Assembleur ARM: Appels de fonction

GIF-1001 Ordinateurs: Structure et Applications
Jean-François Lalonde

Image: vectorstock

Appel de fonction

- Nous avons souvent besoin du même code

    - Exemple: calculer la puissance d'un nombre en effectuant une série de multiplications

$$2^3 = 2 \times 2 \times 2$$

- Plutôt que d'écrire le code à chaque fois que nous en avons besoin, on l'écrit dans une fonction que l'on peut appeler au besoin.

- En ARM, une fonction est un ensemble d'instructions à une adresse donnée.

    - On identifie la fonction par une étiquette

    - On appelle la fonction avec un branchement à cette étiquette

Appel de fonction: mauvais exemple

```
main
B MaFonction
AdresseDeRetour
MOV R0, #0

MaFonction
; Tâche dans la fonction
...
B AdresseDeRetour
```

Appel de fonction: branche à l'adresse de la fonction

Fin de la fonction:
on revient où on était rendus

Pourquoi est-ce un mauvais exemple?

Appel de fonction: mauvais exemple

```
main
B MaFonction
AdresseDeRetour
MOV R0, #0
…
B MaFonction
MOV R0, #1



MaFonction
; Tâche dans la fonction
...
B AdresseDeRetour
```

Appel de fonction: branche à l'adresse de la fonction

2e appel à la même fonction

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 fai

- Instruction `BL` (Branch and Link).

  - Tout d'abord, place l'adresse de retour dans LR

  - Ensuite, branche à l'étiquette

LR est le registre R14.
C'est le registre de liens (*link register*).

```
BL etiquetteDeFonction
```

Adresse de retour
Comment calculer l'adresse de retour?

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):

  - Tout d'abord, place l'adresse de retour dans LR

  - Ensuite, branche à l'étiquette

```
BL etiquetteDeFonction
```

Adresse de retour

Comment calculer l'adresse de retour?

C'est l'adresse immédiatement *après* le BL.
Donc, l'adresse de l'instruction courante + 4.
Donc, PC - 4.

Retour de fonction

- À la fin de la fonction, on veut retourner à l'adresse de retour

- Cette adresse a été stockée dans `LR` par le `BL`

- Il suffit donc d'y brancher avec l'instruction `BX`, qui place le contenu de `LR` dans `PC`:

```
BX LR           ; PC = LR
```

Appel de fonction: exemple corrigé

```
main
BL MaFonction
AdresseDeRetour
MOV R0, #0
…
BL MaFonction
MOV R0, #1


MaFonction
; Tâche dans la fonction
...
BX LR
```

Appel de fonction: branche à l'adresse de la fonction.

`BL` sauvegarde l'adresse de retour dans `LR`.

2e appel à la même fonction.

`BL` sauvegarde l'adresse de retour dans `LR`.

Retour de fonction

`BX` met `LR` dans `PC`
(branche à l'adresse dans `LR`)

Démonstration

(Fonction, problème réglé)

Appels de fonctions imbriquées

```
main
BL MaFonction
AdresseDeRetour
MOV R0, #0


MaFonction
; Tâche dans la fonctio
BL MaFonction2
...
BX LR

MaFonction2
; Tâche dans la fonctio
...
BX LR
```

Appel de fonction imbriquée

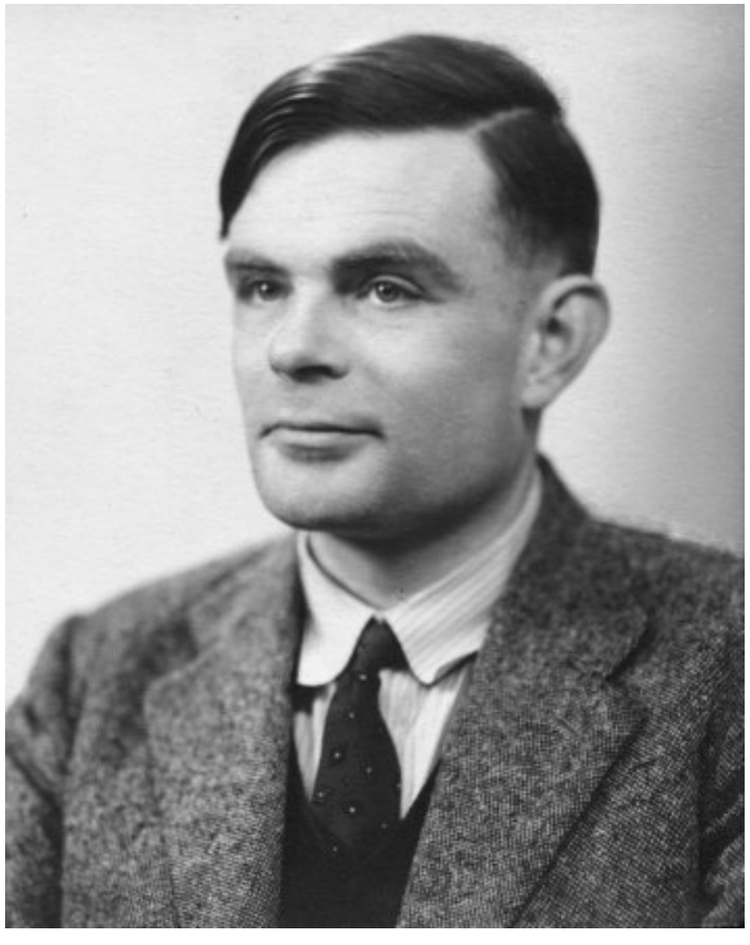
`MaFonction` appelle `MaFonction2`!

`MaFonction2` est une autre fonction

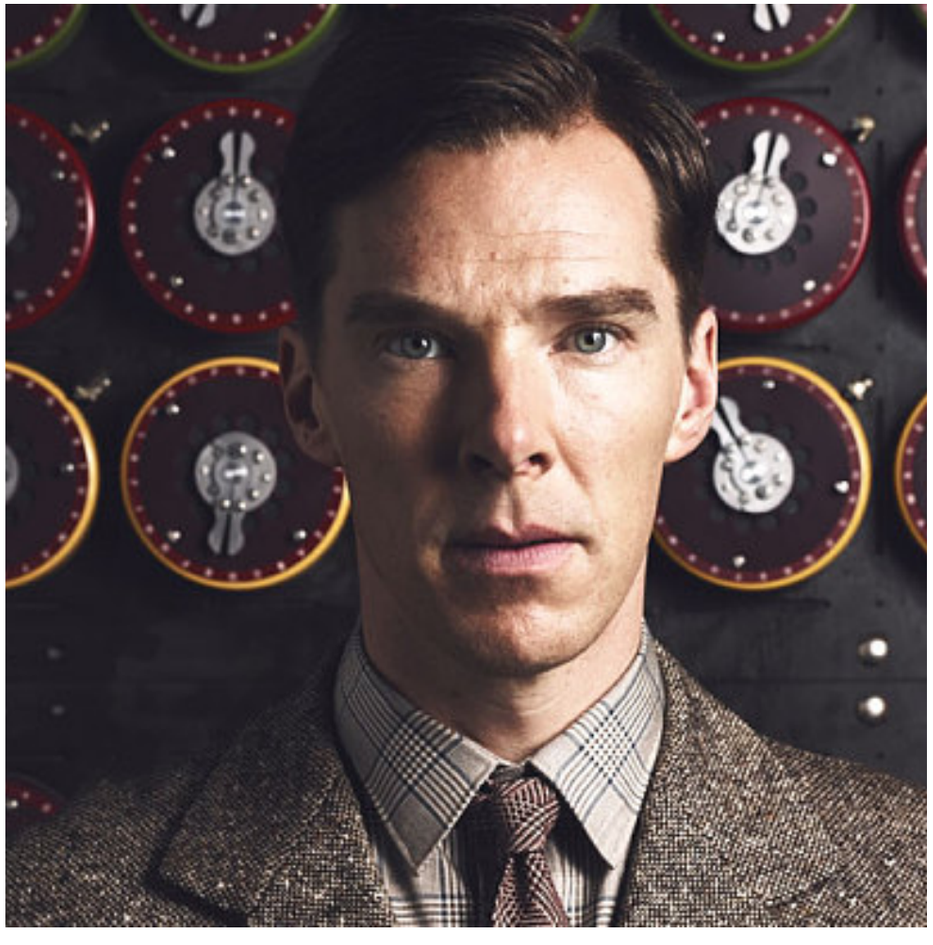
Piles (*stacks*)

