

# L'assembleur ARM d'IAR

# Généralités

# Commentaires et étiquettes

L'assembler ARM d'IAR gère plusieurs types de commentaires :

Les commentaires sur plusieurs lignes : `/* ... */`

Les commentaires de fin de ligne : `// ...` ou `; ...`

Une étiquette est un identificateur qui symbolise l'adresse d'une instruction ou d'une variable.

Exemple

<code>var1</code>	<code>DC8</code>	<code>'c'</code>	<code>; ici var1 est une étiquette</code> <code>; qui désigne une variable</code>
-------------------	------------------	------------------	--

# Les données

# Les valeurs immédiates

---

- Les valeurs immédiates sont des valeurs numériques, des caractères ou des chaînes de caractères définies lors de l'écriture du programme.
- Elles peuvent être placées directement à l'intérieur du code assembleur ou définies à l'aide de pseudo-instructions.

# Les valeurs immédiates

## Exemples

- Nombre binaire : 10110011b
- Nombre hexadécimal : 0A67h ou 0xA67
- Nombre décimal : 2009d ou 2009
- Nombre octal : 3731q
- Caractère : 'C'
- Chaîne de caractères :
  - 'ABCD' : chaîne de 4 caractères
  - "ABCD" : chaîne de 5 caractères ('A', 'B', 'C', 'D' et '\0')

Attention : lorsqu'un nombre hexadécimal commence par une lettre il faut le faire précéder d'un 0 !!!

# Les constantes

- L'assembleur d'IAR ne permet de définir que des constantes contenant des valeurs entières ou des caractères (pas de chaînes de caractères).
- Pour définir une constante on utilise la pseudo instruction EQU.

## Exemples

constante1	EQU	0A12Ch
constante2	EQU	'C'

# Les variables

- Une variable correspond à un emplacement mémoire.
- Cet emplacement est réservé lors de l'assemblage.
- Une variable peut être modifiée durant l'exécution du programme.
- Pour déclarer une variable on utilise une directive qui précise la taille de la zone mémoire dont on a besoin.



# La directive DC8

Cette directive réserve et initialise un ou plusieurs emplacements mémoire d'un octet (8 bits) contenant

- des valeurs numériques comprises entre 0 et 255.
- des caractères.

## Exemples

var1	DC8	'c'	
ch	DC8	"Bonjour"	; déclaration d'une chaîne de caractère
var3	DC8	0x1F	
tab	DC8	1,2,3,4	; déclaration d'un tableau de 4 octets ; initialisés avec les valeurs 1, 2, 3 et 4

# La directive DC16

Cette directive réserve et initialise un ou plusieurs emplacements mémoire d'un demi-mot (2 octets) ne contenant que des valeurs numériques.

## Exemples

```
var1    DC16  0x1234
tab     DC16  1,2,3,4 ; déclaration d'un tableau de 4 demi-
                ; mots de 16 bits initialisés avec les
                ; valeurs 1, 2, 3 et 4
```

# La directive DC32

Cette directive réserve et initialise un ou plusieurs emplacements mémoire d'un mot de 32 bits ne contenant que des valeurs numériques.

## Exemples

```
var1    DC32    0x12345678
tab     DC32    1,2,3,4 ; déclaration d'un tableau de 4 mots
                    ; de 32 bits initialisés avec les
                    ; valeurs 1, 2, 3 et 4
```

# Autres directives

DS8, DS16 et DS32 réservent un ou plusieurs emplacements mémoire de 8, 16 et 32 bits non initialisés.

## Exemple

tab	DS32	10	; déclaration d'un tableau de 10 mots ; de 32 bits non initialisés
-----	------	----	---

# Alignement

Les adresses des mots doivent être des multiples de 4  
et celles des demi-mots des multiples de 2 !!!

Les directives ALIGNROM et ALIGNRAM forcent  
l'alignement d'un variable en mémoire.

```
var1    DC8    'A'  
        ALIGNRAM 2 ; 1 -> adresse variable suivante multiple de 2  
        ; 2 -> adresse variable suivante multiple de 4  
tab     DC32   1,2,3,4
```

# Les instructions

# Les instructions

## Généralités

# Introduction

---

- L'ARM7TDMI dispose
  - de deux jeux d'instructions : 32 Bits (mode ARM) et 16 Bits (mode Thumb)
  - de 7 modes opératoires associés à des ensembles de registres différents.
- Dans ce cours nous étudierons seulement le jeu d'instructions 32 bits en mode utilisateur.



# Jeux d'instructions ARM

Data processing immediate	cond	0	0	1	op				S	Rn		Rd		rotate		immediate																	
Data processing immediate shift	cond	0	0	0	opcode				S	Rn		Rd		shift immediate				shift	0	Rm													
Data processing register shift	cond	0	0	0	opcode				S	Rn		Rd		Rs		0	shift	1	Rm														
Multiply	cond	0	0	0	0	0	0	A	S	Rd		Rn		Rs		1	0	0	1	Rm													
Multiply long	cond	0	0	0	0	1	U	A	S	RdHi		RdLo		Rn		1	0	0	1	Rm													
Move from status register	cond	0	0	0	1	0	R	0	0	SBO		Rd		SBZ																			
Move immediate to status register	cond	0	0	1	1	0	R	1	0	Mask		SBO		rotate		immediate																	
Move register to status register	cond	0	0	0	1	0	R	1	0	Mask		SBO		SBZ					0	Rm													
Branch/exchange instruction set	cond	0	0	0	1	0	0	1	0	SBO		SBO		SBO		0	0	0	1	Rm													
Load/store immediate offset	cond	0	1	0	P	U	B	W	L	Rn		Rd		immediate																			
Load/store register offset	cond	0	1	1	P	U	B	W	L	Rn		Rd		shift immediate				shift	0	Rm													
Load/store halfword/signed byte	cond	0	0	0	P	U	1	W	L	Rn		Rd		High offset		1	S	H	1	Low offset													
Load/store halfword/signed byte	cond	0	0	0	P	U	0	W	L	Rn		Rd		SBZ		1	S	H	1	Rm													
Swap/swap byte	cond	0	0	0	1	0	B	0	0	Rn		Rd		SBZ		1	0	0	1	Rm													
Load/store multiple	cond	1	0	0	P	U	S	W	L	Rn		Register list																					
Coprocessor data processing	cond	1	1	1	0	op1				CRn		CRd		cp_num		op2		0	CRm														
Coprocessor register transfers	cond	1	1	1	0	op1			L	CRn		Rd		cp_num		op2		1	CRm														
Coprocessor load and store	cond	1	1	0	P	U	N	W	L	Rn		CRd		cp_num		8_bit_offset																	
Branch and branch with link	cond	1	0	1	L	24_bit_offset																											
Software interrupt	cond	1	1	1	1	swi_number																											
Undefined	cond	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x		

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

# Exécution conditionnelle

---

- Contrairement aux processeurs qui ne permettent d'exécuter conditionnellement que les instructions de sauts (branchements), toutes les instructions de l'ARM sont exécutables de manière conditionnelle.
- Cette caractéristique permet d'écrire des programmes compacts et rapides.

# Exécution conditionnelle

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-

# Exécution conditionnelle

Pour exécuter une instruction de manière conditionnelle, il suffit de la compléter avec l'extension appropriée traduisant la condition.

## Exemple

- Une instruction d'addition prend la forme suivante :

**ADD** r0, r1, r2 ;  $r0 = R1 + R2$  (ADDAL)

- Pour ne l'exécuter que si  $Z=1$  :

**ADDEQ** r0, r1, r2 ;  $r0 = r1 + r2$  si  $Z=1$

# Exécution conditionnelle

- Par défaut, les opérations de traitement des données n'ont pas d'incidence sur l'état des indicateurs mis à part les comparaisons.
- Pour que l'exécution d'une instruction entraîne la mise à jour de ces indicateurs, elle doit être complétée avec un "S".

## Exemple

ADDS r0, r1, r2 ;  $r0 = r1 + r2$

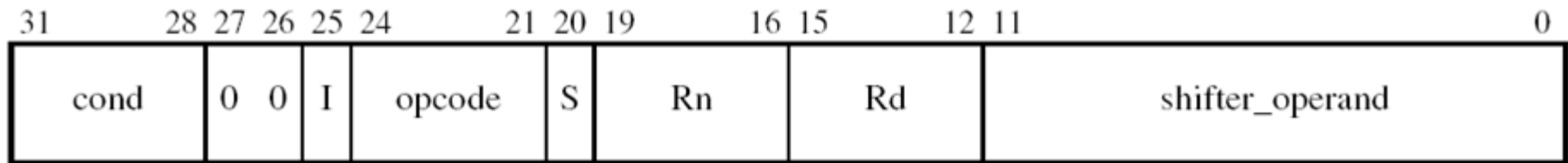
; avec mise à jour des indicateurs

# Les instructions

Les instructions de traitement de  
données

# Instructions de traitement des données

C'est la plus grande famille d'instructions ARM.  
Toutes ces instructions partagent le même format :



- I bit** Distinguishes between the immediate and register forms of <shifter\_operand>.
- S bit** Signifies that the instruction updates the condition codes.
- Rn** Specifies the first source operand register.
- Rd** Specifies the destination register.
- shifter\_operand** Specifies the second source operand.

# Instructions de traitement des données

Cette famille d'instruction regroupe :

- Les opérations arithmétiques et logiques
- Les comparaisons, pas de résultat – indicateurs mis à jour
- Les transferts de données entre les registres

## Attention

- Seules Load et Store peuvent accéder à la mémoire.
- Les instructions de traitement des données ne fonctionnent donc que sur les registres et les constantes.
- Ces instructions manipulent un ou deux opérandes sources  
le premier est toujours un registre  $R_n$ , le deuxième (shifter\_operand) est une valeur immédiate ou un registre.



