

## Elec3A: Architecture

### Leçon 2: Programmation en assembleur

**Pr. El-Bay Bourennane**

[ebourenn@u-bourgogne.fr](mailto:ebourenn@u-bourgogne.fr) ; Tél./Fax : (33) 3 80 39 59 99

Université de Bourgogne / UFR S&T / Licence 2

Année universitaire 2020/2021

# ARM Ltd

2

- Société créée en novembre 1990
  - ▣ Le processeur ARM a été créé par la société britannique appelée Acorn
- Cette société conçoit les processeurs RISC ARM
- ARM ne fabrique pas ses processeurs mais vend des licences à des fabricants de semi-conducteurs.
- Cette société développe également les outils logiciels pour la mise en œuvre des programmes, etc.





# Applications du processeur ARM7

4

L'ARM7 est idéal pour les applications nécessitant des processeurs RISC à partir d'un processeur compact et performant.

**Télécoms-** téléphonie mobile

**Informatique portable-** Ordinateur portable

**Instrument portable-** Unité d'acquisition de données portable

**Automobile-** Unité de gestion de moteur

**Systèmes d'information-** Cartes à puce

**Imagerie-** contrôleur JPEG



- ARM est l'un des cœurs de processeurs les plus licenciés et les plus répandus dans le monde
- Utilisé en particulier dans les appareils portables en raison de sa faible consommation d'énergie et des performances raisonnables (MIPS / watt)



# Organisation du processeur ARM7

5

- ARM7 est une famille d'architectures type RISC
- "ARM" est l'abréviation de "Advanced RISC Machines"
- ARM7 ne fabrique pas ses propres composants VLSI (circuits intégrés).
- ARM7- L'architecture est de type Von Neuman

# Qu'est ce qu'un ARM7TDMI?

6

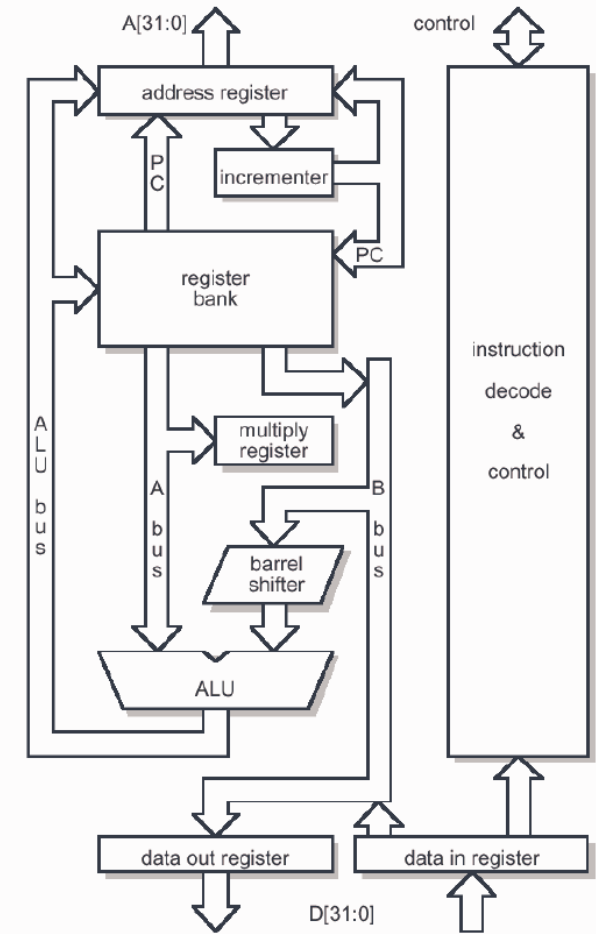
## □ **ARM** : **A**corn **RISC** **M**achine

- Processeur à Architecture « **Von Neumann** »
- **3 étages de pipeline** : Fetch, Decode, Execute
- Instructions sur **32 Bits**
- 2 instructions d'accès à la mémoire **LOAD et STORE**
- **T** : support du mode "Thumb" (instructions sur 16 bits)
- **D** : extensions pour la mise au point
- **M** : Multiplieur 32x32 et instructions pour résultats sur 64bits.
- **I** : émulateur embarqué ("Embedded ICE" )

# Schéma Bloc de ARM7

7

- ✓ Deux blocs principaux:
  - datapath et Décodeur
- ✓ Une banque de registre (r0 à r15)
  - Deux ports de lecture vers A-bus / B-bus
  - Un port d'écriture depuis ALU-bus
  - Ports de lecture / écriture supplémentaires pour Compteur de Programme r15
- ✓ Registre à décalage pour le décalage/rotation du 2<sup>ème</sup> opérande par n'importe quel nombre de bits
- ✓ ALU effectue des opérations arithmétiques/logiques
- ✓ Incrémenteur de PC dédié
- ✓ Registre d'adresse - soit à partir de PC ou de ALU

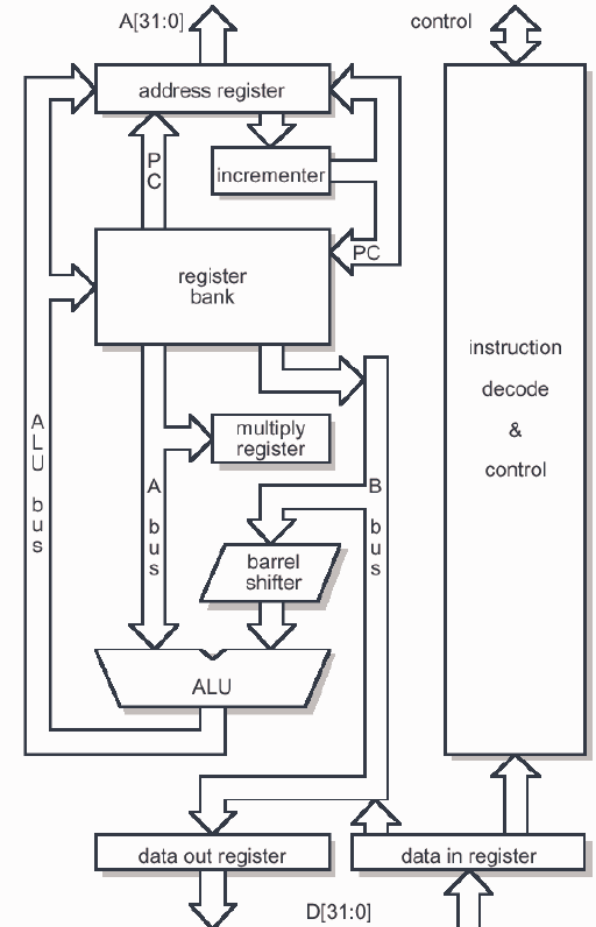


Organisation interne de ARM7

# Schéma Bloc de ARM7(suite)

8

- ✓ Le registre de données contient des données de lecture/écriture de/vers la mémoire
- ✓ Le décodeur d'instructions décode les instructions du code machine pour produire des signaux de contrôle vers le chemin de données
- ✓ Les instructions de traitement de données prennent un seul cycle: les valeurs de données sont lues sur le bus A et le bus B, les résultats d'ALU sont écrits dans la banque de registres

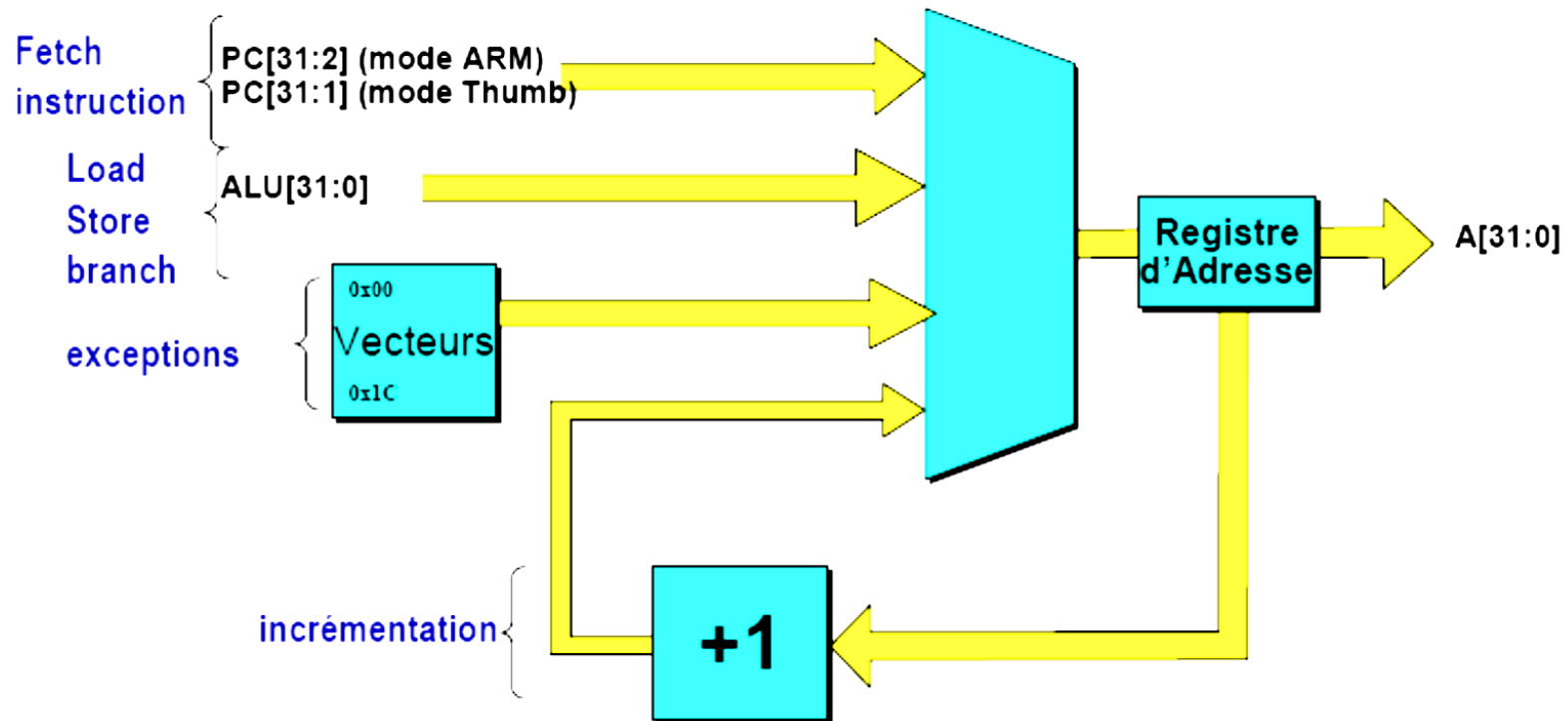


Organisation interne de ARM7



# Génération des Adresses

9

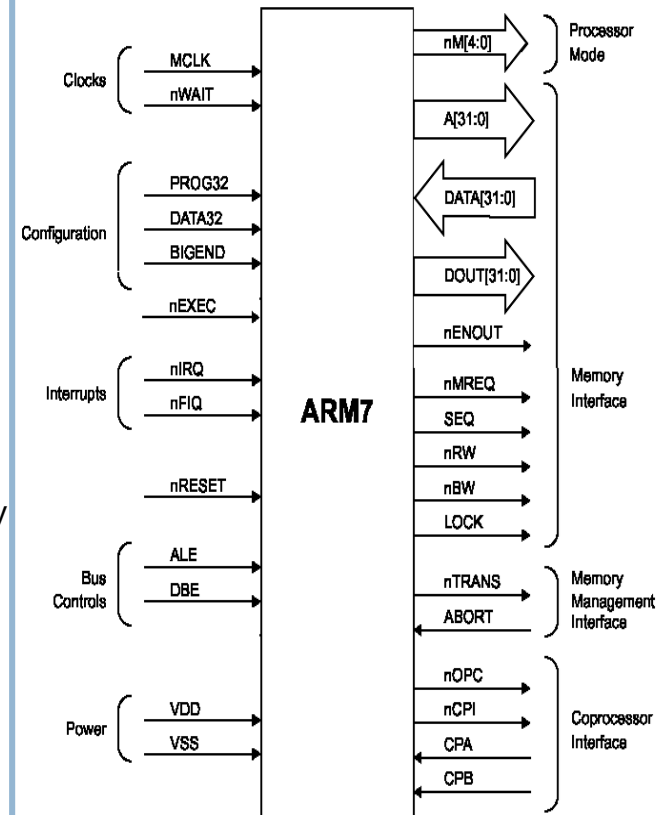


# Description des signaux ARM7

10

- Horloge
  - MCLK Memory Clock Input
  - nWAIT Not wait
- Configuration
  - PROG32 32 bit program configuration
  - DATA32 32 bit data configuration
  - BIGEND Big Endian configuration
- Interruptions
  - nIRQ Not interrupt request
  - nFIQ Not fast interrupt request
- Interface mémoire
  - A[31:0] Addresses
  - DATA[31:0] Data bus in
  - DOUT[31:0] Data bus out
  - nENOUT Not enable data outputs
  - nMREQ Not memory request
  - SEQ Sequential address
  - nRW Not read/write
  - nBW Not byte/word
  - LOCK Locked operation

