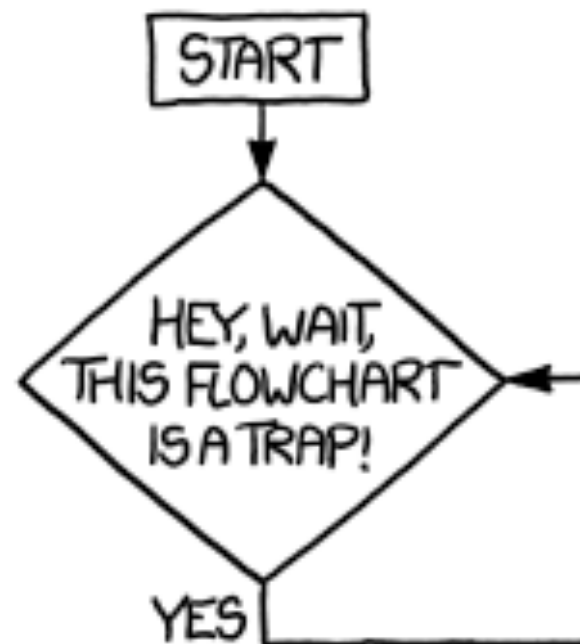
Assembleur ARM: Séquence d’exécution et branchements



xkcd.com

GIF-1001 Ordinateurs: Structure et Applications, Hiver 2019

Jean-François Lalonde

Merci à Yves Roy

Modification de la séquence d'exécution

- Par défaut, les instructions s'exécutent séquentiellement et PC est incrémenté automatiquement par le microprocesseur entre chaque instruction
  - PC = PC + 4 (taille d'une instruction)
- Dans nos programme, il arrive que l'on veuille exécuter autre chose que la prochaine instruction:
  - Saut direct à une instruction
  - Énoncé conditionnel "si"
  - Boucle: "répète N fois" ou "répète tant que"
  - Appel de fonction
- Il est possible de contrôler la séquence d'exécution, en assembleur, avec des instructions qui modifient PC.

Sauts

- PC est un registre et peut être modifié comme les autres registres, avec la plupart des instructions

- Modifier la valeur de PC correspond à effectuer un saut absolu à une adresse.

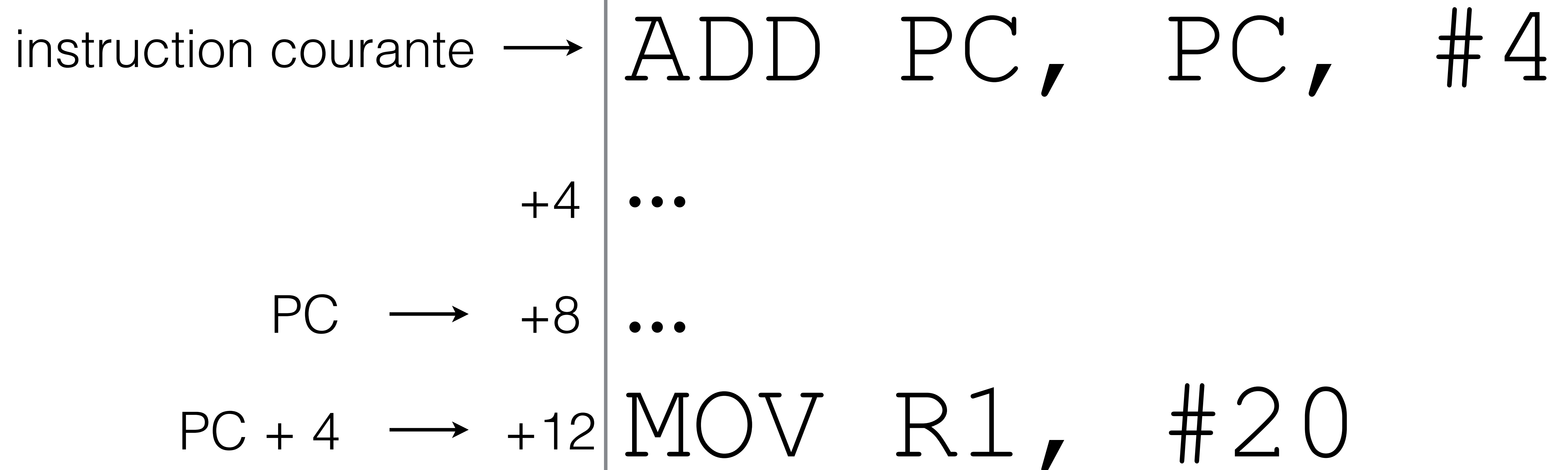
- Exemples:

```
MOV PC, #0x80       ; PC = 0x80
MOV PC, R0          ; PC = R0
LDR PC, [R0]        ; PC = Memoire[R0]
ADD PC, R0, #4      ; PC = R0 + 4
```

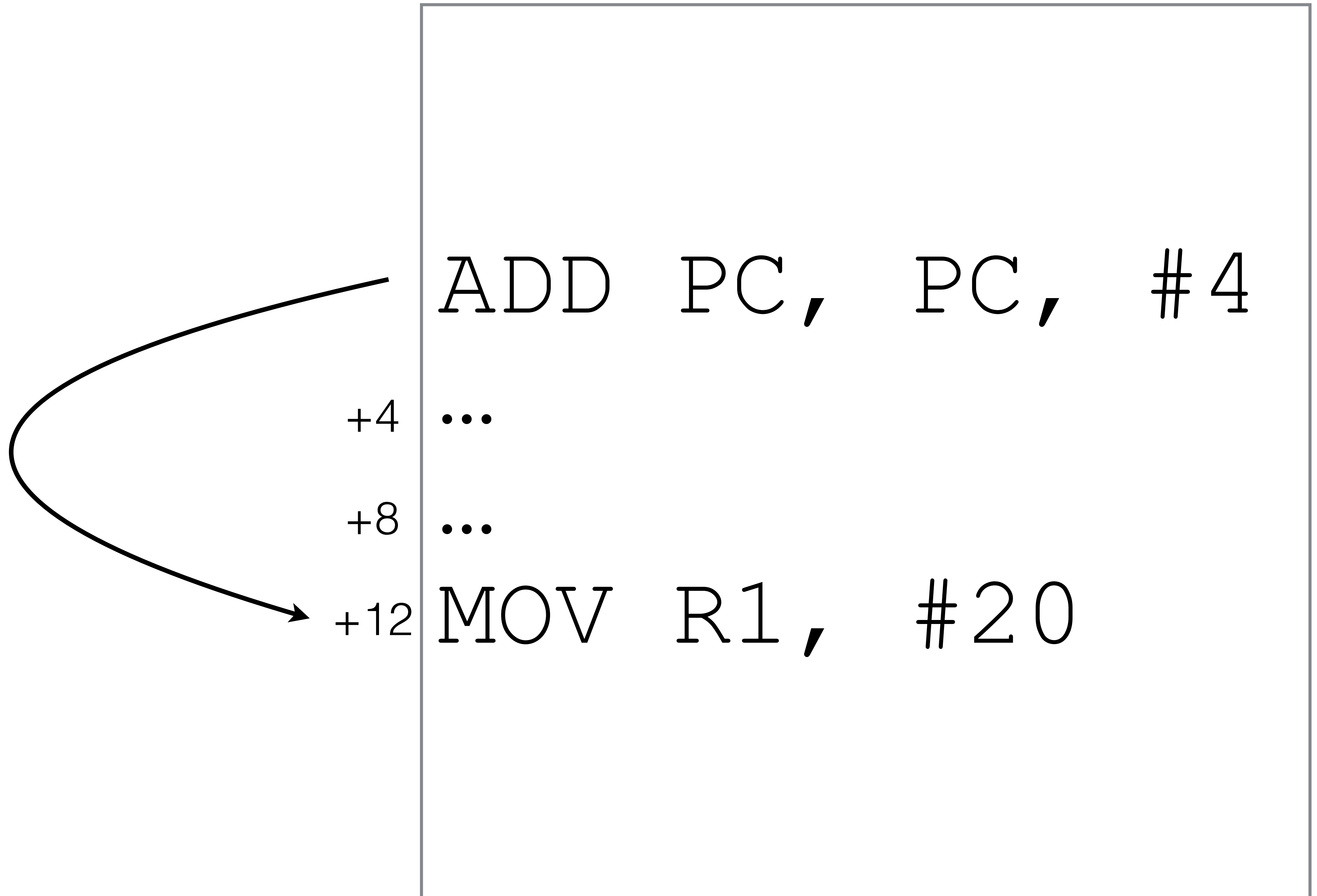
Sauts

```
ADD PC, PC, #4
…
…
MOV R1, #20
```

