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, A. Upegui, F. Vannel

Assembleur
Semestre hiver

Introduction

Ce laboratoire va vous permettre d'écrire votre premier programme en assembleur pour la carte LPC1769. Les premiers exercices, basés sur de l'analyse, vous permettront de vous familiariser avec le langage assembleur de ARM. Dans l'avant dernier exercice, vous devrez programmer une fonction calculant la suite de Fibonacci en assembleur. Le dernier exercice vous permettra de concevoir une fonction manipulant des valeurs sur 64 bits.

1 Notions d'assembleur

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

- **ADD** : Effectue l'addition de deux nombres entiers.
- **ADC** : Effectue l'addition de deux nombres entiers en tenant compte de la retenue (Carry).
- **SUB** : Effectue la soustraction de deux nombres entiers.
- **MOV** : Copie le contenu d'un registre vers un autre registre.
- **CMP** : Effectue la comparaison entre deux nombres, c'est à dire la différence entre deux nombres et la mise à jour des fanions.
- **B** : Effectue un saut dans le programme à une autre instruction que la suivante.
- **LSL, LSR, RRX** : Permet d'effectuer des décalages et des rotations de bits.

1.1 ADD

L'instruction ADD peut être principalement utilisée de deux façons :

- **ADD Rd, Ri, Rj** : Ce qui correspond à $R_d = R_i + R_j$, soit à l'addition de deux registres. Le résultat est toujours stocké dans un registre. d, i, j sont un nombre compris entre 0 et 15 (ils peuvent être différents entre eux, mais également les trois identiques).
- **ADD Rd, Ri, #xxx** : Ce qui correspond à $R_d = R_i + xxx$, soit à l'addition d'un registre avec un nombre. Le résultat est stocké dans un registre. d, i est un nombre compris entre 0 et 15.

L'ajout du suffixe **S** spécifie que les fanions devront être calculés et mis à jour dans le registre **xPSR**. Par exemple **ADDS R2, R1, #18** effectue l'opération suivante $R2 = R1 + 18$ et calcule les fanions **N,Z,C,V**.

1.2 ADC

L'instruction ADC est similaire à l'instruction ADD. Elle tient en compte de l'état du bit **C** présent avant l'opération.

Les opérations effectuées sont :

- **ADC Rd, Ri, Rj** : Ce qui correspond à $R_d = R_i + R_j + C$, soit à l'addition de deux registres plus la valeur de C (qui est soit égal à 0, soit égal à 1).
- **ADC Rd, Ri, #xxx** : Ce qui correspond à $R_d = R_i + xxx + C$, soit à l'addition d'un registre avec un nombre plus la valeur de C.

L'ajout du suffixe **S** spécifie que les fanions devront être calculés et mis à jour dans le registre **xPSR** à l'issu de ce calcul. Notez que la valeur de C pourra ainsi changer lors de cette opération.

1.3 SUB

L'instruction SUB s'utilise de façon très similaire à ADD :

- **SUB Rd, Ri, Rj** : Ce qui correspond à $R_d = R_i - R_j$, soit à la soustraction de deux registres.
- **SUB Rd, Ri, #xxx** : Ce qui correspond à $R_d = R_i - xxx$, soit à la soustraction d'un registre avec un nombre.

Le suffixe **S** peut être ajouté afin que l'instruction mette à jour les fanions.

1.4 MOV

L'instruction **MOV** copie le contenu d'un registre vers un autre registre :

- **MOV Rd, Ri** : Ce qui correspond à $R_d = R_i$

Le suffixe **S** peut être ajouté afin que l'instruction mette à jour les fanions (N,Z uniquement pour l'instruction MOV).

1.5 CMP

L'instruction **CMP** est une instruction de comparaison et est très similaire à SUB :

- **CMP Rd, Ri** : Ce qui correspond à calculer $tmp = R_d - R_i$

Le résultat de la soustraction n'est pas stocké dans un registre, par contre les fanions **N,Z,C,V** sont automatiquement calculés (il n'y a pas de suffixe **S** à ajouter)

1.6 B

L'instruction **B** permet d'effectuer un saut. Normalement le processeur exécute chaque ligne de façon successive. L'instruction **B** crée un saut vers un autre endroit du programme.

- **B LabelOuAller** : Ce qui correspond à faire un saut vers la partie du programme identifiée par le label *LabelOuAller*.