# Les opérations arithmétiques

- ADD Addition

$$(Rd) := (Rn) + (\text{shifter\_operand})$$

- ADC Addition avec retenue

$$(Rd) := (Rn) + (\text{shifter\_operand}) + c$$

- SUB Soustraction

$$(Rd) := (Rn) - (\text{shifter\_operand})$$

- SBC Soustraction avec retenue

$$(Rd) := (Rn) - (\text{shifter\_operand}) + c - 1$$

- RSB Soustraction inverse

$$(Rd) := (\text{shifter\_operand}) - (Rn)$$

- RSC Soustraction inverse avec retenue

$$(Rd) := (\text{shifter\_operand}) - (Rn) + c - 1$$

# Les opérations arithmétiques

---

## Syntaxe

<Operation> {<cond>} {S} Rd, Rn, shifter\_operand

## Exemples

ADD r0, r1, r2

SUBGT r3, r3, #1

RSBLES r4, r5, #5

# Les opérations logiques

- AND ET :  $(Rd) := (Rn) \text{ AND } (\text{shifter\_operand})$
- ORR OU :  $(Rd) := (Rn) \text{ OR } (\text{shifter\_operand})$
- EOR OU exclusif :  $(Rd) := (Rn) \text{ EOR } (\text{shifter\_operand})$
- BIC Bit clear :  $(Rd) := (Rn) \text{ AND NOT}(\text{shifter\_operand})$

Syntaxe :  $\langle \text{Operation} \rangle \{ \langle \text{cond} \rangle \} \{ S \} Rd, Rn, \text{shifter\_operand}$

## Exemples

AND r0, r1, r2

BICEQ r2, r3, #7

EORS r1, r3, r0

# Les comparaisons

Les opérations de comparaison mettent à jour les indicateurs conditionnels NZCV, résultat ignoré.

- CMP comparaison :  $R_n - \text{shifter\_operand}$
- CMN comparaison négative :  $R_n + \text{shifter\_operand}$
- TST test avec ET :  $R_n \text{ AND shifter\_operand}$
- TEQ test avec Ou exclusif :  $R_n \text{ EOR shifter\_operand}$

Syntaxe :  $\langle \text{Operation} \rangle \{ \langle \text{cond} \rangle \} R_n, \text{shifter\_operand}$

## Exemples

CMP r0, r1

TSTEQ r2, #5

# Les transferts de données entre registres

Les opérations de transferts entre registres sont

- MOV :  $(Rd) := (\text{shifter\_operand})$
- MVN :  $(Rd) := \text{NOT}(\text{shifter\_operand})$

Syntaxe :

$\langle \text{Operation} \rangle \{ \langle \text{cond} \rangle \} \{ S \} Rd, \text{shifter\_operand}$

Exemples

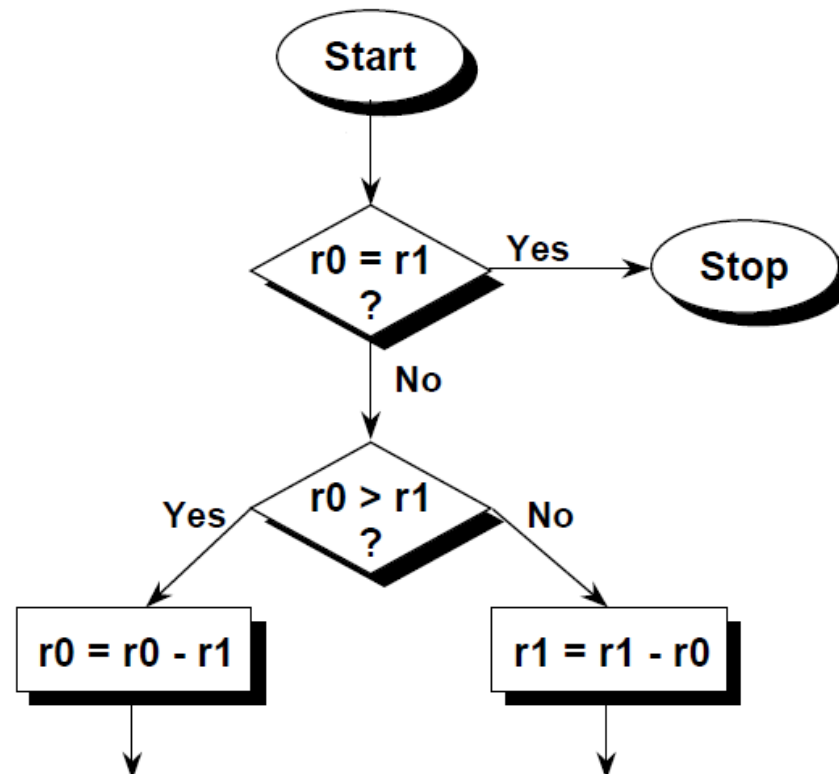
MOV r0, r1

MOVS r2, #10

MVNEQ r1, #0

# Exercice

Ecrire le programme assembleur correspondant au fragment d'organigramme suivant

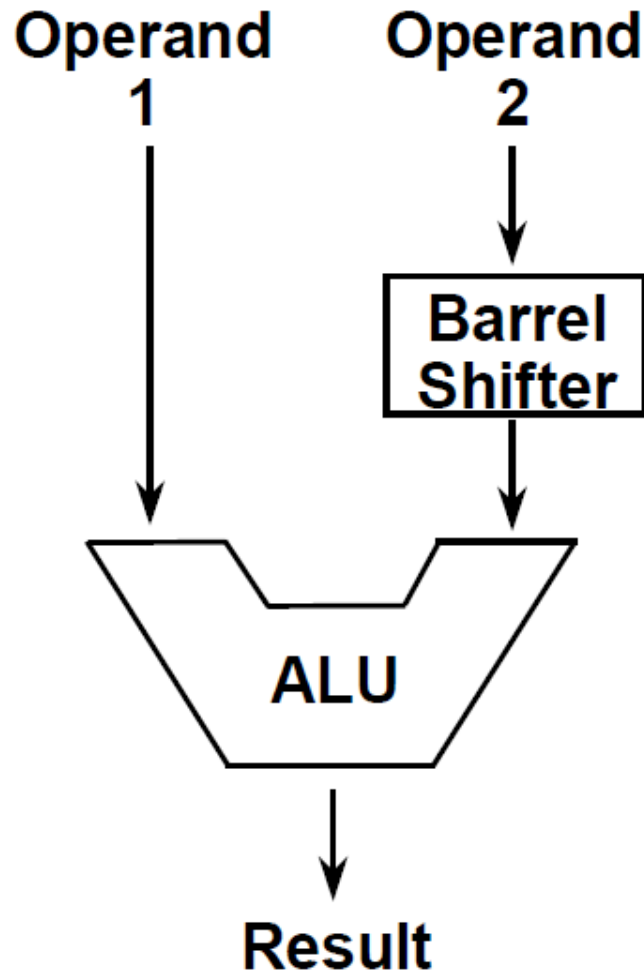


# Exercice

## Solution

```
cmp    r0, r1    ; (r0) - (r1)
beq    EStop     ; branche si (r0) = (r1)
subgt  r0, r0, r1 ; if (r0) > (r1) subtract r1 from r0
sublt  r1, r1, r0 ; else subtract r0 from r1
...
EStop: ...
```

# Utilisation avancée du shifter\_operand



Avant d'atteindre l'ALU, le shifter\_operand passe par le Barrel Shifter et peut donc subir un décalage ou une rotation.



# Utilisation avancée du shifter\_operand

- Lorsque le shifter\_operand est un registre, le nombre de bits sur lequel porte le décalage ou la rotation peut être stocké :
  - Dans une valeur immédiate, 5 bits codés directement dans l'instruction
  - Dans les 5 bits de poids faible d'un registre (pas PC)
- Si aucun décalage ou aucune rotation n'est précisé, le décalage par défaut est appliqué LSL #0 (la valeur du registre n'est pas affectée)

# Utilisation avancée de la shifter\_operand

## Exemples

`; r0 = r1 * 5`

`; r0 = r1 + (r1 * 4)`

`ADD r0, r1, r1, LSL #2`

`; r2 = r3 * 105 ;    r2 = r3 * 15 * 7`

`; r2 = r3 * (16 - 1) * (8 - 1)`

`RSB r2, r3, r3, LSL #4 ; r2 = r3 * 15`

`RSB r2, r2, r2, LSL #3 ; r2 = r2 * 7`

# Utilisation avancée du shifter\_operand

Lorsque le shifter\_operand est une valeur immédiate, cette dernière est stockée directement dans l'instruction :

- Sur 12 bits lorsqu'elle est utilisée directement, d'où une plage de 0 à 4096
- Sur 8 bits lorsqu'elle est utilisée conjointement avec une rotation à droite d'un nombre de bits pair 0, 2, 4, 6, 8, ..., 30

# Utilisation avancée du shifter\_operand

On obtient donc les plages de valeurs suivantes :

0-255 [0-0xFF] sur 8 bits et 0 rotation

256, 260..., 1020 [0x100-0x3FC] = [0x40-0xFF] ROR #30

1024, 1040..., 4080 [0x400-0xFF0] = [0x40-0xFF] ROR #28

4096, 4160..., 16320 [0x1000-0x3FC0] = [0x40-0xFF] ROR #26

0x3FC:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0xFF:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0xFF ror #1

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

..... 0xFF ror #30

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Utilisation avancée de la shifter\_operand

## Exemples

mov R1,#40,26

mov R1, #0x1000 ; converti automatiquement en  
; mov R1,#40,26

mov R1, #4096 ; converti automatiquement en  
; mov R1,#40,26

mov R2, #0x11111111 ; génère une erreur !!!

**Attention : on ne peut donc pas utiliser n'importe qu'elle valeur immédiate avec les instructions MOV et MVN !!!**

# Comment charger une constante 32 bits quelconque ?

- Les instructions MOV et MVN opèrent sur des plages de valeurs limitées.
- Pour charger dans un registre une valeur immédiate hors des plages supportées par MOV et MVN, il faut utiliser l'instruction :

LDR rd,=numeric constant

# Comment charger une constante 32 bits quelconque ?

Cette instruction stocke les valeurs hors des plages supportées par MOV et MVN dans une zone de mémoire particulière le « lit pool » puis opère grâce à un déplacement relatif à partir de PC :

LDR r0, =0x42 ; génère MOV r0,#0x42

LDR r0, =0x55555555

; génère LDR r0,[pc, offset to lit pool]

# Les instructions

## Les instructions de multiplication



# Les multiplications 32 bits

L'assembleur ARM dispose de deux instructions de multiplication de base

La multiplication

MUL {<cond>} {S} Rd, Rm, Rs  
; (Rd) := (Rm) \* (Rs)

La multiplication avec accumulation

MLA {<cond>} {S} Rd, Rm, Rs, Rn  
; (Rd) := (Rm) \* (Rs) + (Rn)

# Les multiplications 32 bits

---

Les restrictions d'utilisation sont

- Rd et Rm doivent être différents
- Impossibilité d'utiliser PC.
- Si le résultat est codé sur plus de 32 bits, il est tronqué.

Les opérandes sont signées ou non signées, à la charge de l'utilisateur de les interpréter correctement.

