TD ASM - 3

Fonction et pile

1 - Fonction

Pour l'instant, nous n'avons fait que du code linéaire, avec juste quelques tests et boucles. Il est temps de voir comment créer des fonctions.

En assembleur, il n'y a pas d'instruction spécifique pour définir une fonction : une fonction n'est qu'un label. Mais contrairement à un simple label, une fonction peut être appelée de n'importe ou, et n'importe quand. Il est donc nécessaire de prévoir du code spécifique, autant au niveau de l'appel de la fonction que dans la fonction elle-même, de manière à ce que le contexte de la fonction soit préservé d'un appel à l'autre.

La première chose qui doit absolument etre préservé, c'est l'adresse de retour, c'est à dire où revenir une fois que la fonction est terminée. En effet, vu que la fonction peut être appelée de n'importe où, il n'est pas possible de mettre à la fin de la fonction une instruction B avec un nom de label de retour.

Pour résoudre ce problème, l'assembleur ARM a une instruction spéciale : BL (Branch with Link). L'instruction BL s'utilise comme l'instruction B, avec le nom d'un label, et « saute » au label en question. La différence est qu'elle met, en plus, l'adresse de retour (c'est à dire l'adresse de l'instruction suivant l'instruction BL) dans un registre spécifique (le seul registre spécifique avec r15 == PC), le registre r14, aussi nommé LR (Link Register).
A la fin de la fonction, il suffit juste de remettre le contenu de LR dans PC pour « sauter » de nouveau à l'endroit de l'appel de la fonction, et continuer le code initial.

Ainsi, le pseudo-code suivant :

```
def maFonction:
    return

i = 1
maFonction()
i += 1
```

Se traduit en :

```
main
        MOV r0, #1
        BL maFonction
        ADD r0, r0, #1
        END

maFonction
        MOV PC, LR
```

Vérifiez que le code assembleur précédent fonctionne bien.

2 - Pile

En vous basant sur le code précédent, traduisez en assembleur le pseudo-code suivant :

```
def maFonctionA:
        return

def maFonctionB:
        maFonctionA()
        return

i = 1
maFonctionB()
i += 1
```

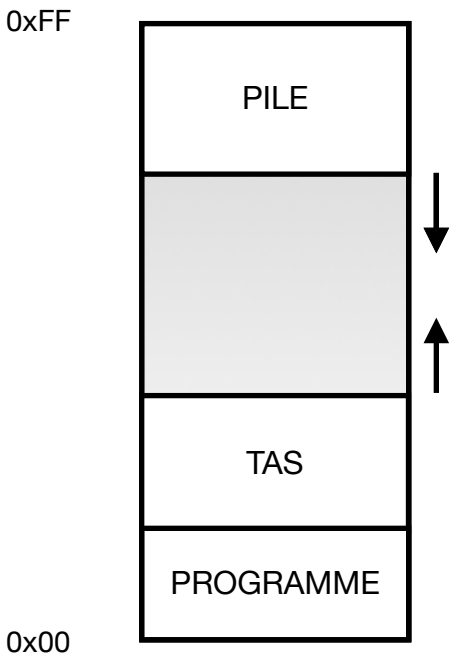
Regardez notamment ce qui se passe en pas à pas. Pourquoi est-ce que cela ne fonctionne pas ? LR est écrasé lors du second appel de fonction (l'appel à maFonctionA), du coup maFonctionB ne peut jamais revenir au code principal.

Pour éviter cela, il va falloir sauvegarder le contenu de LR. Il n'est pas possible de le sauvegarder dans un registre, car ils peuvent être eux-même écrasés (par exemple si maFonctionB s'appelle elle-même). Il faut donc le sauvegarder en mémoire.

Pour l'instant, nous n'avons vu qu'une seule zone mémoire, cette qui est dans les adresses basses, juste au dessus du code. C'est dans cette zone que sont mises les variables réservées avec les instructions DCD ou FILL par exemple. Cette zone, c'est le TAS (HEAP en anglais). C'est la zone utilisée pour les variables globales, définies statiquement à la compilation, lorsque l'émulateur lit le code. L'accès à ces variables est simple, il suffit simplement de donner leur adresse, qui ne change jamais.

Il nous faut autant de sauvegarde de LR qu'il y a d'appel à la fonction. Or, ce nombre est inconnu, il peut même dépendre dynamiquement de l'éxécution du programme, par exemple si l'appel de la fonction dépend d'un test. Il n'est donc pas possible de sauvegarder LR dans le tas, car il n'est pas possible statiquement, à la compilation, de connaitre le nombre de cases à allouer. Il nous faut donc une autre zone de mémoire.