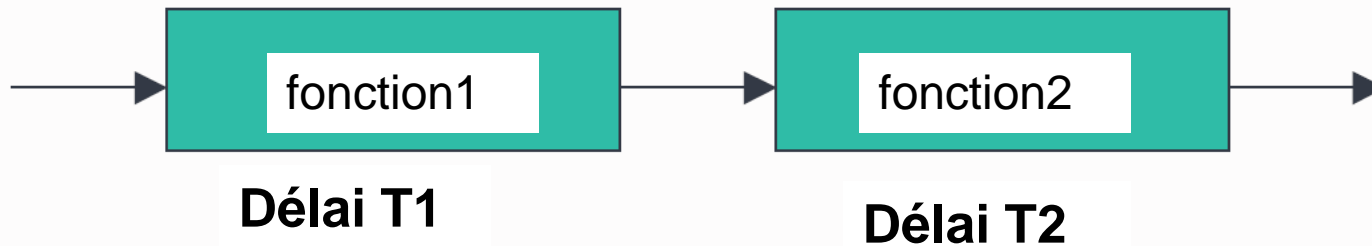
- Unité de gestion de la mémoire
  - nTRANS Not memory translate
  - ABORT Memory abort
- Interface Coprocesseur
  - nOPC Not op-code fetch
  - nCPI Not coprocessor instruction
  - CPA Coprocessor absent
  - CPB Coprocessor busy
- autre
  - nEXEC  
\*\*\*\*\*
  - nRESET Not reset



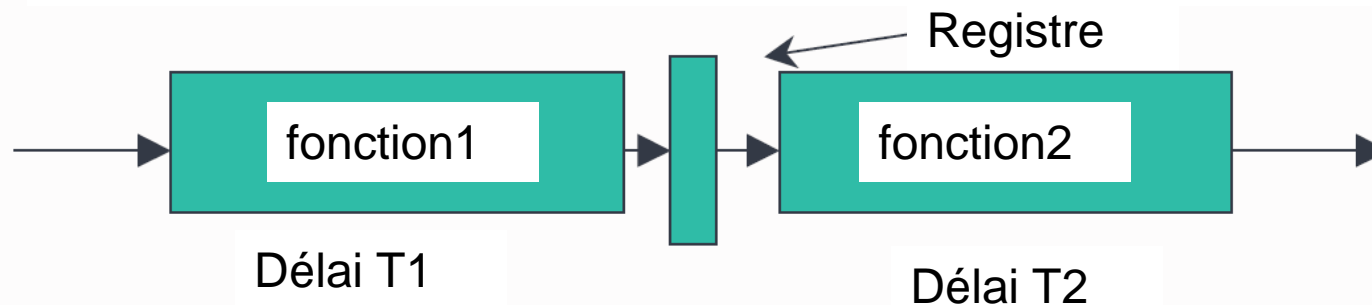
# ARM7 - Pipelining

11

- La fréquence d'exécution maximale est limitée par le chemin de propagation le plus lent.



- Dans ce cas, on ne peut traiter une nouvelle donnée que tous les  $T1+T2$  de délai



- Dans ce cas, on peut traiter une nouvelle donnée tous les  $\text{Max}(T1, T2)$  de délai. Le temps de traitement reste toujours  $T1+T2$

# ARM7 – Le Pipeline d'Instructions

12

- La famille ARM7 utilise un pipeline à 3 étages pour augmenter la vitesse du flot d'instructions dans le microprocesseur.

Mode: ARM Thumb

PC

PC

**FETCH**

L'instruction est lue dans la mémoire

PC - 4

PC-2

**DECODE**

Décodage de l'instruction

PC - 8

PC - 4

**EXECUTE**

Registre(s) lu(s) du banc de registres  
Opérations de décalage et ALU  
Ecriture du résultat vers le banc de registres

Le PC pointe sur l'instruction en cours de lecture (FETCHed), et non sur l'instruction en cours d'exécution.

# ARM7 - Exemple: Pipeline Optimal

13

Cycle	1	2	3	4	5	6
<u>Instruction</u>						
ADD	Fetch	Decode	Execute			
SUB		Fetch	Decode	Execute		
MOV			Fetch	Decode	Execute	
AND				Fetch	Decode	Execute
ORR					Fetch	Decode
EOR						Fetch
CMP						
RSB						

- il faut 6 cycles pour exécuter 6 instructions (CPI "Cycles Per Instruction")=1
- Toutes les operations ne jouent que sur des registres (1 cycle)

# Taille des données et jeu d'instructions

14

- Le processeur ARM7 a une architecture 32 bits.
- Peuvent également être manipulés sous ARM7:
  - ▣ **Octet (Byte)** signifie 8 bits
  - ▣ **Demi-mot (Halfword)** signifie 16 bits (deux octets)
  - ▣ **Mot (Word)** signifie 32 bits (4 octets)
- ARM7 peut manipuler deux jeux d'instructions
  - ▣ 32 bits en mode normal
  - ▣ 16 bits en mode Thumb



# Les Modes du Microprocesseur ARM7

15

- Un microprocesseur ARM a 7 modes opératoires de base :
- **User** : mode sans privilège où la plupart des tâches s'exécutent (mode normal d'exécution)
- **FIQ** : on y entre lors d'une interruption de priorité haute (rapide) (transfert rapide de data)
- **IRQ** : on y entre lors d'une interruption de priorité basse (normale) (gest. Inter. normales)
- **Supervisor** : on y entre à la réinitialisation et lors d'une interruption logicielle (SWI "SoftWare Interrupt") (Mode protégé pour l'OS)
- **Abort** : utilisé pour gérer les violations d'accès mémoire (protection de la mémoire)
- **Undef** : utilisé pour gérer les instructions non définies ("undefined") (gest. Inter. normales)
- **System** : mode avec privilège utilisant les mêmes registres que le mode User

# L'ensemble des registres ARM7

16

## Registres actifs

Mode  
Utilisateur

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr

## Bancs de Registres (Spécifiques à un mode)

FIQ

IRQ

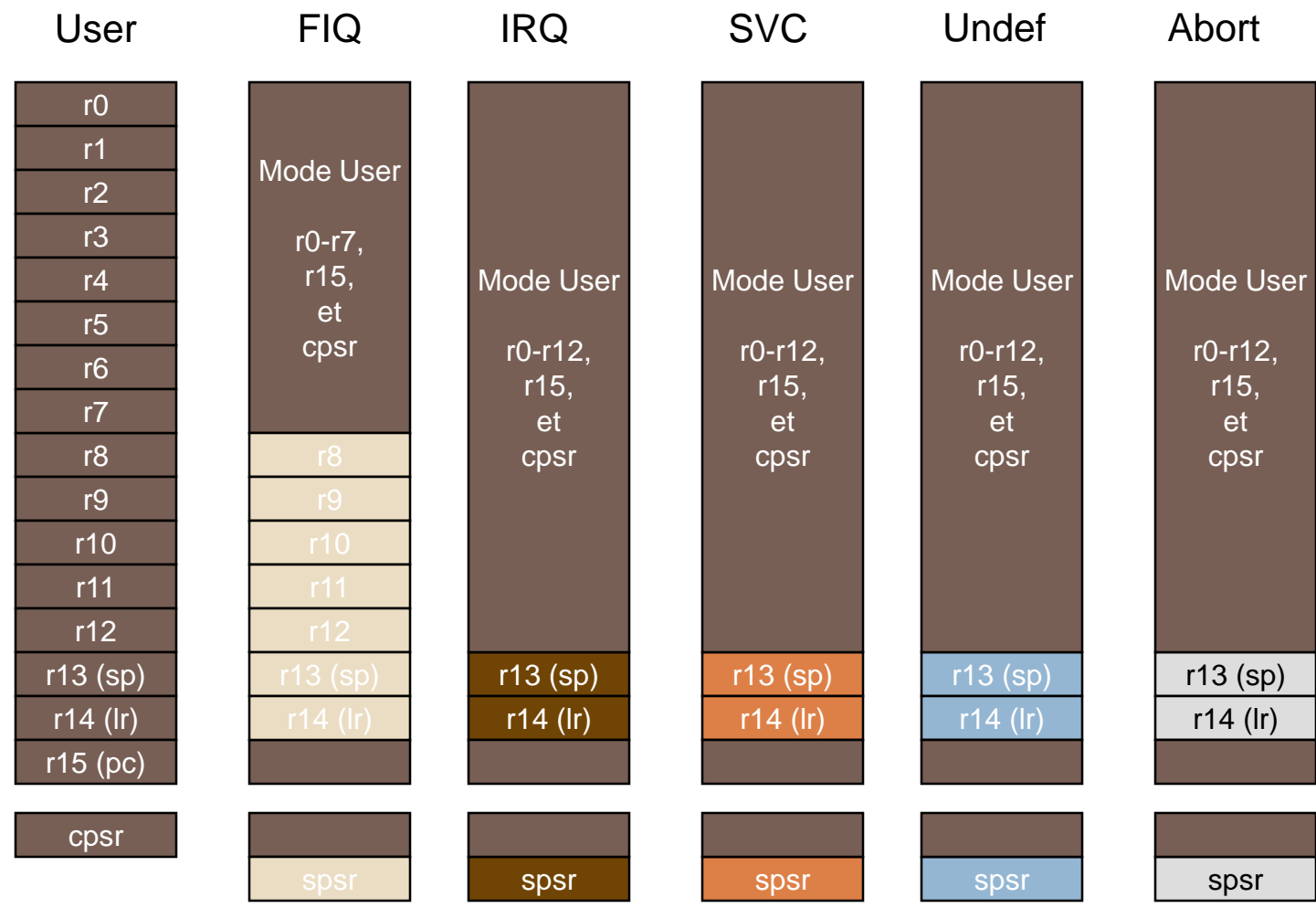
SVC

Undef

Abort

r8				
r9				
r10				
r11				
r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
spsr	spsr	spsr	spsr	spsr

# Résumé de l'organisation des registres



# Les Registres

18

- **ARM dispose de 37 registres de 32-bits.**

- ✓ 1 pointeur de programme (**r15** ou **pc** : "program counter")
- ✓ 1 contenant l'état courant du microprocesseur **cpsr**
- ✓ 5 dédiés à sauvegarder l'état en cas de changement de mode **spsr**
- ✓ 30 registres d'usage général

- **Le mode en cours du microprocesseur gouverne lequel des bancs de registres est accessible. Tous les modes peuvent accéder à :**

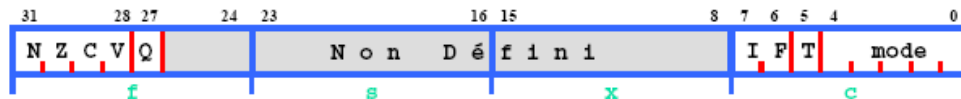
- ✓ un jeu particulier de registres **r0-r12**
- ✓ un registre **r13** particulier (le pointeur de pile, **sp**)
- ✓ un registre **r14** particulier (le registre de lien, **lr**)
- ✓ le pointeur de programme, **r15** (**pc**)
- ✓ le registre d'état courant du microprocesseur, **cpsr**

- **Modes avec privilège (sauf System) peuvent aussi accéder à:**

- ✓ un registre **spsr** particulier

# Les Registres d'État (CPSR et SPSR)

19



## ■ Indicateurs conditionnels

- N = Résultat **N**égatif de l'ALU
- Z = Résultat nul de l'ALU (**Z**éro)
- C = Retenue (**C**arry)
- V = Débordement (**o**Verflow)

## ■ Q = débordement avec mémoire

- Architecture 5TE seulement
- Indique qu'un débordement s'est produit pendant une série d'opérations

## ■ Validation des interruptions

- I = 1 dévalide IRQ.
- F = 1 dévalide FIQ.

## ■ Mode Thumb

- Architecture xT seulement
- T = 0, Processeur en mode ARM
- T = 1, Processeur en mode Thumb

## ■ Indicateurs de mode

- Indiquent le mode actif : système, IRQ, FIQ, utilisateur...