Cette instruction est très souvent utilisée avec une condition. Ces conditions ne dépendent que de l'état des fanions **N,Z,C,V**. (N'hésitez pas à relire votre cours de SLO pour rafraîchir ces notions.)

La liste complète des 15 conditions se trouve dans la documentation du cours. Voici quelques exemples :

- **EQ** (égal) : Z est à l'état 1 (*Z SET*).
- **NE** (différent) : Z est à l'état 0 (*Z Clear*).
- **PL** (positif) : N est à l'état 0 (*N Clear*).

1.7 Appel d'une fonction assembleur

Le passage d'un argument s'effectue au travers des registre R0 à R3

Il n'est possible de retourner qu'une seule valeur : le contenu de R0

Ainsi lors de l'appel dans le fichier **labo2_ASM.c** de la fonction *a = funcexo1(11,32) ;* la valeur 11 va être copié dans le registre R0 et la valeur 32 sera copiée dans le registre R1. Il faudra à la fin de la fonction s'assurer d'avoir le résultat de la fonction dans le registre R0 afin que celui-ci puisse se retrouver copié dans la variable a.

Note : la ligne `MOV PC, R14`

permet d'effectuer la sortie de la fonction et retourner à l'endroit où a été appelée cette fonction.

1.8 Décalages et rotations

Les opérations de décalage et de rotation de bits sont possible à l'aide des instructions ASR (Arithmetic Shift Right), LSL (Logical Shift Left), LSR (Logical Shift Right), ROR (Rotate Right), RRX(Rotate Right with Extend). Le détail complet de ces instructions est disponible dans la documentation de référence.

Pour ce laboratoire, vous pouvez utilisez celle-ci :

- **`LSL` (Logical Shift Left)** Effectue un décalage bit à bit vers la gauche.
  exemple : `LSL R1, R0, #1` copie dans `R1` le contenu de `R0` décalé d'un bit vers la gauche. Le bit de poids faible de R1 sera égal à `0`
- **`LSR` (Logical Shift Right)** Effectue un décalage bit à bit vers la droite.
  exemple : `LSR R1, R0, #5` copie dans `R1` le contenu de `R0` décalé de 5 bits vers la droite. Les 5 bits de poids forts de R1 seront égaux à `0`
- **`RRX` (Rotate Right with eXtended)** Effectue une rotation sur la droite **de 1 bit.** Le bit 31 prendra la valeur du fanion C. Le bit 0 sera copié dans le fanion C (si le suffixe S est ajouté à l'instruction = ordre de mise à jour des fanions).
  exemple : `RRXS R0, R0`.
  Si R0 est égal à 0x0000A50E et que C est égal à 1, alors après exécution R0 contiendra 0x80005287 et C sera égal à 0.
  Après une seconde exécution de l'instruction, R0 sera égal à 0x40002943 et C sera égal à 1.

2 Le projet sous LPCXpresso

Vous importerez le projet **labo2_ASM**.

Le fichier **labo2_ASM.c** contient la fonction **main** et l'appel aux fonctions des quatre exercices.

Le fichier **assembleur.s** contient le code en assembleur que vous devrez analyser et également compléter.

Pour tester votre code et le fonctionnement adéquat par rapport à votre exercice, vous devrez soit modifier la ligne suivante se trouvant dans le début du **main**, soit modifier durant l'exécution la valeur de cette variable :

```
exo=1;
```

Le numéro correspond à l'exercice que vous souhaitez tester.

Info : N'oubliez pas de compiler votre projet.

Info : La vue des registres peut être ajouté via la menu `Window -> Show View -> Other... -> Debug -> Registers`

3 Exercices

3.1 Analyse du code assembleur

Dans cet exercice, Il vous est demandé d'exécuter ce programme pas à pas (bouton *step into*).

Pour chaque instruction de la fonction `funcexo1`, vous noterez le contenu des registres **R0, R1, R2 et R3** ainsi que la valeur de **PC** et des fanions **N,Z,C,V** qui se trouvent dans le registre PSR (bit 31 = N, bit 30 = Z, bit 29 = C, bit 28 = V) après exécution de l'instruction.

Info : Arrêtez vous à la ligne `MOV PC, R14` (non demandée).