# Les multiplications étendues 64 bits

Ces instructions sont

- MULL réalise :  $(RdHi, RdLo) := (Rm * (Rs))$
- MLAL réalise :  $(RdHi, RdLo) := (Rm) * (Rs) + (RdHi, RdLo)$

Le fait d'avoir un résultat codé sur 64 bits nécessite de préciser si les données sont signées ou pas.

- UMULL{<cond>} {S} RdLo, RdHi, Rm, Rs ; Unsigned
- UMLAL{<cond>} {S} RdLo, RdHi, Rm, Rs
- SMULL{<cond>} {S} RdLo, RdHi, Rm, Rs ; Signed
- SMLAL{<cond>} {S} RdLo, RdHi, Rm, Rs

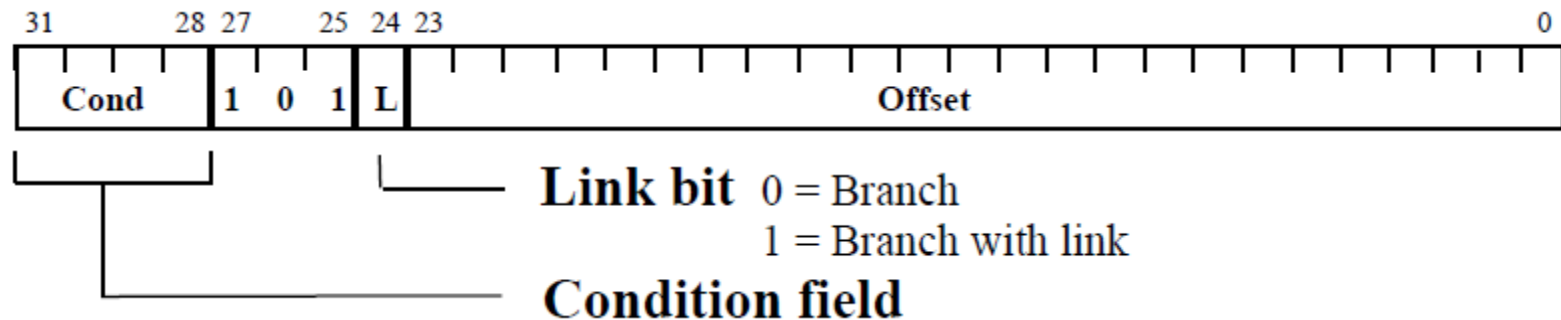
# Les instructions

## Les branchements simples

# Syntaxe

L'instruction de branchement simple à une adresse statique est

$B\{<cond>\} \text{ label}$



# Fonctionnement

L'offset du branchement est calculé par l'assembleur de la manière suivante :

- Il fait la différence entre l'instruction de branchement et l'adresse cible moins 8 à cause du pipeline.
- Il obtient un offset sur 26 bits qui est décalé de 2 bits à droite (les instruction étant alignées sur des mots, les 2 bits de poids faibles sont toujours à zéro) puis stocké dans l'instruction.

Cela donne une amplitude de saut de +/-32 Mo.

# Fonctionnement

---

Quand le processeur exécute l'instruction :

- Il décale l'offset de 2 bits à gauche, l'étend à 32 bits en conservant le signe puis l'ajoute au PC
- L'exécution se poursuit à partir du nouveau PC, une fois le pipeline rempli

# Les instructions

## Les instructions Load / Store



# Généralités

- Les instructions **load** et **store** sont chargées des transferts entre la mémoire et les registres et ce sont les seules instructions (avec swap) qui peuvent travailler avec la mémoire.
- L'architecture ARM distingue plusieurs types de transferts selon le type de données transférées.
- A chaque type sont associés des modes d'adressage particuliers.
- Cette complication relève des modifications apportées peu à peu à l'architecture ARM.

# Transferts de mots et de bytes non signés

Les transferts de mots et de bytes non signés sont

- Load and Store Register Word : LDR / STR
- Load and Store Register Byte : LDRB / STRB

Elles peuvent être exécutées de manière conditionnelle.

Exemple : LDREQB

Syntaxe

<LDR|STR> {<cond>} {<size>} Rd, <addressing\_mode>

Pour store, <Rd> contient la donnée à mettre en mémoire.

# Modes d'adressages des instructions

## LDR, STR, LDRB et STRB

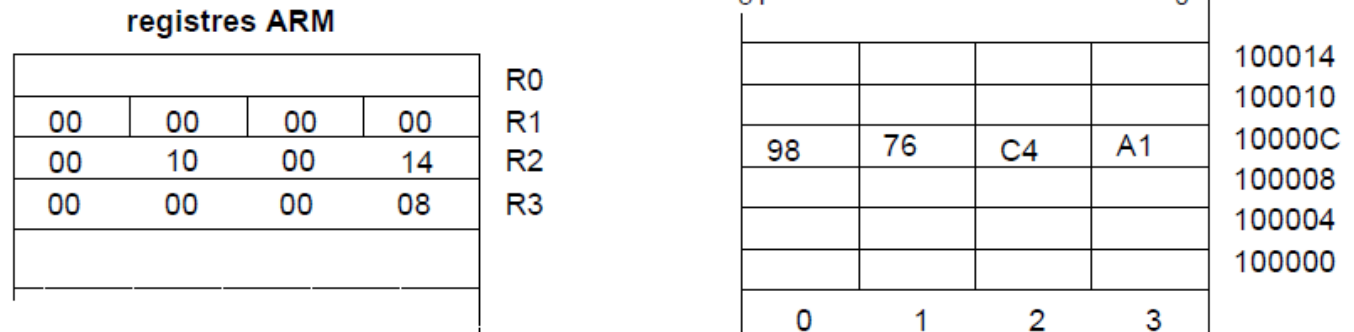
<addressing_mode>	type d'adressage	exemples
[ <Rn>, #+/- <offset_12> ]	registre indirect avec offset signé sur 12 bits <Rn> est inchangé	LDR R1, [ R2, #-24 ] adresse mémoire : R2 - 24
[ <Rn>, +/- <Rm> ]	registre indirect avec offset signé dans un registre <Rn> est inchangé	LDR R1, [ R2, -R3 ] adresse mémoire : R2 - R3
[ <Rn>, +/- Rm, <shift> #shift_imm ]	registre indirect avec offset calculé à partir d'un registre dont le contenu est décalé d'un nombre de bits indiqué en valeur immédiate <Rn> est inchangé	LDR R1, [ R2, R3, LSL #2 ] adresse mémoire : $R2 + R3 * 2^2$
[ <Rn>, #+/- <offset> ] ! pré indexé	registre indirect avec offset signé sur 12 bits mise à jour de <Rn>	LDR R1, [ R2, # 24 ] ! adresse mémoire : R2 + 24 R2 := R2 + 24
[ <Rn>, +/- <Rm> ] ! pré indexé	registre indirect avec offset signé dans un registre mise à jour de <Rn>	LDR R1, [ R2, -R3 ] ! adresse mémoire : R2 - R3 R2 := R2 - R3
[ <Rn>, +/- Rm, <shift> #shift_imm ] ! pré indexé	registre indirect avec offset calculé à partir d'un registre dont le contenu est décalé d'un nombre de bits indiqué en valeur immédiate mise à jour de <Rn>	LDR R1, [ R2, R3, LSL #2 ] adresse mémoire : $R2 + R3 * 2^2$ R2 := R2 + R3 * 2 <sup>2</sup>
[ <Rn> ], #+/- <offset_12> post indexé	registre indirect mise à jour de <Rn> avec l'offset signé sur 12 bits après le transfert	LDR R1, [ R2 ], # 24 adresse mémoire : R2 R2 := R2 + 24
[ <Rn> ], +/- <Rm> post indexé	registre indirect mise à jour de <Rn> avec le contenu signé de <Rm> après le transfert	LDR R1, [ R2 ], -R3 adresse mémoire : R2 R2 := R2 - R3
[ <Rn> ], +/- Rm, <shift> #shift_imm post indexé	registre indirect mise à jour de <Rn> après le transfert avec offset calculé à partir d'un registre dont le contenu est décalé d'un nombre de bits indiqué en valeur immédiate	LDR R1, [ R2 ], R3, LSL #2 adresse mémoire : R2 R2 := R2 + R3 * 2 <sup>2</sup>

<shift> : peut utiliser LSL, LSR, ASR, ROR, RRX

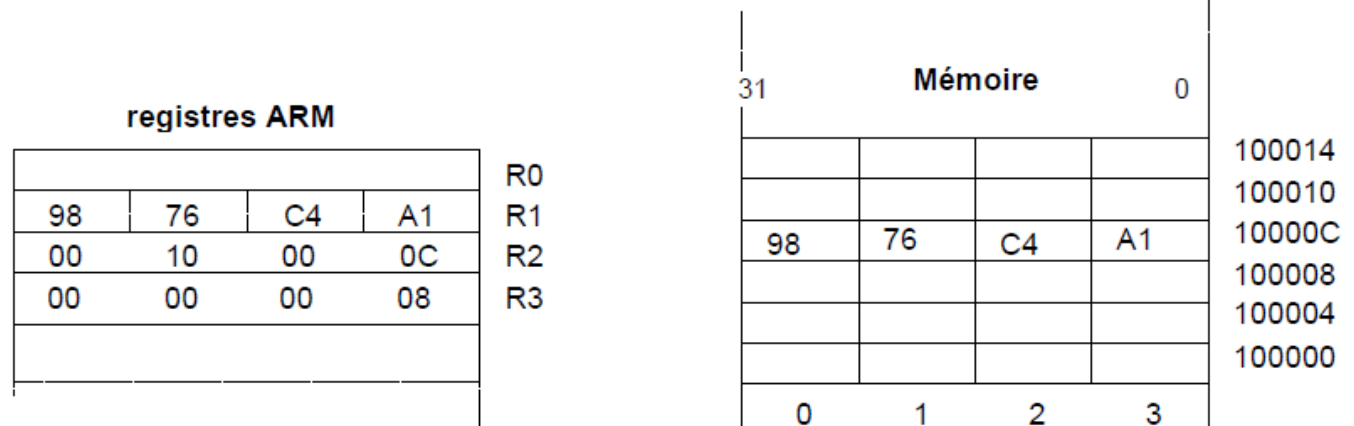
# Transferts de mots et de bytes non signés

## Exemple 1

LDR R1, [R2, - R3]!



