Compte-Rendu de TP1 : Architecture des Microcontrôleurs
Étude du Startup et de la Gestion de la Pile

Youssef Chemrakhi et Théodore Condette

Janvier 2026

1 Travail 5 : Initialisation du Startup et de la RAM

L'objectif est d'observer les premières étapes après un Reset. Nous avons modifié le `Reset_Handler` pour initialiser manuellement les ressources critiques.

1.1 Code d'initialisation

Le code suivant a été inséré pour effacer une zone de la pile et tester l'accès à la RAM :

```
Reset_Handler:
    ; 1. Initialisation de la pile (mise a zero pour debug)
    MOV R6, #0
    LDR R7, =0x20017FF0
    STR R6, [R7]
    STR R6, [R7, #4]
    STR R6, [R7, #8]
    STR R6, [R7, #12]

    ; 2. Initialisation du Stack Pointer (SP)
    LDR SP, =0x20018000

    ; 3. Test d'ecriture en debut de RAM (SRAM1)
    LDR R0, =0x20000000
    MOV R1, #1
    STR R1, [R0]
```

Listing 1 – Insertion dans le Reset_Handler

1.2 Analyse des adresses

Le processeur STM32L476 possède sa RAM (SRAM1) débutant à `0x20000000`. La pile est configurée à `0x20018000` (fin de la RAM). Comme elle est de type **Full Descending**, elle croît vers les adresses décroissantes lors d'un `PUSH`.

Bloc d'instructions 1 : Il initialise la pile et place la valeur 0 dans les trois premières cases mémoire.

Bloc d'instructions 2 : Le registre `SP` prend la valeur `0x20018000` (adresse de fin de la RAM), car le pointeur de pile évolue de manière ascendante.

Bloc d'instructions 3 : Il place la valeur 1 au début de la RAM.

Processus de démarrage : En compilant le code, pour analyser le comportement au démarrage du microprocesseur, on observe le registre `VTOR`. Le `VTOR` contient l'adresse de la pile (qui sera chargée dans le registre `SP` du microprocesseur) ainsi que celle de la première instruction. Ici, cette adresse est `0x0`. On se rend à cette adresse en mémoire, en mode « traditional », pour lire correctement l'adresse. On lit alors `0x20018000`, ce qui indique que l'on se situe dans la RAM. C'est l'adresse que prendra le `SP`. On lit aussi `0x080003D1` qui correspond au code de la première instruction.

2 Travail 6 : Étude de la pile STACKA (Problème LIFO)

Dans cet exercice, nous avons testé l'empilement suivi du dépilement de trois registres : `R4=4, R5=5, R6=6`.

```
1     PUSH {R4}
2     PUSH {R5}
3     PUSH {R6}
4     ; ... modification des registres ...
5     POP {R4}
6     POP {R5}
7     POP {R6}
```

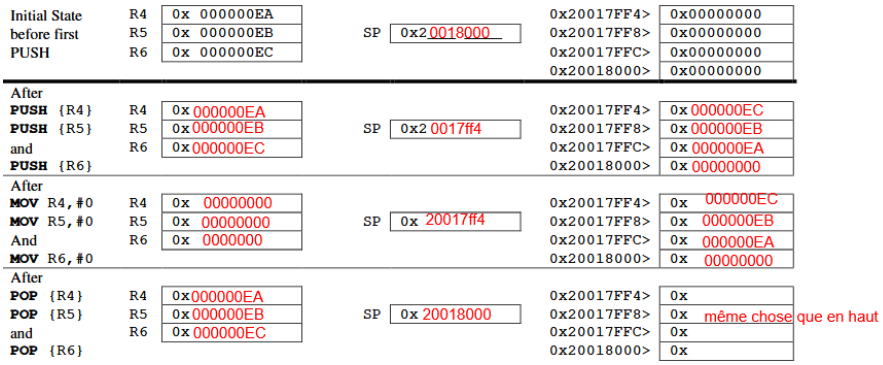

FIGURE 1 – État initial des registres et de la pile avant les instructions PUSH

Résultat : À la fin, `R4` contient la valeur 6 et `R6` contient la valeur 4. **Conclusion :** La pile fonctionne selon le mode *Last In, First Out* (LIFO). En dépilant dans le même ordre que l'entrée, les données sont croisées. Ce mécanisme ne permet pas de restaurer un contexte proprement.

3 Travail 7 : Étude de STACKB (Transparence et coût)

Pour assurer la **transparence** (que l'appelant retrouve ses registres intacts), il faut dépiler dans l'ordre inverse ou utiliser l'instruction multiple :

```
1    PUSH {R4, R5, R6}
2    ; ...
3    POP {R4, R5, R6}
```

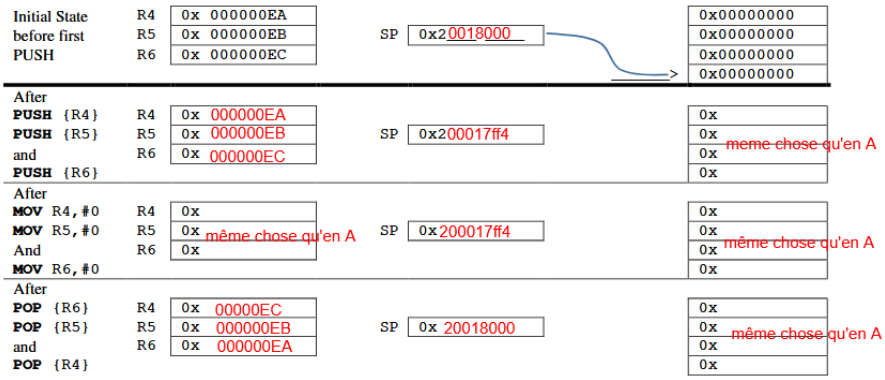

FIGURE 2 – Évolution de la pile après les instructions PUSH et POP

L'architecture ARM gère automatiquement l'ordre correct lors d'un `POP` multiple. **Coût mémoire :** Chaque registre faisant 32 bits (4 octets), l'utilisation de la pile pour 3 registres consomme **12 octets**.