**CPSR** : Current Program Status Register

**SPSR** : Saved Program Status Registers

I: Irq (Interrupt)

F: Firq (Fast Interrupt)

10000: utilisateur

10001: FiQ

10010: Irq

10011: superviseur

10111: Abort

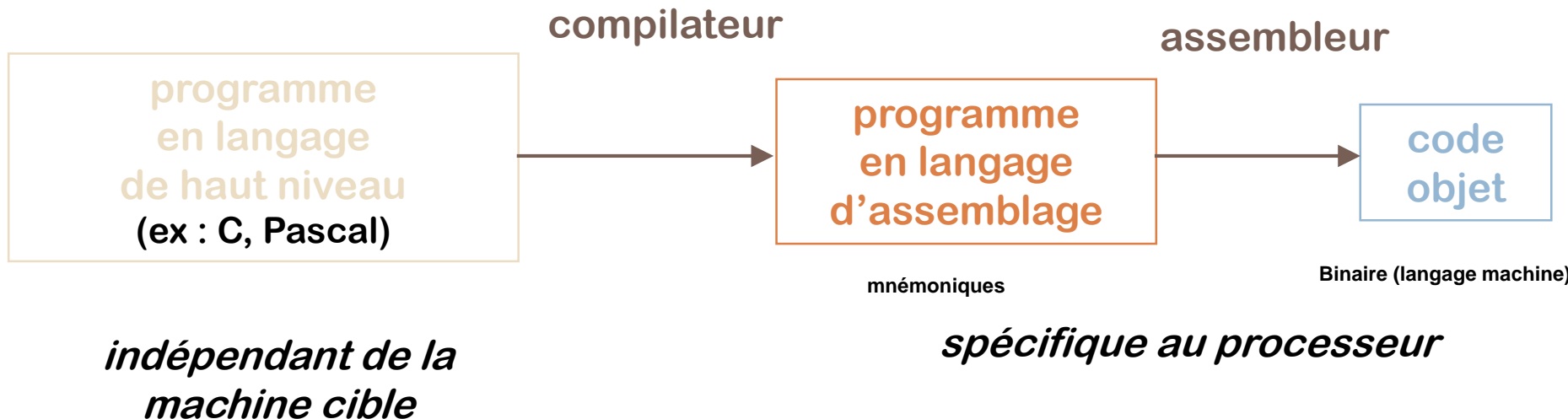
11011 : Undefined

11111 : System

# Langage de bas niveau

20

- Instructions codées en binaire
  - ▣ facilement identifiables par le processeur
  - ▣ n'occupent pas trop de place en mémoire
  - ▣ peuvent être représentées par des mnémoniques pour plus de lisibilité
- Traduction





# Langage de bas niveau

21

- Structure d'un programme en langage assembleur :
  - ✓ Chaque ligne est constituée de 4 champs :
    - Étiquette
    - Mnémonique code opération
    - Opérande
    - Commentaire

# Une ligne d'assembleur

22

<etiquette>   <mnémonique>   <opérandes>

<**Label**>

<**opcode**>

<**operands**>

spgm1                    mov                    r5, #1    ; ceci est un comment.

                         mov                    r6, #0xFF ; r6 ← FF

# Écriture d'un programme en langage d'assemblage

23

## □ Premier exemple

	<b>AREA TP1, CODE, READONLY</b>	; donne un nom au bloc
	<b>ENTRY</b>	; marque la première instruction
debut	<b>MOV r0, #15</b>	; initialise les valeurs
	<b>MOV r1, #20</b>	
	<b>BL addition</b>	; Appel sous-programme <i>addition</i>
	<b>MOV r0, #0x18 ;</b>	
	<b>LDR r1, =0x20026 ;</b>	
	<b>SWI 0x123456</b>	; Termine le programme
addition	<b>ADD r0, r0, r1 ; r0 = r0 + r1</b>	
	<b>MOV pc, lr</b>	; retour du sous-programme
	<b>END</b>	; Marque la fin du fichier

Label                      Opcode                      Opérandes                      Commentaires

# Directives d'assemblage

24

- Define constant Data (DCD)
  - ▣ La directive Define Constant Data (DCD) permet au programmeur d'entrer une valeur entière dans la mémoire programme.
  - ▣ Cette directive traite la donnée comme une partie fixe (permanente) du programme.

Exemples :

```
chaine    DCB "premiere chaine- source",0 ; Define Constant Byte  
Donne32 DCD 0,1,2,3,4,5,6,7,8,9; Define Constant Data
```

# La directive “Equate”

25

- Directive d'assemblage EQU
  - ▣ définir des constantes (**equ**)
  - ▣ réserver de la place en mémoire pour une variable **entière**

	étiquette	equ	valeur_initiale
Exemple de déclaration :	V	equ	8
	Z	equ	3450

Exemple utilisation dans un programme:

```
Add R1, R1, #V;    R1=R1+V = R1+8
MOV R5, #V; R5 reçoit la valeur de V
```

# Instruction de comparaison et codes condition

26

- **cmp r1,r2** effectue la soustraction **r1 - r2** mais n'a pas de registre résultat explicite
- implicitement, affecte les bits **NZVC** du registre CPSR
- **N** : négatif « si MSB de (r1 - r2) est à '1' »
- **Z** : zéro (à 1 si  $r1 - r2 = 0$ )
- **V** : overflow (=1 si dépassement de capacité, nombres signés)
- **C** : carry (=1 si retenue de l'addition est 1, nombres non signés)



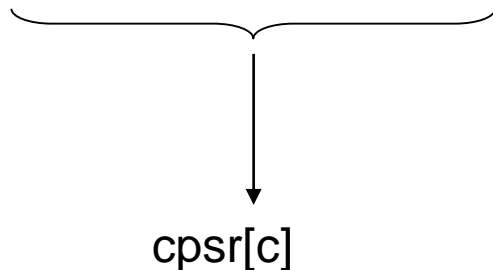
# Autres instructions et codes conditions

27

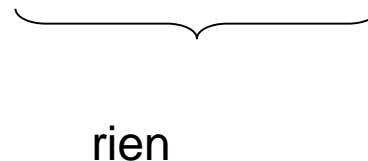
□ D'autres instructions peuvent mettre à jour les codes conditions si elles ont le suffixe « S »

□ `adds r5, r6, r2`

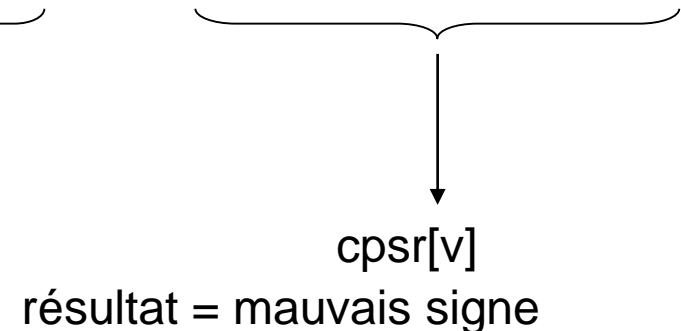
1011	11
+0111	+7
10010	18 (>15)



1011	-5
+0111	+7
10010	2



1011	-5
+1001	+ -7
10100	4 !



# A quoi servent les codes conditions ?

28


- Addition multi-mots (propagation de retenue)
- Détection de dépassement de capacité
- Utilisés par les instructions de contrôle de flot (branchements conditionnels)

# Les Flags et les exécutions conditionnelles

29

- Les instructions ARM peuvent être effectuées de manière conditionnelle en les postfixant avec le champ de code de condition approprié.
- Cela améliore la densité du code et les performances du programme en réduisant le nombre d'instructions de branchements (sauts).

```
CMP    r3,#0
BEQ    saut
ADD    r0,r1,r2
saut    .....
.....
```



```
CMP    r3,#0
ADDNE  r0,r1,r2
```

- Par défaut, les instructions de traitement de données n'affectent pas les drapeaux du code de condition mais les drapeaux (flags) peuvent être mis à jour en utilisant le suffixe "**S**". CMP n'a pas besoin de "**S**" car elle affecte par défaut les Flags.

Boucle

```
...
SUBS   r1,r1,#1
BNE    Boucle
```

décrémente r1 et affecte les flags

Si le flag Z vaut zéro alors brancher vers Boucle

# Les Codes Condition

30

- Les codes condition possibles sont listés ci-dessous:

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V
AL	Always	

# Exemples d'exécution conditionnelle

31

- Utilisation d'une séquence de plusieurs instructions conditionnelles

```
if (a==0) x=0;
if (a>0)  x=1;

    CMP      r0,#0; ici r0=a et r1=x
    MOVEQ    r1,#0; si égalité, x=0
    MOVGT    r1,#1; si plus grand, x=1
```

- Utilisation des instructions de comparaison conditionnelle

```
if (a==4 || a==10) x=0; "||" Signifie OR (OU logique)

    CMP      r0,#4
    CMPNE    r0,#10
    MOVEQ    r1,#0
```

# Exemples d'exécution conditionnelle

32

## Exemple Addition multi-mots (propagation de retenue)

On veut additionner deux nombres de 64-bits X et Y et sauvegarder le résultat dans Z. Nous avons besoins de deux registres par nombre. Sauvegarder X dans r1:r0, Y dans r3:r2, et Z dans r5:r4 (notation – MSW:LSW)

Alors :

```
ADDS r4, r0, r2
```

```
ADC r5, r1, r3
```

“S” à la fin de l’instruction signifie qu’on veut mettre à jour les codes conditions à la fin de l’addition.

De même, si nous voulions soustraire les deux nombres :

```
SUBS r4, r0, r2
```

```
SBC r5, r1, r3
```



# Instructions de branchement

33

## □ **B : *Branch*.**

Branchement vers une adresse absolue codée sur 24 bits.

```
b routine ; pc:=@routine
```

Les 24 bits du déplacement ne permettent, une fois décalés, d'atteindre que des adresses éloignées de  $\pm 16\text{Mo}$ . Pour effectuer un saut lointain, on effectue un chargement ou un calcul avec le pc comme destination.

## □ **bl : *Branch with Link*.**

Saut avec sauvegarde du pc dans r14.

```
bl routine ; r14:=pc puis pc:=@routine
```

Permet de limiter les accès mémoire lors des appels de sous-programmes.

# Instructions de branchement

34

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

# Instructions de contrôle

35

**B**{<cond>} **label**

**cond** : EQ, NE, HS, LO, ... (14 au total dont une partie pour les nombres non signés, une autre pour les nombres signés)

- Teste les codes conditions
- **Branchement** si condition vérifiée
- **label** sera remplacé par le déplacement entre l'adresse de l'instruction **B** et l'adresse de l'instruction cible par l'assembleur.

# Instructions de contrôle

36

- **tst : TeST.** Test de bit(s).  
tst r1,#3 ; *r1 ET 3*, bits d'état **N** et **Z** mis en place selon le résultat.  
Le bit **C** est mis en place par le décalage éventuel.
- **teq : Test EQuality.** Teste l'égalité de deux registres avec un OU exclusif.  
teq r0,r1 ; *r0 (XOR) r1*, bits d'état **N** et **Z** mis en place selon le résultat. Le bit **C** est mis en place par le décalage éventuel.
- **cmp : CoMPare.**  
Comparaison de deux valeurs par soustraction. cmp r0,r1 ; *r0-r1*, bits d'état **N** et **Z** mis en place selon le résultat. Les bits **C** et **V** sont mis en place par l'ALU.
- **cmn : CoMpare Negative.**  
Comparaison avec négation du second opérande.  
cmn r0,r1 ; *r0+r1*, bits d'état **N** et **Z** mis en place selon le résultat. Les bits **C** et **V** sont mis en place par l'ALU.

# Instructions arithmétiques et logiques

37

## □ Constituées de:

□ Arithmétiques:	ADD	ADC	SUB	SBC	RSB	RSC
□ Logiques:	AND	ORR	EOR	BIC		
□ Comparaisons:	CMP	CMN	TST	TEQ		
□ Affectation de données:		MOV	MVN			

- Toutes ces instructions ne fonctionnent qu'entre registres mais jamais avec la mémoire. C'est un architecture type Load & Store

## Syntaxe:

**<Operation>{<cond>}{S} Rd, Rn, Operand2**

- Les instructions de comparaison mettent à jours les flags – Elles n'ont pas besoin de registre destinataire de résultat Rd
- Le deuxième opérande est transmis à l'ALU au travers le registre à décalage.