Avant l'exécution de l'instruction

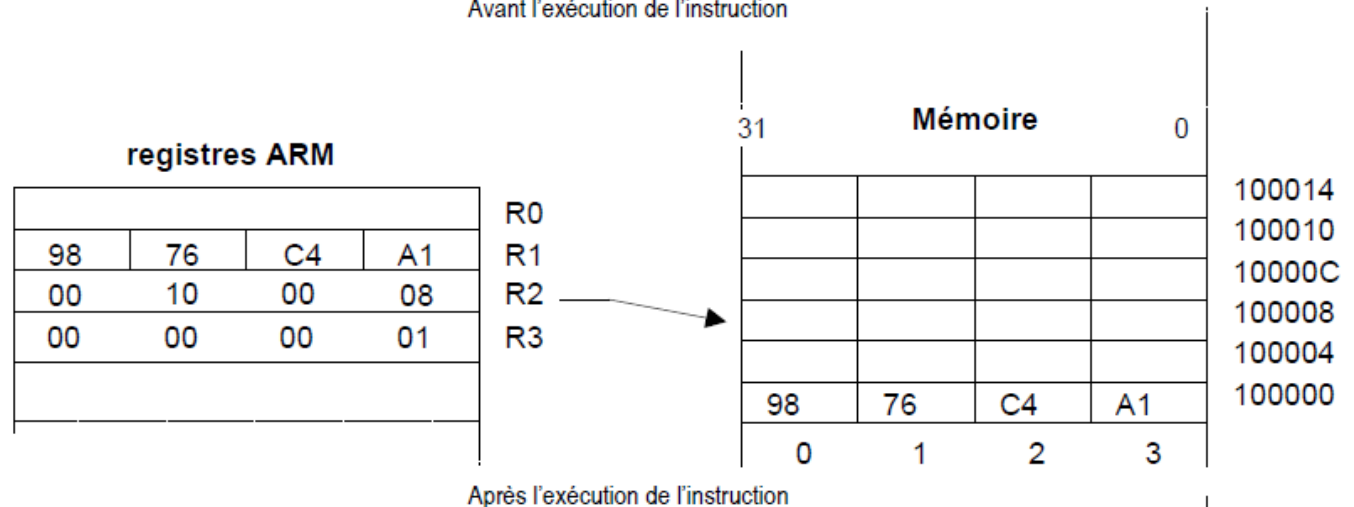
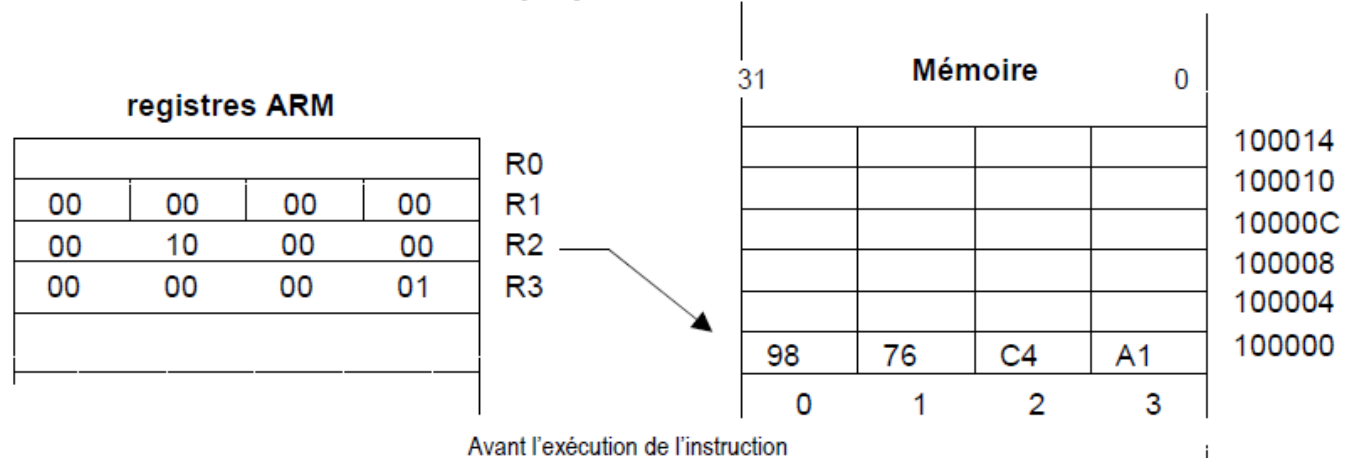


Après l'exécution de l'instruction

# Transferts de mots et de bytes non signés

## Exemple 2

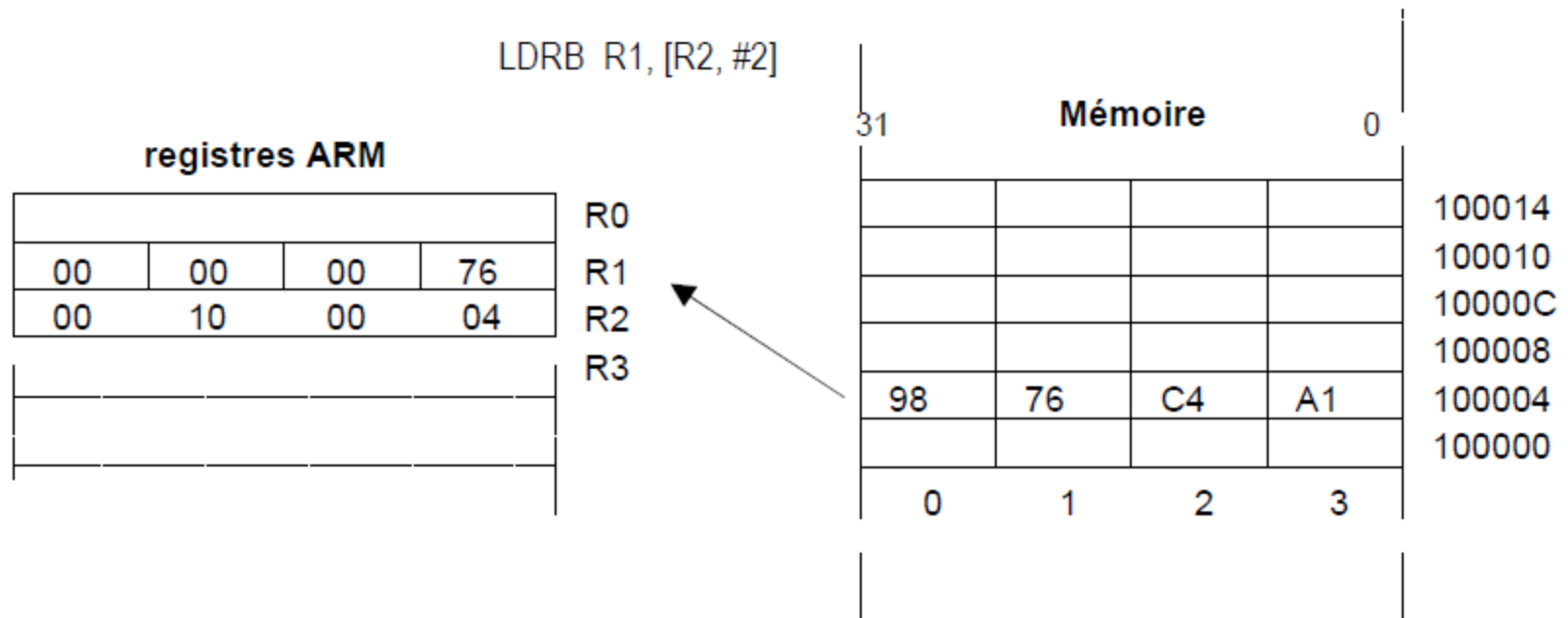
LDR R1, [R2], R3, LSL #3



# Transferts de mots et de bytes non signés

## Remarque 1

Lors du chargement d'un byte non signé dans un registre, les 24 bits de forts poids sont mis à 0.



# Transferts de mots et de bytes non signés

## Remarque 2

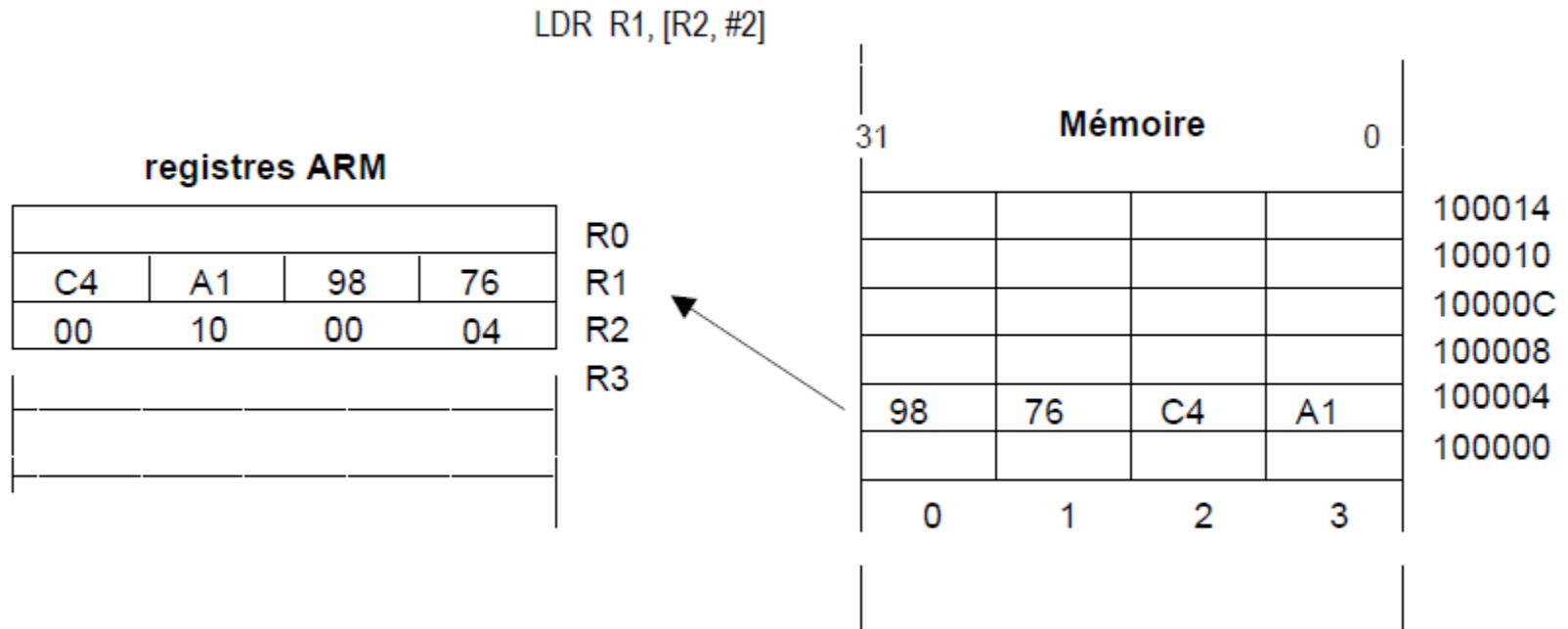
Lorsqu'un mot est non aligné (adresse non divisible par quatre), c'est le mot qui contient l'octet adressé qui est chargé dans le registre.

