

Systemes à Microprocesseurs

Cycle Ingénieur Troisième Année

Sébastien Bilavarn



Plan

- Ch1 – Représentation de l'information
- Ch2 – ARM Instruction Set Architecture
- Ch3 – Accès aux données
- **Ch4 – Programmation structurée**
- Ch5 – Cycle d'exécution
- Ch6 – Codage binaire
- Ch7 – Microcontrôleur ARM Cortex-M



Programmation structurée en assembleur ARM

- Appels de sous-programmes
 - Structure de pile
- Mise en place d'une pile
- Passage de paramètres



Notions

- Sous-programme: terme générique désignant un sous-ensemble d'un programme
 - Procédure: un sous-programme qui ne renvoie pas de résultat.
 - Ex: `printf ("Hello world\n");`
 - Fonctions: un sous-programme effectuant un traitement sur des données et qui renvoie un résultat.
 - Ex: `c = max (a, b);`
- Appel et retour de sous-programmes
 - Un sous-programme doit mémoriser l'adresse du code appelant pour poursuivre l'exécution à l'adresse de retour correspondante
- Mécanismes d'échanges de données
 - Passage de paramètre par valeur: le code appelé dispose d'une copie de la valeur
 - Passage de paramètre par référence: le code appelé dispose de l'adresse du paramètre. Il peut modifier sa valeur.
 - Valeur de retour d'une fonction: donnée fournie par le code appelé au code appelant

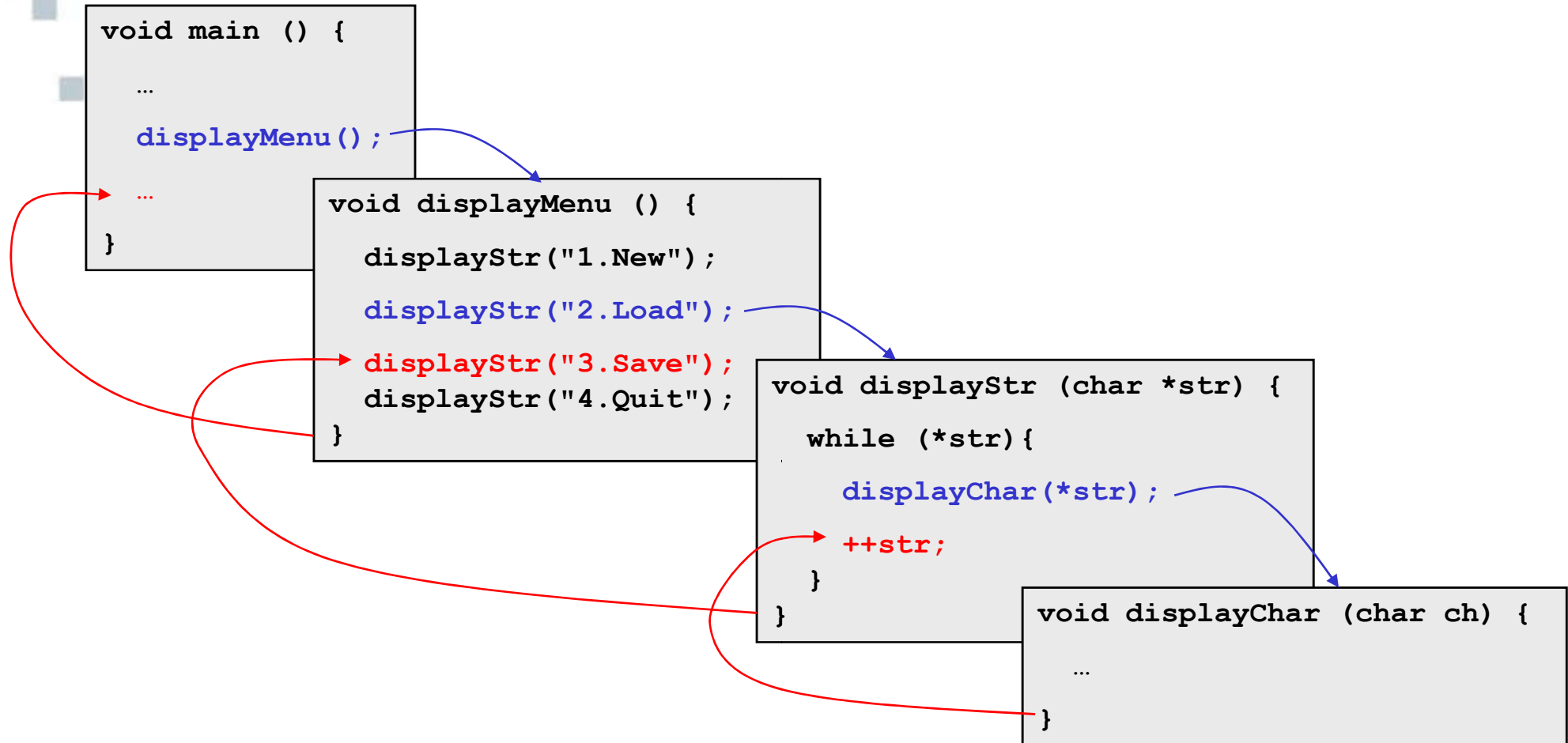


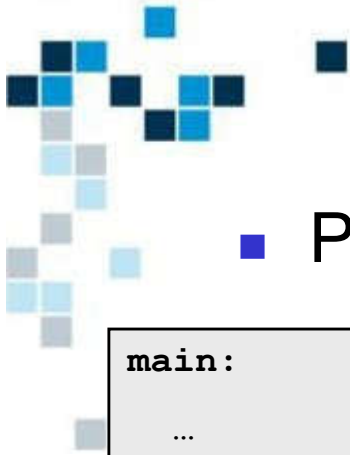
Appel de sous-programmes

- Appel et retour de sous-programmes
 - Pour appeler un sous programme:
 - Branch and Link: BL *label*
 - L'exécution se poursuit à l'instruction correspondant au label
 - L'adresse de retour est conservée dans le registre LR (Link Register, r14)
 - Pour revenir d'un sous-programme:
 - MOV PC, LR (équivalent à MOV r15, r14)
 - L'adresse de retour est récupérée dans LR
 - L'exécution se poursuit à l'instruction qui suit l'instruction d'appel

Enchaînement d'appels

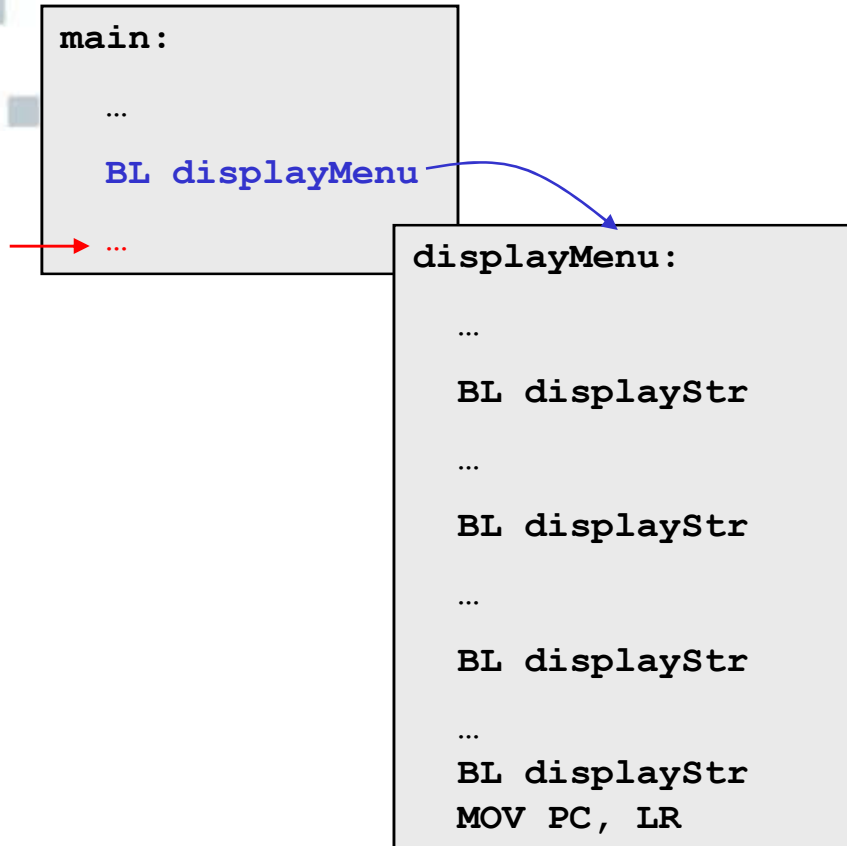
■ Principe





Enchaînement d'appels

■ Problème de la traduction en assembleur (1)