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

Alan Turing



Imitation Game



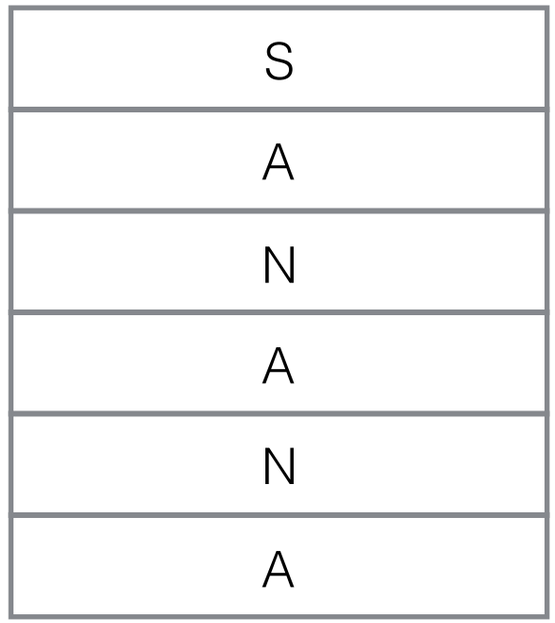
Piles (*stacks*)

- 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

Image: clker.com

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

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

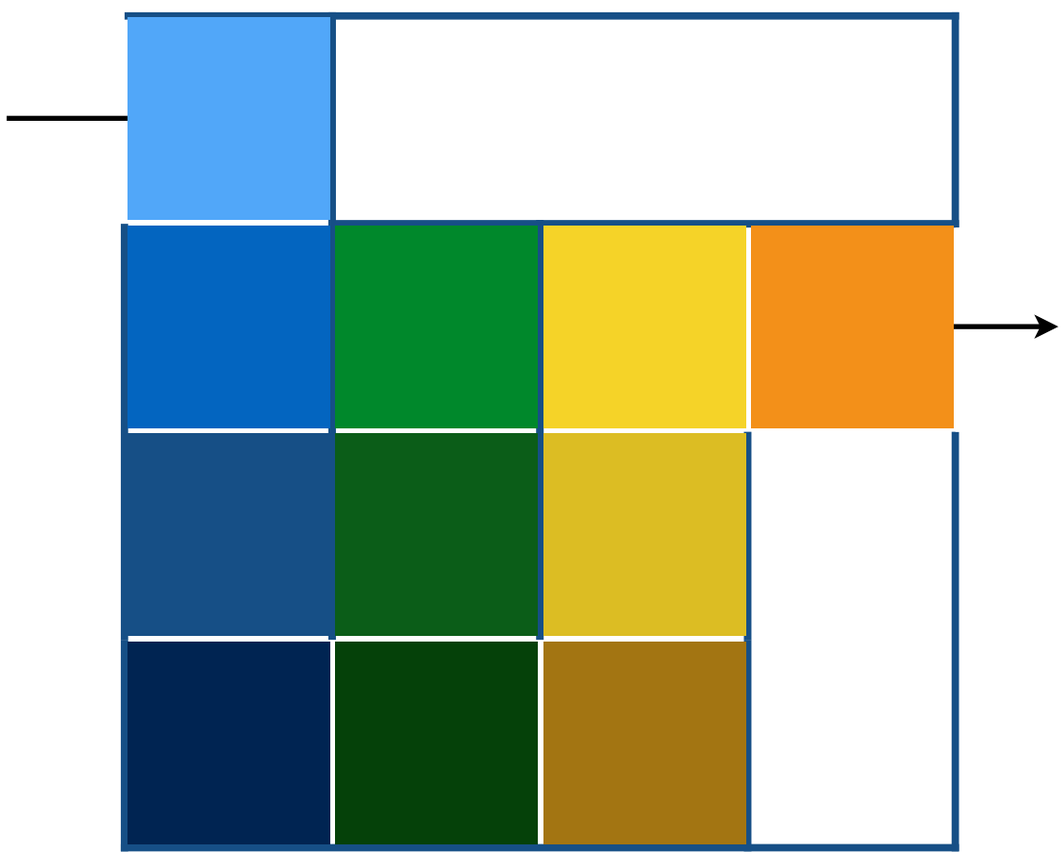


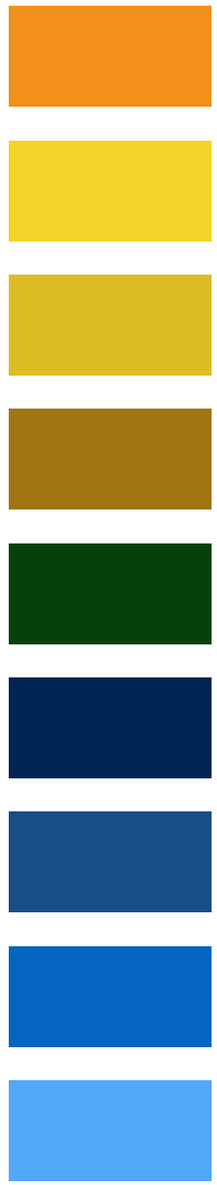
2. Dépile (POP) les lettres

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 pointeur de pile (R13)

- Le registre `R13` est le pointeur de pile (*stack pointer* ou `SP`*)*

- `SP` indique le dessus de la pile
  - l'endroit qui contient la donnée retournée si on `POP`

- Bien qu'il est possible de le faire, modifier la valeur de `SP` directement peut être dangereux!

Conseil d'ami
N'utilisez *jamais* `R13` (ni `SP`) dans votre code!
N'utilisez que les instructions `PUSH` et `POP`.

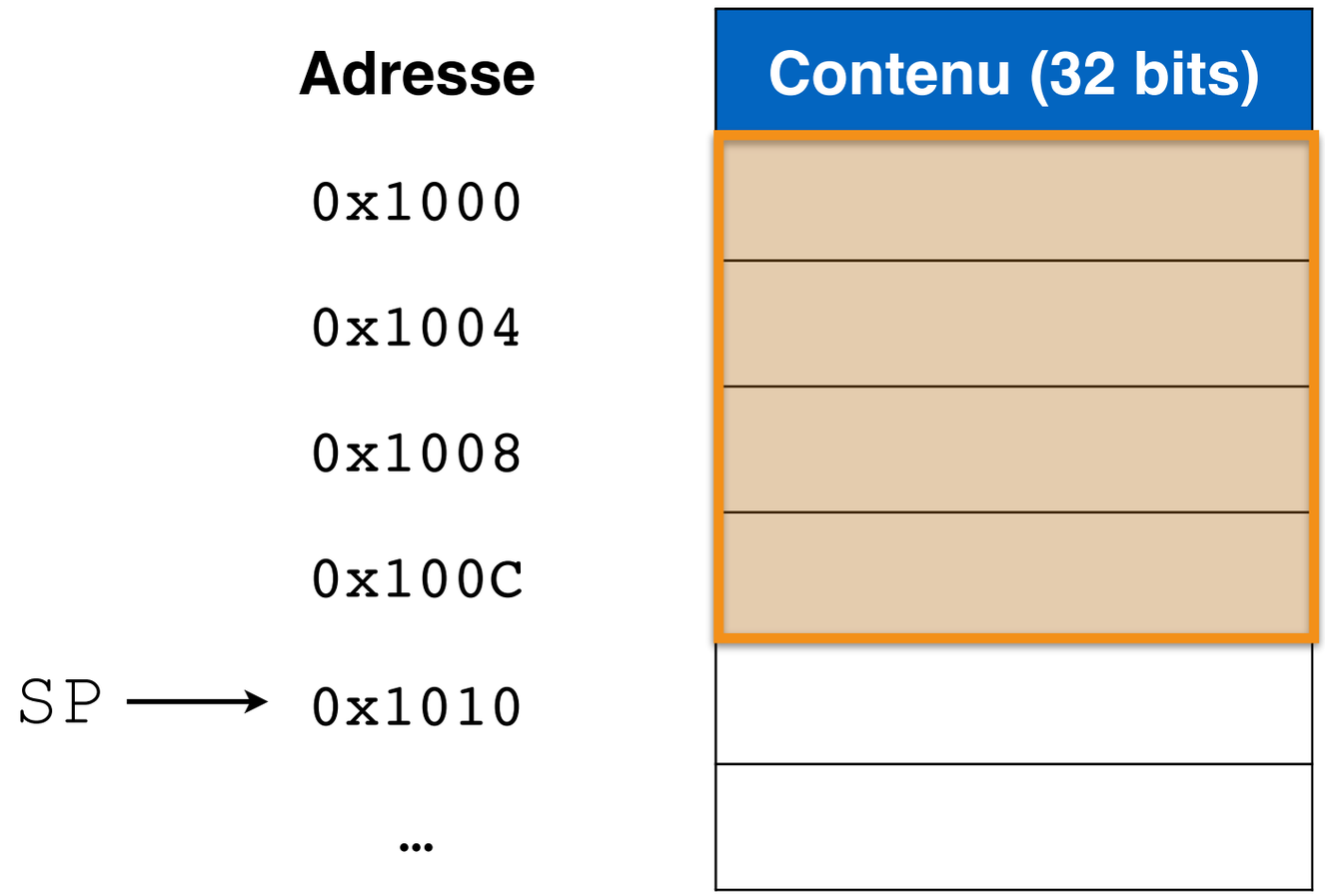
PUSH: empile

```
PUSH {Rs}    ; 1) SP = SP - 4
             ; 2) place le contenu de Rs à l'adresse
                  indiquée par SP
```

```
MOV R0, #0
PUSH {R0}
PUSH {R0}
PUSH {R0}
```

