `es.s` et `assemble.sh` contenus dans le Placard.

1 Des additions

L'instruction `ADD` effectue la somme de deux entiers. L'instruction `ADDS` effectue la somme de deux entiers et provoque la mise à jour des codes de conditions arithmétiques `N, Z, C, V` contenus dans le registre d'état du processeur (`cpsr`).

On s'intéresse lors de ce travail à la valeur de la somme et aux codes de conditions arithmétiques `N, Z, C, V` rangés respectivement dans les bits 31, 30, 29 et 28 du registre d'état du processeur.

Le programme `add.s` donné en annexe calcule la somme des entiers V1 et V2 déclarés en zone data et affiche **en hexadécimal** la valeur de cette somme et le contenu du registre d'état juste après l'exécution de l'instruction `ADDS`.

1.1 Compilation et exécution d'un programme en assembleur ARM

Le programme `add.s` fait appel à une bibliothèque d'entrées-sorties (`es.s`) pour afficher à l'écran des entiers et des chaînes de caractères. Le fonctionnement de cette bibliothèque sera explicité en fin de semestre. Pour compiler ce programme il faut taper les commandes suivantes :

- `source /opt/gnu/bin/setenvarm.csh`
  cela vous donne alors accès aux commandes GNU ARM dont nous avons besoin pour la suite.
- `arm-elf-gcc -c -Wa,--gdwarf2,-L add.s`
  cela produit le fichier `add.o` qui contient du binaire translatable (il n'est pas complet, il ne peut être exécuté).
- `arm-elf-gcc -c -Wa,--gdwarf2,-L es.s`
  cela produit le fichier `es.o` qui contient du binaire translatable.
- `arm-elf-gcc -g -o add add.o es.o`
  cela produit le fichier `add` qui contient du binaire exécutable.

Ces trois dernières commandes peuvent être remplacées par la commande `assemble.sh add.s`.

Lorsque la compilation se termine sans erreur, le programme executable obtenu (`add`) peut alors être exécuté via la commande suivante : `arm-elf-run add`

1.2 Questions

1. Calculez, à la main, la somme de V1 et V2 par exemple en l'écrivant en binaire. Exécutez le programme. Vérifiez votre addition. Quelles sont les valeurs de `N, Z, C` et `V` ? Expliquez ces valeurs par rapport au calcul que vous avez fait à la main.
   On prendra soin d'interpréter les valeurs de V1 et V2 en supposant qu'ils sont codés soit en base 2, soit en complément à 2. On pourra ne s'intéresser qu'aux 4 bits de poids fort et considérer les résultats comme si le processeur travaillait sur 4 bits.
   Exemple : 0x5 + 0x8 = 0xd ; interprétations :
   – en base 2 sur 4 bits : 5+8=13,
   – en complément à 2 sur 4 bits : 5+(-8)= -3
2. Créez une copie du programme `add.s` (par exemple `add1.s`) dans laquelle vous modifierez les valeurs de V1 et V2 : respectivement `0x40000000` et `0x50000000`. Recompilez `add1.s` avec la commande `assemble.sh add1.s`. Effectuez le même travail que précédemment en exécutant `add1`.
3. Même chose avec les valeurs :
   `0x50000000 et 0xe0000000, 0x10000000 et 0xa0000000, 0xd0000000 et 0x60000000, 0xb0000000 et 0x40000000, 0xc0000000 et 0xb0000000, 0xd0000000 et 0xc0000000`

2 Utilisation du metteur au point : gdb

Un metteur au point (ou "débogueur") sert à suivre l'exécution d'un programme en pas à pas c'est-à-dire une ligne de programme après l'autre ou à modifier un programme en cours d'exécution. C'est ce que nous allons observer dans l'exercice suivant.

Un metteur au point sert aussi à chercher des erreurs dans un programme en stoppant celui-ci justement à l'endroit où l'on soupconne l'erreur... C'est une des raisons pour lesquelles ce genre d'outils doit être maîtrisé.

Reprenez le programme `add.s` et tapez les commandes suivantes :

1. `arm-elf-gdb add`
   On lance le débogueur, nous sommes désormais dans l'environnement `gdb`.
2. `target sim`
   On active le simulateur, ce qui permet d'exécuter des instructions en langage d'assemblage ARM.
3. `load`
   On charge le programme à exécuter dont on a donné le nom à l'appel de `gdb`.
4. `list` On voit 10 lignes du fichier source.
5. `list` On voit les 10 suivantes.
6. `list 10,13` On voit les lignes 10 à 13.