LR

@ retour dans main

PC

@ displayMenu

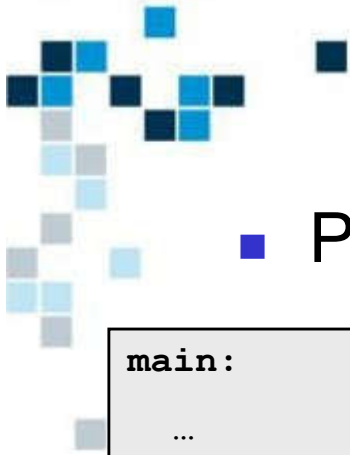
Traduction en assembleur:

Branchement temporaire (BL displayMenu)

Dans la fonction main, à l'exécution de l'instruction BL displayMenu :

-l'adresse de retour (adresse de l'instruction BL displayMenu + 4) est sauvegardée dans le registre LR (r14)

-puis on saute à l'adresse de la première instruction de displayMenu (PC ← @displayMenu)



Enchaînement d'appels

■ Problème de la traduction en assembleur (2)

```
main:
...
BL displayMenu
...
```

```
displayMenu:
...
BL displayStr
...
BL displayStr
...
BL displayStr
MOV PC, LR
```

LR

@ retour dans displayMenu

PC

@ displayStr

Dans la fonction displayMenu, à l'exécution de l'instruction BL displayStr :

-l'adresse de retour (adresse de l'instruction BL displayMenu + 4) est sauvegardée dans le registre LR (r14)

-puis on saute à l'adresse de la première instruction de displayStr (PC ← @displayStr)

```
displayStr:
...
BL displayChar
...
MOV PC, LR
```

On écrase la valeur précédente de LR (adresse de retour dans main)!!

Enchaînement d'appels

■ Problème de la traduction en assembleur (3)

```
main:
...
BL displayMenu
...
```

```
displayMenu:
...
BL displayStr
...
BL displayStr
...
BL displayStr
...
BL displayStr
MOV PC, LR
```

LR

@ retour dans displayStr

PC

@ displayChar

Dans la fonction displayStr, à l'exécution de l'instruction BL displayChar :

-l'adresse de retour (adresse de l'instruction BL displayStr + 4) est sauvegardée dans le registre LR (r14)

-puis on saute à l'adresse de la première instruction de displayChar (PC ← @displayChar)

```
displayStr:
...
BL displayChar
...
MOV PC, LR
```

On écrase la valeur précédente de LR (adresse de retour dans displayMenu)!!

```
displayChar:
...
MOV PC, LR
```

Enchaînement d'appels

■ Problème de la traduction en assembleur (4)

```
main:
...
BL displayMenu
...
```

```
displayMenu:
...
BL displayStr
...
BL displayStr
...
BL displayStr
MOV PC, LR
```

```
displayStr:
...
BL displayChar
...
MOV PC, LR
```

```
displayChar:
...
MOV PC, LR
```

LR

@ retour dans displayStr

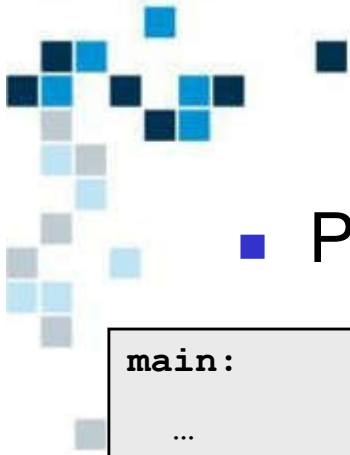
PC

@ retour dans displayStr

A la fin de l'exécution de displayChar :

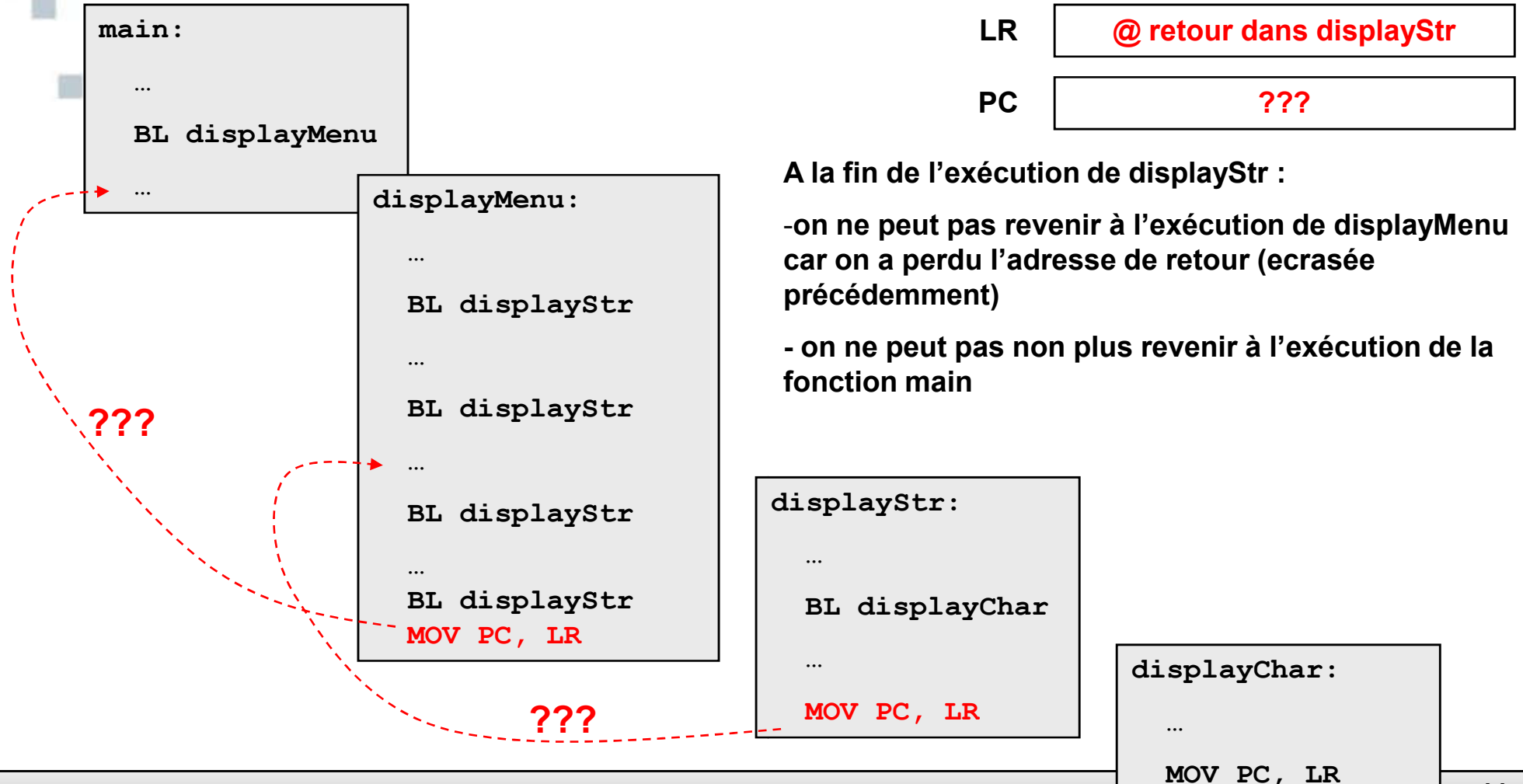
-on récupère l'adresse de retour dans displayStr, que l'on place dans le registre PC (par l'instruction MOV PC, LR)