Exemple

Pile de 16 octets



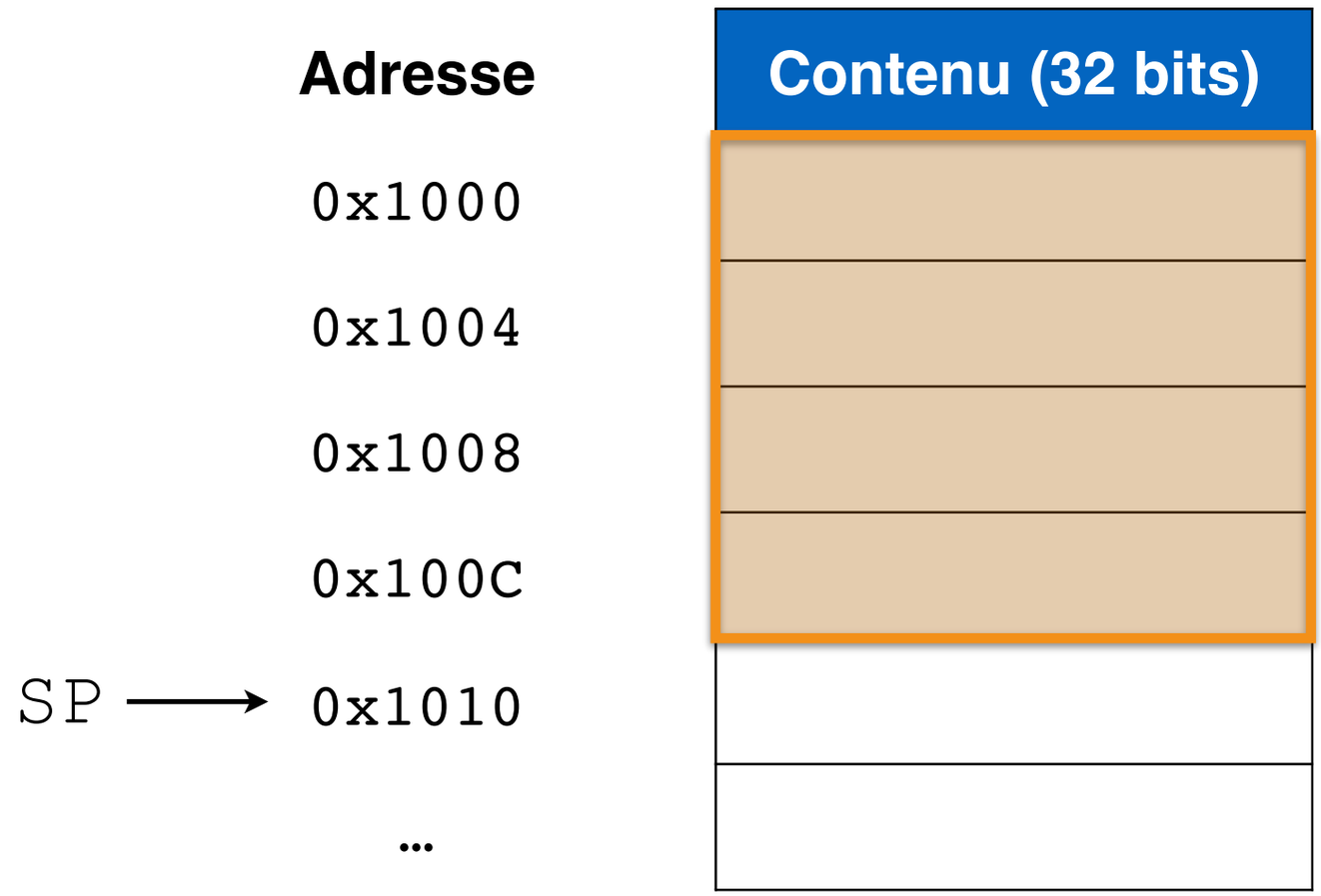
PUSH: empile

```
PUSH {Rs}    ; 1) SP = SP - 4
             ; 2) place le contenu de Rs à l'adresse
                  indiquée par SP
```

```
MOV R0, #0
PUSH {R0}
PUSH {R0}
PUSH {R0}
```

Exemple

Pile de 16 octets



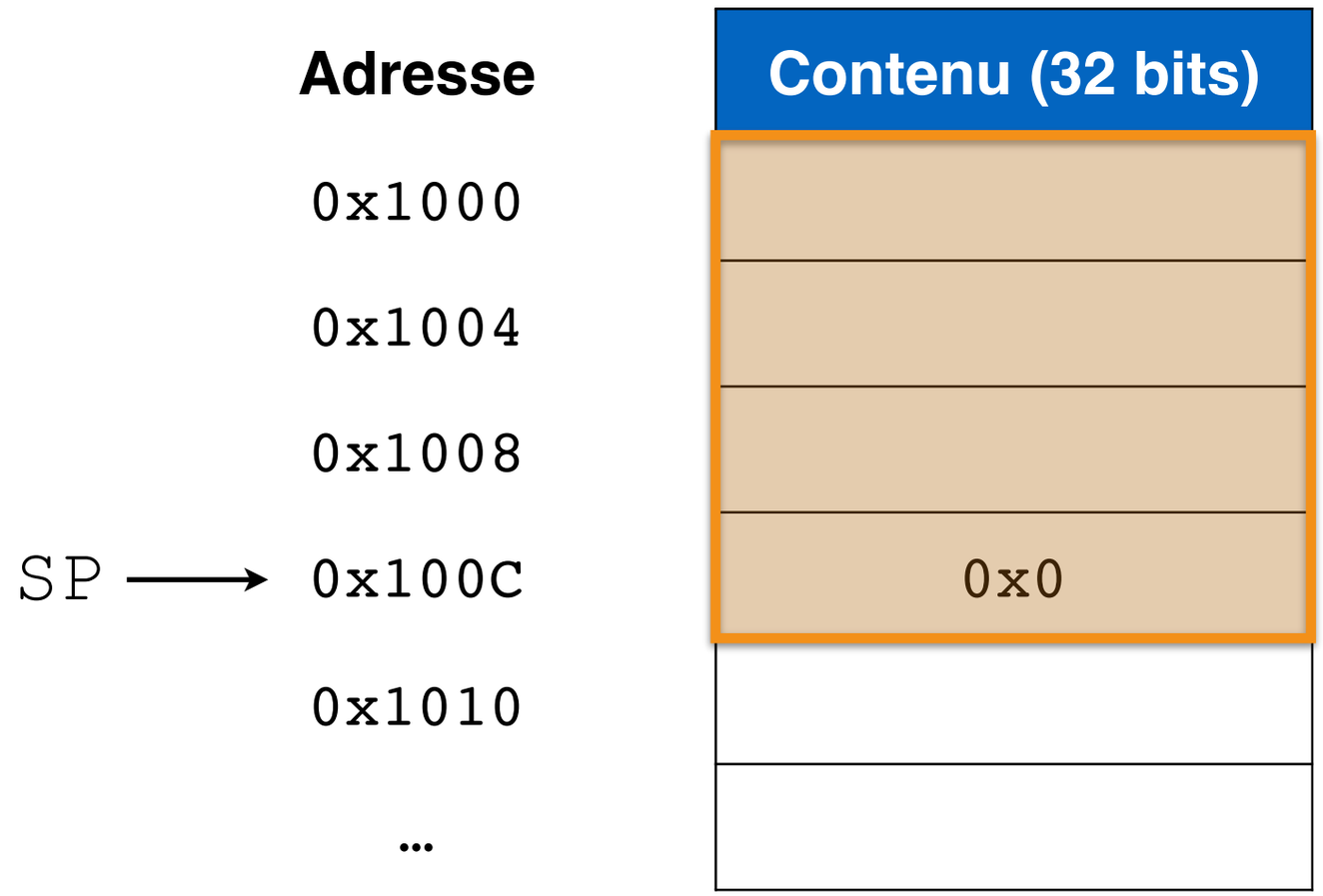
PUSH: empile

```
PUSH {Rs}    ; 1) SP = SP - 4
             ; 2) place le contenu de Rs à l'adresse
                  indiquée par SP
```

```
MOV R0, #0
PUSH {R0}
PUSH {R0}
PUSH {R0}
```