Sauts



Sauts



Démonstration
(Modifications à PC)

Sauts à une étiquette

- Il n’est pas très pratique d’avoir à déterminer l’adresse directement

- L’instruction B (Branch) saute à une « étiquette » déterminée par le programme:

```
B etiquette
```

- L’assembleur convertit l’étiquette par son adresse relative à PC.

```
B main ; on saute à l’adresse indiquée par l’étiquette « main »
...

main
MOV R0, #1
...
```

Branchements conditionnels

- Un énoncé conditionnel se code habituellement avec au moins deux instructions:
  - une pour faire le test ou la comparaison
  - une pour effectuer (ou pas) le branchement en fonction de la comparaison.
- Le résultat du test ou de la comparaison est entreposé dans les drapeaux de l'ALU et l'instruction de branchement est conditionnelle. Si la condition est rencontrée, on effectue le branchement. Sinon, on exécute la ligne suivante… Pour cette raison, les conditions de branchement sont souvent inversées.
- Exemple: Si `maVar` vaut 4, exécute une tâche:

exemple en C

```
if (maVar == 4) {
  // exécute une tâche…
}

// le programme continue…
```

assembleur

```
LDR R0, maVar      ; Met la valeur de maVar dans R0
CMP R0, #4         ; Change les drapeaux comme R0-4

BNE PasEgal        ; NE = "Not Equal"
; execute une tâche…

PasEgal
; le programme continue…
```

Branchements conditionnels

- Autre exemple: if/else

exemple en C

```
if (maVar == 4) {
  // exécute une tâche…
} else {
  // exécute une autre tâche…
}
// le programme continue…
```

assembleur

```
LDR R0, maVar      ; Met la valeur de maVar dans R0
CMP R0, #4         ; Change les drapeaux comme R0-4
BNE PasEgal        ; NE = “Not Equal”
; execute une tâche (nous sommes dans le “if”)

B Continue
PasEgal
; exécute une autre tâche (nous sommes dans le “else”)

Continue
; le programme continue...
```

Démonstration
(if/else avec branchements)

Boucles

- Une boucle (répète N fois ou tant que) est constituée de trois opérations:
  1. initialiser la boucle
  2. condition d'arrêt
  3. opération mathématique qui mènera à la réalisation de la condition d'arrêt.
- En assembleur, le début de la boucle est identifié par une étiquette
- Voici un exemple de boucle qui se répète cinq fois:

exemple en C

```
for (int i = 0; i < 5; i++) {
  // tâche à l'intérieur de la boucle
}
// le programme continue…
```

assembleur

```
InitDeBoucle
MOV R4, #5

DebutDeBoucle
; tâche à l'intérieur de la boucle
SUBS R4, R4, #1      ; R4 = R4 - 1, change les drapeaux
BNE DebutDeBoucle    ; Condition d'arrêt

; le programme continue…
```

Démonstration
(Tableaux avec boucle)

Appel de fonction

- Une fonction est un ensemble d'instructions à une adresse donnée. Cette adresse est identifiée par une étiquette.

- Appeler une fonction signifie mettre PC au début de la fonction pour exécuter les instructions constituant la fonction.

- Retourner d'une fonction signifie reprendre l'exécution d'où l'appel s'est fait. Il faut revenir à l'endroit où nous étions: c'est l'adresse de retour.

Appel de fonction: mauvais exemple

Main	Fonction (ailleurs dans le code)
B MaFonction AdresseDeRetour MOV R0, #0	MaFonction ; Tâche dans la fonction **B AdresseDeRetour**

- Pourquoi est-ce un mauvais exemple?
- Comment faire pour appeler la fonction deux fois?
  - on revient toujours au même endroit!

Démonstration
(Fonction, problème)

Appel de fonction et adresse de retour

- Pour pouvoir revenir à plus d'un endroit, il faut sauvegarder l'adresse de retour avant de faire le branchement.

- Instruction `BL` (Branch and Link):

```
BLcc Adresse ; met l'adresse de retour dans LR
```

- Après l'exécution de la fonction, on place `PC = LR` pour continuer l'exécution après l'endroit où la fonction a été appelée, avec l'instruction `BX` (Branch and eXchange):

```
BX Rm         ; PC = Rm (Rm peut être n'importe quel registre
```

Main	Fonction

BL MaFonction ; LR = PC-4
MOV R0, #0
 	
BL MaFonction ; LR = PC-4
ADD R0, R0, #1
	MaFonction
	; Tâche dans la fonction
	BX LR ; PC = LR

Démonstration
(Fonction, problème réglé)

Appel de fonction et adresse de retour

- Que ce passe-t-il si une fonction appelle une autre fonction?

Main	Fonction1	Fonction2

`; Debut Main`
`BL Fonction1`
`; Suite Main`
	`Fonction1`
	`; Code Fonction 1`
	`BL Fonction2`
	`; Suite 1 Fonction1`
	
	`BL Fonction2`
	`; Suite 2 Fonction1`
	`BX LR`
		`Fonction2`
		`; Code Fonction 2`
		`BX LR`