Le mot subit une rotation pour que le byte correspondant à l'adresse utilisée devienne le byte le plus significatif.

Le kit IAR provoque une exception :

**Next label is a Thumb label**

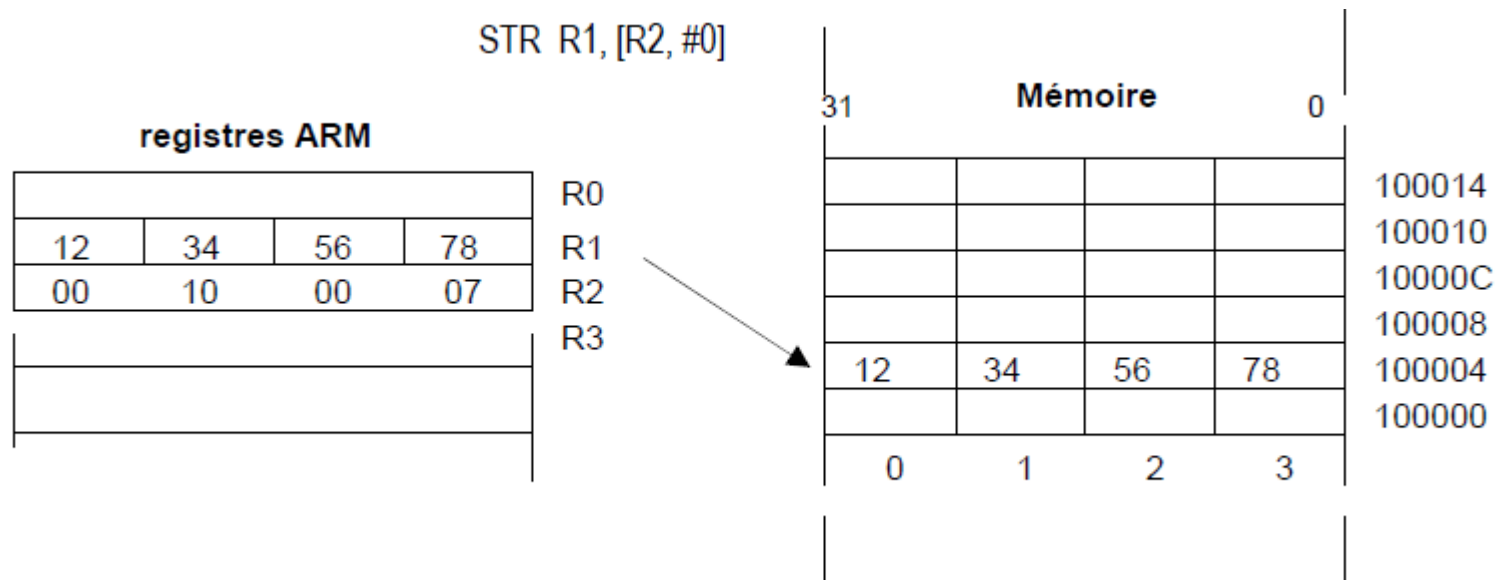
# Transferts de mots et de bytes non signés





# Transferts de mots et de bytes non signés

Remarque 3 : Lorsqu'un mot est stocké en mémoire à une adresse non divisible par quatre, les deux derniers bits de l'adresse sont mis à 0 et le mot est stocké avec cette adresse.



# Transferts de halfwords (16 bits) et de bytes signés

Les transferts de demi-mots et de bytes signés sont

- Load Halfword and Signed Halfwords : LDRH / LDRSH
- Store Halfwords : STRH
- Load Signed Bytes : LDRSB
- Store Bytes : STRB

Elles peuvent être exécutées de manière conditionnelle.

Exemple : LDREQH

Syntaxe

- LDR {cond} H | SH | SB <Rd>, <addressing\_mode>
- STR {cond} H | B <Rd>, <addressing\_mode>

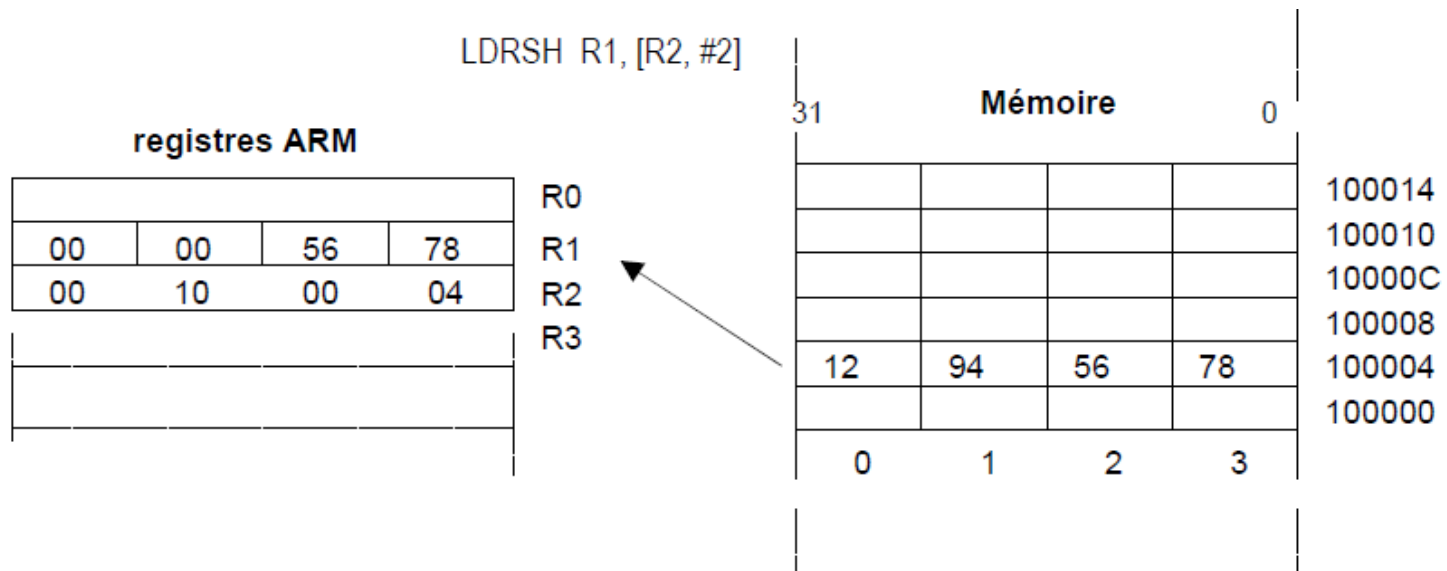
# Modes d'adressages des instructions LDRH, LDRSH, STRH, LDRSB

<addressing_mode>	type d'adressage	exemples
[ <Rn>, #+/- <offset_8bit> ]	registre indirect avec offset sur 8 bits <Rn> est inchangé	LDRH R1, [ R2, # -6] adresse mémoire : R2 -6
[ <Rn>, +/- <Rm> ]	registre indirect avec offset dans un registre <Rn> est inchangé	LDRSB R1, [ R2,R3 ] adresse mémoire : R2 + R3
[ <Rn>, #+/- <offset_8bit> ] ! pré indexé	registre indirect avec offset sur 8 bits mise à jour de <Rn>	LDRH R1, [ R2, # -6 ] ! adresse mémoire : R2 -6 R2 := R2 -6
[ <Rn>, +/- <Rm> ] ! pré indexé	registre indirect avec offset dans un registre mise à jour de <Rn>	LDRSB R1, [ R2,R3 ] ! adresse mémoire : R2 + R3 R2 := R2 + R3
[ <Rn> ], #+/- <offset_8bit> post indexé	registre indirect avec offset sur 8 bits mise à jour de <Rn> après le transfert	LDRH R1, [ R2 ], # -6 adresse mémoire : R2 R2 := R2 -6
[ <Rn> ], +/- <Rm> post indexé	registre indirect avec offset dans un registre mise à jour de <Rn> après le transfert	LDRSB R1, [ R2 ], R3 adresse mémoire : R2 R2 := R2 + R3

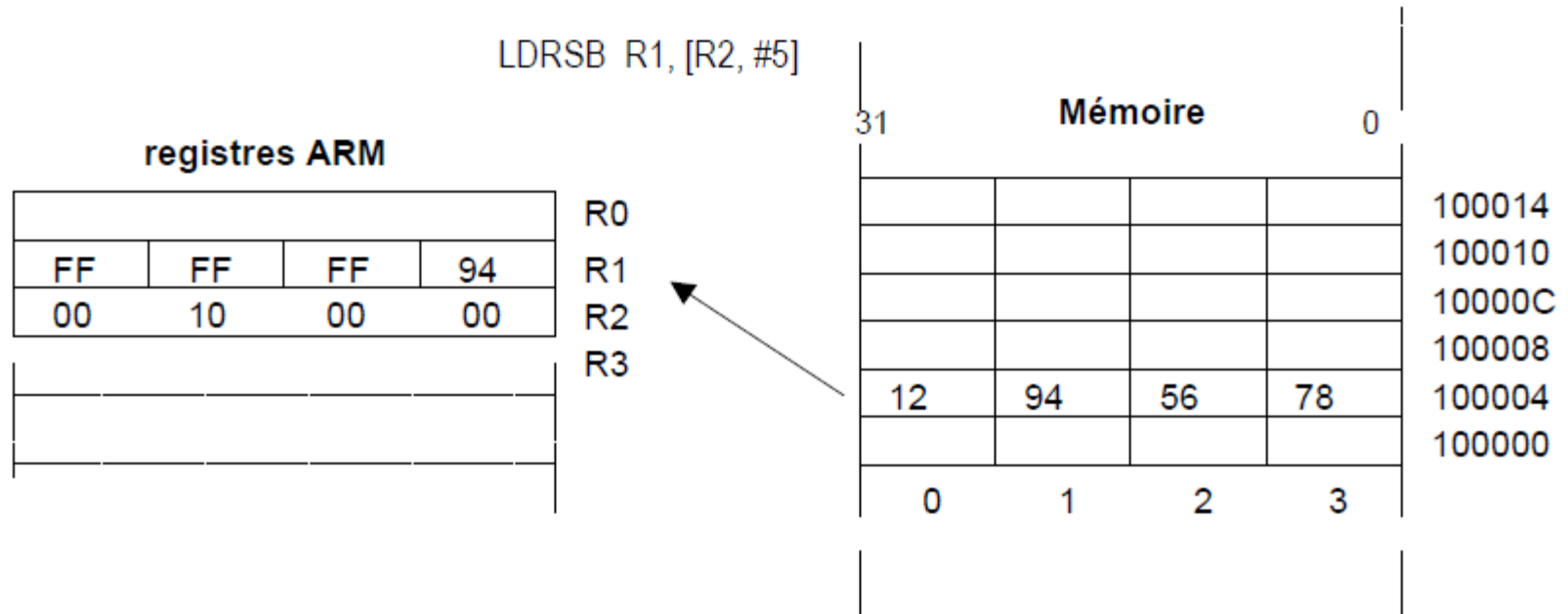
# Transferts de halfwords (16 bits) et de bytes signés

## Remarque 1

Lors d'un chargement d'un halfword ou d'un byte signé de la mémoire, le signe est étendu pour donner un mot de 32 bits de même valeur.