Exemple

Pile de 16 octets



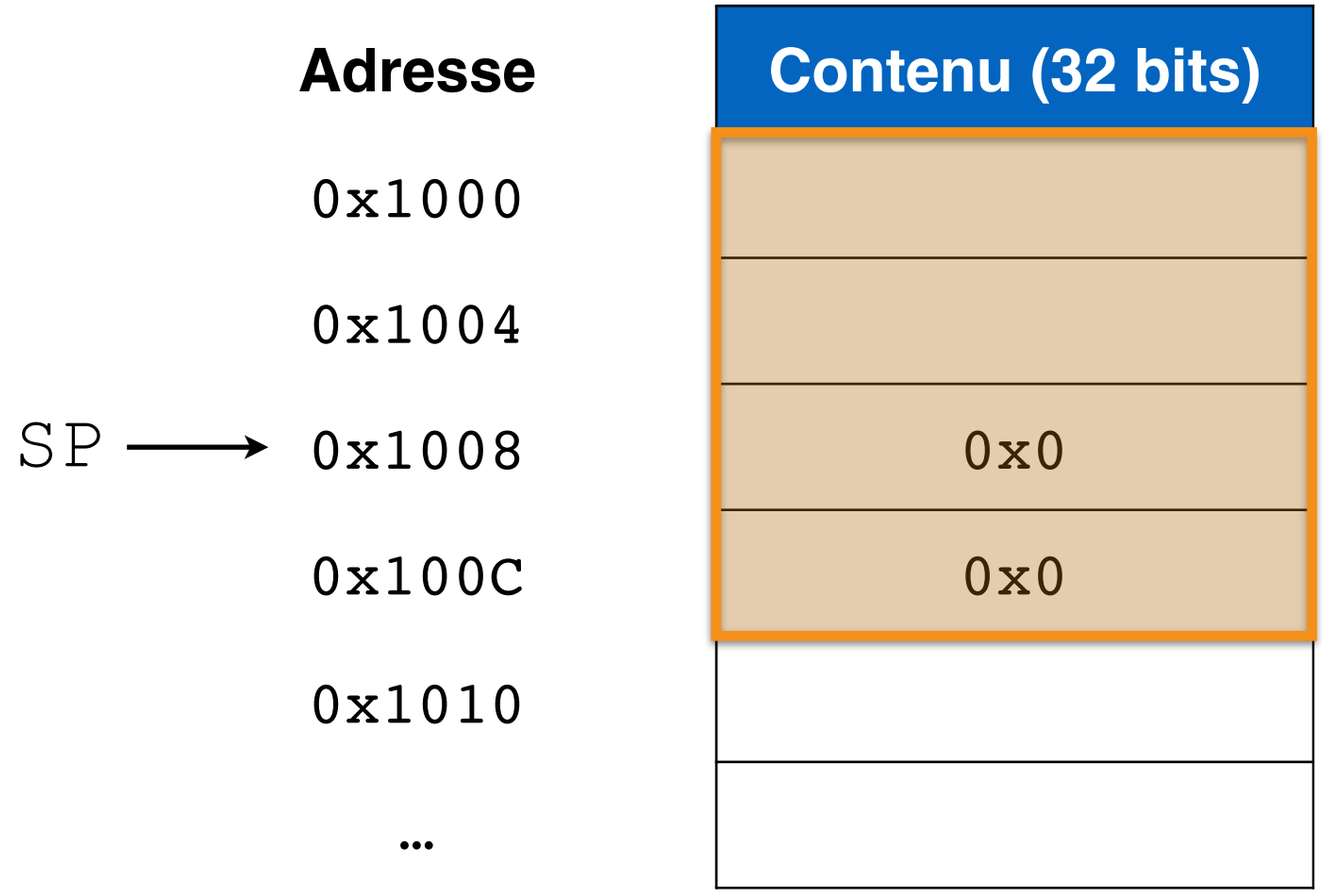
PUSH: empile

```
PUSH {Rs}    ; 1) SP = SP - 4
             ; 2) place le contenu de Rs à l'adresse
                  indiquée par SP
```

```
MOV R0, #0
PUSH {R0}
PUSH {R0}
PUSH {R0}
```

Exemple

Pile de 16 octets



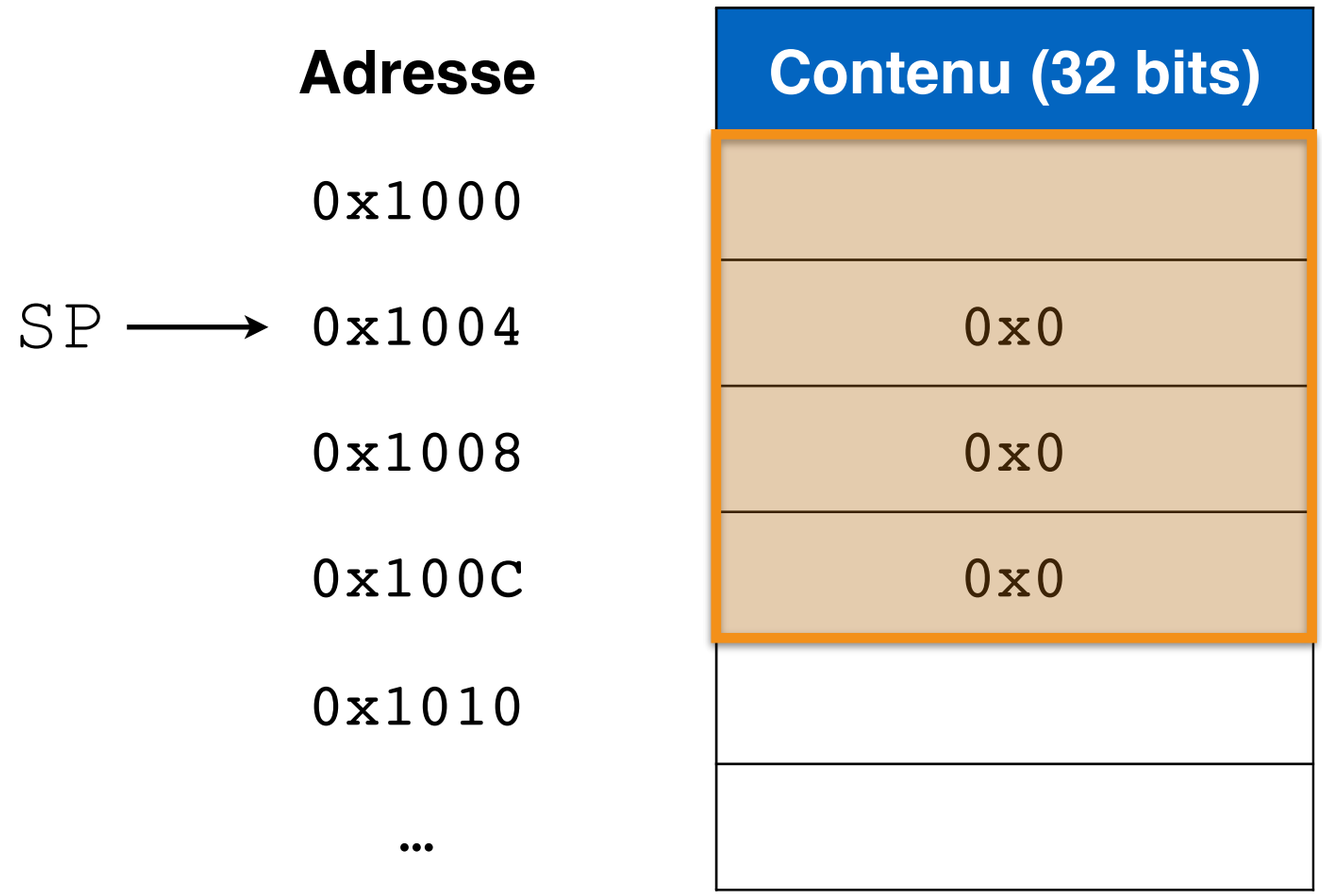
PUSH: empile

```
PUSH {Rs}    ; 1) SP = SP - 4
             ; 2) place le contenu de Rs à l'adresse
                  indiquée par SP
```

```
MOV R0, #0
PUSH {R0}
PUSH {R0}
PUSH {R0}
```