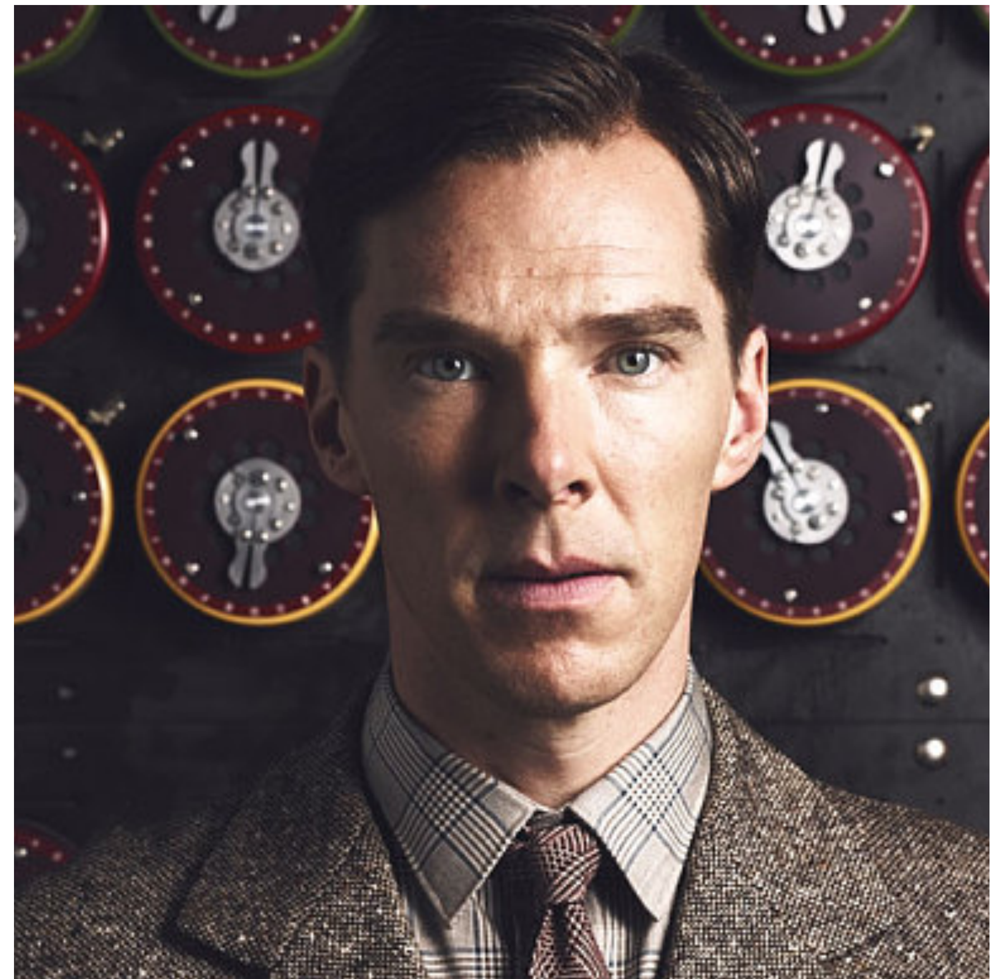
- La séquence de gestion des adresses de retour devient:
  - `LR` = “Suite Main”
  - `LR` = “Suite 1 Fonction1”
  - `PC` = `LR`, donc `PC` = “Suite 1 Fonction1”
  - `LR` = Suite 2 Fonction1
  - `PC = LR`, donc `PC` = “Suite 2 Fonction1”
  - `PC = LR`, donc `PC` = “Suite 2 Fonction1”… Le `BX LR` de Fonction1 “retourne" au mauvais endroit!

Piles («stacks»)

- Structure de données très utilisée en informatique
- Inventée par Alan Turing (1946)

Alan Turing

Imitation Game

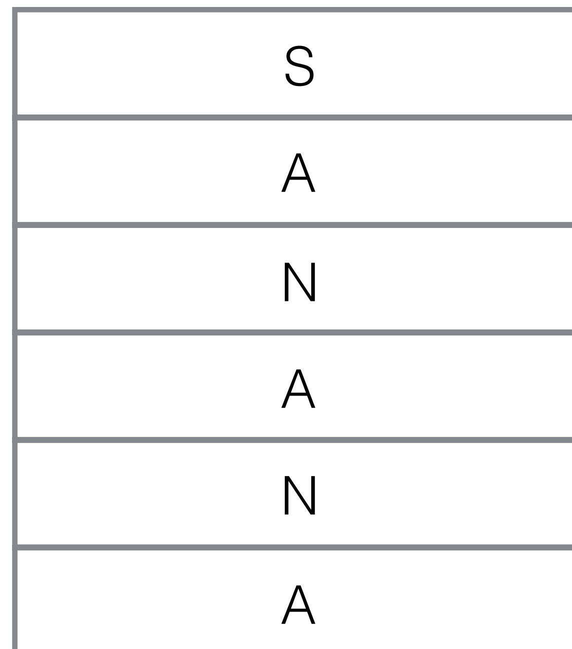


Piles («stacks»)

- Structure de données “LIFO”
  - LIFO = Last In, First Out
- Deux opérations principales:
  - PUSH: rajoute un élément sur “le dessus” la pile
  - POP: enlève un élément du “dessus” de la pile

Exemple #1: épeler «ananas» à l’envers

1. Empile (PUSH) les lettres: A-N-A-N-A-S

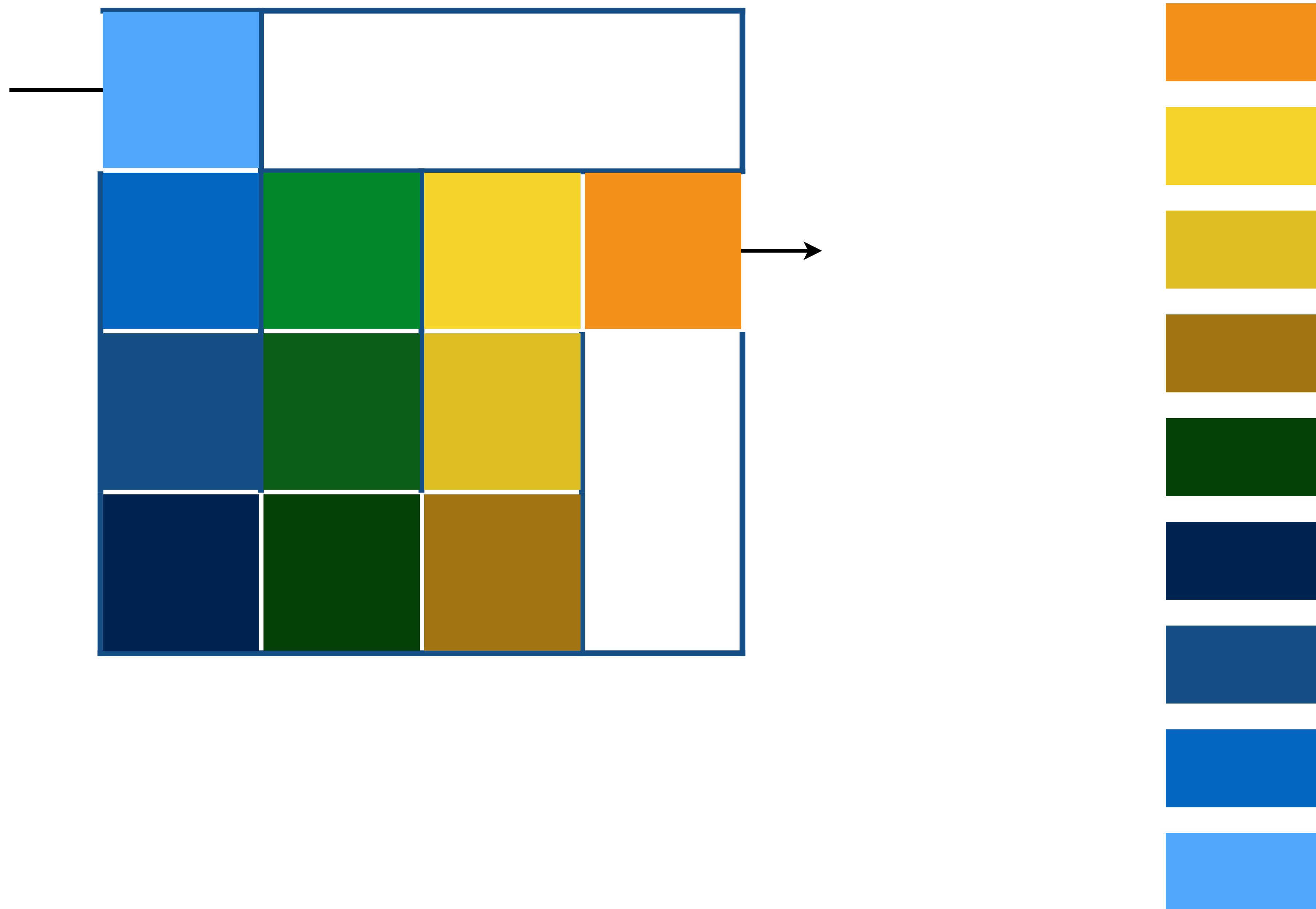


2. Dépile (POP) les lettres

S-A-N-A-N-A

Exemple #2: naviguer un labyrinthe

1. On empile (PUSH) les cases
2. Cul-de-sac? Dépile (POP) jusqu'à jonction



Le “Stack Pointer” (R13)