# Transferts de halfwords (16 bits) et de bytes signés



## Remarque 2

Ces instructions ne permettent pas l'utilisation de décalage.

# Transferts de halfwords (16 bits) et de bytes signés

---

## Remarque 3

Les transferts de bytes ou de halfwords vers les registres dépendent de la représentation signée ou non de la valeur.

L'extension du signe aux 32 bits du registre marque la différence.

Pas de distinction pour les transferts de 32 bits.

# Chargement d'une variable dans un registre

Pour charger une variable dans un registre, on peut utiliser la pseudo instruction :

```
LDR R0, MaVar
```

Le programme d'assemblage transforme cette pseudo instruction en une instruction réelle plus complexe :

```
LDR R0, [PC,#-24]
```

Ne marche pas si le déplacement est trop grand, codé sur plus de 12 bits.

# Chargement d'une variable dans un registre

## Solution générale

```
LDR R0, =MaVar
```

```
LDR R1, [R0]
```

Si l'adresse de la variable est compatible avec `<immed_8r>` par ex 0x21, l'instruction est transformée en :

```
MOV r0, #0x21
```

Adresse de MaVar



# Chargement d'une variable dans un registre

Sinon si l'adresse n'est pas compatible avec <immed\_8r>

Pour 0xAE274F21 l'instruction est transformée :

LDR r0,[pc,#0x???] ——— Déplacement relatif

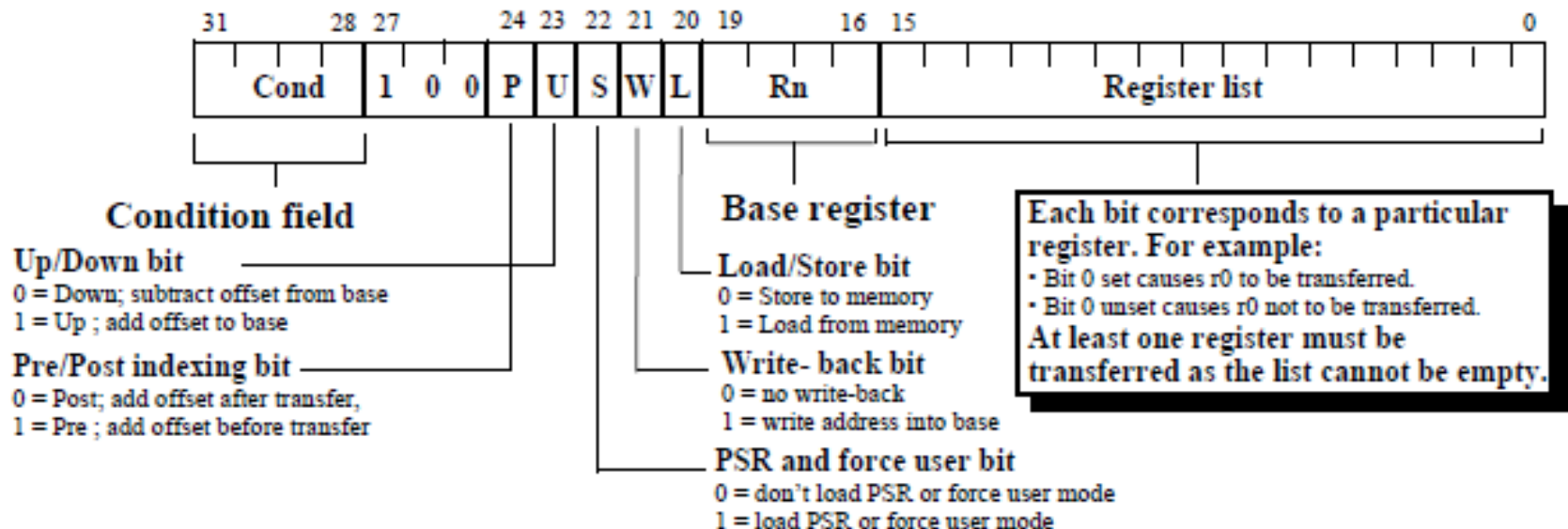
...

...@ Fin des instructions

.WORD 0xAE274F21 ← Adresse de MaVar

# Les instructions Load multiple et Store multiple

Les instructions load et store multiples sont capables d'effectuer des échanges entre la mémoire et l'ensemble (ou une partie) des registres (R0 à R15).



# Les instructions Load multiple et Store multiple

Forme générale des instructions de transfert multiple :

LDM | STM {cond} IA|IB|DA|DB <Rn> { ! } <register\_list>

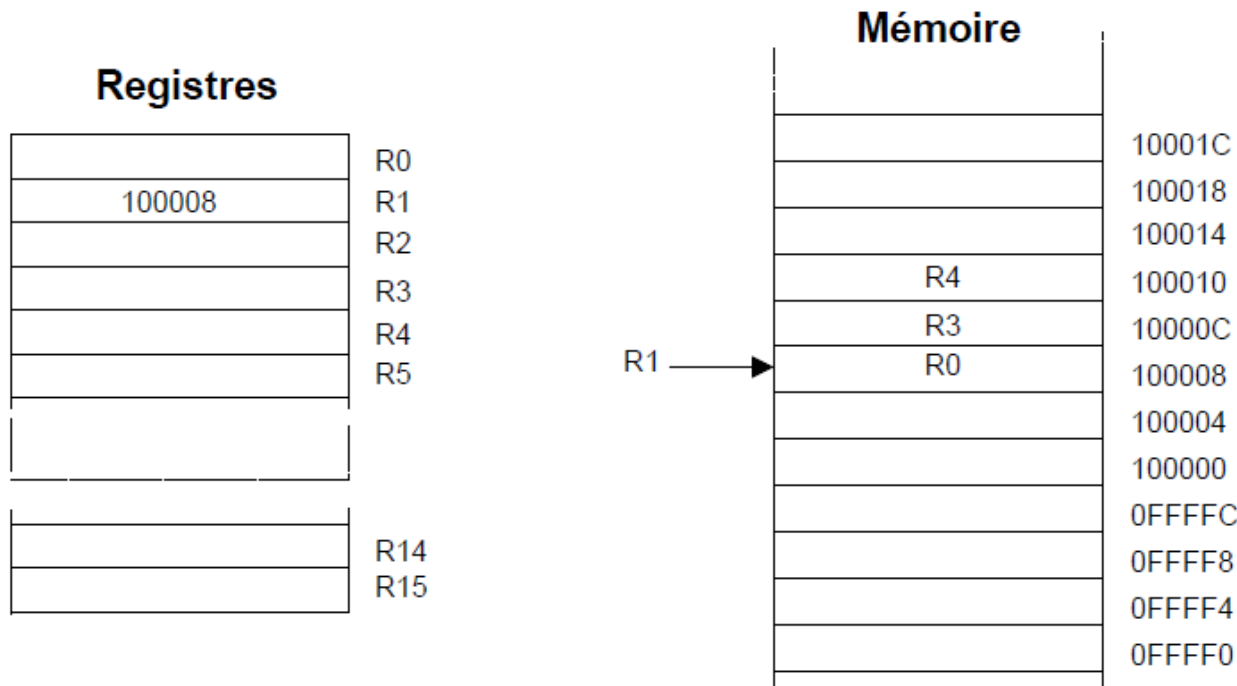
Les formes de LDM load multiple et de STM store multiple définies par IA, IB, DA, DB permettent différentes organisations des données transférées en mémoire.

# Les instructions Load multiple et Store multiple

## Exemple simple

Enregistrement du contenu des registres R0, R3, R4 à l'adresse mémoire pointée par R1. Les registres sont stockés dans l'ordre croissant des adresses.

STMIA R1, { R0, R3, R4 }



# Les instructions Load multiple et Store multiple

- Les registres sont déposés en mémoire dans l'ordre croissant des adresses, de R0 à R15, quel que soit leur ordre dans la liste de l'instruction.
- Ils sont repris de la mémoire (LDM) dans l'ordre inverse, de R15 à R0...
- Par habitude, les registres sont écrits dans l'ordre croissant dans la liste de l'instruction.
- L'adresse pointée par Rn doit être une adresse de mot, divisible par quatre. Ce pointeur peut être fixe ou mis à jour après les transferts grâce au symbole « ! ».