# Instructions arithmétiques et logiques

38

- Modes d'adressage : reg, reg, reg | imm
- add[s] rd, rs1, rs2                      ;rd  $\leftarrow$  rs1 + rs2
- adc[s] rd, rs1, rs2                      ;rd  $\leftarrow$  rs1 + rs2 + bit C de CPSR
- sub[s] rd, rs1, rs2                      ;rd  $\leftarrow$  rs1 - rs2
- sbc[s] rd, rs1, rs2                      ;rs  $\leftarrow$  rs1 - rs2 - !CPSR[C]
- rsb[s] rd, rs1, #imm                      ;rd  $\leftarrow$  #imm - Rs1
- rsc[s] rd, rs1, #imm                      ;rd  $\leftarrow$  #imm - rs1 - !CPSR[C]
- mul et ses variantes
- and orr eor bic

**!CPSR[C] = Not (Carry Flag)**

# Les opérations logiques

39

- Ci-dessous quelques opérations sur des bits

AND	r0, r1, r2	; r0 := r1 and r2 (bit-by-bit for 32 bits)
ORR	r0, r1, r2	; r0 := r1 or r2
EOR	r0, r1, r2	; r0 := r1 xor r2
BIC	r0, r1, r2	; r0 := r1 and not r2

- BIC signifie Bit-Clear. Chaque bit à « 1 » dans le deuxième opérande va mettre à « 0 » le bit correspondant dans le premier opérande

r1:	0101 0011 1010 1111 1101 1010 0110 1011
r2:	1111 1111 1111 1111 0000 0000 0000 0000
r0:	0000 0000 0000 0000 1101 1010 0110 1011

# Les instructions d'affectation (MOV)

40

Ci-dessous des exemples d'affectations de registres avec l'instruction MOV

<b>MOV</b>	<b>r0, r2</b>	<b>; r0 := r2</b>
<b>MVN</b>	<b>r0, r2</b>	<b>; r0 := not r2</b>

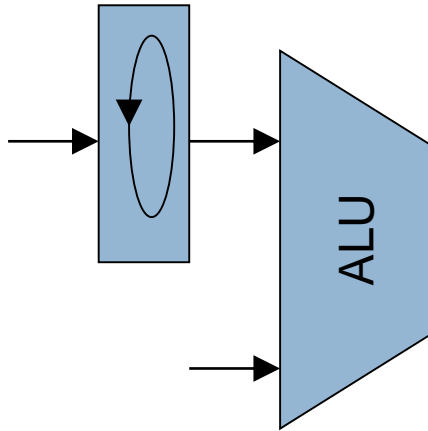
MOVN veut dire MOV négatif (valeurs inversées)

r2:	0101 0011 1010 1111 1101 1010 0110 1011
r0:	1010 1100 0101 0000 0010 0101 1001 0100



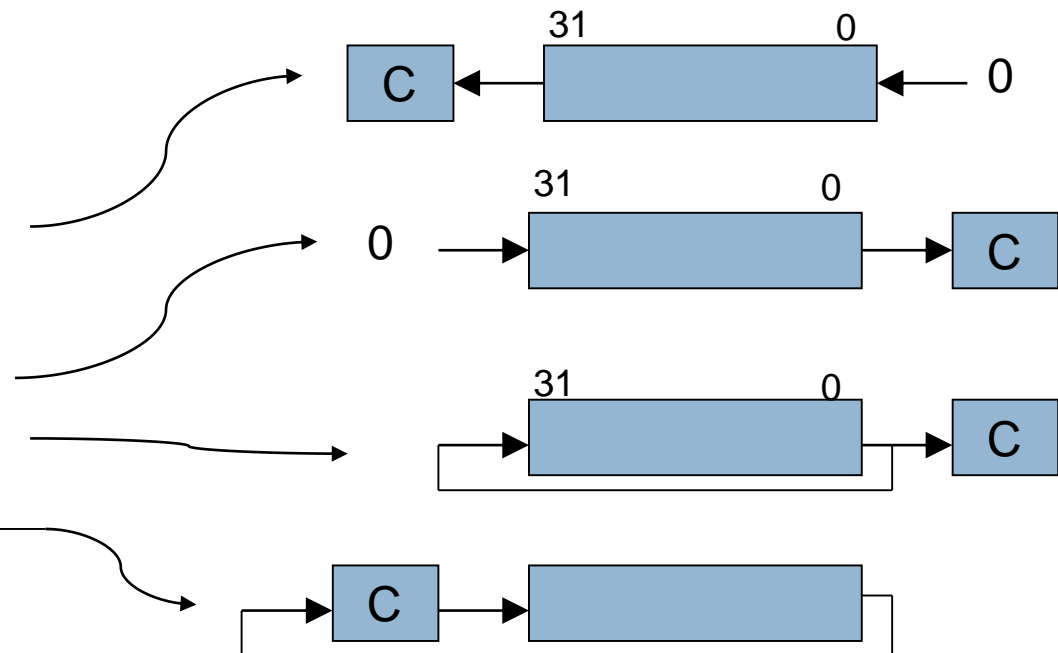
# Décalage d'un opérande

41



LSL #imm ou Ri  
LSR #imm ou Ri  
ASR #imm ou Ri  
ROR #imm ou Ri  
RRX

Décalage possible du 2<sup>ème</sup> opérande



# Décalage d'un opérande

42

- **lsl** : *Logical Shift Left*.

Décalage logique des bits d'un registre vers la gauche.

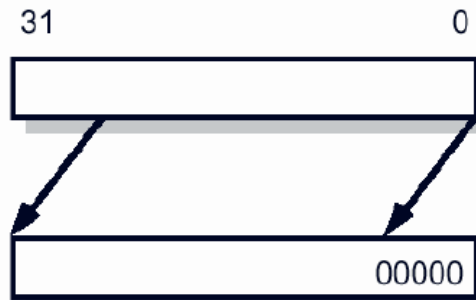
`add r0,r1,r2,lsl#2 ;  $r0 := r1 + r2 * (2^{**}2) = r1 + r2 \ll 2$`

`Mov r0, r0, lsl#3`

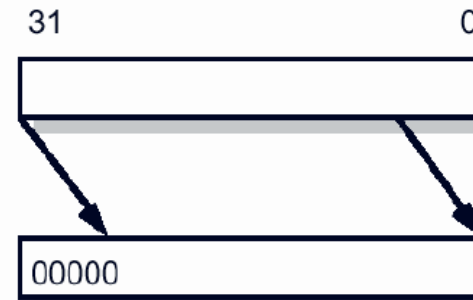
- **lsr** : *Logical shift Right*.

Décalage logique des bits d'un registre vers la droite.

`add r0,r1,r2,lsr r3 ;  $r0 := r1 + r2 / (2^{**}r3) = r1 + r2 \gg r3$`



LSL #5



LSR #5

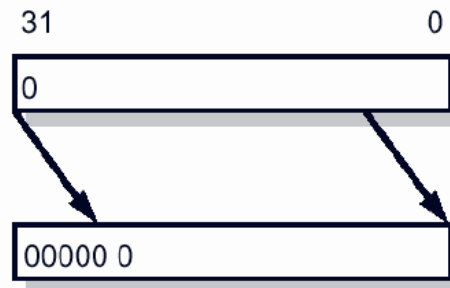
# Décalage d'un opérande

43

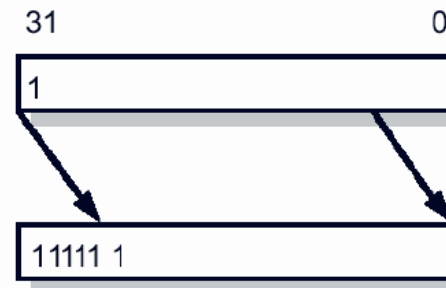
- **asl** : *Arithmetic Shift Left*.

asl est un synonyme de lsl. Il est préférable d'utiliser le mnémonique lsl à la place.

- **asr** : *Arithmetic Shift Right*. Décale un registre vers la droite en conservant son signe.



ASR #5, positive operand



ASR #5, negative operand

# Décalage d'un opérande

44

## •ror : *ROtate Right*.

Fait "tourner" le registre sur lui-même de gauche à droite.

```
add r0,r0,r1,ror #1 ; r0:=r0+(r1>>>1+ (r1 ET 1)<<31)
```

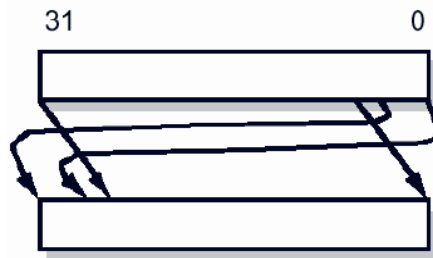
Remarquez que rol n'existe pas, il faut utiliser cette instruction avec (32-n) rotations pour l'émuler.

## •rrx : *Rotate Right eXtended*.

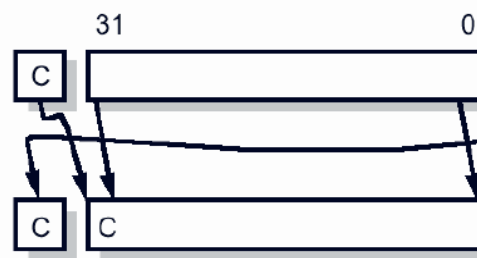
Rotation de 1 bit de gauche vers la droite sur 33 bits, avec le Carry.

```
add r0,r1,r2,rrx ; r0:=r1+(r2|c)>>>1+ (c<<32)
```

La retenue **C** est écrasé par le dernier bit décalé hors du registre.



ROR #5



RRX

# Utilisation du registre à décalage

45

## □ Exemple1:

$$r0 = r1 * 5 = r1 + (r1 * 4)$$

⇒ ADD r0, r1, r1, LSL #2

## □ Exemple2:

$$r2 = r3 * 105 = r3 * 15 * 7 = r3 * (16 - 1) * (8 - 1)$$

⇒ RSB r2, r3, r3, LSL #4 ;  $r2 = (r3 * 2^4 - r3) = r3 * 15$

⇒ RSB r2, r2, r2, LSL #3 ;  $r2 = (r2 * 2^3 - r2) = r2 * 7$

□ RSB rd, rs1, rs2 ;  $rd \leftarrow (rs2 - rs1)$

# Valeurs immédiates (constantes)

46

- Aucune instruction ARM7 ne peut contenir une constante codée sur 32 bits.
- Les valeurs immédiates doivent être codées sur 8 bits:

MOV r5, r6 ;  $r5 \leftarrow r6$

MOV r8, #255 ;  $r8 \leftarrow 255$  (ou 0xFF)

MOV r4, #256; Pas autorisé

MVN r9, #0 ;  $r9 \leftarrow 255$

# Comment charger une constante à 32 bits?

47

- Utiliser la pseudo-instruction:

- ▣ `LDR rd, =const`

- Exemple

pseudo-instructions

- ▣ `LDR r0,=0xFF`

- ▣ `LDR r0,=0x55555555`

=>

=>

Vraies instructions

`MOV r0,#0xFF`

`LDR r0,[PC,#Imm12]`

...

...

`DCD 0x55555555`

Adresse DCD

# Multiplication

48

## □ Syntaxe:

□ **MUL**{<cond>}{S} Rd, Rm, Rs

$$Rd = Rm * Rs$$

□ **MLA**{<cond>}{S} Rd, Rm, Rs, Rn

$$Rd = (Rm * Rs) + Rn$$

□ **MLA** = **M**ultiplication + **A**ccumulation

## □ Nombre de cycles d'horloge

□ L'instruction de base MUL

■ 2-5 cycles sur ARM7TDMI



# Premiers modes d'adressage

49

- Adressage d'un registre

`r5`

- Valeur immédiate (indiquée par #)

`#1`

`#0xF0`

`#0b10001100`

- Adressage d'une case mémoire

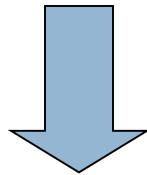
- Divers modes existent :

- **str** : *STore Register* : Écriture d'un mot (ou octet si *strb*) en mémoire
    - **ldr** : *LoaD Register* : Lecture d'un mot (ou octet si *strb*) en mémoire
    - **stm** : Écriture d'un ou plusieurs mots en mémoire
    - **ldm** : *LoaD Multiple* : Lecture d'un ou plusieurs mots en mémoire

# Mettre l'adresse d'une variable dans un registre

50

```
debut      add    r3,pc,#4      ; r3 ← pc+4
           ...                ; r3 pointe vers src
           ...
src         DCD    1,2,3,4,5,6,7,8
```



```
debut      adr     r3,src      ; pseudo-instruction
                                   ; obtention de l'adresse de src
           ...                ; r3 pointe vers src
src         DCD    1,2,3,4,5,6,7,8
```


# Mettre l'adresse d'une variable dans un registre

51

Utilisation de la pseudo instruction ADR – C'est comme une instruction normale sauf que c'est une directive d'assemblage . L' assembleur va la traduire en une ou plusieurs vraies instructions.