-L'exécution reprend à l'adresse de retour dans displayStr



Enchaînement d'appels

■ Problème de la traduction en assembleur (5)





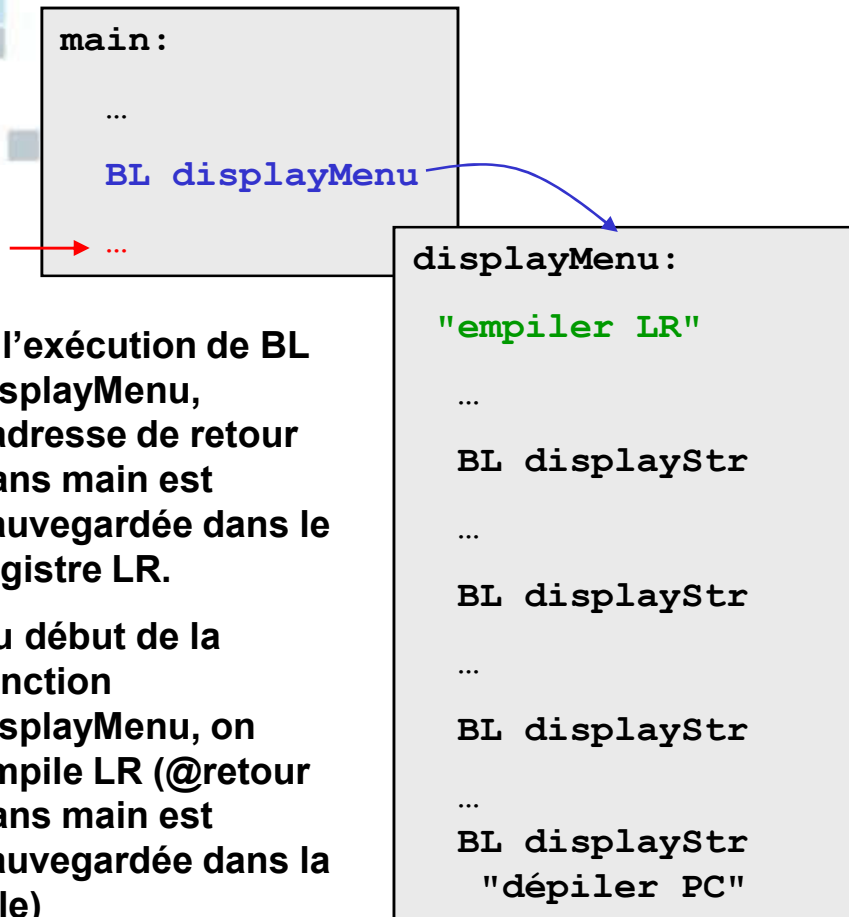
Enchaînement d'appels

- Si un sous-programme est "terminal" (s'il n'appelle pas d'autre sous-programme), le registre LR suffit pour sauvegarder et restaurer l'adresse de retour
 - Ex: fonction *displayChar*
- Si un sous-programme en appelle un autre, la valeur de LR doit être sauvegardée avant d'effectuer le saut à l'instruction BL
- Solution: utilisation d'une **structure de pile**

La procédure de sauvegarde de l'adresse de retour ne peut fonctionner telle quelle en cas d'enchaînement d'appels à plusieurs sous programmes (cas le plus fréquent)

Utilisation d'une pile

■ Principe (1)



LR

@ retour dans main

PC

@ displayMenu

Pile

@ retour dans main

Utilisation d'une pile

■ Principe (2)

```
main:
```

```
...
```

```
BL displayMenu
```

```
...
```

À l'exécution de BL displayStr, l'adresse de retour dans displayMenu est sauvegardée dans le registre LR.

Au début de la fonction displayStr, on empile LR (@retour dans displayMenu est sauvegardée en mémoire)

```
displayMenu:
```

```
"empiler LR"
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
"dépiler PC"
```

```
displayStr:
```

```
"empiler LR"
```

```
...
```

```
BL displayChar
```

```
...
```

```
"dépiler PC"
```

LR

@ retour dans displayMenu

PC

@ displayStr

Pile

@ retour dans displayMenu

@ retour dans main

Utilisation d'une pile

■ Principe (3)

```
main:
```

```
...
```

```
BL displayMenu
```

```
...
```

BL displayChar: on saute à l'exécution de displayChar

Programme terminal (pas de sous programme) donc il n'est pas obligatoire d'empiler l'adresse de retour (LR suffit dans ce cas)

```
displayMenu:
```

```
"empiler LR"
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
"dépiler PC"
```

```
displayStr:
```

```
"empiler LR"
```

```
...
```

```
BL displayChar
```

```
...
```

```
"dépiler PC"
```

LR

@ retour dans displayStr

PC

@ displayChar

Pile

@ retour dans displayMenu

@ retour dans main

```
displayChar:
```

```
...
```

```
MOV PC, LR
```

Utilisation d'une pile

■ Principe (4)

```
main:
```

```
...
```

```
BL displayMenu
```

```
...
```

À la fin de l'exécution de displayChar, on récupère l'adresse de retour dans displayStr (contenue dans LR)

```
displayMenu:
```

```
"empiler LR"
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
"dépiler PC"
```

```
displayStr:
```

```
"empiler LR"
```

```
...
```

```
BL displayChar
```

```
...
```

```
"dépiler PC"
```