STMIA R1, { R0, R3, R4 } ; R1 n'est pas modifié

STMIA R1 !, { R0, R3, R4 } ; R1 := R1 + 12 (3 registres de 4 bytes)

# Les instructions Load multiple et Store multiple

---

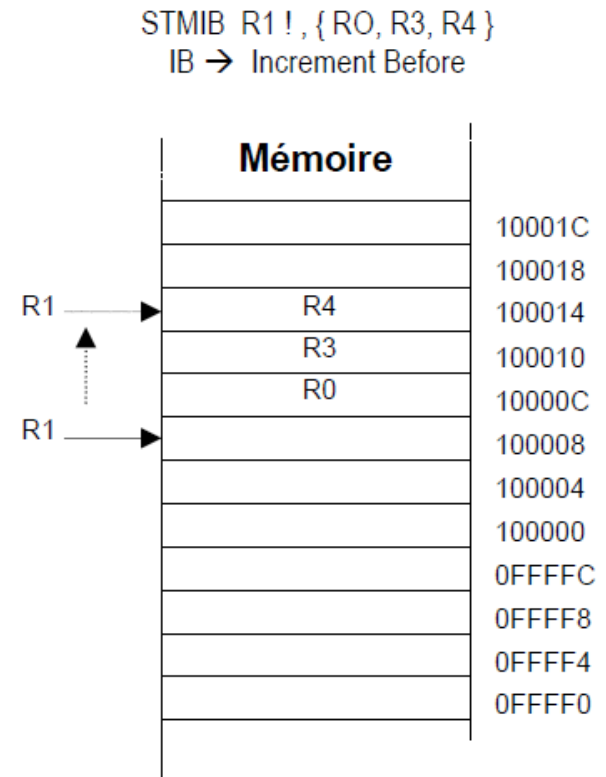
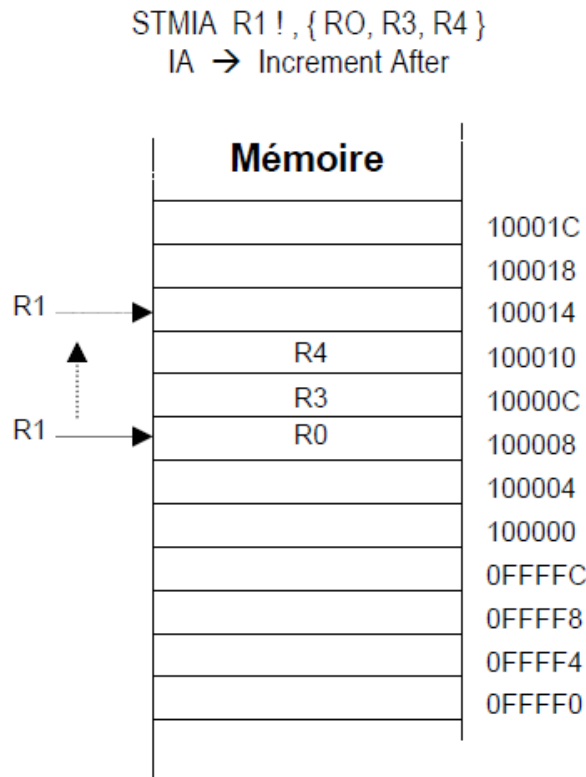
Quatre manières de disposer les registres en mémoire :

- Dépôt en mémoire avec une adresse montante ou avec une adresse descendante.
- Pointeur sur le dernier élément déposé ou la première place libre.

Pour chaque disposition, il existe un LDM et un STM.

# Les instructions Load multiple et Store multiple

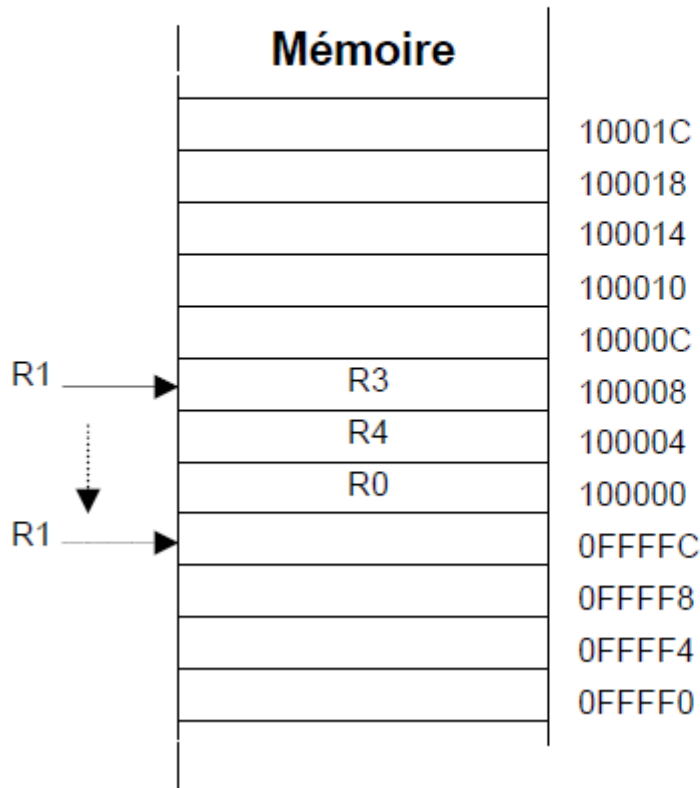
## Exemples



# Les instructions Load multiple et Store multiple

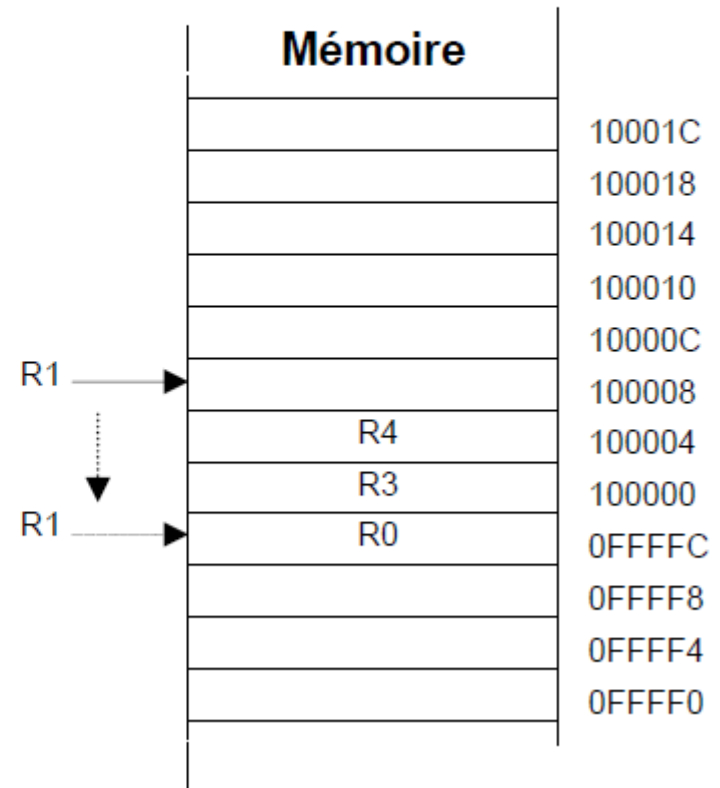
STMDA R1!, { R0, R3, R4 }

DA → Decrement After



STMDB R1!, { R0, R3, R4 }

DB → Decrement Before





# Les instructions

## Manipulation de la pile

# Généralités

---

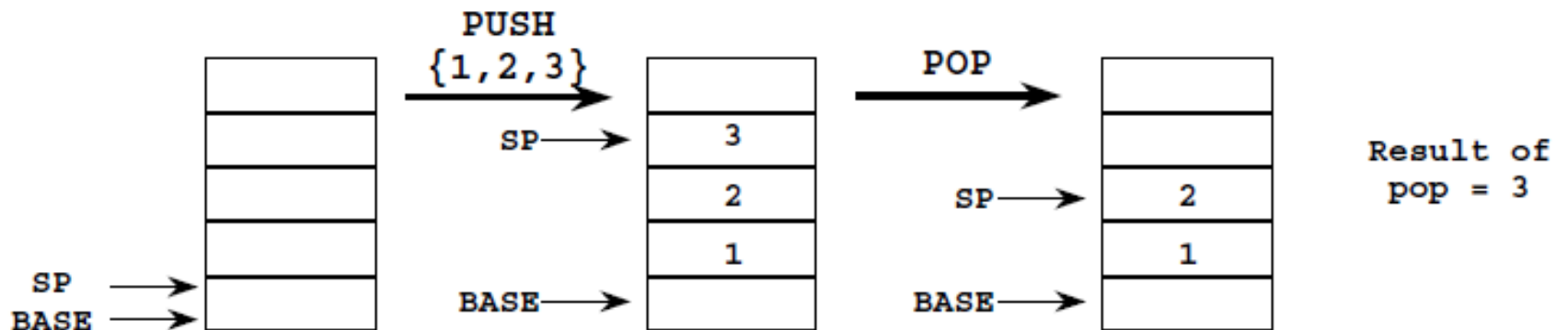
Une pile est une zone de mémoire

- qui grandit à mesure que de nouvelles données sont ajoutées (empilées, « PUSH ») sur son sommet (« TOP ») et
- diminue lorsque des données sont enlevées (dépilées, « POP ») à partir de son sommet.

# Généralités

Deux pointeurs définissent les limites d'une pile :

- Le pointeur de base pointe le "bas" de la pile, le premier endroit.
- Le pointeur de pile pointe le sommet de la pile.



# Généralités

Les piles sont très utilisées pour

- Sauver / restaurer les valeurs de registres
- Stocker des variables locales temporaires
- Passer les paramètres aux sous programmes
- Gérer le retour des sous programmes (récursions...)

Sur ARM7, les données sont empilées à l'adresse

- la plus basse, la pile grandit dans la mémoire de manière descendante (« Descending stack »).
- La plus haute, la pile grandit dans la mémoire de manière ascendante (« Ascending stack »).

# Les opérations sur la pile

Le pointeur de pile (SP ou R13) peut indiquer

- L'adresse du dernier emplacement occupé. Il est donc incrémenté (Full Ascending stack) ou décrémenté (Full Descending stack) avant le PUSH.
- L'adresse du premier emplacement vide. Il est donc incrémenté (Empty Ascending stack) ou décrémenté (Empty Descending stack) après le PUSH.

Par défaut, sur ARM7 la pile est gérée avec la convention « Full Descending »

# Les opérations sur la pile

Elles dérivent des transferts multiples et ont la forme :

LDM | STM {cond} FD|FA|ED|EA <Rn> { ! } <register\_list> {^}

Le type de pile est donnée par le postfix pour l'instruction

STMFD / LDMFD : Full Descending stack

STMFA / LDMFA : Full Ascending stack.