- Par convention, on définit un bloc de mémoire qu’on nomme “pile” (stack)
- La pile est utilisée pour sauvegarder temporairement des valeurs de travail
  - exemples: variables locales, paramètres et valeurs de retour des fonctions
- Le Stack Pointer contient une adresse de la pile à laquelle on peut écrire ou lire une valeur
- Changer la valeur de SP directement peut être dangereux! On utilise plutôt des instructions dédiées:
  - PUSH: rajouter un élément sur la pile
  - POP: enlever un élément de la pile

La pile

- Le registre SP (Stack Pointer) indique le dessus de la pile en tout temps.

- PUSH

```
PUSH {Rs}          ; 1) SP = SP - 4
                   ; 2) place le contenu de Rs sur la pile

PUSH {R0, R1, R2}  ; 1) SP = SP - 4*(nombre de registres),
                   ; 2) Place le contenu de R2, R1, et R0 (dans l'ordre)
                   ;    sur la pile.
```

  - Permet de mettre une (ou plusieurs) donnée(s) sur la pile.

- POP

```
POP {Rd}           ; 1) Place le contenu sur la pile dans Rd
                   ; 2) SP = SP + 4

POP {R0, R1, R2}   ; 1) Place le contenu sur la pile dans R0, R1, et R2
                   ;    (dans l'ordre)
                   ; 2) SP = SP + 4*(nombre de registres)
```

  - Permet de récupérer une (ou plusieurs) donnée(s) de la pile.

PUSH / POP

- Lors d'un PUSH, les registres sont empilés du *plus grand au plus petit* numéro de registre

```
PUSH {R0, R1, R2, LR}
```

R0
R1
R2
LR

- Lors d'un POP, les registres sont dépilés du *plus petit au plus grand* numéro de registre

```
POP {R0, R1, R2, LR}
```

PUSH / POP

- Lors d'un PUSH, les registres sont empilés du *plu*…

Peu importe l'ordre dans lequel les registres sont indiqués dans l'instruction. Par exemple:

```
PUSH {R3, R1, R0, R2}
```

va tout de même empiler les registres du plus grand au plus petit numéro.

R0
R1
R2
LR

- Lor… *plus petit au plus grand* numéro de registre

```
POP {R0, R1, R2, LR}
```

Préparation d'une pile

- La pile est habituellement:
  - descendante: lors d'un PUSH, SP *diminue* de 4 octets (pourquoi 4?)
  - placée après les données en RAM, donc à une adresse supérieure
- Exemple:

```
SECTION CODE

main
  LDR   SP, =maPile     ; Préparons une pile (de 64 octets)
  ADD   SP, SP, #64     ; La pile descend, donc il faut commencer à la fin

SECTION DATA

maPile      DS32    16 ; Pile de 64 octets (16*4 = 64)
```

Appel de fonction et adresse de retour, avec pile

- Que se passe-t-il si une fonction appelle une autre fonction?

Main	Fonction1	Fonction2
; Debut Main BL Fonction1 ; Suite Main	; Debut Fonction1 BL Fonction2 ; Suite 1 Fonction1 BL Fonction2 ; Suite 2 Fonction1 BX LR	; Code Fonction 2 BX LR

Main	Fonction1	Fonction2
; Debut Main BL Fonction1 ; Suite Main	**PUSH {LR}** ; Debut Fonction1 BL Fonction2 ; Suite 1 Fonction1 BL Fonction2 ; Suite 2 Fonction1 **POP {LR}** BX LR	**PUSH {LR}** ; Code Fonction 2 **POP {LR}** BX LR

Exemple d’appel de fonction

- L’instruction BL commande un branchement
- Le nom de la fonction suit l’instruction BL
- L’éditeur de lien convertira le nom de la fonction en adresse

Code C

```
Retour = FonctionSansParametre();
```

Assembleur

```
BL    FonctionSansParametre
```

Appel de fonctions — conventions

- Paramètres:
  - On se sert de R0 à R3 lorsqu'il y en a 4 ou moins
  - S'il y en a plus?
    - On utilise la pile… et/ou la mémoire
- Valeur de retour:
  - On se sert de R0 lorsqu'il y en a 1 ou moins
  - S'il y en a plus?
    - On utilise la pile… et/ou la mémoire

Appel de fonction: 1 paramètre et 1 retour

- On place la valeur du paramètre dans `R0` juste avant l'appel
- L'instruction `BL` commande un branchement
- `R0` contient la valeur de retour une fois la fonction exécutée

Code C

```
Retour = FonctionAUnParametre(0x12);
```

Code Assembleur

```
MOV   R0, #0x12              ; R0 contient le paramètre de la fonction
BL    FonctionAUnParametre ; Appel de la fonction
MOV   R3, R0                 ; Récupère le résultat de la fonction
```

Code C

```
int FonctionAUnParametre (int param)
{ return param + 1; }
```

Code Assembleur

```
FonctionAUnParametre
ADD  R0, R0, #1              ; R0 contient le paramètre de la fonction
BX   LR                      ; R0 contient le résultat de la fonction
```

Exemple

- Appel de fonction, place le résultat dans R2

```
MOV  R0, #8          ; Nous voulons que R2 = Fonction(8)
BL   MaFonction
MOV  R2, R0
```

```
MaFonction           ; R0 = paramètre, R0 = valeur de retour

PUSH {LR}            ; Préserve LR

ADD R1, R0, R0       ; Calcule le double du paramètre passé en entrée
MOV R0, R1           ; R0 = valeur de retour

POP {LR}             ; Restaure R4 à sa valeur initiale
BX LR
```

- Quel est le problème?
  - Qu'arrive-t-il à R1?

Préservation de l'environnement

- Le nombre de registres étant limité, on ne veut pas qu'une fonction remplace le contenu des registres
- Problème:
  - La fonction ne connait pas le nom des registres qui sont utilisés par le code qui fait l'appel de la fonction
  - Le code qui fait appel à la fonction ne connait pas le nom des registres qui sont utilisés par la fonction
- Solution:
  - La fonction “protège” le contenu des registres qu'elle utilise
- Méthode utilisée:
  - On protège le contenu d'un registre en le sauvegardant sur la pile avec `PUSH`
  - On récupère ce contenu avec `POP`

Retour sur l'exemple

- Comment adapter notre exemple?

```
MOV  R0, #8                ; Nous voulons appeler MaFonction(8)
BL   MaFonction
MOV  R2, R0
```

```
MaFonction                 ; R0 = paramètre, R0 = valeur de retour



ADD R1, R0, R0             ; Calcule le double du paramètre passé en entrée
MOV R0, R1                 ; R0 = valeur de retour


BX LR
```

Retour sur l'exemple

- La fonction préserve le contenu de R1 (en utilisant la pile) pour ne pas le corrompre en faisant ses calculs

```
MOV  R0, #8               ; Nous voulons appeler MaFonction(8)
BL   MaFonction
MOV  R2, R0
```

```
MaFonction                ; R0 = paramètre, R0 = valeur de retour

PUSH {R1, LR}             ; Préserve LR et R4 car nous allons nous en servir

ADD R1, R0, R0            ; Calcule le double du paramètre passé en entrée
MOV R0, R1                ; R0 = valeur de retour

POP {R1, LR}              ; Restaure R1 et LR à leur valeur initiale
BX LR
```