LR

@ retour dans displayStr

PC

@ retour dans displayStr

Pile

@ retour dans displayMenu

@ retour dans main

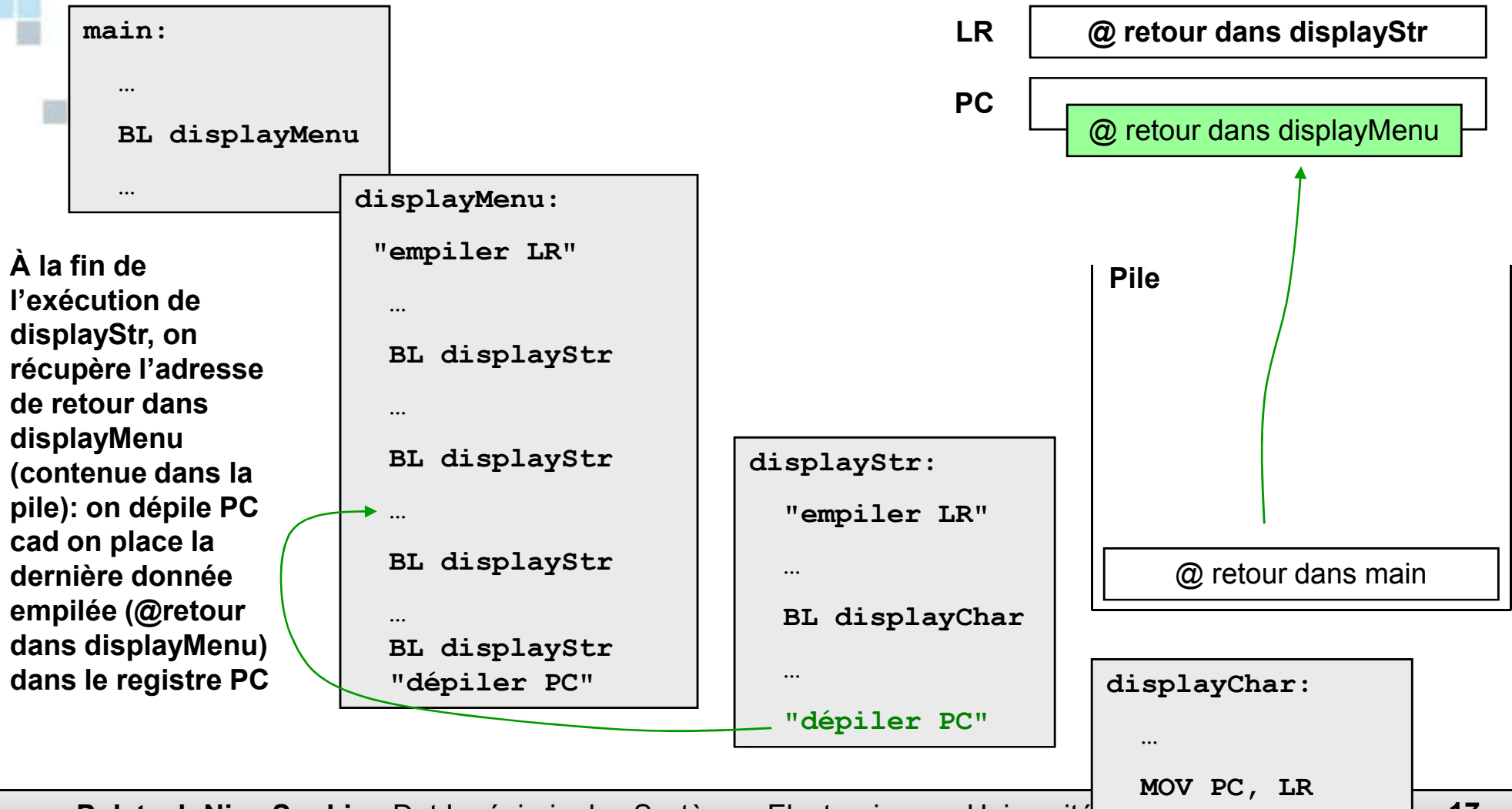
```
displayChar:
```

```
...
```

```
MOV PC, LR
```

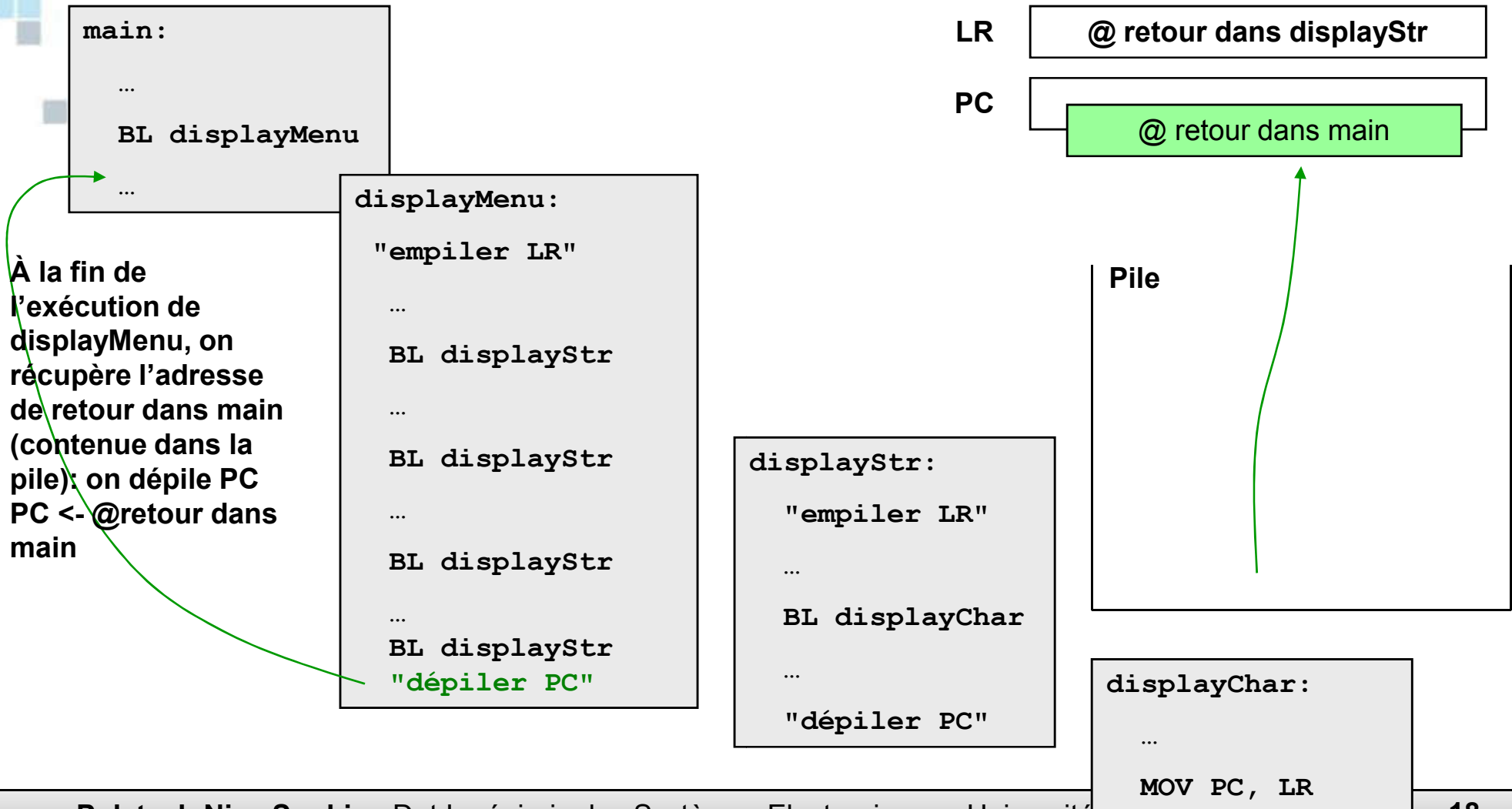

Utilisation d'une pile

■ Principe (5)



Utilisation d'une pile

■ Principe (6)