Pour cela, nous allons utiliser la PILE (STACK en anglais). La pile est une zone de mémoire qui se trouve dans les adresses hautes de la mémoire. Contrairement au TAS qui monte au fur et à mesure des allocations, la PILE, elle, descend.



Comme son nom l'indique, la pile va etre utilisée pour empiler des informations concernant le contexte des fonctions, au fur et à mesure de leur appel.

Pour cela, l'assembleur ARM a deux instructions : LDMFD (Load Multiple registers) et STMFD (Store Multiple registers). Ces deux instructions utilisent un registre comme pointeur de pile, c'est à dire qu'il va y avoir un registre dont le seul but va être de maintenir l'adresse de la dernière case de la pile. N'importe quel registre peut etre utilisé, mais généralement, on utilise r13 (vu que r14 est utilisé pour LR et r15 pour PC).

Ainsi, la sauvegarde de LR au début du code de la fonction s'écrit :

```
STMFD r13, {LR}
```

A la fin de la fonction, il est possible de remettre la valeur de la pile dans LR puis LR dans PC, mais généralement on saute le passage par LR :

```
LDMFD r13, {PC}
```

Il ne faut bien sur pas oublier d'incrémenter et décrémenter r13 à chaque appel de STMFD et LDMFD afin de prendre en compte l'empilage ou le dépilage d'un item dans la pile.
Cela peut être fait à la main avec un ADD, mais cela peut aussi etre fait automatiquement par l'assembleur en utilisant les modes d'indexations vus au TD précédent (ici, le !).

Ainsi, le code correct pour le pseudo-code vu plus haut est :

```
main
        MOV         r0, #1
        BL          maFonctionB
        ADD         r0, r0, #1
        END

maFonctionA
        STMFD       r13!, {LR}
        LDMFD       r13!, {PC}

maFonctionB
        STMFD       r13!, {LR}
        BL          maFonctionA
        LDMFD       r13!, {PC}
```

Vérifiez que le code assembleur précédent fonctionne bien. Utilisez notamment le pas à pas et le bouton « stack » à coté du code lorsque vous être sur une instruction de pile, ainsi que la fenetre de visualisation de la mémoire à la fin pour voir l'état de la pile.

Faites un dessin de la pile, sous forme de tableau, avec à gauche l'adresse mémoire, au milieu le contenu, et à droite le nom ou le sens de la valeur.
Par exemple :

0xFF000000 — 0x0C — Adresse de retour de la fonction maFonctionA

Pour info, les adresses de début de chacune des zones (tas, pile) est défini dans les settings de l'assembleur (« Instruction Memory Size » donne la taille de la zone réservée pour le code, le tas se trouve juste après).

ARM étant un processeur très générique, il n'est pas du tout obligatoire d'avoir une pile qui descend, il est aussi possible d'avoir une pile qui monte. Dans ce cas, on utilise les instructions LDMFA et STMFA (A pour « Ascending) au lieu de LDMFD et STMFD (D pour « Descending »). De même, ici r13 pointe sur la dernière case de la pile, c'est à dire une case « pleine », contenant une valeur. Il est possible de gérer la pile avec r13 qui pointe vers la première case après la pile, c'est à dire une case vide. Dans ce cas, on utilise les instructions LDMED et STMED (E pour « Empty ») au lieu de LDMFD et STMFD (F pour « Full »).

Si l'on veut gérer une pile qui monte, avec r13 qui pointe sur la première case vide, quelles instructions va-t-on utiliser ?

3 - Passage de paramètres

Le passage de paramètre le plus simple est de passer par des registres.

Par exemple, pour la fonction suivante :

```
maFonction(i):
    i = i+1
```

Il est possible de définir que la valeur de i doit être passé dans r0 par exemple.

Ainsi le pseudo-code suivant :

```
def maFonction(i):
    i = i+1
i = 1
maFonction(i)
```

Sera traduit en assembleur en :

```
main
    MOV     r0, #1
    BL      maFonction
    END

maFonction
    STMFD   r13!, {LR}
    ADD r0, r0, #1
    LDMFD   r13!, {PC}
```

Mais lorsqu'il y a plus d'une fonction, et que les autres fonctions prennent elles aussi un paramètre, nous allons retrouver le problème de l'écrasement des valeurs. Il serait possible d'affecter un registre de passage spécifique à chaque fonction, mais cela serait compliqué à gérer, sans compter que le nombre de registres est limité.