Retour sur l’exemple

- Le code qui fait l’appel préserve le contenu de R0 pour ne pas le perdre lors du retour de la fonction

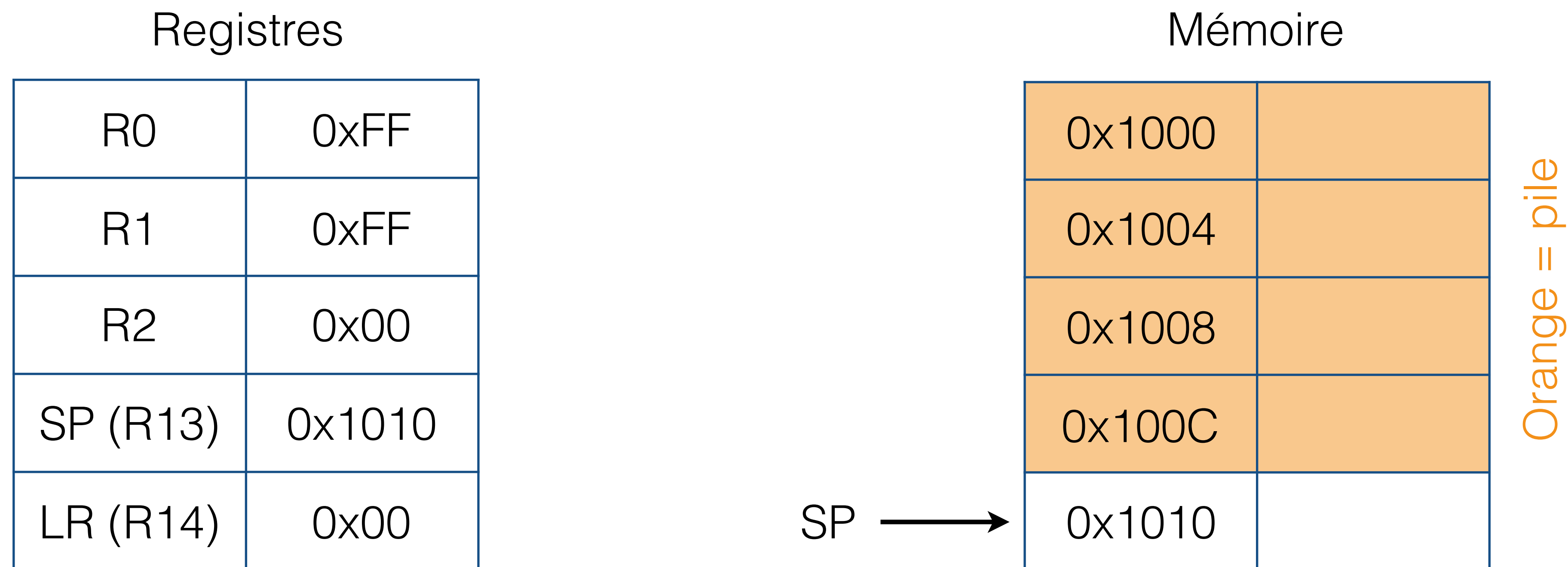
```
PUSH {R0}
MOV  R0, #8                  ; Nous voulons appeler MaFonction(8)
BL   MaFonction
MOV  R2, R0
POP {R0}
```

```
MaFonction                  ; R0 = paramètre, R0 = valeur de retour

PUSH {R1, LR}               ; Préserve LR et R4 car nous allons nous en servir

ADD R1, R0, R0              ; Calcule le double du paramètre passé en entrée
MOV R0, R1                  ; R0 = valeur de retour

POP {R1, LR}                ; Restaure R1 et LR à leur valeur initiale
BX LR
```

Exemple: illustration

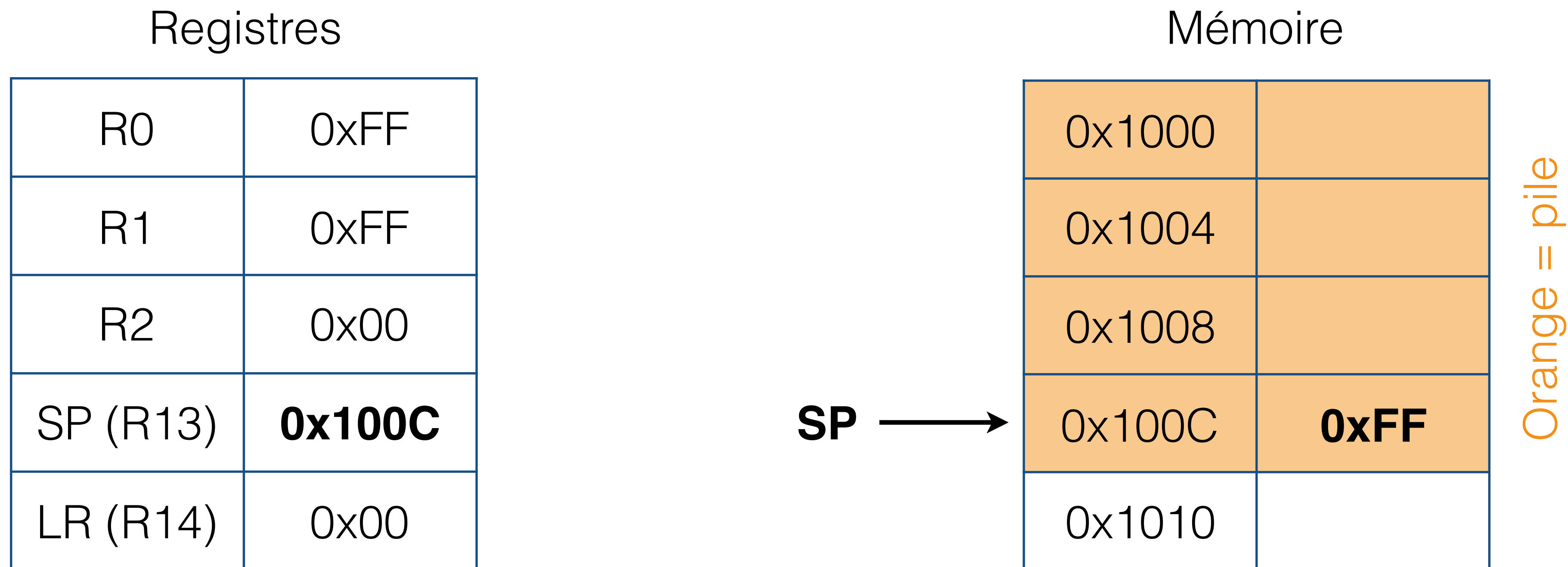
```
0x90   PUSH {R0}
0x94   MOV  R0, #8          ; Nous voulons appeler MaFonction(8)
0x98   BL   MaFonction
0x9C   MOV  R2, R0
0xA0   POP {R0}

       MaFonction           ; R0 = paramètre, R0 = valeur de retour

0xBC   PUSH {R1, LR}        ; Préserve LR et R1 car nous allons nous en servir

0xC0   ADD R1, R0, R0       ; Calcule le double du paramètre passé en entrée
0xC4   MOV R0, R1           ; R0 = valeur de retour

0xC8   POP {R1, LR}         ; Restaure R1 et LR à leur valeur initiale
0xCC   BX LR
```