1. Complétez le tableau fourni. Effectuez ce travail sur les deux appels de cette fonction.
2. Expliquez les instructions exécutées en fonctions des valeurs des fanions.
3. Quelle est la fonction mathématique calculée par ce code ? abs(a-b)
   Info : Vous pouvez modifier la valeur passée en argument dans le fichier *labo2_ASM.c* afin de vérifier vos hypothèses.
4. Bonus :
   - Transposez le code de la fonction *funcexo1* en language C.
   - Comparez le code assembleur issu de votre fonction C avec le code assembleur d'origine.

3.2 Analyse du code assembleur II

Le code assembleur suivant permet de faire un calcul mathématique d'une suite. Malheureusement le code n'ayant pas été commenté nous ne savons plus exactement ce qu'effectue ce programme. De mémoire il s'agit d'un programme calculant la suite du *triangle de pascal* ou la suite des *nombres triangulaire*.
`https://fr.wikipedia.org/wiki/Triangle_de_Pascal`
`https://fr.wikipedia.org/wiki/Nombre_triangulaire`

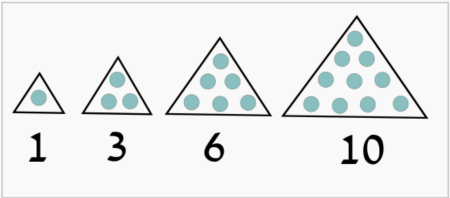

FIGURE 1 – Nombre triangulaires calculés pour 1,2,3 et 4

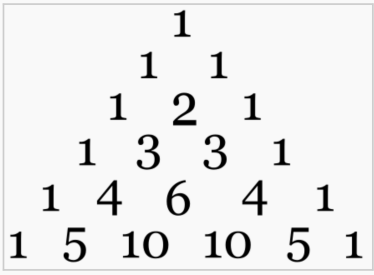

FIGURE 2 – Triangle de Pascal. 6 itérations calculées

Réalisez le même travail que dans l'exercice précédent, à savoir :

1. Complétez le tableau fourni.
2. Justifier l'influence des fanions sur les instructions exécutées.
3. Retrouvez la fonction mathématique calculée par ce code et indiquez à quoi correspond l'argument passé à la fonction ainsi que la valeur retournée ?
4. Bonus :
   - Transposez le code de la fonction *funcexo2* en language C.
   - Comparez le code assembleur issu de votre fonction C avec le code assembleur d'origine.

Pour réaliser cet exercice, vous devez tout d'abord modifier la valeur `exo` et remplacer **1** par le numéro de cet exercice, soit **2**.

Retourne le premier nombre triangulaire plus grand ou égal à l'argument

3.3 Suite de Fibonacci

Dans cet exercice, vous devez écrire un programme en assembleur permettant de calculer la suite de Fibonacci.

```
int funcexo3 (int n)
{
  int a, b;
  int i;

  a = 2;
  b = 1;
  for(i=0; i<n; i ++)
  {
    a = a + b;
    b = a - b;
  }
  return a;
}
```

Nous considérerons $n > 0$

Pour information le début de la suite de Fibonacci renvoyée par cette fonction est la suivante : 3,5,8,13,21,34...(la première valeur correspondant à n=1).

La suite de Fibonacci permet de calculer le fameux nombre d'or. Celui-ci peut être calculé par $\frac{funcexo3(n+1)}{funcexo3(n)}$. Plus n sera grand, plus le nombre obtenu convergera vers le nombre d'or (φ).

3.4 Manipulation de bits

Les type `long long` en C permet de représenter des valeurs entières en 64 bits. Pour garder une valeur de ce type le processeur a besoin de 2 registres (chacun de 32 bits bien évidement...).

Lors du passage d'une valeur du type `long long` (alias `uint64_t`) comme argument ou comme retour d'une fonction, les 32 bits plus faibles se trouvent sur le premier registre à disposition (`R0`) et les 32 bits plus forts se trouvent sur le registre suivant (`R1`).

Dans cet exercice la valeur `dir` se trouve dans le registre `R2`.

Dans cet exercice vous devez écrire la fonction `uint64_t funcexo3(uint64_t, uint32_t)` qui gère l'opération décalage :

Pour `B = funcexo4(A, dir)`

la fonction fera :

$B = A << 1$	si $dir = 0$	// décalage de un bit à gauche sur 64 bits
$B = A >> 1$	si $dir = 1$	// décalage de un bit à droite sur 64 bits
$B = A$	dans tous les autres cas.	

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.