A la place, la norme est que chaque fonction a la responsabilité de préserver les valeurs des registres qu'elle va utiliser. Pour cela, elle va sauvegarder ses registres sur la pile, en utilisant les instructions LDMFD et STMFD, qui permettent de sauver directement plusieurs registres sur la pile. Il suffit de donner en paramètre, entre les accolades, non pas un registre mais une liste de registres. Il est possible de spécifier soit des registres individuels (« {r0, r1, r2, r14} »), soit un range de registres (« {r0-r2} »), soit une combinaison des deux (« {r0-r4, r14} »).
Le plus important est de donner la même liste dans le même ordre (à part LR et PC) dans LDMFD et STMFD.

Ainsi, le pseudo-code suivant :

```
def maFonctionA(i):
    i = i + 1

def maFonctionB(j):
    j = j + 1
    maFonctionA(j)

k = 1
maFonctionB(k)
```

Se traduit en assembleur par :

```
main
        MOV         r0, #1
        BL          maFonctionB
        END

maFonctionA
        STMFD       r13!, {r0, LR}
        ADD         r0, r0, #1
        LDMFD       r13!, {r0, PC}

maFonctionB
        STMFD       r13!, {r0, LR}
        ADD         r0, r0, #1
        BL          maFonctionA
        LDMFD       r13!, {r0, PC}
```

Vérifiez en pas à pas que le code assembleur précédent fonctionne bien, et faites un dessin de la pile.

4 - Valeur de retour

De la même manière, il est possible de passer la valeur de retour de la fonction par un registre. Le choix du registre de retour, comme pour le paramètre, est libre pour le développeur. Cette fois-ci, la valeur initiale du registre n'est pas gardée par la fonction (puisque son but est de le modifier), mais par la fonction appellante (puisque la valeur va être modifiée).

Ainsi, le pseudo-code suivant :

```
def maFonctionA(i):
        return i + 1

def maFonctionB(j):
        j = j + 1
        k = maFonctionA(j)
        j = j + 2
        k += maFonctionA(j)
        return k
l = 1
maFonctionB(l)
```

Sera traduit ainsi en assembleur :

```
main
        MOV         r0, #1
        BL          maFonctionB
        END

maFonctionA
        STMFD       r13!, {r0, LR}
        ADD         r1, r0, #1
        LDMFD       r13!, {r0, PC}

maFonctionB
        STMFD       r13!, {r0, LR}
```

```
ADD     r0, r0, #1
BL      maFonctionA
MOV     r2, r1
ADD     r0, r0, #2
BL      maFonctionA
ADD     r1, r2, r1
LDMFD   r13!, {r0, PC}
```

Chacune des deux fonctions prend son paramètre en r0 et retourne sa valeur en r1.
Vérifiez que le code fonctionne et qu'il retourne la bonne valeur en r1 à la fin. Quelle est la bonne valeur attendue ? N'oubliez pas de dessiner la pile.

5 - Base Pointer

Le passage des paramètres et de la valeur de retour par des registres est simple, mais limité. Il faut que chacune de ces valeurs tiennent dans un registre, et le nombre de registre est limité. Ce n'est donc pas générique.

Une manière plus générique de faire cela est d'utiliser la pile comme zone de stokage des paramètres et de la valeur de retour.

Dans ce cas, c'est à la fonction appellante d'empiler les valeurs des paramètres et de réserver une place pour la valeur de retour. Réserver une place revient ni plus ni moins qu'à bouger le pointeur de pile, c'est à dire décrémenter (pour une pile qui va vers le bas) la valeur du registre de pile (r13 généralement).

Se pose alors la question : comment la fonction appelée va-t-elle accéder aux paramètres et à la valeur de retour ? On pourrait utiliser un décalage par rapport au pointeur de pile, mais si la pile bouge, cela peut être compliqué.
Pour garder une référence vers les paramètres et la valeur de retour qui soit indépendante du pointeur de pile, on va utiliser un autre registre (généralement r11), que l'on appellera le BASE POINTER. Ce pointeur est initalisé au début de la fonction appellée à la valeur du STACK POINTER. Comme ca, si la pile change, le BASE POINTER, lui, reste à la même place, et ainsi les références vers les paramètres et la valeur de retour sont des décalages constants.