L' exemple suivant copie les données de TABLE 1 à TABLE2

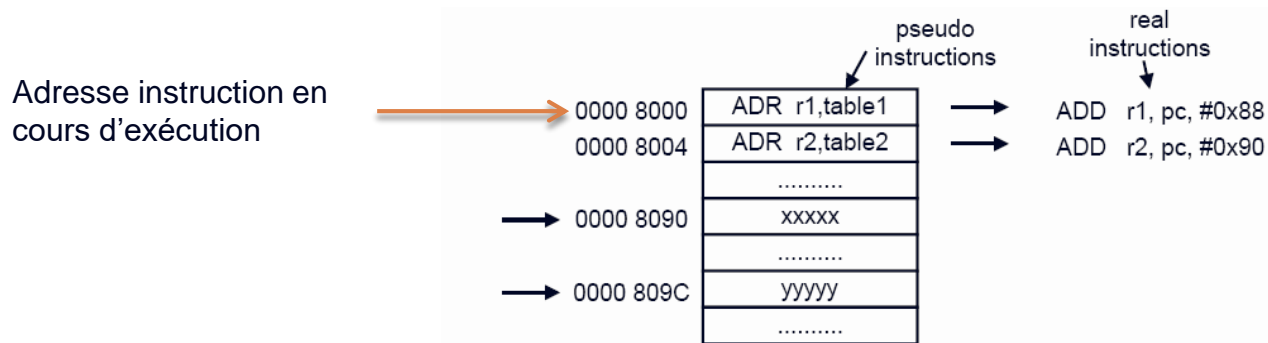
copy	ADR r1, TABLE1	; r1 pointe vers TABLE1
	ADR r2, TABLE2	; r2 pointe vers TABLE2
	LDR r0, [r1]	; charge 1ère valeur
	STR r0, [r2]	; et la sauvgde ds TABLE2
	.....	
TABLE1	DCD 6,3,9,14,33	; <source des données>
	.....	
TABLE2	DCD 0,0,0,0,0	; <destination des données>



# Mettre l'adresse d'une variable dans un registre

52

## Comment fonctionne ADR?



- Solution: Le compteur programme PC (**r15**) est souvent proche de l'adresse de la donnée désirée
- **ADR r1, TABLE1** est traduite en une instruction qui rajoute ou soustrait une valeur constante au PC (**r15**) et met le résultat dans r1
- Cette constante est connue sous le nom d'offset relatif au PC (**PC-relative offset**), et est calculée comme :  
 $\text{adr\_de\_table1} - (\text{valeur\_PC})$
- **Valeur\_PC** = Adresse instruction en cours d'exécution + 8 = **0000 8008**
- **Le saut (offset) est de**  $(00008090 - 00008008) = \mathbf{0x88}$

# Mettre l'adresse d'une variable dans un registre

53

Utilisation de la pseudo instruction ADR –  
Copions la suite des données de TABLE 1 à TABLE2

copy

```
ADR r1, TABLE1           ; r1 pointe vers TABLE1
ADR r2, TABLE2           ; r2 pointe vers TABLE2
LDR r0, [r1]               ; charge 1ère valeur
STR r0, [r2]               ; et la sauvgde ds TABLE2
ADD r1, r1, #4             ; r1 pointe le mot suivant
ADD r2, r2, #4             ; r2 pointe le mot suivant
LDR r0, [r1]               ; charge 2ème valeur ...
STR r0, [r2]               ; et la sauvgde ds TABLE2
.....
TABLE1 DCD 6,3,9,14,33     ; <source des données>
.....
TABLE2 DCD 0,0,0,0,0       ; <destination des données>
```

# Adressage d'une case de la mémoire

54

- Une adresse = 32 bits
- Une instruction = 32 bits et contient code opératoire, ...
- On ne peut pas mettre une adresse sous forme immédiate dans le code de l'instruction
  - Mettre l'adresse dans un registre et utiliser le contenu de ce registre pour accéder à la mémoire

# Instructions d'accès à la mémoire

55

- `ldr <reg>,<am>` ; `<am>= adresse mémoire`
  - ▣ copie d'un mot de la mémoire dans un registre
- `ldrb <reg>,<am>`
  - ▣ copie d'un octet
- `str <reg>,<am>`
  - ▣ écriture d'un mot en mémoire
- `strb <reg>,<am>`
  - ▣ écriture d'un octet

# Exemple pour Load

56

## □ Avant:

- `r0 = 0x00000000`
- `r1 = 0x00070000`
- `mem32[0x00070000] = 0x00000005`
- ✓ `LDR r0, [r1]`

## □ Après:

- `r0 = 0x00000005`
- `r1 = 0x00070000`



# Modes d'adressage de la mémoire

57

- $[R_i]$
- $[R_i, \#imm]$ 
  - ▣ Pré-indexé
  - ▣ Accès à des champs d'une structure, accéder à des paramètres et des variables locales dans la pile
- $[R_i, R_j]$ 
  - ▣ Pré-indexé
  - ▣ Accès à un élément d'un tableau (base + index)
  - ▣ Possibilité de décalage sur  $R_j$  (LSL, LSR, ...)

# Modes d'adressage pré-indexé

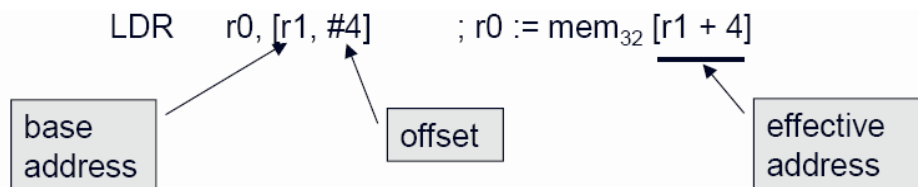
58

## ▣ Avant:

- $r0 = 0x00000000$
- $r1 = 0x00007000$
- $\text{mem}_{32}[0x00007000] = 0x00001000$
- $\text{mem}_{32}[0x00007004] = 0x00002000$
- ✓ Pré-indexation: `LDR r0, [r1, #4]`

## ▣ Après:

- $r0 = 0x00002000$
- $r1 = 0x00007000$



# Modes d'adressage pré-indexé

59

## Amélioration du programme précédent

copy	ADR r1, TABLE1	; r1 pointe vers TABLE1
	ADR r2, TABLE2	; r2 pointe vers TABLE2
	LDR r0, [r1]	; charge 1ère valeur
	STR r0, [r2]	; et la sauvgde ds TABLE2
	LDR r0, [r1, #4]	; charge 2ème valeur ...
	STR r0, [r2, #4]	; et la sauvgde ds TABLE2
	.....	
TABLE1	.....	; <source des données>
	.....	
TABLE2	.....	; <destination des données>

# Modes d'adressage pré-indexé

60

## ▣ Avant:

- $r0 = 0x00000000$
- $r1 = 0x00007000$
- $\text{mem32}[0x00007000] = 0x00001000$
- $\text{mem32}[0x00007004] = 0x00002000$

✓ Pré-indexation avec mise à jour: `LDR r0, [r1, #4] !`

## ▣ Après:

- $r0 = 0x00002000$
- $r1 = 0x00007004$

```
LDR    r0, [r1, #4]!    ; r0 := mem32[r1 + 4]
                        ; r1 := r1 + 4
```

- ▣ « ! » Indique que l'instruction doit mettre à jour le registre de base à la fin du transfert de données.

# Modes d'adressage de la mémoire

61

- $[R_n], \#imm$ 
  - ▣ post indexé (accès avec  $R_n$  puis incrémentation de  $R_n$ )
- $[R_n, \#imm] !$  ou  $[R_i, R_j] !$ 
  - ▣ auto indexé
  - ▣ accès à  $mem[R_i + \#imm]$  ou  $mem[R_i + R_j]$
  - ▣ Mise à jour :  $R_i = R_i + \#imm$  ou  $R_i = R_i + R_j$

# Modes d'adressage post-indexé

62

## ▣ Avant:

- `r0 = 0x00000000`
- `r1 = 0x00007000`
- `mem32[0x00007000] = 0x00001000`
- `mem32[0x00007004] = 0x00002000`

✓ Post-indexation : `LDR r0, [r1], #4`

## ▣ Après:

- `r0 = 0x00001000`
- `r1 = 0x00007004`

```
LDR    r0, [r1], #4    ; r0 := mem32[r1]
                        ; r1 := r1 + 4
```

# Modes d'adressage post-indexé

63

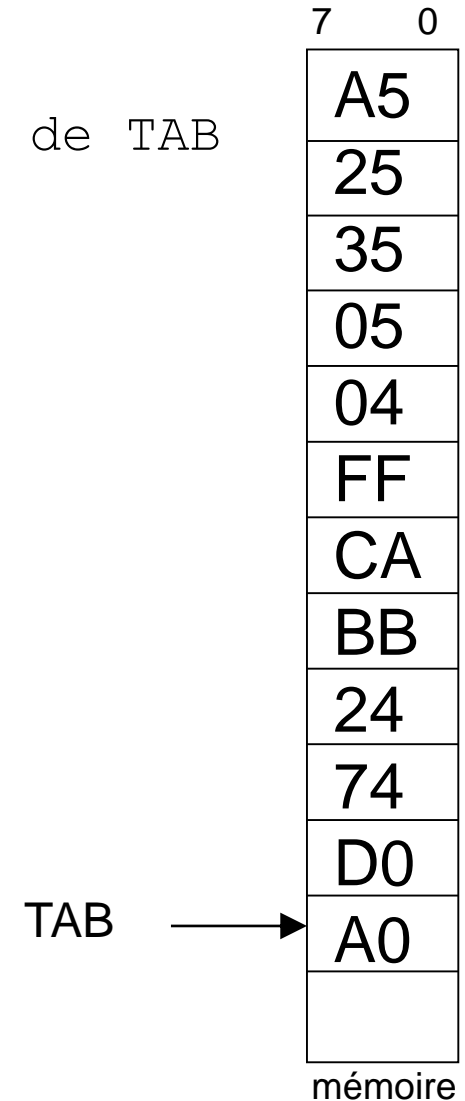
## Amélioration du programme précédent

copy	ADR r1, TABLE1	; r1 pointe vers TABLE1
	ADR r2, TABLE2	; r2 pointe vers TABLE2
<b>Boucle</b>	LDR r0, [r1], #4	; charge 1ère valeur
	STR r0, [r2], #4	; et la sauvgde ds TABLE2
	???	; si plus, aller à <b>Boucle</b>
TABLE1	.....	; <source des données>
	.....	
TABLE2	.....	; <destination des données>

# Modes d'adressage de la mémoire

64

```
ADR    R1, TAB          ;Obtenir l'adresse de TAB
LDRB   R2, [R1]          R2 : XXXXXXA0
LDR    R2, [R1]          R2 : 2474D0A0
LDR    R2, [R1], #4      R2 : 2474D0A0
                          R1 : TAB+4
LDRB   R2, [R1, #4]      R2 : XXXXXX05
LDR    R2, [R1, #4] !    R2 : A5253505
                          R1 : TAB+8
```





# Initialisation des éléments d'un tableau de 10 octets avec les 10 premiers entiers

65

## □ Programme

```

                                AREA tableau, CODE, READONLY                                ; donne le nom du bloc
                                ENTRY                                                ; marque la première instruction
debut                          MOV      r0, #0                                ; r0 = i
                                ADR      r1, tab                                ; r1 = Adresse de tab
boucle                         CMP      r0, #10
                                BCS      fin                                ; Carry Set. Bran. Si r0 ≥ 10
                                ; pour nombres non signés
                                STRB     r0, [r1], #1                        ; tab[i] ← I
                                ; r1 ← r1 + 1
                                ADD      r0, r0, #1                        ; r0 ← r0 + 1
                                B        boucle
Fin                            SWI      0x123456                        ; sortie du programme
Tab                            DCB      0,0,0,0,0,0,0,0,0,0 ; Define Constant Byte (octet) 8 bits
                                ; réserve place pour 10 bytes init. à 0
                                END      ; fin de fichier
```

## □ Pour un tableau de mots de 32 bits

□ **STR** r0, [r1], #4

# Mettre une valeur de 32 bits dans un registre

66

- Coder la valeur 32 bits dans le programme de sorte qu'elle soit en mémoire à l'aide de DCD
- Placer dans un registre l'adresse de cette valeur en la calculant par rapport à l'adresse de l'instruction

```
ldr    r5,=#0xFFFFFFFF    ; pseudo-instruction
                                ; remplacée
                                ; par mov si possible
                                ; sinon ...
```

```
v321   ldr    r5, [pc, #zz]    ; r5 <- mem[pc+zz]
```

...

```
v32    DCD    0,1,2,3,4,5,6,7,8,9
```

# Astuce pour éviter quelquefois un ldr ,=

67

- Chargement depuis la mémoire long
- Quelquefois on peut remplacer un ldr ,= par un mvn
- Exemple :

```
mov  r7, #0xFF00FFFF      ; impossible  
mvn  r7, #0x00FF0000      r7 <- 0xFF00FFFF
```

# Charger/Sauvegarder instructions multiples

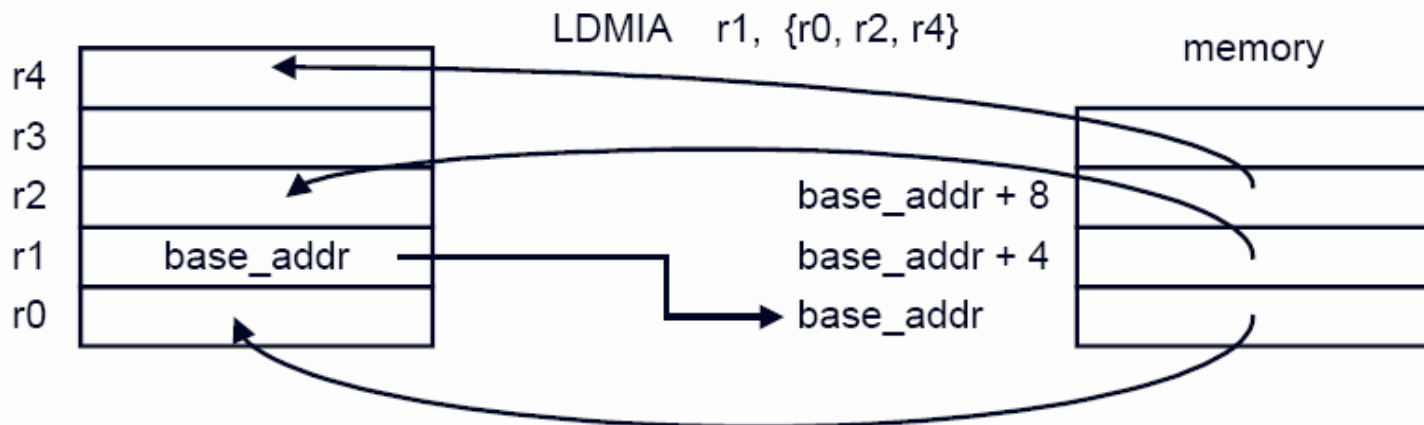
68

- Les instructions LDR et STR peuvent uniquement charger/sauvegarder un unique mot de 32 bits.
- ARM peut charger/sauvegarder n'importe quel sous-ensemble des 16 registres en une seule instruction. Par exemple:

**LDMIA r1, {r0, r2, r4} ; r0 := mem32[r1]**

**; r2 := mem32[r1+4]**

**; r4 := mem32[r1+8]**



# Charger/Sauvegarder instructions multiples

69

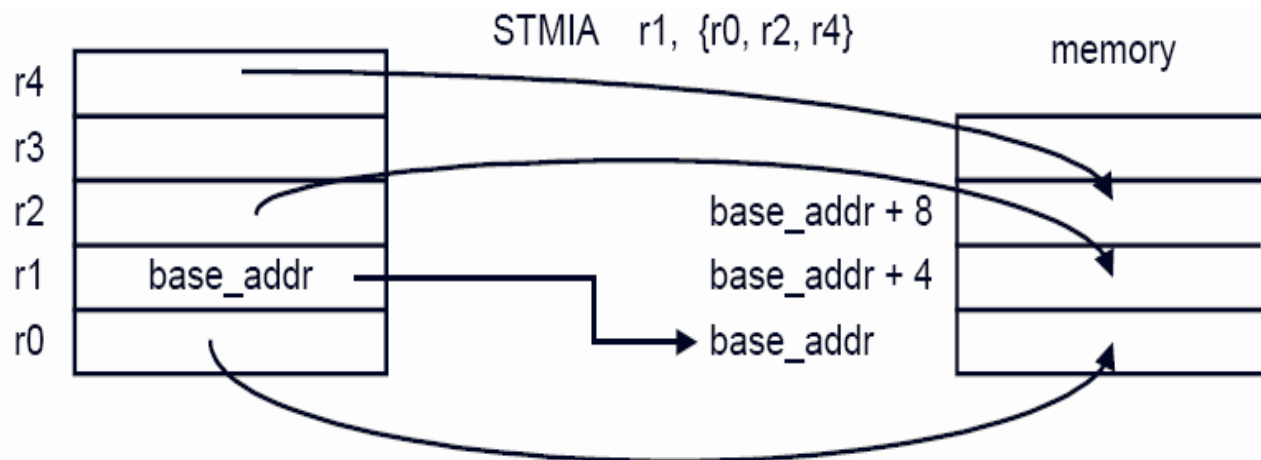
- N'importe quels registres peuvent être spécifiés. Cependant, il faut faire attention si vous incluez **r15 (PC)**, vous forcez un saut (branchement) dans votre programme.
- L'instruction complémentaire à LDMIA est l'instruction STMIA :

## Par exemple:

**STMIA r1, {r0, r2, r4} ; mem32[r1] := r0**

```
; mem32[r1+4] := r2
```

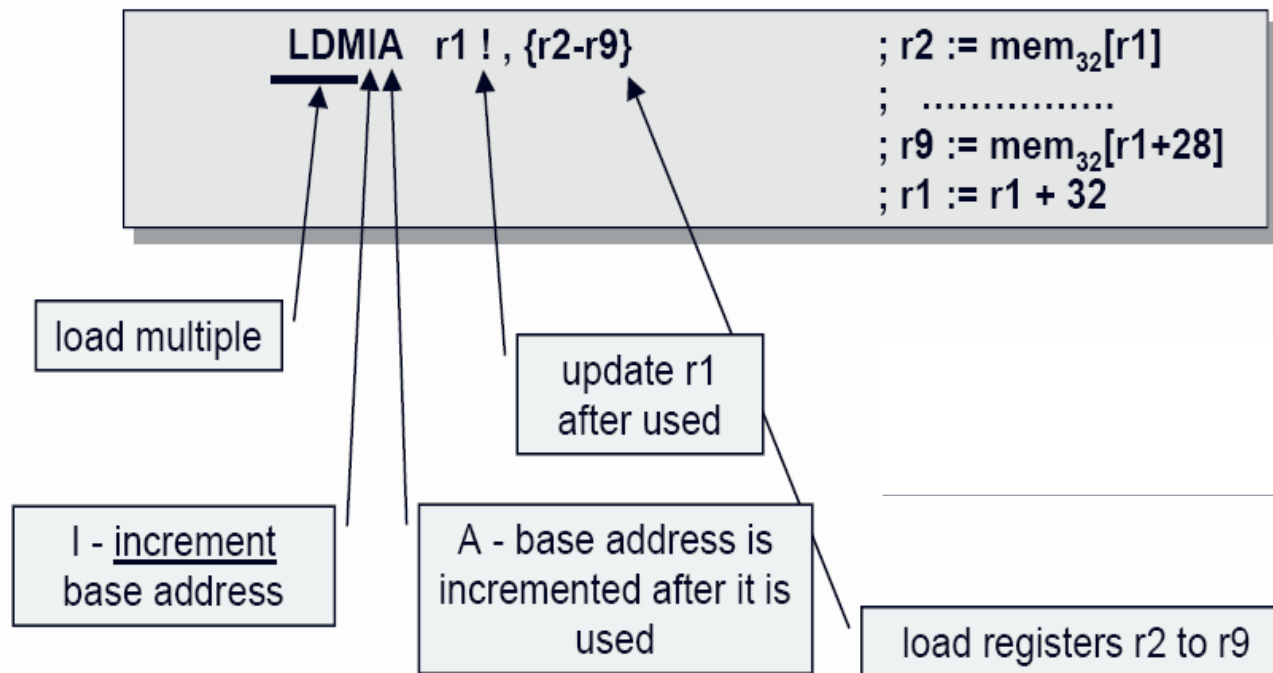
```
; mem32[r1+8] := r4
```



# Mise à jour de l'adresse de base dans le cas Chargt./Sauvegarde multiples

70

- Jusqu'ici, r1, le registre de base, n'a pas été changé. Vous pouvez mettre ce registre pointeur à jour en rajoutant **!** après r1 dans l'instruction :



# Les 4 variantes de STM

71

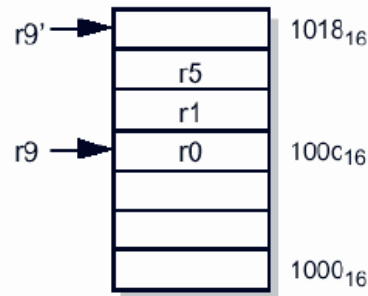
- Les registres avec des numéros élevés sont dans les adresses hautes

**IA** : Incrémenter Après (**A**fter)

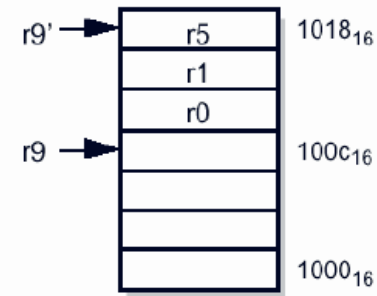
**IB** : Incrémenter Avant (**B**efore)

**DA** : Décrémenter Après (**A**fter)

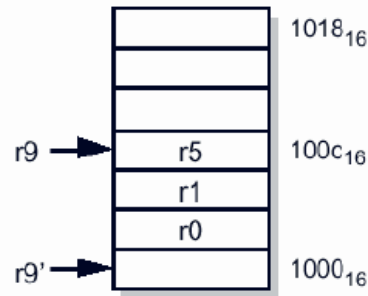
**DB** : Décrémenter Avant (**B**efore)



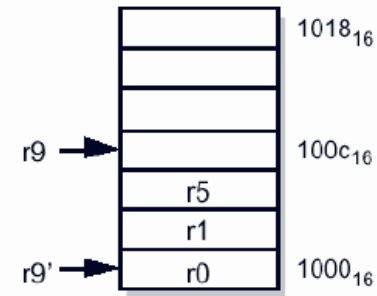
STMIA r9!, {r0,r1,r5}



STMIB r9!, {r0,r1,r5}



STMDA r9!, {r0,r1,r5}

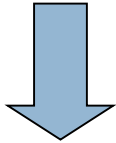


STMDB r9!, {r0,r1,r5}

# Si ... alors

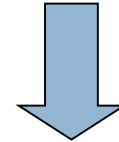
72

```
si condition  
alors corps_alors  
...
```



```
B_condition_inverse etq_suite_si  
instructions du corps_alors  
etq_suite_si  
...
```

```
si a ==0  
alors a=a+1  
b=a
```



```
tst R1 ; cmp a,#0  
bne finsi  
add R1, R1,#1; a+1  
finsi R2=R1; b=a
```



# Si ... alors

73

**; Si ( (a==b) && (c==d)) alors e := e + 1;**

**CMP r0, r1** ; r0 contient a, r1 contient b

**CMPEQ r2, r3** ; r2 contient c, r3 contient d

**ADDEQ r4, r4, #1** ; e := e+1

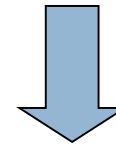
- Il faut remarquer que si la première comparaison trouve une inégalité, les deux instructions suivantes ne seront pas exécutées.
- Le « && » logique dans le « Si » est implémenté en mettant la deuxième comparaison conditionnelle. On aurait pu utiliser les branchements!

# Si ... alors ... sinon

74

```
si condition  
alors corps_alors  
sinon corps_sinon
```

```
si a ==0  
alors a=a+1  
sinon a=a+2  
b=a
```



```
tst R1  
bne sinon  
add R1, R1, #1  
b finisi  
sinon add R1,R1,#2  
finisi mov R1,R1
```

# Tant que

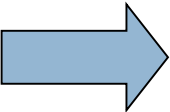
75

tant que <cond>  
faire  
corps tantque



**tantque:**  
si not <cond>  
    alors goto **fintantque**  
corps tantque  
    goto tantque  
**fintantque:** ...

tant que i>j  
faire  
    i=i-1

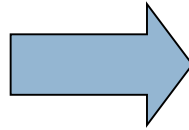


**tantque**  
    cmp    r1,r2  
    bls    **fintantque**; Branch.si ≤  
    sub    r1,r1,#1  
    b tantque  
**fintantque** ...

# Astuce : utilisation de l'extension S

76

```
tantque
    cmp r1,#0
    bls fintantque
    sub r1,r1,#1
    b tantque
fintantque ...
```



```
    cmp r1,#0
tantque
    bls fintantque
    subs r1,r1,#1
    b tantque
fintantque ...
```

```
    cmp r1,#0
    bls fintantque
tantque
    subs r1,r1,#1
    bgt tantque
fintantque ...
```

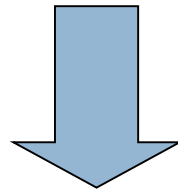
} Inutile si on est sûr  
que  $r1 > 0$

# Pour = tant que

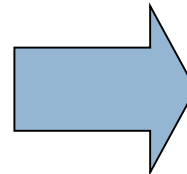
77

Pour i allant de 1 à N par pas de 1

...



i = 1  
Tant que i <= N  
...  
i = i + 1



Voir le  
transparent  
précédent !

# Somme des 10 premiers entiers

78

## □ *Algorithme* :

somme  $\leftarrow$  0;

pour i = 1 à 10 faire

    somme  $\leftarrow$  somme + i;

## □ *Programme* :

**AREA** somme, **CODE**, **READONLY** ; donne le nom du bloc  
**ENTRY**

Debut        **MOV**     r0, #0                ; r0 = somme

**MOV**     r1, #1                ; r1 = I

Boucl        **CMP**     r1, #10

**BHI**     fin        ; Branchement si supérieur

**ADD**     r0, r0, r1 ; somme  $\leftarrow$  somme + i

**ADD**     r1, r1, #1        ; i  $\leftarrow$  i + 1

**B**        boucl

Fin           **SWI**     0x123456 ; arrêt exécution program.

**END**                                ;Fin fichier

# L'idée de la Pile

79

Une pile (en anglais stack) est une structure de données (partie de la mémoire) fondée sur le principe « dernier arrivé, premier sorti » (ou LIFO pour Last in, First out),.

Propriétés intéressantes de la pile:

- Allocation et libération par un simple ajustement de pointeur.
- Gestion simple :
  - Push et Pop.

Définition : La **PILE** est une zone mémoire RAM gérée par des pointeurs qui permettent de transférer rapidement des données dans des cases mémoires selon un protocole bien établi.

# L'idée de la Pile

80

- Les instructions de load/store multiples peuvent être utilisées pour implémenter une **LIFO** appelée **Pile**.
- La Pile est une portion de la mémoire principale utilisée pour sauvegarder les données temporairement.
- L'opération **PUSH** permet de sauvegarder de nombreux registres dans la Pile .