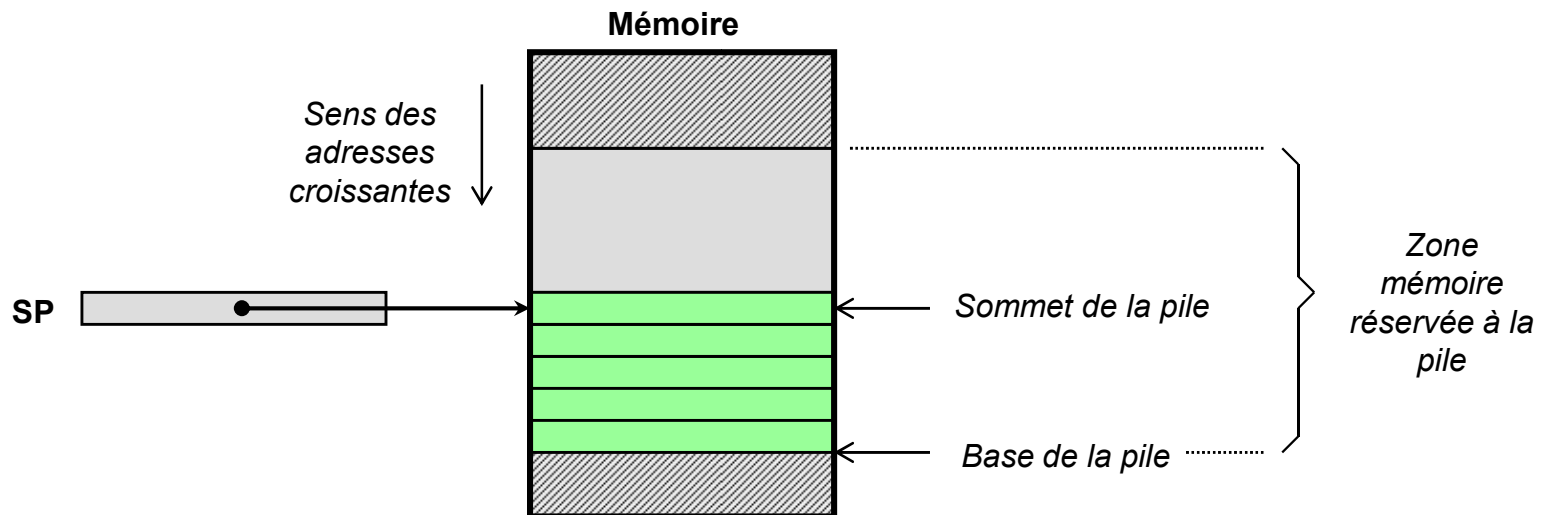
Mise en place d'une pile

- La pile réside dans un espace mémoire qui sera réservé spécifiquement à cet usage
- Les données sont rangées dans des cellules adjacentes au fur et à mesure qu'elles sont empilées
- Pour repérer le niveau de remplissage de la pile, on utilise un registre pointeur de pile (Stack Pointer SP)
 - C'est le registre r13 du processeur ARM qui remplit cette fonction



Mise en place d'une pile

- Modèles d'évolution de la pile: **Full Descending**



La pile se remplit dans le sens des adresses décroissantes

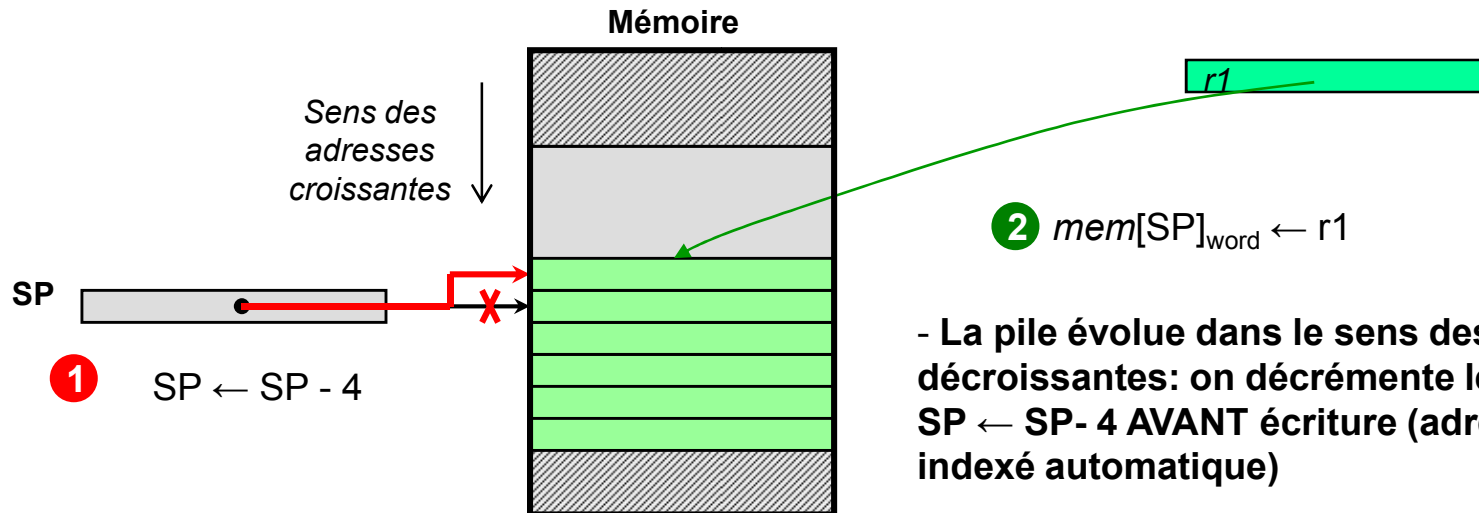
Le pointeur de pile SP repère la dernière donnée empilée

Mise en place d'une pile

- Modèles d'évolution de la pile: **Full Descending**
 - Pour **empiler**

Empiler:

- On ECRIT après la dernière la donnée pointée dans la pile (à l'adresse SP-4)



- La pile évolue dans le sens des adresses décroissantes: on décrémente les adresses $SP \leftarrow SP - 4$ AVANT écriture (adressage pré-indexé automatique)

- Ecrire la donnée

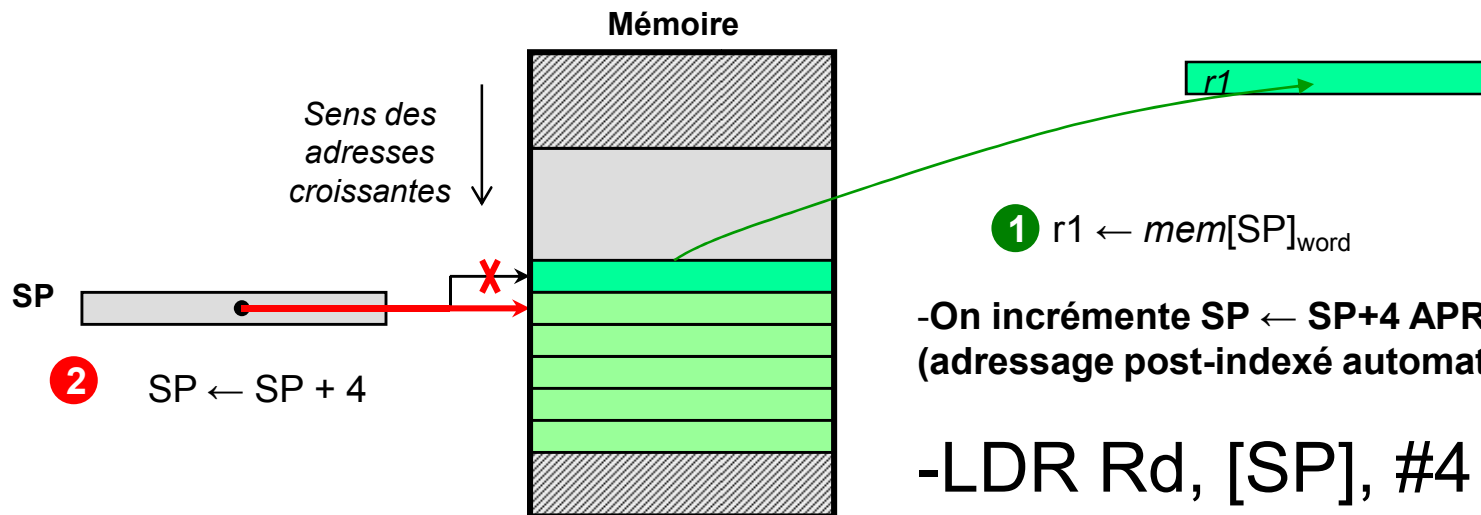
■ **STR Rd, [SP, #-4]!**

Mise en place d'une pile

- Modèles d'évolution de la pile: **Full Descending**
 - Pour **dépiler**

Dépiler:

- On LIT la dernière donnée dans la pile
- accès à l'adresse SP



Mise en place d'une pile

■ Exemple d'utilisation en mode Full Descending

```
main:
```

```
...
```

```
BL DisplayMenu
```

```
...
```

```
displayMenu:
```

```
STR LR, [SP, #-4]!
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
...
```

```
BL displayStr
```

```
LDR PC, [SP], #4
```

```
displayStr:
```

```
STR LR, [SP, #-4]!
```

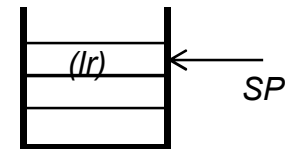
```
...
```

```
BL DisplayChar
```

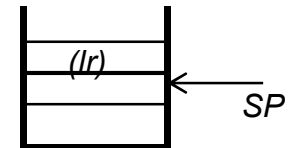
```
...
```

```
LDR PC, [SP], #4
```

Sens des
adresses
croissantes



Sens des
adresses
croissantes



```
displayChar:
```

```
...
```

```
MOV PC, LR
```

A chaque appel d'une sous-fonction non terminale:
on empile LR au début, on dépile PC à la fin.

A l'appel d'une sous fonction terminale, LR suffit
pour sauvegarder / récupérer l'adresse de retour.

Full descending: remplissage dans le sens des
adresses décroissantes:

Empiler: STR LR, [SP, #-4]!

Dépiler: LDR LR, [SP], #4



Appel de sous-programmes

- Précautions nécessaires lors d'appels à sous-programmes
 - Un sous-programme est vu comme une boîte noire, la partie visible concerne ses paramètres et sa valeur de retour

Mais

- Le programme principal et ses sous-programmes manipulent tous les mêmes registres
 - → un sous-programme peut modifier des valeurs de registres à l'insu du programme appelant
- Lors de l'appel à un sous-programme, l'appelant souhaite trouver les registres dans le même état qu'avant l'appel

Appel de sous-programmes

■ Exemple

```
void main () {  
    int i;  
    for (i=0; i<10; i++)  
        func1(i);  
}
```

```
main:    mov r1, #0  
loop1:   cmp r1, #10  
        bge eloop1  
        mov r2, r1  
        bl func1  
        add r1, r1, #1  
        b loop1  
eloop1:  b eloop1
```

Dans chacun des deux sous-programmes, le symbole i désigne une variable distincte

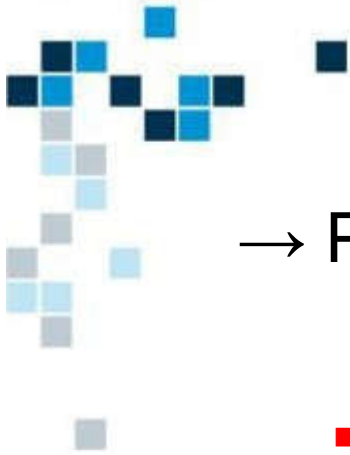
Ces deux sous-programmes utilisent le même registre r1 comme compteur de boucle

Au moment d'exécuter cette instruction, r1 a été modifié par le sous-programme func1 et a donc peu de chances d'être à la valeur souhaitée

```
void func1 (int x) {  
    int i;  
    for (i=0; i<2*x; i++)  
        ...;  
}
```

```
func1:   mov r1, #0  
loop2:   cmp r1, r2, lsl#1  
        bge eloop2  
        ...  
        ...  
        add r1, r1, #1  
        b loop2  
eloop2:  mov pc, lr
```

En plus de l'adresse de retour, il faut également sauvegarder certaines données contenues dans des registres !!



Appel de sous-programmes

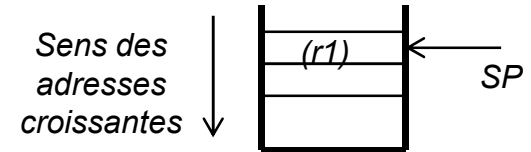
→ Précautions nécessaires:

- A l'entrée d'un sous programme, on sauvegarde les registres utilisés comme variables locales
- Avant de sortir, on restaure la valeur de ces registres
- La pile offre un moyen d'allouer dynamiquement de la mémoire pour effectuer cette sauvegarde



Appel de sous-programmes

■ Exemple en utilisant la pile



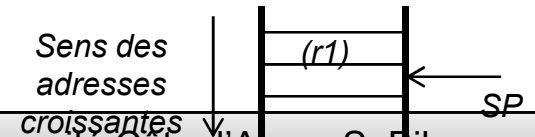
Sauvegarde de r1

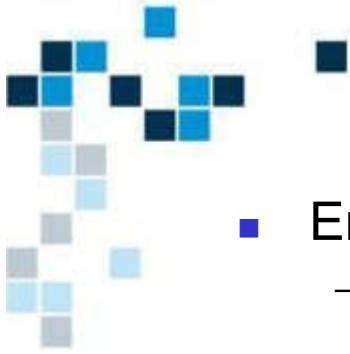
```
main:    mov r1, #0
loop1:   cmp r1, #10
        bge eloop1
        mov r2, r1
        bl func1
        add r1, r1, #1
        b loop1
eloop1:  b eloop1
```

Au moment d'exécuter cette instruction, r1 a été utilisé par le sous-programme func1, mais sa valeur a été sauvegardée et rétablie

```
func1:   STR r1, [SP, #-4]!
        mov r1, #0
loop2:   cmp r1, r2, lsl#1
        bge eloop2
        ...
        add r1, r1, #1
        b loop2
eloop2:  LDR r1, [SP], #4
        mov pc, lr
```

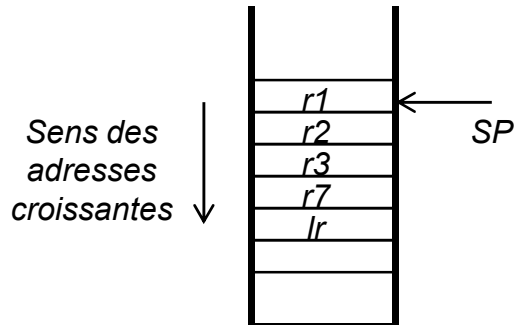
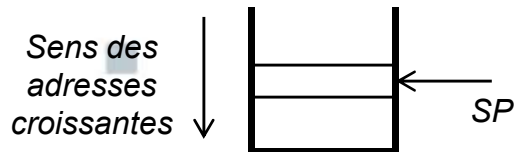
Restauration de r1





Appel de sous-programmes

- Empiler/dépiler une liste de registres (1)
 - Il faut sauvegarder `lr`, ainsi que les registres modifiés par le sous programme



```
foo:    str lr, [sp, #-4]!  
        str r7, [sp, #-4]!  
        str r3, [sp, #-4]!  
        str r2, [sp, #-4]!  
        str r1, [sp, #-4]!
```

...