Ainsi, le pseudo-code précedent :

```
def maFonction(i, j):
        return i + j
k = 1
l = 2
maFonction(k,l)
```

Sera maintenant traduit en assembleur comme ceci :

```
main
                MOV     r0, #1
                MOV     r1, #2
                STMFD   r13!, {r0}
                STMFD   r13!, {r1}
                SUB     r13, r13, #4
                BL      maFonction
                END

maFonction
                STMFD   r13!, {r4, r5, LR}
                MOV     r11, r13
```

```
LDR     r4, [r11, #4 * 5]
LDR     r5, [r11, #4 * 4]
ADD     r4, r4, r5
STR     r4, [r11, #4 * 3]
MOV     r13, r11
LDMFD   r13!, {r4, r5, PC}
```

Vérifiez que le code fonctionne et qu'il retourne la bonne valeur à la fin. Ou se trouve la valeur de retour ? Mettez un commentaire à coté de chaque ligne de code. N'oubliez pas de dessiner la pile.

Remarquez qu'à la fin de la fonction, on remet r13 à r11, ce qui permet de remettre la pile à sa bonne valeur.

Que se passe-t-il si on veut appeler plusieurs fonctions à la suite ? La même chose que pour LR : la valeur de r11 sera écrasée. Les mêmes causes produisant les mêmes effets, on pensera donc à sauver r11 en même temps que LR.

Ainsi, le pseudo-code suivant :

```
def maFonctionA(i):
    return i + 1

def maFonctionB(j):
    j = j + 1
    k = maFonctionA(j)
    j = j + 2
    k += maFonctionA(j)
    return k
l = 1
maFonctionB(l)
```

Sera maintenant traduit en assembleur comme ceci :

```
main
        MOV     r0, #1
        STMFD   r13!, {r0}
        SUB     r13, r13, #4
        BL      maFonctionB
        END

maFonctionA
        STMFD   r13!, {r4, r11, LR}
        MOV     r11, r13
        LDR     r4, [r11, #4 * 4]
        ADD     r4, r4, #1
        STR     r4, [r11, #4 * 3]
        MOV     r13, r11
        LDMFD   r13!, {r4, r11, PC}

maFonctionB
        STMFD   r13!, {r4, r5, r11, LR}
        MOV     r11, r13
        LDR     r4, [r11, #4 * 5]
        ADD     r4, r4, #1
        STMFD   r13!, {r4}
        SUB     r13, r13, #4
        BL      maFonctionA
        LDMFD   r13!, {r5}
```

```
ADD         r13, r13, #4
ADD         r4, r4, #2
STMFD       r13!, {r4}
SUB         r13, r13, #4
BL          maFonctionA
LDMFD       r13!, {r4}
ADD         r4, r4, r5
STR         r4, [r11, #4 * 4]
MOV         r13, r11
LDMFD       r13!, {r4, r5, r11, PC}
```

Vérifiez que le code fonctionne et qu'il retourne la bonne valeur à la fin. Mettez un commentaire à coté de chaque ligne de code. N'oubliez pas de dessiner la pile.

6 - Variables locales

Dans une fonction, il faut parfois créer des variables locales. Comme pour les paramètres et la valeur de retour, ces variables locales peuvent utiliser des registres, ou bien, si elles sont trop nombreuses ou trop grosses, être stockées dans la pile.

Ces variables locales sont empilées à la suite de l'adresse de retour et du base pointer. Elles sont accessibles par décalage du base pointer, comme pour les paramètres de retour, mais là ou les paramètres sont accessibles par un décalage positif (on remonte la pile), les variables locales sont accessibles par un décalage négatif (on descend la pile).

L'ensemble des valeurs mises en place pour l'appel d'une fonction (paramètre, valeur de retour), et l'empilage des valeurs sauvegardées (adresse de retour, registres, variables locales) forme ce que l'on appelle le STACK FRAME, le cadre de contexte d'une fonction.

7 - Exercices

- Faire la fonction factorielle en assembleur

- Ecrivez le code assembleur (gérant notamment les stack frames] des codes suivants :

```
=====
= 1 =
=====

-------------
def main():
  f1(3)

def f1(i):
  f2(i+1)

def f2(j):
  j+2
-------------


=====
= 2 =
=====
```

```
-------------
def main():
  f1(3)

def f1(i):
  return f2(i+1)

def f2(j):
  return j+2
-------------
```

```
=====
= 3 =
=====
```

```
-------------
def main():
  o = f1(3)
  o = o + 6
  p = f2(o)

def f1(i):
  j = f2(i+1)
  return j+2

def f2(j):
  k = j+2
  return k
-------------
```

```
=====
= 4 =
=====
```

```
-------------
def main():
  o, p = f1(3, 4, 5, 6, 7)
  f2(o+11, p+12, o+13, p+14, p+15)

def f1(i, j, k, l, m):
  g, h = f2(i+1, j+2, k+3, l+4, m+5)
  return g-1, h-2

def f2(a, b, c, d, e):
  y = a + b + c
  z = d + e
  return y, z
-------------
```