Exemple

Pile de 16 octets



POP: dépile

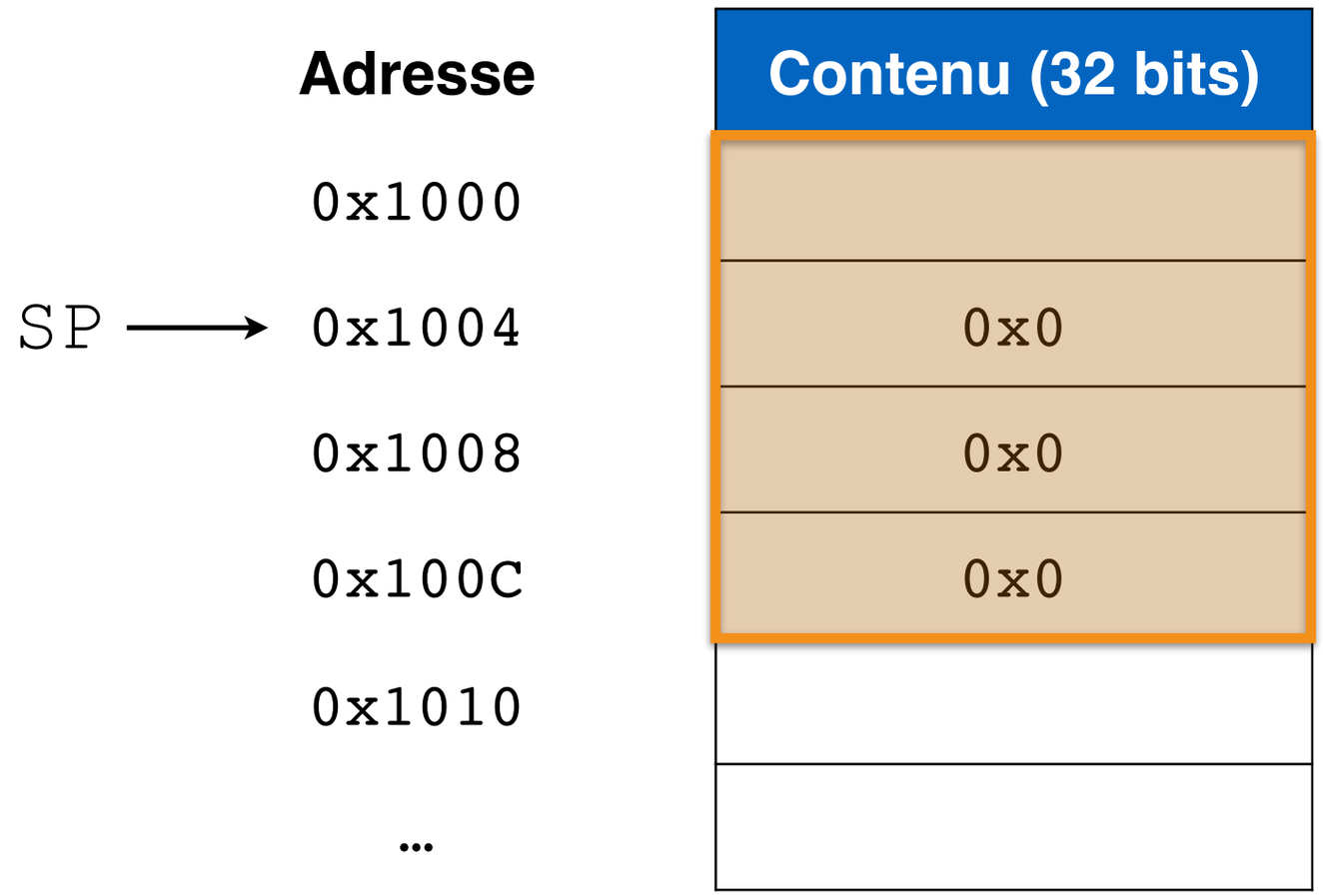
```
POP {Rd}     ; 1) Place le contenu à l'adresse indiquée
                  par SP
             ; 2) SP = SP + 4
```

```
POP {R0}
POP {R1}
POP {R2}
```

R0	
R1	
R2	

Exemple

Pile de 16 octets



POP: dépile

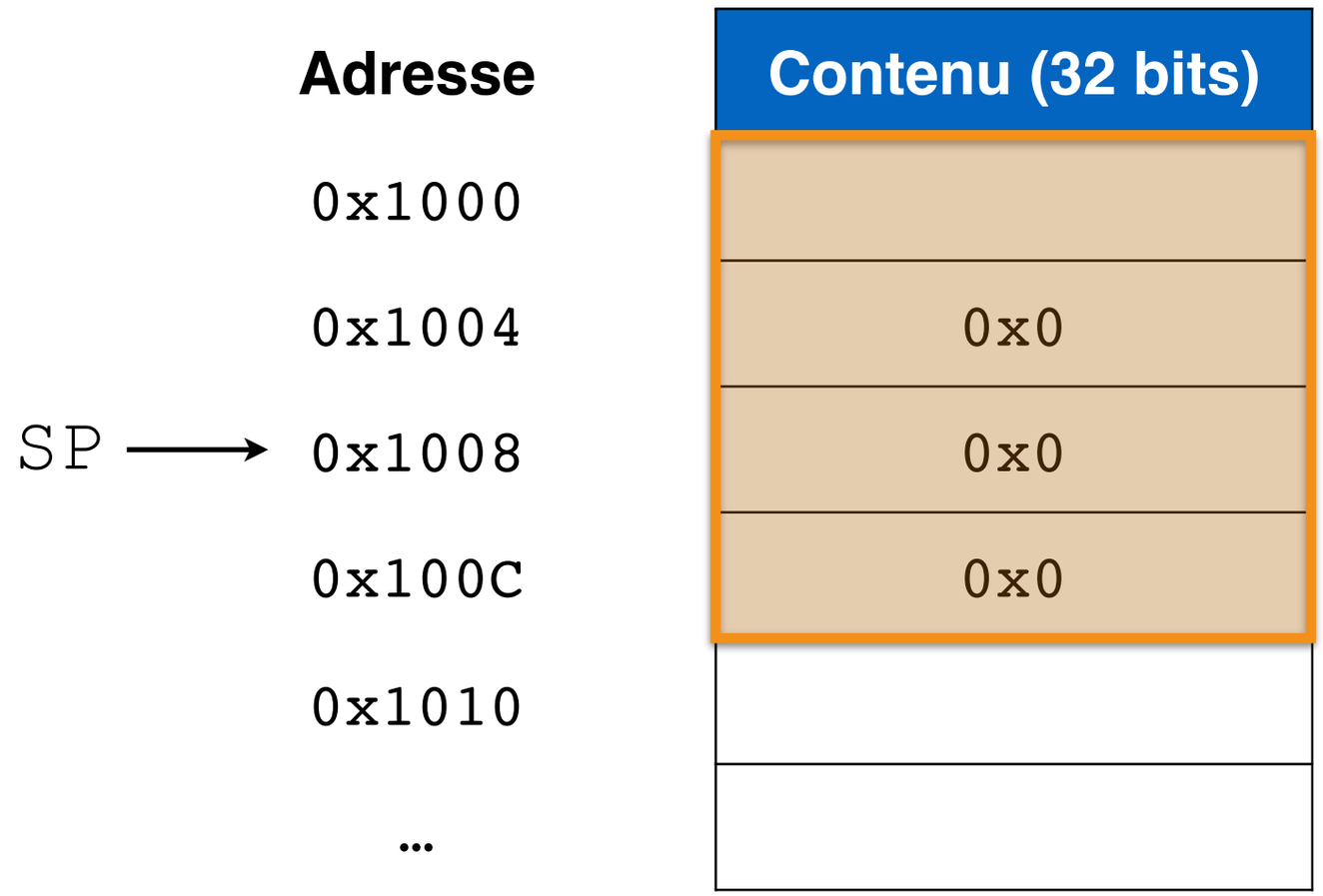
```
POP {Rd}     ; 1) Place le contenu à l'adresse indiquée
                  par SP
             ; 2) SP = SP + 4
```

```
POP {R0}
POP {R1}
POP {R2}
```