...

```
Fin:    ldr r1, [sp], #4  
        ldr r2, [sp], #4  
        ldr r3, [sp], #4  
        ldr r7, [sp], #4  
        ldr pc, [sp], #4
```

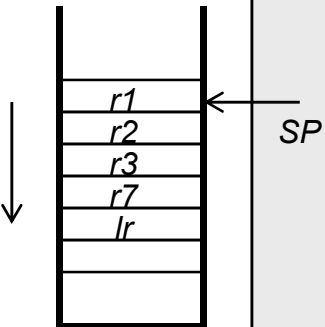
Empiler LR en premier

Dépiler PC en dernier



Appel de sous-programmes

- Empiler/dépiler plusieurs registres à la fois (2)
 - Utilisation des instructions de transfert multiples pour accéder à la pile



```
foo:    str lr, [sp, #-4]!  
        str r7, [sp, #-4]!  
        str r3, [sp, #-4]!  
        str r2, [sp, #-4]!  
        str r1, [sp, #-4]!  
  
        ...  
  
        ldr r1, [sp], #4  
        ldr r2, [sp], #4  
        ldr r3, [sp], #4  
        ldr r7, [sp], #4  
        ldr pc, [sp], #4
```

```
foo:    STMDB sp!, {r1-r3, r7, lr}  
  
        ...  
  
        ...  
  
        ...  
  
        LDMIA sp!, {r1-r3, r7, pc}
```



Appel de sous-programmes

- Empiler/dépiler plusieurs registres à la fois (3)
 - Utilisation des instructions de transfert multiples pour accéder à la pile

Mode	Pour empiler	Pour dépiler
Full Ascending (FA)	STMIB/STMFA	LDMDA/LDMFA
Empty Ascending (EA)	STMIA/STMEA	LDMDB/LDMEA
Full Descending (FD)	STMDB/STMFD	LDMIA/LDMFD
Empty Descending (ED)	STMDA/STMED	LDMIB/LDMED

Le mode utilisé par défaut sur ARM est le Full Descending:

Pour empiler un registre:	<code>STR Rd, [SP, #-4]!</code>
Pour empiler une liste de registres :	<code>STMFD SP!, {reglist}</code>
Pour dépiler un registre :	<code>LDR Rd, [SP], #4</code>
Pour dépiler une liste de registres :	<code>LDMFD SP!, {reglist}</code>



Récapitulatif sur l'utilisation de la pile (1)

- Changement de contexte
 - Au début d'un sous programme: sauvegarde de l'adresse de retour et des registres.
 - A la fin d'un sous programme: restauration des registres et de l'adresse de retour.

- Autre utilisation de la pile
 - Passage de paramètres



Passage de paramètres

- L'utilisation de variables globales pour passer des informations à un sous-programme est fortement déconseillée
- Les paramètres sont considérés comme des variables locales de chaque sous-programme
- Ils peuvent être transmis de deux façons:
 - Passage par valeur
 - Passage par référence (ou par adresse)
- Mise en place
 - Passage par registres
 - Passage par la pile



Passage de paramètres

- Mise en place: passage par registres
 - Facile à mettre en place
 - Rapide à l'exécution (pas d'accès mémoire pour lire ou écrire les données)

Mais

- les registres sont en nombre limité, souvent utilisés pour d'autres usages
- mêmes inconvénients que les variables globales (risque de modifications non contrôlées)



Passage de paramètres

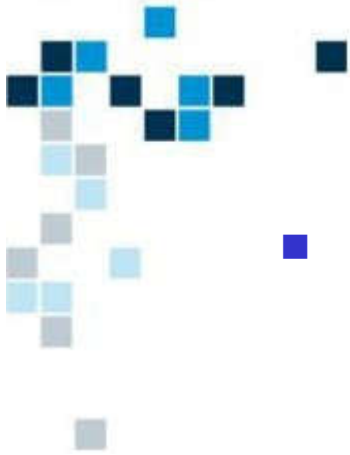
- Mise en place: passage par registres
- Exemple: calcul de PGCD
 - Le sous-programme prend deux paramètres entiers passés dans r1 et r2
 - Il renvoie le résultat dans r0

```
main:    mov r1, #24
         mov r2, #18
         bl  pgcd
```

Les paramètres 24 et 18 sont passés par le programme appelant vers le programme appelé via les registres r1 et r2.

// Ces paramètres sont utilisés et modifiés par la fonction pgcd!! Les valeurs 24 et 18 sont perdues à la fin de l'appel à pgcd.

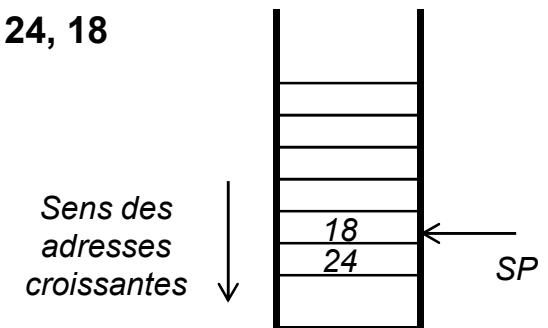
```
pgcd:    mov r0, #0
         cmp r1, #0
         beq epgcd
         cmp r2, #0
         beq epgcd
loop1:   cmp r1, r2
         moveq r0, r1
         beq epgcd
         subgt r1, r1, r2
         sublt r2, r2, r1
         b loop1
epgcd:   mov pc, lr
```



Passage de paramètres

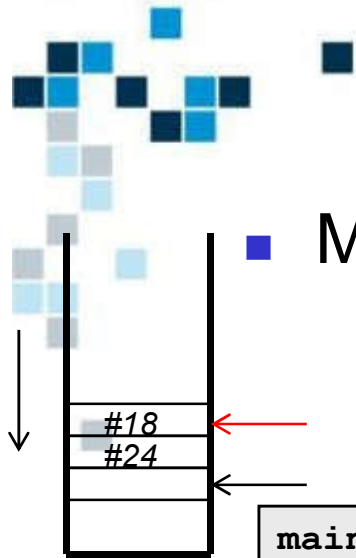
- Mise en place: passage par la pile
 - Offre un mécanisme simple d'allocation et de libération dynamique d'espace mémoire: il suffit de déplacer le pointeur de pile
 - Le programme appelant empile les paramètres
 - Le sous-programme appelé les lit dans la pile et libère l'espace occupé au moment de sortir

Exemple: les paramètres à passer sont: 24, 18



Passage de paramètres

- Mise en place: passage par la pile



```
main:  mov r0, #24
        str r0, [sp, #-4]!
        mov r0, #18
        str r0, [sp, #-4]!
        bl pgcd
        add sp, sp, #8
```

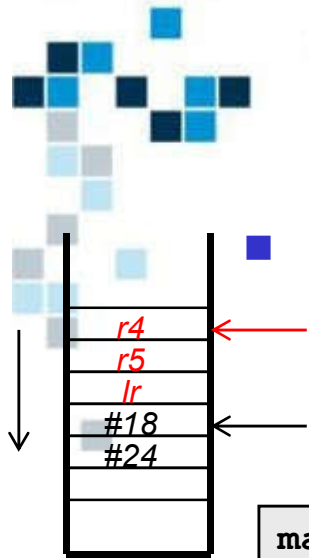
Avant l'appel à la fonction (bl pgcd), on empile les paramètres.

Après l'appel à la fonction (bl pgcd), on libère les paramètres.

```
pgcd:
        ldr r4, [sp, #4]
        ldr r5, [sp]
        mov r0, #0
        cmp r4, #0
        beq epgcd
        cmp r5, #0
        beq epgcd
loop1:  cmp r4, r5
        moveq r0, r4
        beq epgcd
        subgt r4, r4, r5
        sublt r5, r5, r4
        b loop1
epgcd:  mov pc, lr
```

Passage de paramètres

■ Mise en place: passage par la pile



```
main:  mov r0, #24
        str r0, [sp, #-4]!
        mov r0, #18
        str r0, [sp, #-4]!
        bl pgcd
        add sp, sp, #8
```

Même exemple avec prise en compte de la sauvegarde de l'adresse de retour et des registres modifiés.