Exemple: illustration



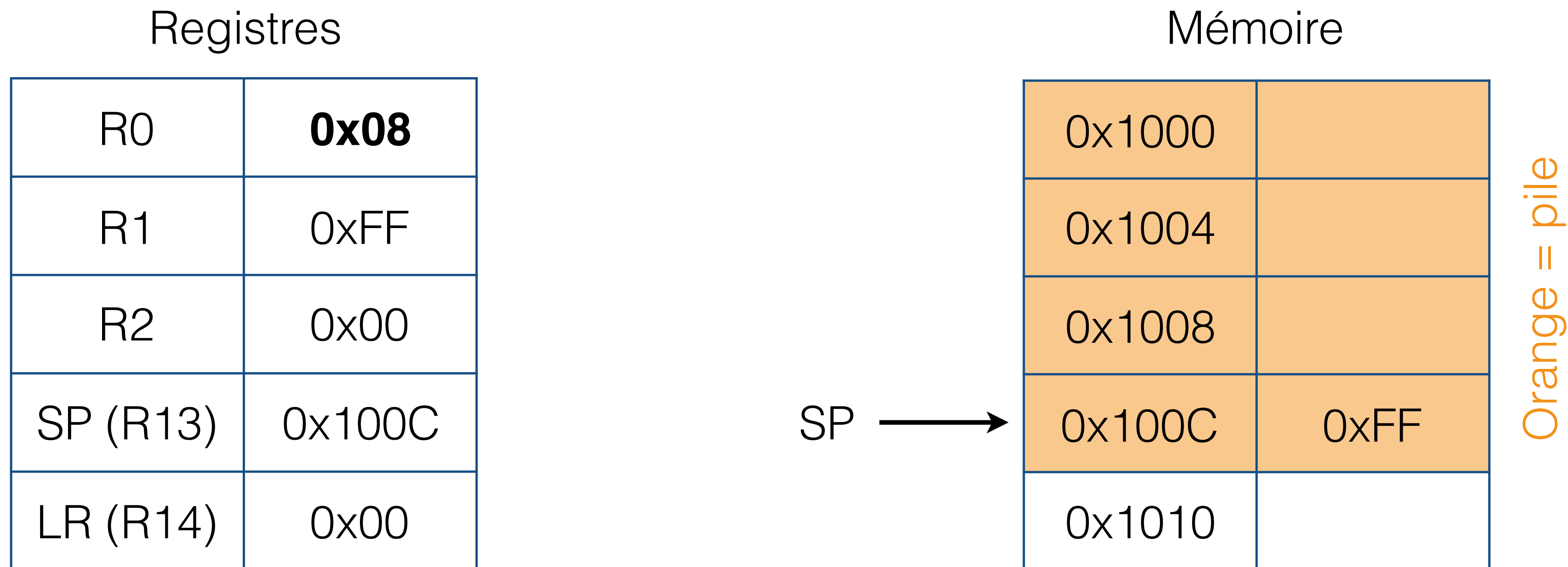

```
0x90   PUSH {R0}
0x94   MOV  R0, #8        ; Nous voulons appeler MaFonction(8)
0x98   BL   MaFonction
0x9C   MOV  R2, R0
0xA0   POP {R0}

       MaFonction          ; R0 = paramètre, R0 = valeur de retour

0xBC   PUSH {R1, LR}       ; Préserve LR et R1 car nous allons nous en servir

0xC0   ADD R1, R0, R0      ; Calcule le double du paramètre passé en entrée
0xC4   MOV R0, R1          ; R0 = valeur de retour

0xC8   POP {R1, LR}        ; Restaure R1 et LR à leur valeur initiale
0xCC   BX LR
```

Après PUSH {R0}



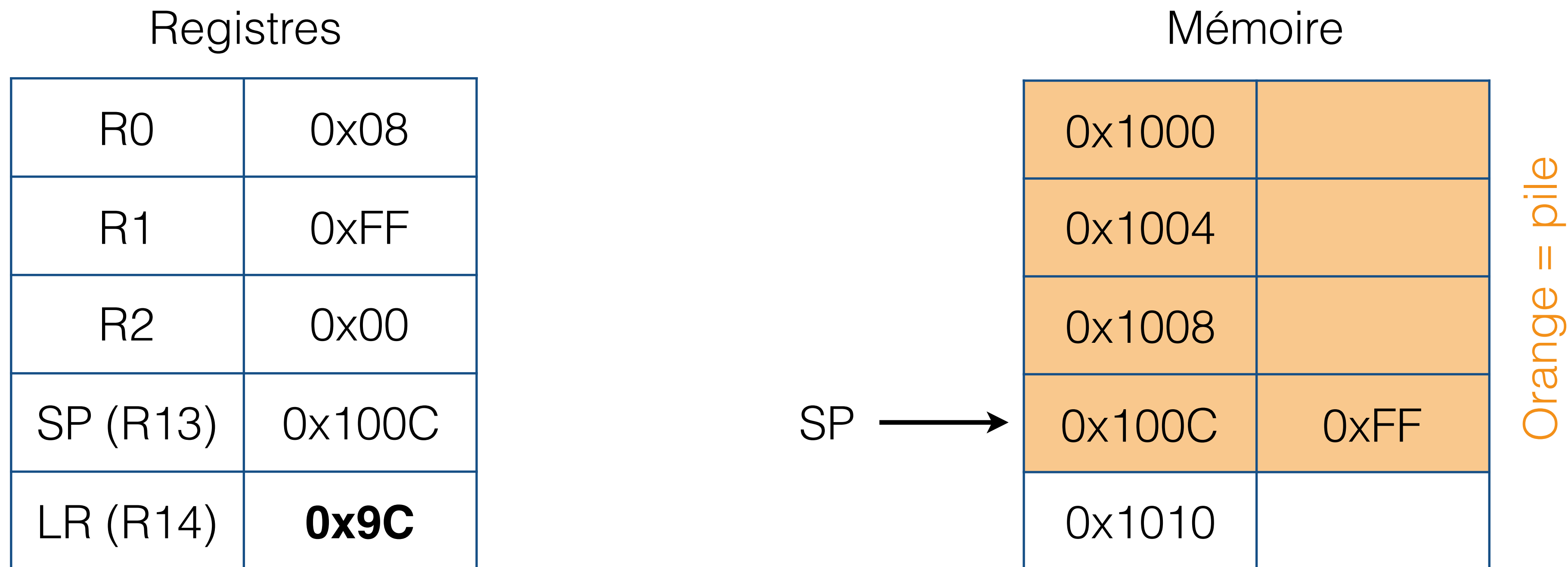

```
0x90   PUSH {R0}
0x94   MOV  R0, #8          ; Nous voulons appeler MaFonction(8)
0x98   BL   MaFonction
0x9C   MOV  R2, R0
0xA0   POP {R0}

       MaFonction           ; R0 = paramètre, R0 = valeur de retour

0xBC   PUSH {R1, LR}        ; Préserve LR et R1 car nous allons nous en servir

0xC0   ADD R1, R0, R0       ; Calcule le double du paramètre passé en entrée
0xC4   MOV R0, R1           ; R0 = valeur de retour

0xC8   POP {R1, LR}         ; Restaure R1 et LR à leur valeur initiale
0xCC   BX LR
```