R0	0x0
R1	
R2	

Exemple

Pile de 16 octets



POP: dépile

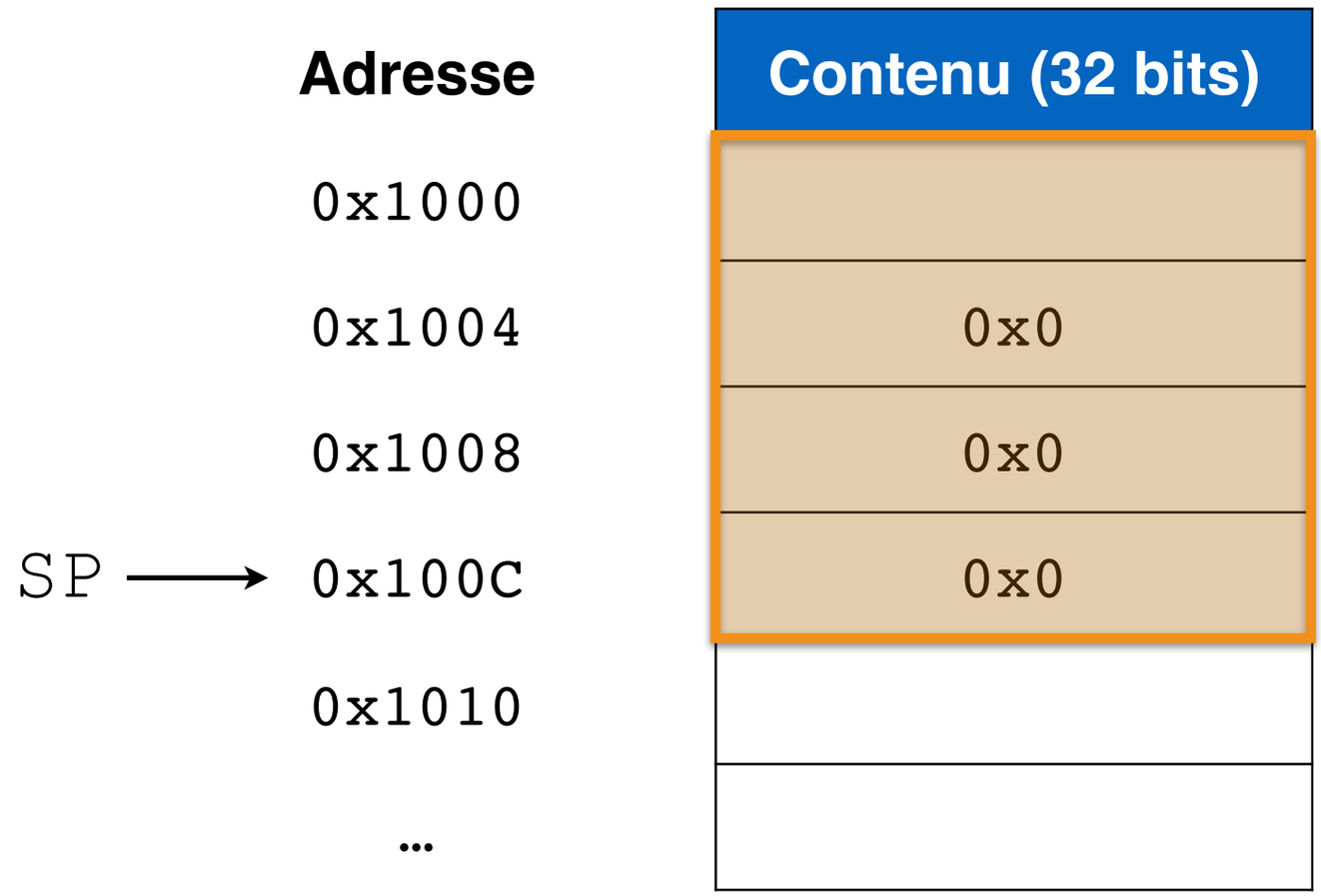
```
POP {Rd}     ; 1) Place le contenu à l'adresse indiquée
                  par SP
             ; 2) SP = SP + 4
```

```
POP {R0}
POP {R1}
POP {R2}
```

R0	0x0
R1	0x0
R2	

Exemple

Pile de 16 octets



POP: dépile

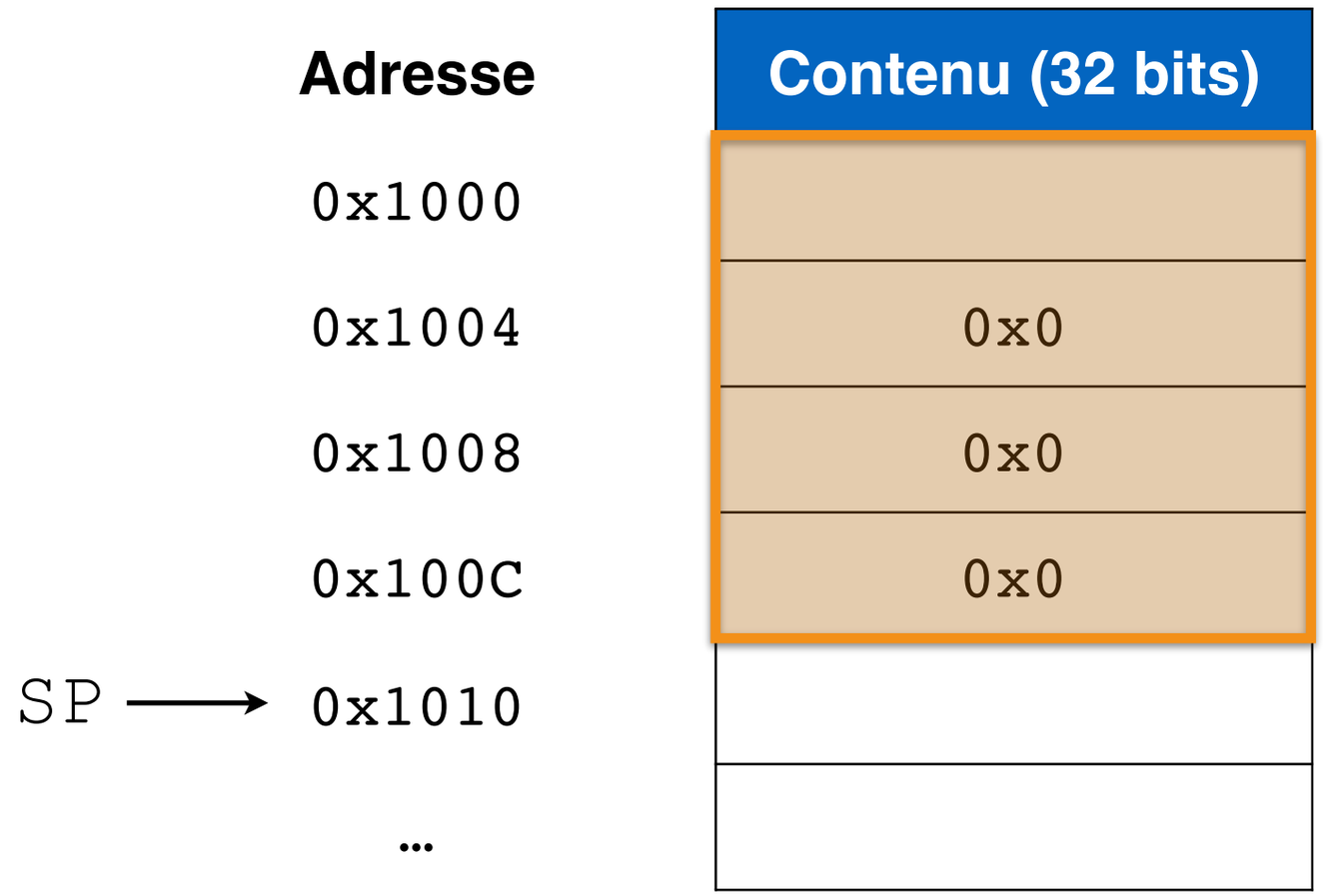
```
POP {Rd}     ; 1) Place le contenu à l'adresse indiquée
                  par SP
             ; 2) SP = SP + 4
```

```
POP {R0}
POP {R1}
POP {R2}
```