7. `break main`
   On met un point d'arrêt juste avant l'étiquette `main`.
8. `run`
   Le programme s'exécute jusqu'au premier point d'arrêt exclus : ici, on est donc juste avant la première instruction.
9. `info reg`
   Permet d'afficher en hexadécimal et en décimal les valeurs stockées dans tous les registres. Notez la valeur de `r15` aussi appelé `pc`, le compteur de programme. Elle représente l'adresse de la prochaine instruction qui sera exécutée.
10. `s` Une instruction est exécutée. `gdb` affiche une ligne du fichier source qui est la prochaine instruction (et qui n'est donc pas encore exécutée).
11. `info reg $pc`
    On peut spécifier un registre particulier. Quelle est la différence entre la nouvelle valeur de `pc` et celle que vous aviez notée précédemment ?
12. `s` puis encore une fois `s`
13. `info reg $r6 $r7`
    Les registres `r6` et `r7` contiennent respectivement les adresses des chaines `Ch1` et `Ch2`. Quelle est leur différence ? A quoi correspond-elle ?
14. `break 21` Point d'arrêt a la ligne 21 du fichier source.
15. `c` On continue l'exécution jusqu'au prochain point d'arrêt (exclus).
16. `info reg`
    Regardez en particulier les registres `r2`, `r3`, `r4`, `r5` et `cpsr`.
17. `s`
18. `info reg`
    A votre avis, a quoi sert l'instruction de la ligne 21 ?
19. `set $pc=` <la première valeur de `pc` que vous aviez observée>
    Nous replaçons le compteur de programme au début du programme !
20. `delete break` et vous répondez `y` à la question `Delete All Breakpoints ?`
21. `break 16`
22. `c`
23. `info reg $r2` `r2` contient l'adresse de V1.
    Nous allons le vérifier :
24. `x` <la valeur contenue dans `r2`, c'est-à-dire celle que l'on vient de lire>
    On reconnait la valeur de V1. On aurait pu faire aussi : `x &V1`.
    Nous allons modifier cette valeur :
25. `set V1 = 0x44000000`
26. `x &V1`
27. `break 20`
28. `c`
29. `info reg`
    Nous allons modifier aussi le deuxième paramètre de l'addition en modifiant directement le contenu du registre `r3` :
30. `set $r3 = 0x11111111`

31. `info reg $r3`
32. `s`
    On vient de faire exécuter l'addition : `ADDS r4, r2, r3`.
33. `info reg $r4 $cpsr`
    Vous observez dans `r4` la somme de `0x44000000` et `0x11111111` et les codes `N, Z, C` et `V` dans `cpsr`. Ainsi, vous auriez pu faire toute la première partie de ce travail pratique sans quitter `gdb`, avec un seul fichier source, et sans recompiler !

3 Des soustractions

On s'intéresse maintenant à `SUB` et `SUBS`.

1. Copiez le fichier `add.s` en `sub.s`. Remplacez l'instruction `ADDS` par `SUBS`. Pour V1 et V2 prenez les valeurs `0x60000000` et `0x50000000`. Exécutez ce programme. Il a effectué le calcul V1 - V2. Que valent `Z` et `C` ?
2. Recommencez cette expérience en inversant les valeurs de V1 et V2.
   C vaut 1 quand il n'y a pas d'emprunt pour une soustraction. Autrement dit lorsque la différence V1 - V2 est possible dans l'ensemble des ebtiers naturels (V1 $\geq$ V2), le bit C vaut 1.
   Prenez votre documentation technique ARM et regardez la figure 2. Que lisez-vous dans la colonne signification en face de la condition C ? et $\overline{C}$?
   Même question pour Z et $\overline{Z}$. Trouvez des valeurs pour V1 et V2 qui permettent de mettre en évidence vos conclusions sur Z et $\overline{Z}$.
3. On s'intéresse aux conditions dont le mnémonique est `HI` et `LS`. Pour quels types de valeurs de V1 et V2 la condition associée est-elle vraie ?
4. Dans le tableau de la figure 2, vous voyez des lignes pour lesquelles le mnémonique est associé à des comparaisons sur des nombres signés (`GE, LT, GT, LE`). Considérez les couples de valeurs suivantes pour V1 et V2 et interprétez les codes conditions après le calcul de la différence V1 - V2 en considérant les entiers V1 et V2 comme des entiers relatifs : `0x40000000` et `0xc0000000`, `0x40000000` et `0xe0000000`, `0xb0000000` et `0x60000000`, `0xf0000000` et `0xe0000000`.
   Refaites les interprétations en considérant les entiers V1 et V2 codés en base 2 (non signé : HS, LO, HI, LS).

4 Utilisation du metteur au point : `ddd`

Prenez votre documentation technique et retrouvez les principales commandes de `gdb` dans le paragraphe "Exécution avec un débogueur".

Dans cette documentation on vous explique comment utiliser une version graphique du metteur au point appelée : `ddd`. En utilisant cette documentation reproduisez la manipulation précédente mais en utilisant cette fois-ci `ddd`.

Annexe : le programme `add.s`

```
.data
```

```
V1: .word 0x30000000
V2: .word 0x40000000
Ch1: .asciz "impression de la somme"
Ch2: .asciz "impression du contenu du registre d'etat"

   .text
   .global main
main: STMFD sp!, {lr}

      LDR r6, relaisCh1
      LDR r7, relaisCh2

      LDR r2, relaisV1
      LDR r2, [r2]

      LDR r3, relaisV2
      LDR r3, [r3]

      ADDS r4, r2, r3 @ addition des deux entiers
      MRS r5, cpsr    @ sauvegarde du registre d'etat dans r5

      MOV r1, r6
      BL EcrChaine
      MOV r1, r4
      BL EcrHexa32    @ impression de la somme

      MOV r1, r7
      BL EcrChaine
      MOV r1, r5
      BL EcrHexa32    @ impression du registre d'etat

fin:  LDMFD sp!, {pc}

relaisV1: .word V1
relaisV2: .word V2
relaisCh1: .word Ch1
relaisCh2: .word Ch2
```