Après `MOV R0, #8`




```
0x90    PUSH {R0}
0x94    MOV  R0, #8          ; Nous voulons appeler MaFonction(8)
0x98    BL   MaFonction
0x9C    MOV  R2, R0
0xA0    POP {R0}

        MaFonction           ; R0 = paramètre, R0 = valeur de retour

0xBC    PUSH {R1, LR}        ; Préserve LR et R1 car nous allons nous en servir

0xC0    ADD R1, R0, R0       ; Calcule le double du paramètre passé en entrée
0xC4    MOV R0, R1           ; R0 = valeur de retour

0xC8    POP {R1, LR}         ; Restaure R1 et LR à leur valeur initiale
0xCC    BX LR
```

Après `BL MaFonction`




```
         0x90   PUSH {R0}
         0x94   MOV  R0, #8        ; Nous voulons appeler MaFonction(8)
         0x98   BL   MaFonction
LR  -->  0x9C   MOV  R2, R0
         0xA0   POP {R0}

                MaFonction         ; R0 = paramètre, R0 = valeur de retour

         0xBC   PUSH {R1, LR}      ; Préserve LR et R1 car nous allons nous en servir

         0xC0   ADD R1, R0, R0     ; Calcule le double du paramètre passé en entrée
         0xC4   MOV R0, R1         ; R0 = valeur de retour

         0xC8   POP {R1, LR}       ; Restaure R1 et LR à leur valeur initiale
         0xCC   BX LR
```

Après PUSH {R1, LR}

Registres

R0	0x08
R1	0xFF
R2	0x00
SP (R13)	**0x1004**
LR (R14)	0x9C

Mémoire

0x1000	
0x1004	**0xFF**
0x1008	**0x9C**
0x100C	0xFF
0x1010	

SP → 0x1004

Orange = pile

```
      0x90   PUSH {R0}
      0x94   MOV  R0, #8          ; Nous voulons appeler MaFonction(8)
      0x98   BL   MaFonction
LR →  0x9C   MOV  R2, R0
      0xA0   POP {R0}

             MaFonction           ; R0 = paramètre, R0 = valeur de retour

      0xBC   PUSH {R1, LR}        ; Préserve LR et R1 car nous allons nous en servir

      0xC0   ADD R1, R0, R0       ; Calcule le double du paramètre passé en entrée
      0xC4   MOV R0, R1           ; R0 = valeur de retour

      0xC8   POP {R1, LR}         ; Restaure R1 et LR à leur valeur initiale
      0xCC   BX LR
```

Après `ADD R1, R0, R0`

Registres

R0	0x08
R1	**0x10**
R2	0x00
SP (R13)	0x1004
LR (R14)	0x9C

Mémoire

0x1000	
0x1004	0xFF
0x1008	0x9C
0x100C	0xFF
0x1010	

SP → 0x1004

Orange = pile

```
     0x90   PUSH {R0}
     0x94   MOV  R0, #8        ; Nous voulons appeler MaFonction(8)
     0x98   BL   MaFonction
LR → 0x9C   MOV  R2, R0
     0xA0   POP {R0}

            MaFonction          ; R0 = paramètre, R0 = valeur de retour

     0xBC   PUSH {R1, LR}       ; Préserve LR et R1 car nous allons nous en servir

     0xC0   ADD R1, R0, R0      ; Calcule le double du paramètre passé en entrée
     0xC4   MOV R0, R1          ; R0 = valeur de retour

     0xC8   POP {R1, LR}        ; Restaure R1 et LR à leur valeur initiale
     0xCC   BX LR
```

Après `MOV R0, R1`

Registres

R0	**0x10**
R1	0x10
R2	0x00
SP (R13)	0x1004
LR (R14)	0x9C

Mémoire

0x1000	
0x1004	0xFF
0x1008	0x9C
0x100C	0xFF
0x1010	

SP → 0x1004

Orange = pile

```
      0x90   PUSH {R0}
      0x94   MOV  R0, #8       ; Nous voulons appeler MaFonction(8)
      0x98   BL   MaFonction
LR → 0x9C   MOV  R2, R0
      0xA0   POP {R0}

             MaFonction        ; R0 = paramètre, R0 = valeur de retour

      0xBC   PUSH {R1, LR}     ; Préserve LR et R1 car nous allons nous en servir

      0xC0   ADD R1, R0, R0    ; Calcule le double du paramètre passé en entrée
      0xC4   MOV R0, R1        ; R0 = valeur de retour

      0xC8   POP {R1, LR}      ; Restaure R1 et LR à leur valeur initiale
      0xCC   BX LR
```

Après `POP {R1, LR}`

Registres

R0	0x10
R1	**0xFF**
R2	0x00
SP (R13)	**0x100C**
LR (R14)	**0x9C**

Mémoire

0x1000	
0x1004	0xFF
0x1008	0x9C
SP → 0x100C	0xFF
0x1010	

Orange = pile

```
      0x90   PUSH {R0}
      0x94   MOV  R0, #8        ; Nous voulons appeler MaFonction(8)
      0x98   BL   MaFonction
LR →  0x9C   MOV  R2, R0
      0xA0   POP {R0}

             MaFonction         ; R0 = paramètre, R0 = valeur de retour

      0xBC   PUSH {R1, LR}      ; Préserve LR et R1 car nous allons nous en servir

      0xC0   ADD R1, R0, R0     ; Calcule le double du paramètre passé en entrée
      0xC4   MOV R0, R1         ; R0 = valeur de retour

      0xC8   POP {R1, LR}       ; Restaure R1 et LR à leur valeur initiale
      0xCC   BX LR
```

Après `BX LR`

Registres

R0	0x10
R1	0xFF
R2	0x00
SP (R13)	0x100C
LR (R14)	0x9C

Mémoire

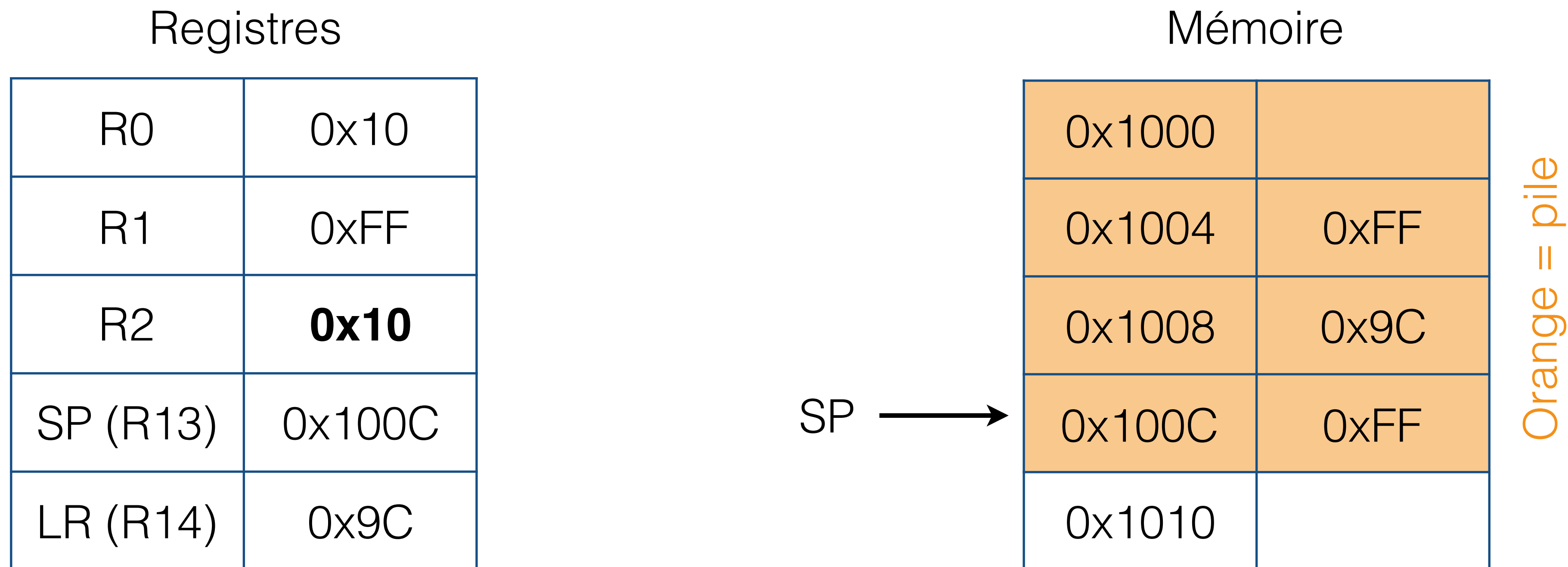
0x1000	
0x1004	0xFF
0x1008	0x9C
0x100C	0xFF
0x1010	