R0	0x0
R1	0x0
R2	0x0

Exemple

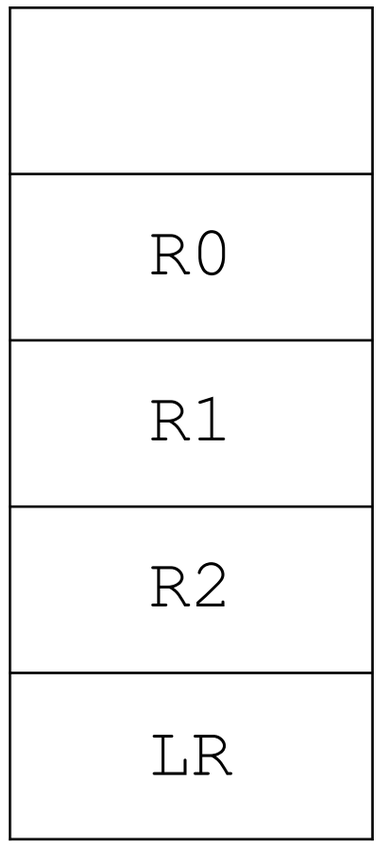
Pile de 16 octets



PUSH / POP

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

```
PUSH {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

- Lor
plu

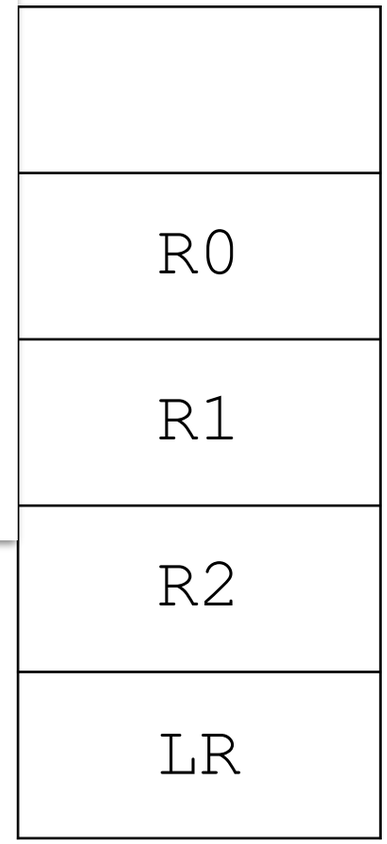
```
PUSH {R3, R1, R0, R2}
```

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

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

- Lor
plus petit au plus grand numéro de registre

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



Préparation d'une pile

- Pour préparer la pile:
  - on alloue un espace mémoire (ALLOC)
  - on charge l'adresse de la pile dans SP
  - on incrémente SP pour aller « à la fin » de la pile (car la pile est descendante)

```
SECTION CODE

main
LDR SP, =maPile    ; Charger l'adresse dans SP
ADD SP, SP, #64    ; Il faut commencer à la fin car
                   ; la pile descend

SECTION DATA

maPile ALLOC32 16 ; Pile de 64 octets (16*4 = 64)
```

Appels de fonctions imbriquées

```
main
BL MaFonction
MOV R0, #0


MaFonction
; Tâche dans la fonct
BL MaFonction2
...
BX LR

MaFonction2
; Tâche dans la fonct
...
BX LR
```

Appel de fonction imbriquée

`MaFonction` appelle `MaFonction2`!

`MaFonction2` est une autre fonction

Appels de fonctions imbriquées

```
main
BL MaFonction
MOV R0, #0


MaFonction
PUSH {LR}
; Tâche dans la fonct
BL MaFonction2
...
POP {LR}
BX LR

MaFonction2
; Tâche dans la fonction 2
...
BX LR
```

Comme `MaFonction` modifie `LR` (dans l'instruction `BL`), on sauvegarde `LR` sur la pile.

On restaure `LR` à sa valeur originale juste avant le `BX LR`.

Passage de paramètres

- Les paramètres sont les « entrées » de la fonction
  - Exemple, pour une fonction qui calcule la puissance ($base^{exposant}$), on doit lui donner la base et l'exposant.

- Paramètres:
  - Lorsqu'il y en a 4 ou moins, on peut utiliser les registres R0 à R3.
  - S'il y en a plus (pas dans le cadre du cours), on peut utiliser la pile.

Valeur de retour

- La valeur de retour est la « sortie » de la fonction (le résultat)
  - On se sert de R0 lorsqu'il y en a 1.
  - S'il y en a plus (pas dans le cadre du cours), on peut utiliser la pile.

1 paramètre et 1 valeur de 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
```

1 paramètre et 1 valeur de retour

```
main
MOV R1, #2
MOV R0, #8
BL  MaFonction
MOV R2, R0
CMP R1, R2
```

Nous voulons comparer le résultat de `MaFonction(8)` avec 2.

1 paramètre et 1 valeur de retour

```
MaFonction
; paramètre: R0
; valeur de retour: R0

PUSH {LR}

ADD R1, R0, R0
ADD R1, R1, R0

MOV R0, R1
POP {LR}
BX LR
```

La fonction `MaFonction`
prend un paramètre en entrée (dans `R0`)
retourne une valeur en sortie (dans `R0`)

La fonction utilise `R1` pour stocker un résultat temporaire.

Le résultat final est placé dans `R0` pour la valeur de retour.

1 paramètre et 1 valeur de retour

```
main
MOV R1, #2
MOV R0, #8
BL  MaFonction
MOV R2, R0
CMP R1, R2

MaFonction
; paramètre: R0
; valeur de retour: R0

PUSH {LR}

ADD R1, R0, R0
ADD R1, R1, R0

MOV R0, R1
POP {LR}
BX LR
```

Quel est le problème?

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`

1 paramètre et 1 valeur de retour

```
main
MOV R1, #2
MOV R0, #8
BL  MaFonction
MOV R2, R0
CMP R1, R2

MaFonction
; paramètre: R0
; valeur de retour: R0

PUSH {LR}

ADD R1, R0, R0
ADD R1, R1, R0

MOV R0, R1
POP {LR}
BX LR
```

1 paramètre et 1 valeur de retour

```
main
MOV R1, #2
MOV R0, #8
BL  MaFonction
MOV R2, R0
CMP R1, R2

MaFonction
; paramètre: R0
; valeur de retour: R0

PUSH {R1,LR}

ADD R1, R0, R0
ADD R1, R1, R0

MOV R0, R1
POP {R1,LR}
BX LR
```

La pile sauvegarde tous les registres modifiés par la fonction.

La pile restaure tous les registres modifiés par la fonction.

1 paramètre et 1 valeur de retour

```
main
MOV R1, #2
PUSH {R0}
MOV R0, #8
BL  MaFonction
MOV R2, R0
POP {R0}
CMP R1, R2

MaFonction
; paramètre: R0
; valeur de retour: R0
PUSH {R1,LR}

ADD R1, R0, R0
ADD R1, R1, R0

MOV R0, R1
POP {R1,LR}
BX LR
```

La pile sauvegarde R0,
qui est utilisé par la fonction.

La pile restaure R0,
qui est utilisé par la fonction.

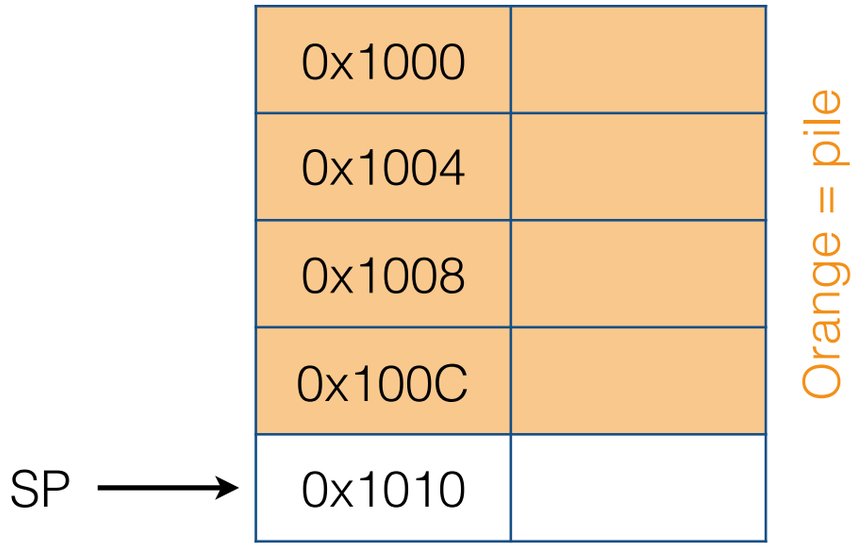
Exemple: illustration

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	0xFF
R1	0xFF
R2	0x00
SP (R13)	0x1010
LR (R14)	0x00