/!\ Ne pas sauvegarder le registre utilisé pour le résultat (ici r0)

```
pgcd:   stmfd sp!, {r4, r5, lr}

        ldr r4, [sp, #16]
        ldr r5, [sp, #12]
        mov r0, #0
        cmp r4, #0
        beq epgcd
        cmp r5, #0
        beq epgcd

loop1:  cmp r4, r5
        moveq r0, r4
        beq epgcd
        subgt r4, r4, r5
        sublt r5, r5, r4
        b loop1

epgcd:  ldmfd sp!, {r4, r5, pc}
```

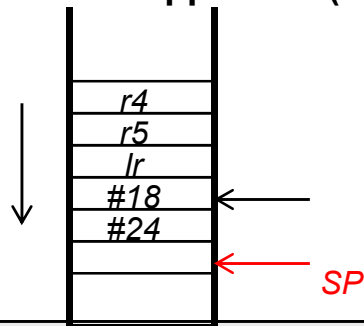


Passage de paramètres

- Mise en place: passage par la pile

```
main:    mov r0, #24
         str r0, [sp, #-4]!
         mov r0, #18
         str r0, [sp, #-4]!
         bl pgcd
```

Même chose en libérant l'espace des paramètres dans la fonction pgcd, donc juste avant de retourner à la fonction appelante (mov pc, lr)



```
pgcd:    stmfd sp!, {r4, r5, lr}
         ldr r4, [sp, #16]
         ldr r5, [sp, #12]
         mov r0, #0
         cmp r4, #0
         beq epgcd
         cmp r5, #0
         beq epgcd
loop1:   cmp r4, r5
         moveq r0, r4
         beq epgcd
         subgt r4, r4, r5
         sublt r5, r5, r4
         b loop1
epgcd:   ldmfd sp!, {r4, r5, lr}
         add sp, sp, #8
         mov pc, lr
```



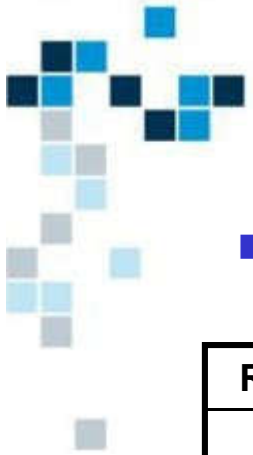
Récapitulatif sur l'utilisation de la pile (2)

- Changement de contexte
 - Au début d'un sous programme: sauvegarde de l'adresse de retour et des registres modifiés.
 - A la fin d'un sous programme: restauration des registres et de l'adresse.
- Passage de paramètres par la pile
 - Dans le programme appelant
 - Empiler les paramètres
 - Dans le sous programme appelé
 - Sauvegarder les registres (le contexte)
 - Lire les paramètres dans la pile
 - Traitement de la fonction
 - Restaurer les registres (le contexte)
 - Libérer l'espace occupé par les paramètres dans la pile



■ ARM Procedure Call Standard (APCS)

- Pour faciliter l'interfaçage de routines issues de sources différentes (compilateurs, programmeur), un ensemble de règles a été défini pour standardiser les entrées et sorties de sous-programmes.
- L'APCS permet
 - de définir une utilisation spécifique des registres
 - de définir quel type de pile à utiliser (full/empty, ascending/descending supporté par le jeu d'instruction)
 - de définir les mécanismes de passage de paramètres et de résultat utilisés par les fonctions et procédures
 - etc



ARM Procedure Call Standard (APCS)

■ Convention d'utilisation des registres

Registre	Nom APCS	Description
0	a1	Argument 1
1	a2	Argument 2
2	a3	Argument 3
3	a4	Argument 4
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	v6	Register variable 6
10	sl	Stack Limit
11	fp	Frame Pointer
12	ip	Intra-Procedure Scratch register
13	sp	Stack Pointer
14	lr	Link Address
15	pc	Program Counter

Passage des 4 premiers paramètres (au delà les paramètres sont passés la pile)

Variables locales (registres à sauvegarder en entrée d'un sous-programme)

R11 = fp

R13 = sp

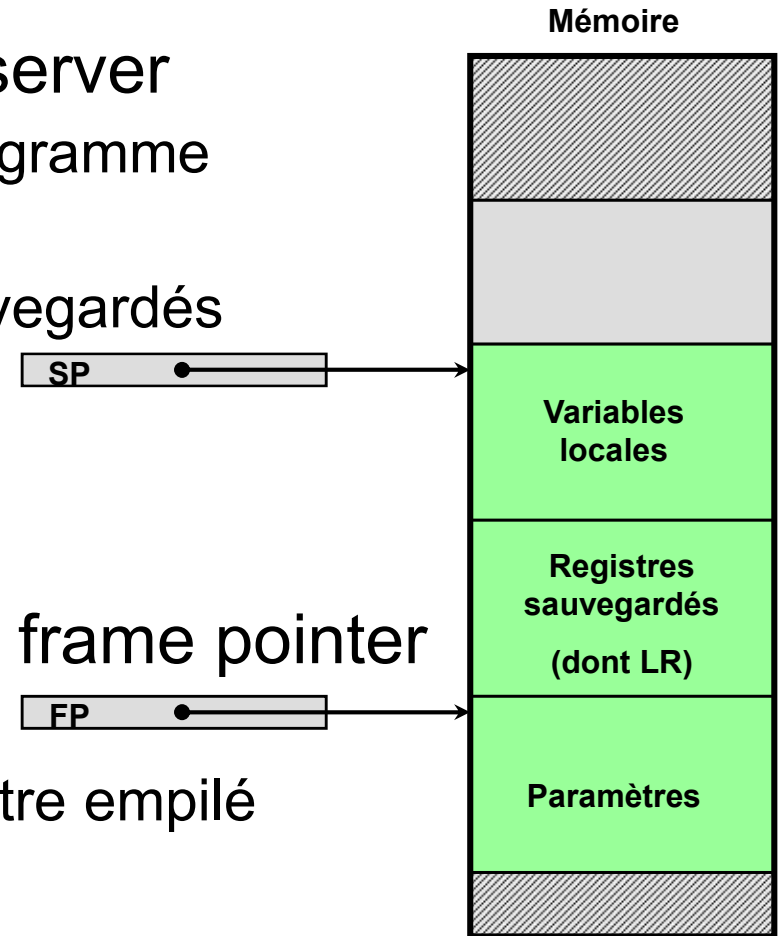
R14 = lr

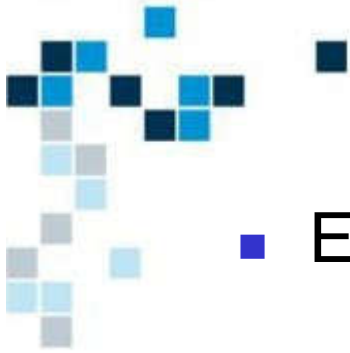
R15 = pc



Cadre de pile (stack frame)

- La pile est utilisée pour conserver
 - Les paramètres d'un sous programme
 - L'adresse de retour
 - Les valeurs des registres sauvegardés
 - Les variables locales
- On utilise un registre appelé frame pointer
 - Le registre R11
 - Il pointe sur le dernier paramètre empilé
- Le pointeur de pile SP pointe lui sur le sommet de la zone des variables locales





Cadre de pile (stack frame)

■ Exemple d'utilisation

```
main:  str r4, [sp, #-4]!
       str r5, [sp, #-4]!
       str r6, [sp, #-4]!
       bl  foo
```

```
foo:   stmfd sp!, {r3-r8, fp, lr}      @ save registers
       sub fp, sp, #40                 @ set FP
       sub sp, sp, #100                @ allocate 100
                                           @ octets for
                                           @ local var

       ...

       add sp, sp, #100                @ free local
                                           @ var space

       ldmfd sp!, {r3-r8, fp, lr}      @ reload
                                           @ register
                                           @ context

       add sp, sp, #12                 @ free parameter
                                           @ space

       mov pc, lr                     @ return
```



ARM Procedure Call Standard (APCS)

- Modèle de pile: Full Descending
- Cadre de pile