SP → 0x100C

Orange = pile

```
      0x90   PUSH {R0}
      0x94   MOV  R0, #8        ; Nous voulons appeler MaFonction(8)
      0x98   BL   MaFonction
LR →  0x9C   MOV  R2, R0
      0xA0   POP {R0}

             MaFonction         ; R0 = paramètre, R0 = valeur de retour

      0xBC   PUSH {R1, LR}      ; Préserve LR et R1 car nous allons nous en servir

      0xC0   ADD R1, R0, R0     ; Calcule le double du paramètre passé en entrée
      0xC4   MOV R0, R1         ; R0 = valeur de retour

      0xC8   POP {R1, LR}       ; Restaure R1 et LR à leur valeur initiale
      0xCC   BX LR
```

Après `MOV R2, R0`




```
       0x90   PUSH {R0}
       0x94   MOV  R0, #8       ; Nous voulons appeler MaFonction(8)
       0x98   BL   MaFonction
LR --> 0x9C   MOV  R2, R0
       0xA0   POP {R0}

              MaFonction        ; R0 = paramètre, R0 = valeur de retour

       0xBC   PUSH {R1, LR}     ; Préserve LR et R1 car nous allons nous en servir

       0xC0   ADD R1, R0, R0    ; Calcule le double du paramètre passé en entrée
       0xC4   MOV R0, R1        ; R0 = valeur de retour

       0xC8   POP {R1, LR}      ; Restaure R1 et LR à leur valeur initiale
       0xCC   BX LR
```

Après `POP {R0}`

Registres

R0	**0xFF**
R1	0xFF
R2	0x10
SP (R13)	**0x1010**
LR (R14)	0x9C

Mémoire

0x1000	
0x1004	0xFF
0x1008	0x9C
0x100C	0xFF
0x1010	

SP → 0x1010

Orange = pile

```
      0x90   PUSH {R0}
      0x94   MOV  R0, #8        ; Nous voulons appeler MaFonction(8)
      0x98   BL   MaFonction
LR →  0x9C   MOV  R2, R0
      0xA0   POP {R0}

             MaFonction         ; R0 = paramètre, R0 = valeur de retour

      0xBC   PUSH {R1, LR}      ; Préserve LR et R1 car nous allons nous en servir

      0xC0   ADD R1, R0, R0     ; Calcule le double du paramètre passé en entrée
      0xC4   MOV R0, R1         ; R0 = valeur de retour

      0xC8   POP {R1, LR}       ; Restaure R1 et LR à leur valeur initiale
      0xCC   BX LR
```

Démonstration
(Fonction, paramètres et pile)

Appel de fonction: 1—4 paramètres et 1 retour

- R0, R1, R2 et R3 servent à passer les paramètres
- R0 sert à retourner la valeur de retour

Code C

```
Retour = FonctionAMoinsDe5Parametres(0x12, 0x23, 0x34, 0x45);
```

Code Assembleur

```
MOV R0, #0x12              ; Paramètre 1
MOV R1, #0x23              ; Paramètre 2
MOV R2, #0x34              ; Paramètre 3
MOV R3, #0x45              ; Paramètre 4

BL FonctionAUnParametre    ; Appel de fonction

MOV R4, R0                 ; Valeur de sortie
```

Code C

```
int FonctionAMoinsDe5Parametres(int P0, int P1, int P2, int P3)
{ return P0 + P1 + P2 + P3; }
```

Code Assembleur

```
FonctionAMoinsDe5Parametres
ADD R0, R0, R1             ; Calcul de la somme
ADD R0, R0, R2
ADD R0, R0, R3
BX  LR                     ; Fonction terminée
```

Cas à plus de 4 paramètres: par la pile

- On utilise `R0` à `R3`
- Si on en veut plus, on utilise la pile
- Il faut tenir compte des registres qu'on préserve en début de fonction avant de lire des valeurs dans la pile

```
MOV R0, #1
MOV R1, #2
MOV R2, #3
MOV R3, #4
MOV R5, #5                  ; Plaçons le 5e paramètre dans R5

BL  FonctionA5Parametres    ; Appel de fonction

MOV R8, R0
```

```
FonctionsA5Parametres
PUSH {R9, LR}               ; Sauvegarde LR et R9 (nous l'utiliserons)

LDR R9, [SP, #8]            ; R9 = paramètre 5
ADD R0, R0, R9              ; retour = paramètre 0 + paramètre 5

POP {R9, LR}                ; Restaure LR et R9
BX LR
```

Autres cas à plus de 4 paramètres

- Les paramètres sont indépendants les uns des autres:
  - On les place un par un sur la pile et ils sont lus un par un
- Les paramètres font partie d’une chaîne ou d’un vecteur:
  - On place l’adresse du début des valeurs sur la pile
  - On place le nombre de valeurs sur la pile
  - La fonction peut lire en boucle
- Les paramètres font partie d’une structure dont les éléments n’occupent pas tous le même nombre d’octets:
  - On place l’adresse du début de la structure sur la pile
  - On place le nombre de mots utilisés pour mémoriser la structure
  - La fonction doit lire la pile en tenant compte des longueurs variées des valeurs de la structure
- Ça correspond à passer un pointeur ou une référence en langage plus évolué que l’assembleur (natif ou évolué)

Cas à plusieurs retours

- R0 peut quand même servir à retourner une valeur
- Le code qui fait appel à la fonction passe des valeurs d’adresse en paramètre (par R0 jusqu’à R3 et/ou par la pile)
- Les adresses passées pointent sur des espaces mémoires où la fonction pourra écrire sans causer de problème au code qui lui fait appel
- La fonction fait ses calculs et elle utilise les valeurs d’adresses pour savoir où sauvegarder les résultats à retourner
- Le code qui a fait l’appel retrouve les valeurs retournées aux adresses qu’il a passé en paramètre
- Le principe fonctionne tant pour des variables indépendantes que pour des chaînes, des vecteurs, des structures, etc.

Annexe: La pile, plus que pour les retours

- La pile sert à entreposer les adresses de retours comme vu précédemment.
- La pile sert aussi à passer des paramètres
- La pile sert à sauvegarder les registres utilisés dans une fonction.
- Souvent, les variables locales (dont la portée se limite à une fonction), sont de l'espace mémoire sur la pile allouée dynamiquement (le compilateur modifie SP au début de la fonction et à la fin pour réserver de l'espace mémoire). Sinon les variables locales sont des registres.
- La pile sert à effectuer des calculs mathématiques comme réaliser une chaînes d'additions et de multiplications
- La pile sert à sauvegarder le contexte d'une tâche lors d'interruptions (voir prochain cours!)
- La pile est une structure de donnée fondamentale pour les ordinateurs. Tous les processeurs ont un pointeur de pile…