Mémoire

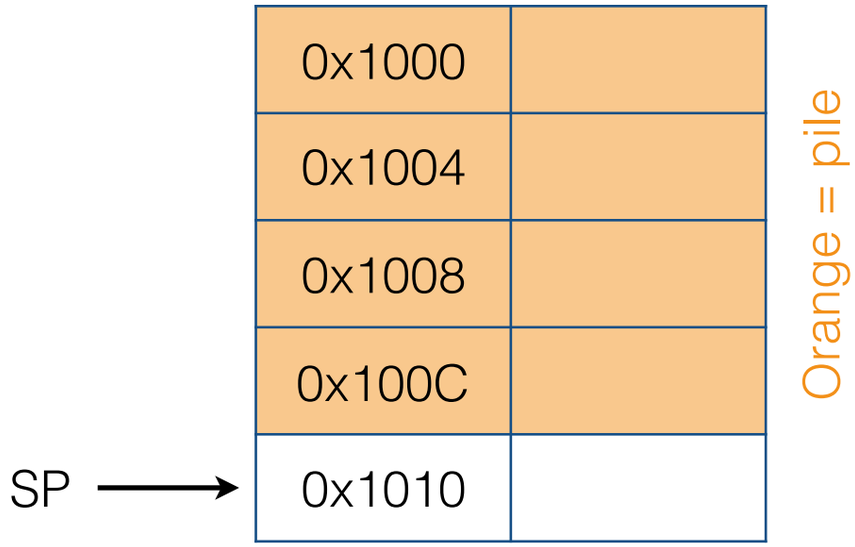



```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	0xFF
R1	**2**
R2	0x00
SP (R13)	0x1010
LR (R14)	0x00

Mémoire




```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	0xFF
R1	2
R2	0x00
SP (R13)	**0x100C**
LR (R14)	0x00

Mémoire

SP →

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

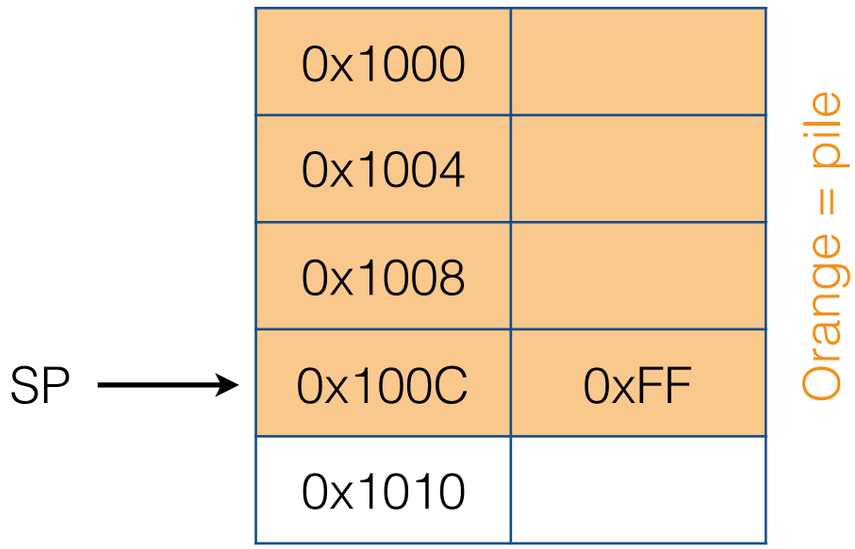
Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	**8**
R1	2
R2	0x00
SP (R13)	0x100C
LR (R14)	0x00

Mémoire

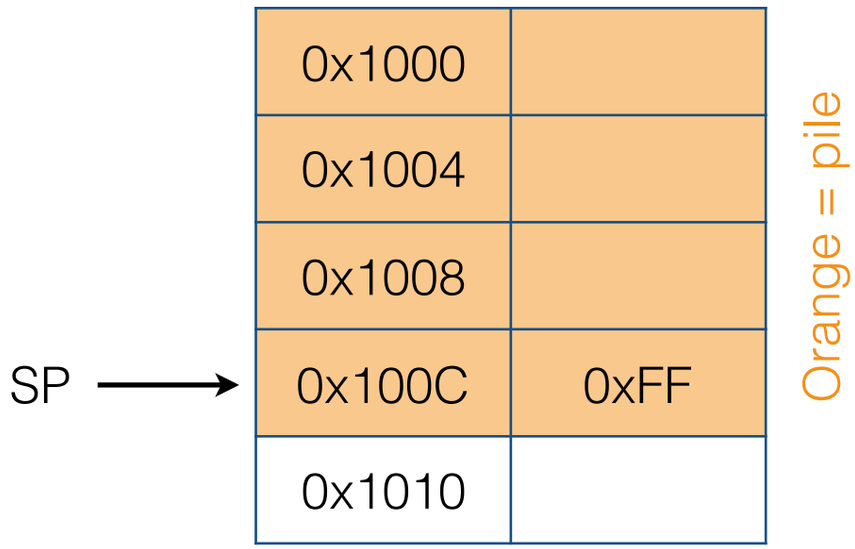



```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	8
R1	2
R2	0x00
SP (R13)	0x100C
LR (R14)	**0xA0**

Mémoire




```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	8
R1	2
R2	0x00
SP (R13)	**0x1004**
LR (R14)	0xA0

Mémoire

SP →

0x1000	
0x1004	0x2
0x1008	0xA0
0x100C	0xFF
0x1010	

Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	8
R1	**16**
R2	0x00
SP (R13)	0x1004
LR (R14)	0xA0

Mémoire

SP →

0x1000	
0x1004	0x2
0x1008	0xA0
0x100C	0xFF
0x1010	

Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	8
R1	**24**
R2	0x00
SP (R13)	0x1004
LR (R14)	0xA0

Mémoire

SP →

0x1000	
0x1004	0x2
0x1008	0xA0
0x100C	0xFF
0x1010	

Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	**24**
R1	24
R2	0x00
SP (R13)	0x1004
LR (R14)	0xA0

Mémoire

SP →

0x1000	
0x1004	0x2
0x1008	0xA0
0x100C	0xFF
0x1010	

Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	24
R1	**2**
R2	0x00
SP (R13)	**0x100C**
LR (R14)	**0xA0**

Mémoire

SP →

0x1000	
0x1004	0x2
0x1008	0xA0
0x100C	0xFF
0x1010	

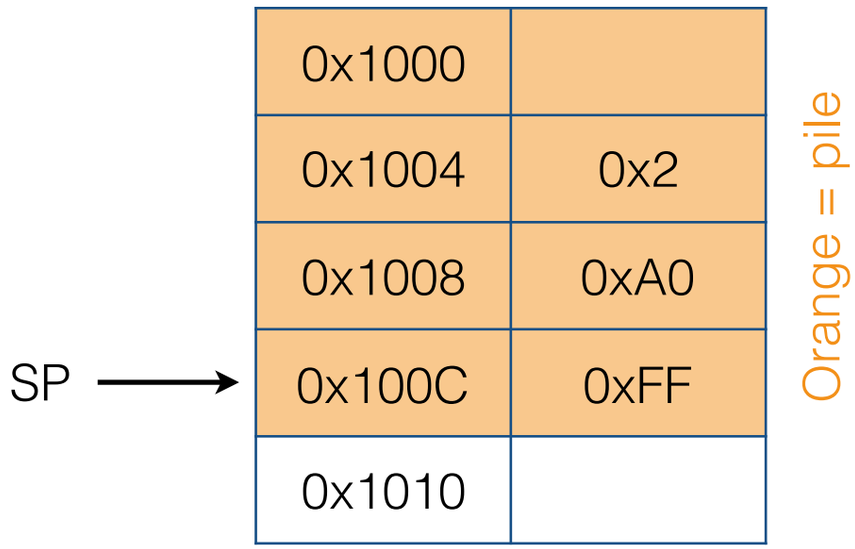
Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	24
R1	2
R2	0x00
SP (R13)	0x100C
LR (R14)	0xA0

Mémoire




```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	24
R1	2
R2	24
SP (R13)	0x100C
LR (R14)	0xA0

Mémoire

SP →

0x1000	
0x1004	0x2
0x1008	0xA0
0x100C	0xFF
0x1010	

Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

Registres

R0	0xFF
R1	2
R2	24
SP (R13)	**0x1010**
LR (R14)	0xA0

Mémoire

0x1000	
0x1004	0x2
0x1008	0xA0
0x100C	0xFF
0x1010	

SP →

Orange = pile

```
        main
0x90    MOV R1, #2
0x94    PUSH {R0}                   ; Sauvegarde R0
0x98    MOV R0, #8
0x9C    BL Triple                   ; Appel de fonction
0xA0    MOV R2, R0
0xA4    POP {R0}                    ; Restaure R0
0xA8    CMP R1, R2

        Triple
        ; paramètre: R0
        ; valeur de retour: R0
0xC0    PUSH {R1,LR}                ; Sauvegarde R1 et LR

0xC4    ADD R1, R0, R0              ; Calcule 3*R0 en utilisant des additions.
0xC8    ADD R1, R1, R0

0xCC    MOV R0, R1                  ; Place résultat dans R0 (valeur de retour)
0xD0    POP {R1,LR}                 ; Restaure R1 et LR
0xD4    BX LR                       ; Fonction terminée
```

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…