**PUSH {r1, r3-r5, r14}**

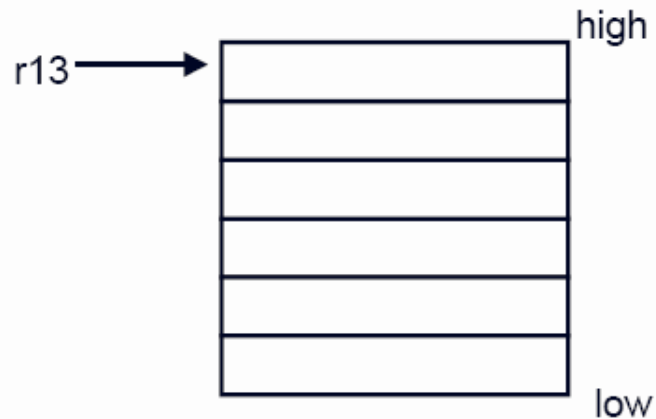


# L'idée de la Pile

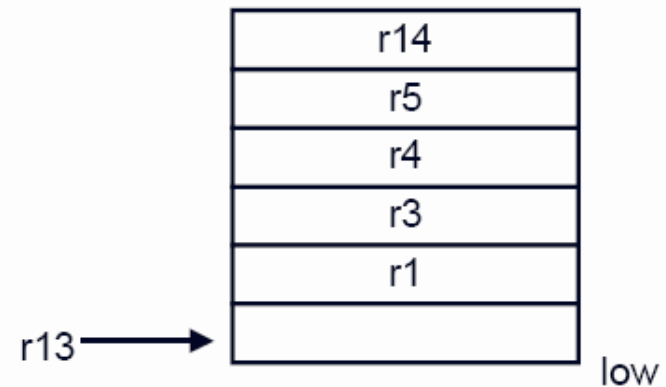
81

**PUSH {r1, r3-r5, r14}**

Pile avant PUSH



Pile après PUSH

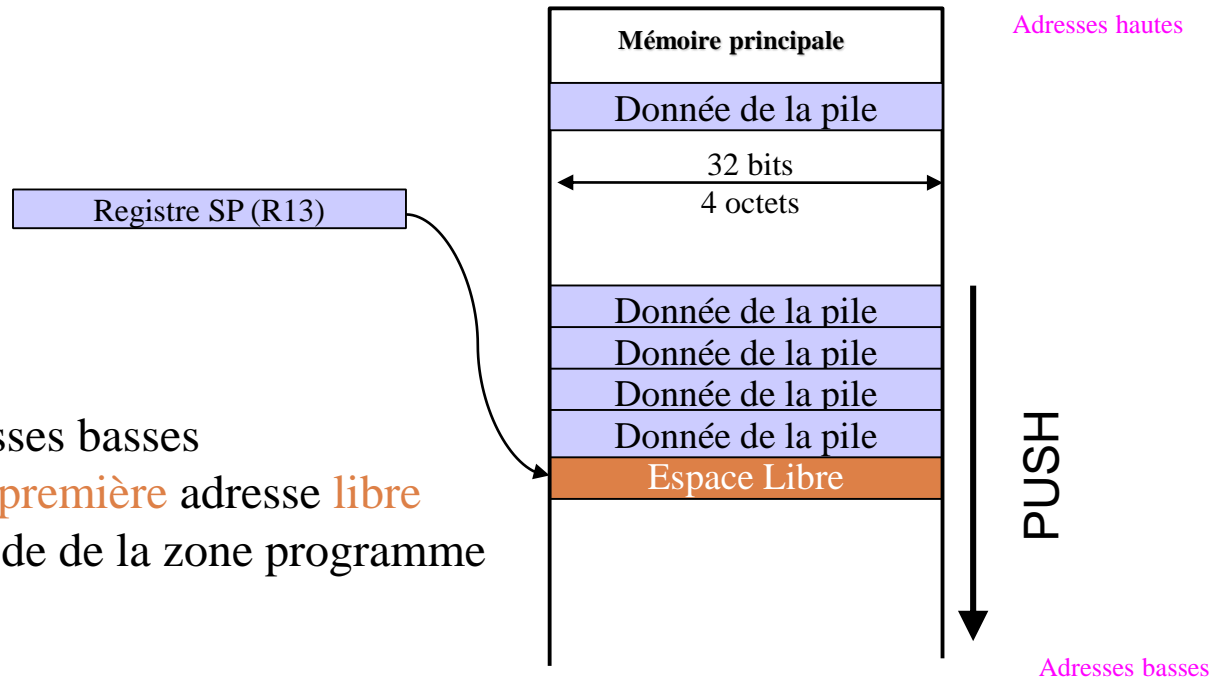


# Mise en œuvre de la Pile

82

## Conventions:

- Registre dédié :  
SP (Stack Pointer)
- Remplissage vers les adresses basses
- Registre **SP** pointe vers **la première** adresse **libre**
- Réservation mémoire loin de la zone programme



**Autre implémentation:**  
Remplissage vers les adresses Hautes.

# PUSH vers la Pile

83

Bien noter les propriétés de l'opération PUSH :

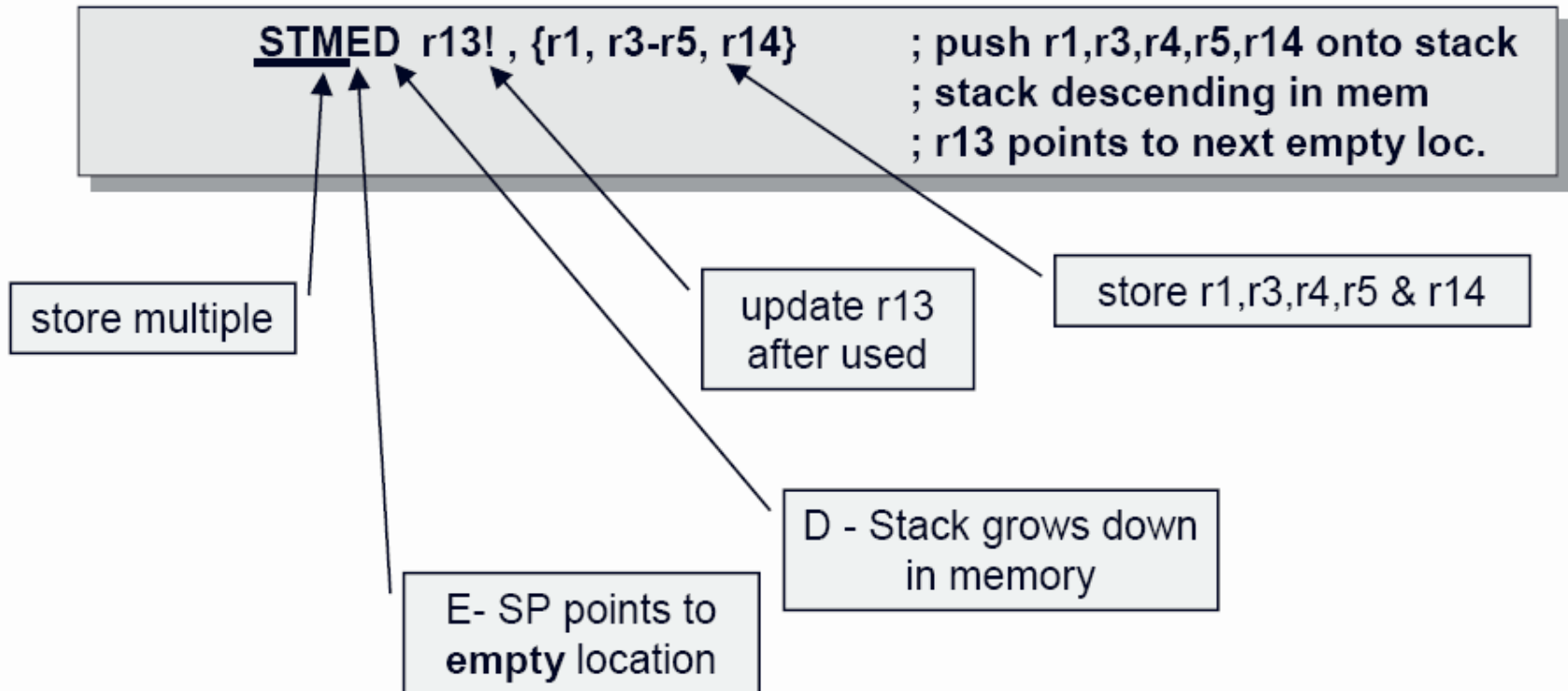
- **r13** est comme le pointeur d'adresses. On l'appelle **STACK POINTER (SP)**.
- La Pile évolue vers les adresses mémoire basses. Pour sauvegarder des valeurs dans la Pile, le Pointeur de Pile est **décrémenté** après chaque usage (cas du mode STMDA).
- ARM n'a pas l'instruction PUSH mais on peut utiliser l'une des instructions STM pour son implémentation.

**STMDA r13!, {r1, r3-r5, r14} ; Push r1, r3-r5, r14 vers Pile**  
**; Pile évolue vers le bas de Mém.**  
**; r13 pointe vers la place libre.**

# PUSH vers la Pile

84

Instruction équivalente : **STMED**

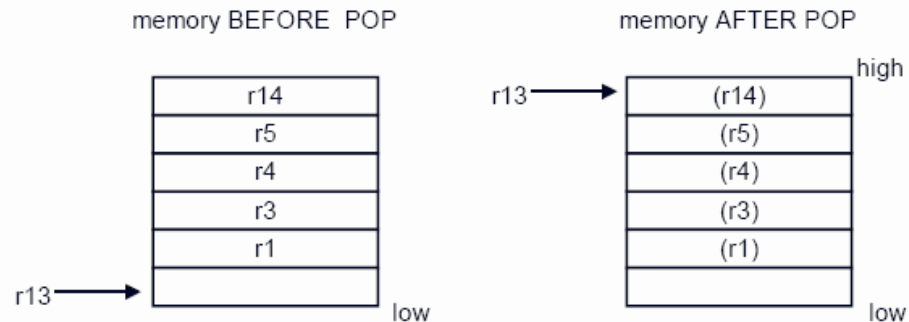


# L'opération POP

85

L'opération complémentaire de PUSH est l'opération **POP**.

**POP {r1, r3-r5, r14}**



# L'opération POP

86

- ARM ne dispose pas de l'instruction POP. On peut utiliser :

**LDMIB r13!, {r1, r 3-r5, r 14}** ; Pop r1, r3-r5, r 14 de la Pile

- Qui est équivalente à :

**LDMED r13!, {r1, r 3-r5, r 14}** ; Pop r1, r3-r5, r 14 de la Pile

# Manipuler la Pile

87

- 4 modèles de pile existent
- Le modèle **ascending/Descending** : on empile vers les adresses croissantes (**ascending** ) ou décroissantes (**Descending** )
- Le modèle **Full/Empty** : le pointeur de pile (SP) pointe le dernier élément rangé dans la pile (Full) ou bien vers la case libre (**Empty**)

# Manipuler la Pile

88

---

STMFA	r13!, {r0-r5};	Push onto a Full Ascending Stack
LDMFA	r13!, {r0-r5};	Pop from a Full Ascending Stack
STMFD	r13!, {r0-r5};	Push onto a Full Descending Stack
LDMFD	r13!, {r0-r5};	Pop from a Full Descending Stack
STMEA	r13!, {r0-r5};	Push onto an Empty Ascending Stack
LDMEA	r13!, {r0-r5};	Pop from an Empty Ascending Stack
STMED	r13!, {r0-r5};	Push onto Empty Descending Stack
LDMED	r13!, {r0-r5};	Pop from an Empty Descending Stack

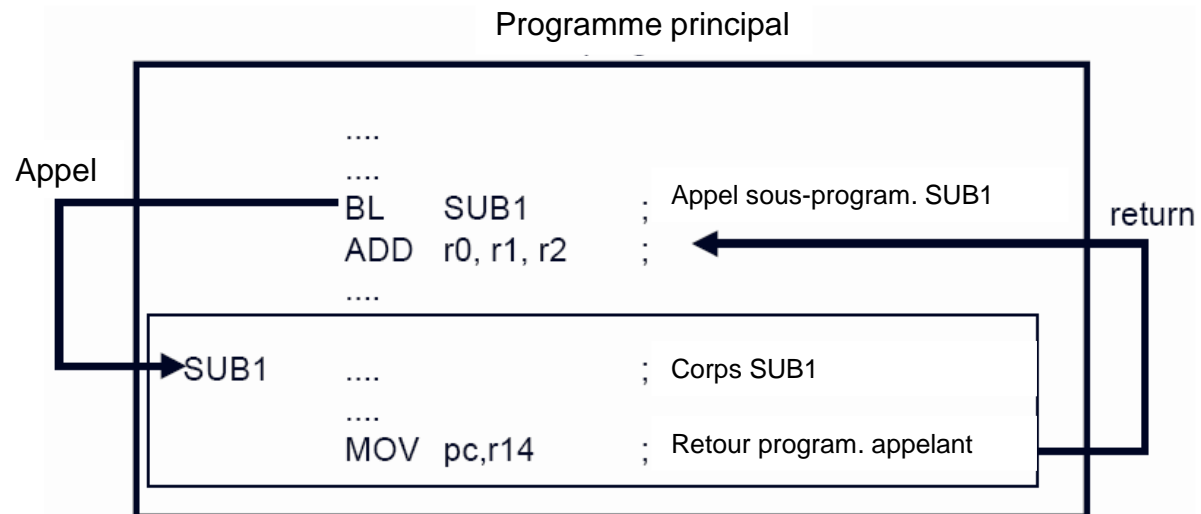
---



# Sous-programme

89

- ❑ Les sous-programmes nous permettent de rendre notre code plus modulaire et donc plus ré-utilisable et compréhensible.
- ❑ La structure générale d'un sous-programme dans un programme est :



# Sous-programme

90

- **BL nom\_sous\_pgme** (Branch-and-Link) est l'instruction de saut vers le sous-programme.

```
BL label      ; r14 ← pc - 4 ; pc ← label
```

Elle effectue les opérations suivantes :

- 1) Elle sauvegarde la valeur du **PC** (qui pointe vers l'instruction suivante) dans r14. C'est l'adresse du retour.
  - 2) Elle charge le **PC** avec l'adresse du sous-programme. Ce qui permet d'effectuer le saut (Branch).
- **BL** utilise toujours **r14** pour l'adresse de retour. **r14** est appelé **link-register** (peut être noté **lr** ou bien **r14**).
  - Le retour depuis le sous-programme est simple : remettre **r14** dans le **PC (r15)**.  
« **MOV PC, R14** ; pc ← r14 »

# Sous-programmes imbriqués

91

- ❑ Écrasement de R14 suite au second appel !
- ❑ Nécessité de sauvegarder R14 dans la pile avant le second appel au sous-programme.

