h e p i a

Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences
Western Switzerland

Laboratoire de systèmes numériques
D. Bechevet, A. Lescourt, V. Pilloux, F. Vannel

Assembleur avancé
Semestre hiver

Introduction

Dans ce laboratoire, vous allez approfondir vos notions d'assembleur et mieux comprendre le fonctionnement interne d'un microprocesseur. Les exercices proposés vous permettront de valider les méthodes d'appel de fonctions et passage d'arguments.

1 Notions d'assembleur

Pour ce laboratoire vous aurez besoin d'utiliser les instructions suivantes :

- **LDR** : *Copie* le **contenu d'une adresse** de l'espace d'adressage du microprocesseur vers un **registre**.
- **STR** : *Copie* le **contenu d'un registre** vers une **adresse** de l'espace d'adressage du microprocesseur.
- Les décalages ou rotations.

1.1 LDR

L'instruction LDR s'utilise de deux façons :

- **LDR Rd, [Rp]** : ceci correspond à copier dans le registre Rd le contenu de l'adresse pointée par le registre Rp. Soit en langage C, l'équivalent de `Rd = *Rp;` En assembleur il faut écrire le nom du registre qui contient l'adresse de l'objet pointé entre [].
- **LDR Rd, =nom** : ceci correspond à copier dans le registre Rd la valeur ***nom***. Pour cela ***nom*** doit être définit ailleurs dans le code à l'aide de la syntaxe suivante : **.equ nom, valeur** (exemple **.equ var, 0x12345678**). Dans ce mode d'utilisation **LDR** fait référence au registre **PC** auquel une valeur est additionnée ou soustraite à ce dernier. L'utilisation de cette instruction dans ce mode peut être assimilée à **MOV Rd, #nombre** mais permet de copier une valeur de 32 bits, ce que l'instruction **MOV** ne permet pas (référez vous à la documentation de l'instruction MOV pour vous en rendre comptes par vous-même).

1.2 STR

L'instruction STR s'utilise de la façon suivante :

- **STR Rd, [Rp]** : ceci correspond à copier le contenu du registre Rd à l'adresse pointée par le registre Rp. Soit en langage C, l'équivalent de `*Rp = Rd;` (Attention : autant pour **LDR** que pour **STR** le registre faisant référence de pointeur est écrit à la fin de l'instruction.

1.3 La stack

L'utilisation de la pile est un sujet abordé durant le cours. Cependant pour pouvoir rapidement mettre des données sur la pile puis les récupérer voici les instructions à utiliser

- **PUSH { R0 ,R3}** : ceci correspond à empiler le contenu du registre R0 puis R3 sur la pile. Il est possible de préciser dans l'instruction de 1 à plusieurs registres.
- **POP { R0 ,R3}** : ceci correspond à récupérer de la pile (dépiler) deux variables et les enregistrer dans R3 et R0. Il est possible de préciser dans l'instruction de 1 à plusieurs registres.

2 Le projet sous Eclipse

Vous importerez dans votre *workspace* le projet **laboratoire3**.

Le fichier **labo3_ASM2.c** contient la fonction **main** et l'appel aux fonctions des trois exercices.

Le fichier **labo3_assembleur.s** contient les fonctions assembleur à compléter

Pour tester votre code et le fonctionnement adéquat par rapport à votre exercice, vous devrez modifier la valeur de `exo` se trouvant dans le `main`

```
exo=1;
```

Il est également possible de modifier à l'aide du débugger la valeur de cette variable globale.

3 Exercices

Pour ces exercices, il vous est demandé de respecter les conventions sur les registres de ARM, à savoir que dans une fonction seuls les registres R0 à R3 peuvent être librement utilisés. Pour pouvoir utiliser les autres registres, il faut les sauvegarder au préalable (à l'aide de la pile par exemple) et les restaurer en quittant la fonction.

3.1 Variables en mémoire

Une fonction `UINT32 funcexo1(UINT32* valA, UINT32* valB)` reçoit en paramètre **deux pointeurs**. Ces pointeurs pointent vers deux adresses mémoire dont le contenu correspond à des nombres entiers non-signés.

Lors de l'appel de cette fonction, la valeur de ces pointeurs se trouvent dans les registres `R0` et `R1`.

Ecrire en assembleur une fonction capable de lire les valeurs passées en référence, faire l'addition de ces deux nombres et retourner le résultat s'il est correct. Si un dépassement de capacité provoque un résultat erroné, il faudra retourner la valeur `0`.

Le code C équivalent à cette fonction est :

```
c = *a + *b
```

3.2 Accès à un tableau

La fonction `UINT32 funcexo2(UINT32* tabA, UINT32* tabB)` reçoit en paramètres **deux tableaux** (c'est à dire deux pointeurs... ce qui est identique) et retourne une valeur non-signée.

Dans cet exercice vous devez implémenter une fonction en code assembleur qui effectuera l'opération **B[] = 2*A[]** lors de l'appel `ok = funcexo2(A,B);`

La valeur en retour (`ok`) est **1** s'il n'y a pas d'erreurs de calcul. Si lors des opérations il y a une erreur causé par un dépassements de capacité, la copie s'arrête immédiatement et la valeur retournée est **0** (pas ok).

La taille du tableau est considéré connu et fixe : `taille=10`

3.2.1 Bonus

Modifiez la fonction de l'exercice précédent pour effectuer la même opération mais sur une taille donnée du tableau. Le prototype de la fonction contient un nouvel argument **taille du tableau** :

```
UINT32 funcexo2(UINT32* tabA, UINT32* tabB, UINT32 taille)
```

Veuillez respecter les conventions sur les registres de ARM et si nécessaire utilisez la pile (stack).

3.3 Passage d'une structure en argument

Lorsque l'on désire passer une structure (de type C) à une routine assembleur, le compilateur aligne les variables de la structure selon des critères qui lui sont propres. L'alignement peut être configurable ou non, cela dépend du compilateur. Ce TP consiste à comprendre comment le compilateur de l'Arm Cortex 3 passe une structure à l'assembleur et à utiliser son contenu.

La structure suivante doit être passée à une fonction assembleur qui s'occupera d'additionner les valeurs de chaque champ. Les types des variables de la structure sont volontairement différents, mais l'addition doit s'effectuer sur un mot de 32 bits signé.

```
typedef struct  {
int32_t a;
int8_t b;
int16_t c;
int8_t d;
int32_t e
} numbers_t;
```

Calculez a+b+c+d+e en assembleur et retournez le résultat sous forme d'un entier signé de 32 bits.

4 Travail à présenter

Pour ce laboratoire, il vous est demandé de présenter à la fin de la séance votre travail. Vous devrez être en mesure d'expliquer les notions que vous avez acquises sur le langage assembleur et la manipulation de données.

Rappel : COMMENTEZ VOTRE CODE.

Bon travail.