Résultat : Contrairement à `stackA` et `stackB`, les valeurs de `R4` et `R6` sont modifiées, tandis que celle de `R5` reste inchangée. Grâce à cela, nous avons pu modifier la valeur des registres.

Dans la partie précédente de ce compte rendu, nous avons présenté les comportements de `stackA` et `stackB` de manière inversée. En réalité, `stackA` a pour effet d'inverser l'ordre des registres lors de leur sauvegarde, tandis que `stackB` permet une sauvegarde et une restauration correctes des registres, dans le bon ordre.

Conclusion : Ainsi, `stackB` correspond à l'implémentation correcte du mécanisme de pile et doit être privilégiée afin de garantir un comportement conforme et prévisible du programme.

4 Travaux 8 à 10 : Mécanismes de Fonctions

Nous avons analysé trois approches pour les fonctions :

1. **FUNCT1 :** L'adresse de retour de **FUNCT1** est stockée dans le registre `LR`. On lit l'adresse `0x0203` (avec le bit de poids faible indiquant que le code est en 16 bits), ce qui correspond en réalité à l'adresse `0x0202`.

   En mode désassemblage, on observe que, juste à côté de l'instruction `ADD R4, R5`, cette adresse (adresse de retour) est visible.

   Pour revenir à l'instruction `ADD R4, R5, R6`, le `PC` exécute l'instruction `BL FUNCT1`, qui lui indique de prendre la valeur contenue dans le registre `LR` pour revenir à l'instruction immédiatement suivante à l'appel de la fonction.

   Dans une fonction, le microcontrôleur impose que les paramètres soient passés dans les registres `R0`, `R1`, `R2`, etc., dans cet ordre.

   Si les registres ont perdu leur valeur initiale, cela signifie que la fonction n'est pas transparente, car elle a modifié le contexte initial. Il est donc préférable de privilégier la transparence des fonctions, c'est-à-dire veiller à préserver l'état des registres utilisés.

2. **FUNCT2 :** La fonction 2 n'est pas transparente, car les valeurs des registres `R4`, `R5` et `R6` sont modifiées définitivement par la fonction. Autrement dit, les valeurs des registres avant l'appel de la fonction sont perdues. C'est une mauvaise pratique selon M. Laroche.

3. **FUNCT3 :** Dans la fonction 3, on sauvegarde la pile avec `push R4 R5 LR`. La fonction met à zéro les deux registres, puis récupère de la mémoire les valeurs des registres avant l'appel de la fonction et copie dans le `PC` l'adresse

de retour. Comme dans la fonction 2, cette fois-ci les valeurs des registres sont conservées : la fonction est transparente, ce qui constitue la principale amélioration.

5 Travaux 11 et 12 : Appels imbriqués et gestion du registre LR

Nous avons analysé deux situations mettant en jeu des appels de fonctions imbriqués et la gestion du registre `LR` :

1. **Requested work 11 — Appel imbriqué correct mais incomplet :** Lors de l'exécution, on entre dans la **fonction 4**, qui sauvegarde le registre `LR` sur la pile, puis appelle la **fonction 5**. La fonction 5 se termine classiquement par l'instruction `BX LR`. Le registre `PC` récupère alors l'adresse de retour contenue dans `LR`, ce qui permet de revenir à l'instruction suivant l'appel de la fonction 5 dans la fonction 4. Ensuite, la fonction 4 restaure l'adresse de retour initiale en dépilant la valeur sauvegardée dans `LR` directement dans `PC`.

   Cependant, la fonction 4 ne réalise pas correctement l'addition des valeurs contenues dans les registres `R4`, `R5` et `R6`. En effet, lors de l'appel de la fonction 5, ces registres ne sont pas sauvegardés sur la pile. Ils sont modifiés par la fonction 5 et remis à zéro lors du retour dans la fonction 4, ce qui conduit à un résultat d'addition nul.

   Pour corriger ce problème, il est nécessaire de sauvegarder les registres `R4`, `R5` et `R6` au début de la fonction 5 à l'aide d'une instruction `PUSH`, puis de les restaurer avant la sortie de la fonction avec une instruction `POP`. Ce problème n'apparaît pas dans le test de la fonction 4, car celle-ci prend correctement cette précaution.
2. **Requested work 12 — Appel imbriqué incorrect :** Lors de l'appel de la **fonction 7**, la valeur précédente du registre `LR` est écrasée. En conséquence, le registre `PC` ne peut plus récupérer l'adresse de retour correcte et reste bloqué dans la **fonction 6**. Il s'agit donc d'un appel imbriqué incorrect, dû à l'absence de sauvegarde du registre `LR` avant l'appel de la fonction 7.