# Sous-programmes imbriqués

92

**BL SUB1** ; saut vers un sous-programme 1

.....

**SUB1 STMED r13!, {r0-r2, r14}** ; Push les registres de travail & de link vers pile

....

**BL SUB2** ; saut vers un sous-programme imbriqué

...

**LDMED r13!, {r0-r2, r14}** ; POP les registres de travail & de link

**MOV pc, r14** ; Retour vers programme principal

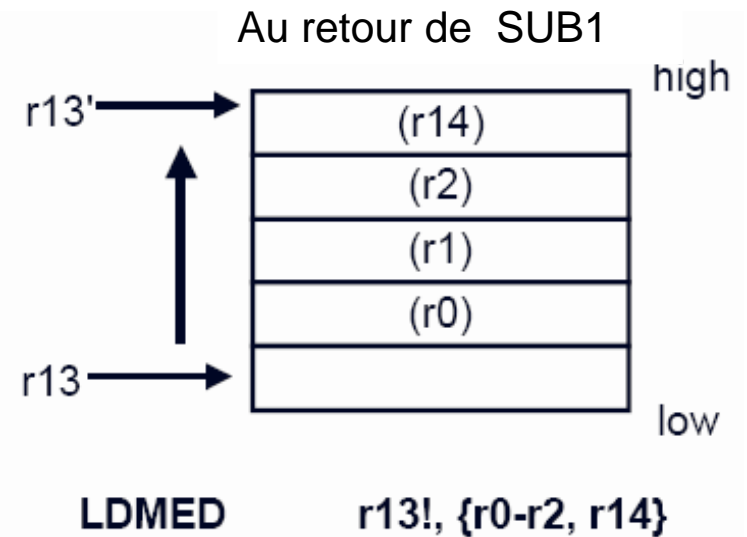
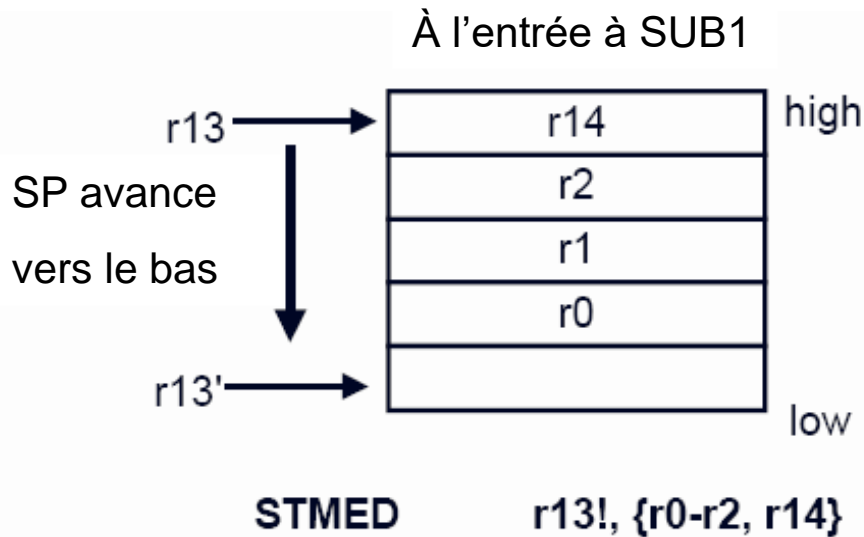
**SUB2** ...

...

**MOV pc, r14** ; Retour vers programme appelant **SUB1**

# Sous-programmes imbriqués

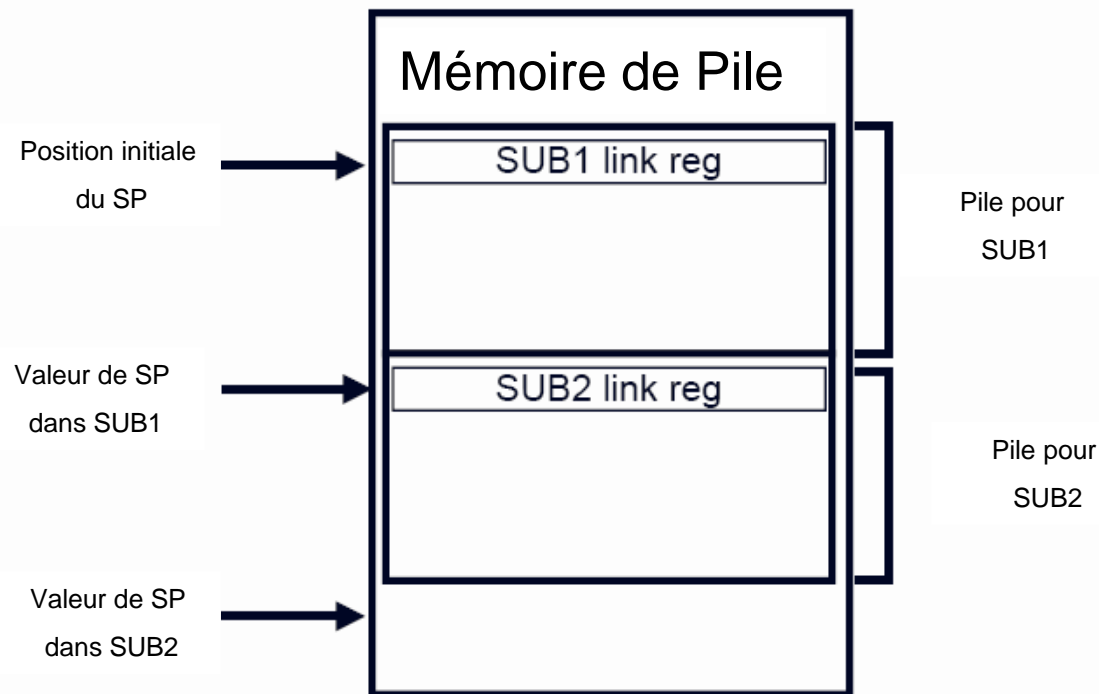
93



# Effets des sous-programmes imbriqués

94

**SUB1** appelle un autre sous programme **SUB2**. Supposant que **SUB2** sauvegarde aussi son registre de lien (r14) ainsi que ses registres de travail, la pile va ressembler à :



# Variables locales

95

- Les registres sont des variables globales
- Pour les utiliser comme des variables locales
  - ▣ Empiler leur contenu en début de sous-programme
  - ▣ Dépiler en fin de sous-programme

```
sp1    STMFA r13!, {r5, r14}      ; r13 est le SP
...
ici je fais ce que je veux avec r5
...
LDMFA r13!, {r5, r15}           ; r15=PC←r14
```

# Fonctions(retour de valeur)

96

- Utiliser une variable globale
  - ▣ registre libre ou une case mémoire ☹
  - ▣ registre : ne pas empiler et dépiler le registre au début et à la fin du sous-programme !
- Utiliser la pile
  - ▣ Dans le programme principal, réserver une case dans la pile en incrémentant le pointeur r13
  - ▣ Dans le sous-programme, écrire dans cette case
    - Attention, entre-temps, on a peut-être empilé des valeurs !
    - Calculer le déplacement
  - ▣ Dans le programme appelant, après l'appel, récupérer la valeur et décrémenter le pointeur de pile



# Fonctions(retour de valeur)

97

appelant

...

```
add r13,r13,#4 ; (1)
```

```
bl sp1
```

```
LDMFA r13!,{r7} ; r7←r1
```



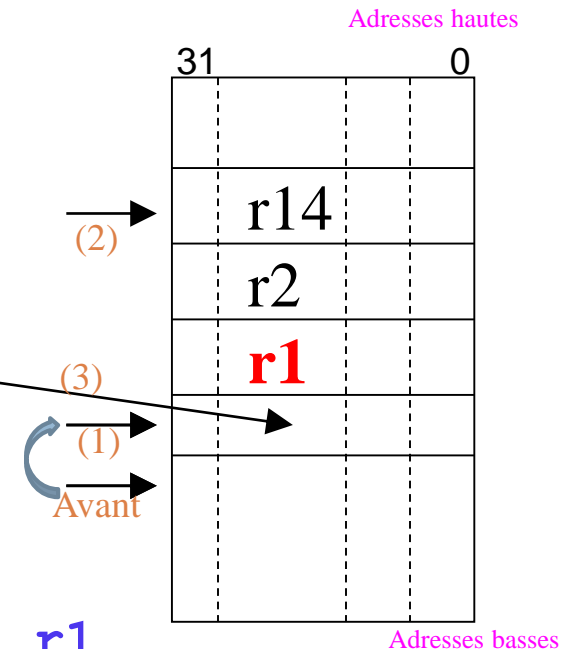
sp1

```
STMFA r13!,{r1,r2,r14}; (2)
```

ici, calcul de la nouvelle valeur de **r1**

```
STR r1,[r13,#-12] ; nb elts empiles * 4 (3)
```

```
LDMFA r13!,{r1,r2,r15} ; r15=PC←r14
```



# Passage de paramètres

98

- Utiliser une variable globale
  - ▣ registre libre ou une case mémoire ☹
- Par la pile
  - ▣ Dans le programme appelant, empiler les paramètres
  - ▣ Dans le programme appelé, les récupérer en utilisant le bon déplacement par rapport à r13

# Passage de paramètres par la pile

99

```
STR R13!, R4;  passage du paramètre (1)
```

```
BL  sp1
```

```
SUB R13, R13, #4
```

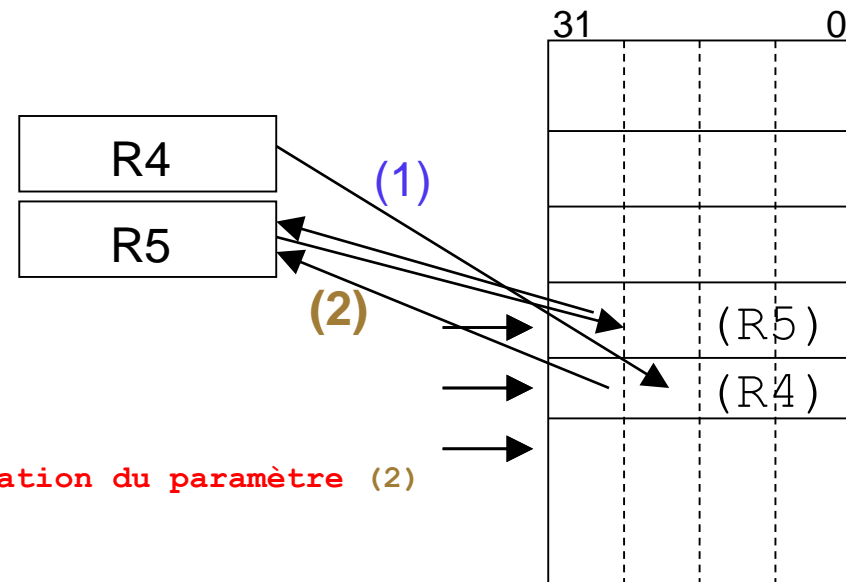
```
sp1
```

```
STMFA    R13!, R5
```

```
LDR      R5, [R13, #-4];  récupération du paramètre (2)
```

```
...
```

```
LDMFA    R13!, R5
```



Astuce : l'emplacement du paramètre peut être utilisé pour renvoyer une valeur...