STMED / LDMED : Empty Descending stack

STMFA / LDMEA : Empty Ascending stack

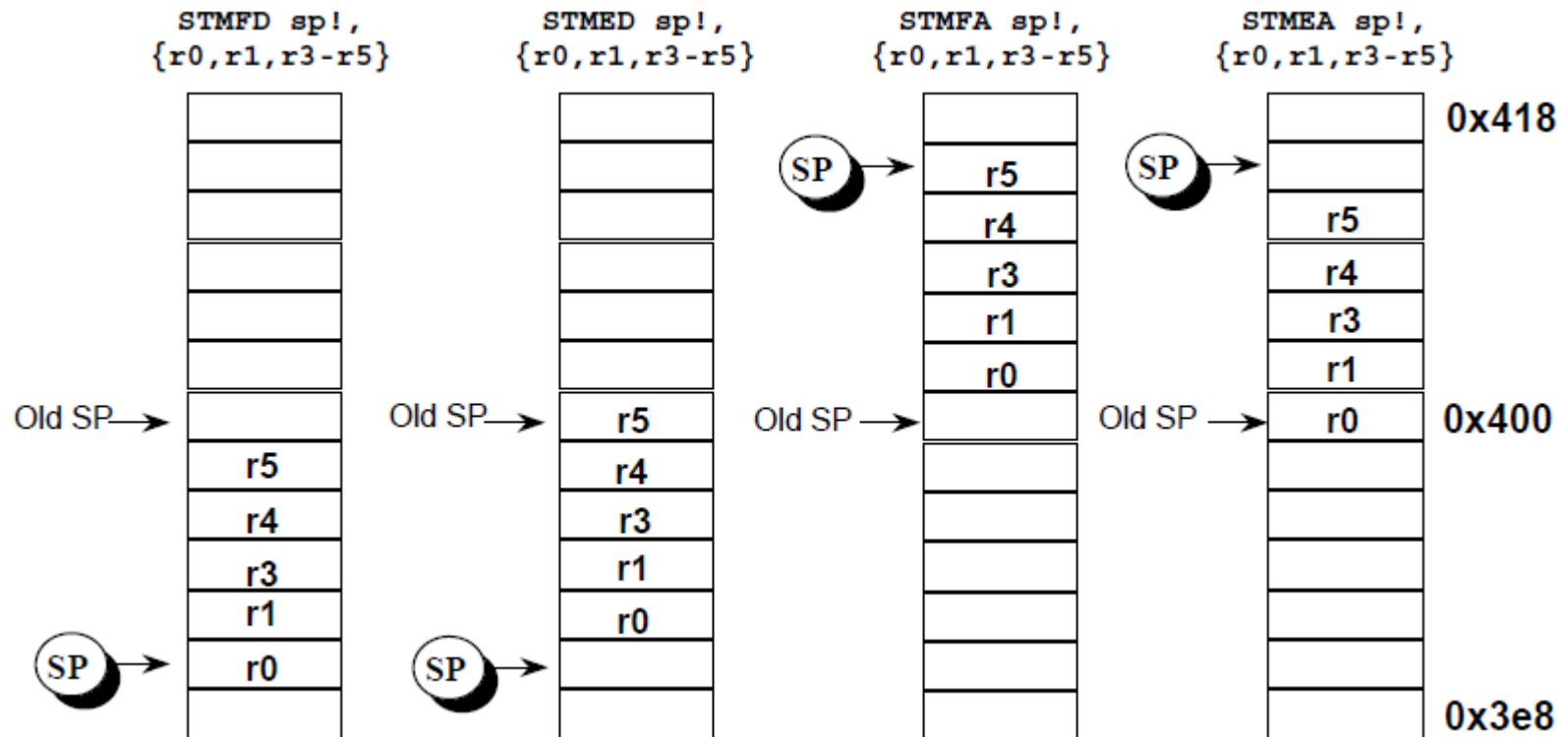
Alias

PUSH {<reglist>} pour STMFD sp!, <reglist>

POP {<reglist>} pour LDMFD sp!, <reglist>

# Les opérations sur la pile

## Exemples



# Les instructions

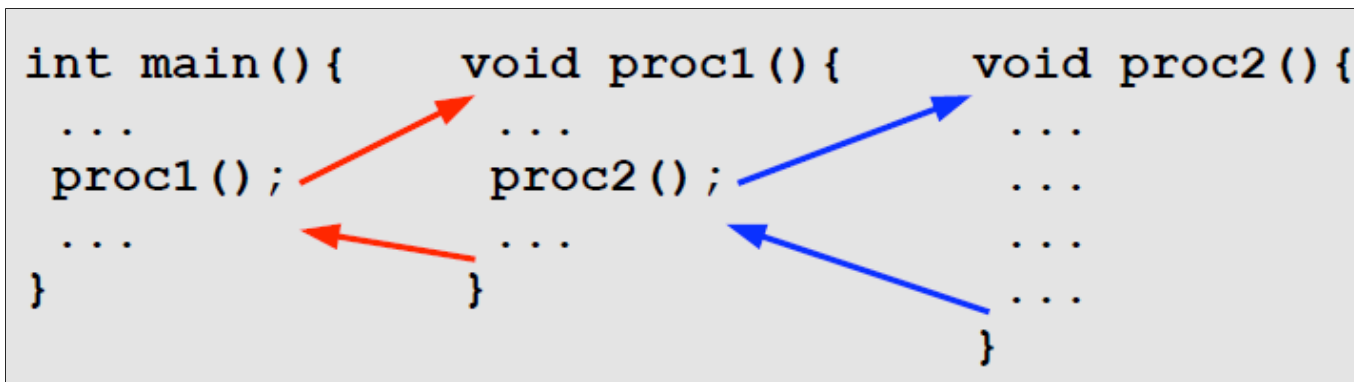
## Gestion des sous-programmes



# Appels des sous-programmes

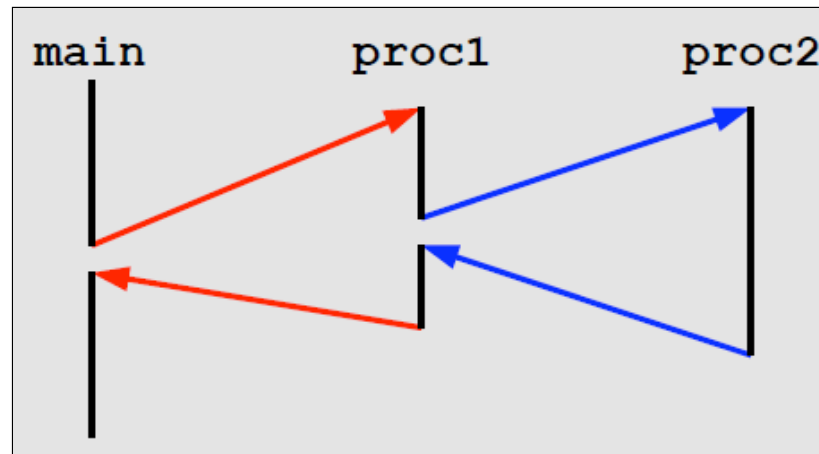
Un sous-programme (procédure ou fonction) est un bloc d'instructions dont on peut lancer l'exécution depuis le programme principal ou depuis un autre sous-programme (appel).

A la fin de l'exécution du sous-programme l'exécution après l'appel.



# Appels des sous-programmes

L'exemple précédent peut être schématisé ainsi

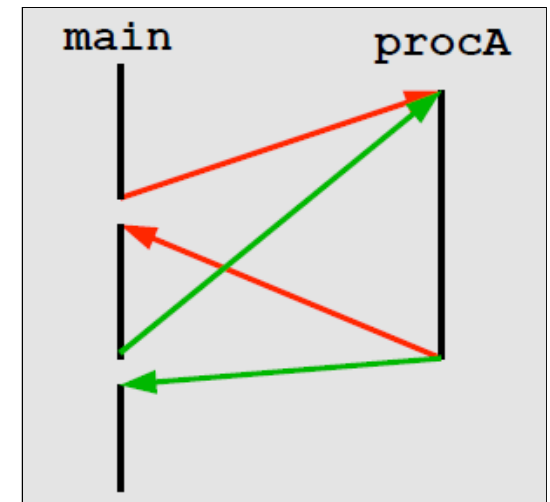
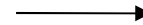


On pourrait l'implanter facilement avec des branchements...

# Appels des sous-programmes

Mais un sous programme peut être appelé depuis différents emplacements du programme :

```
int main() {  
    ...  
    procA() ;  
    ...  
    procA() ;  
    ...  
}  
  
void procA() {  
    ...  
    ...  
    ...  
}
```



L'implantation n'est pas réalisable avec des branchements à des adresses statiques.

# Appels des sous-programmes

Il est donc nécessaire de communiquer au sous programme appelé l'adresse de retour.

L'instruction BL (Branch with Link) appelle un sous programme après avoir enregistré l'adresse de retour (PC – 4 car pipeline) dans le registre R14 (LR pour Link Register) :

BL nomProc ; Appel de la procédure nomProc

# Appels des sous-programmes

Le retour s'effectue en mettant la valeur du registre LR dans le registre PC, ou avec l'instruction BX fait un branchement à une adresse contenue dans un registre :

- MOV PC, LR ; retour à l'appelant méthode 1
- BX LR ; retour à l'appelant méthode 2

Ce mécanisme ne gère pas les appels imbriqués (main → proc1 → proc2) car le contenu de LR est écrasé par le deuxième appel...

La solution consiste à utiliser la pile pour stocker LR avant l'appel de proc2...

# Appels des sous-programmes

```
proc1:
    PUSH    {lr}    @ Sauver adr. retour
    ...
    BL      proc2 @ Ecrase l'adr. retour
    ...
    POP     {lr}    @ Restaure adr. retour
    BX      lr      @ Retour à l'appelant
```

# Passage de paramètres à un sous-programme

## Passage de paramètres à un sous-programme par

- Variables globales (adresse mémoire absolue) → (utilisation mémoire excessive), incompatible avec la récursivité. A utiliser pour des données fixes du programme.
- Registres → limite les registres disponibles dans le sous programme. Les registre sont peu nombreux.
- Pile → accès plus lent qu'à un registre, nombre de paramètres non limité.
- Pointeur vers un bloc de paramètres en mémoire.

# Passage de paramètres à un sous-programme

Convention utilisée avec l'ARM :

- Les quatre premiers paramètres sont passés à l'aide des registres R0, R1, R2 et R3, s'il y a moins de quatre paramètres, les registres excédentaires ne sont pas utilisés.
- Si il y a plus de quatre paramètres, les paramètres supplémentaires sont empilés, le cinquième paramètre est au sommet de la pile, le sixième en dessous, etc. La fonction appelée doit les dépiler avant de revenir à la fonction appelante.



# Passage de paramètres à un sous-programme

La valeur de retour d'une fonction est retournée dans le registre R0.

Exemple : Une fonction retourne la somme de ses deux paramètres.

```
// Prototype pour le C :  
int asmAdd (int a, int b);  
  
@ Le code assembleur :  
asmAdd:  
    ADD    r0 , r0 , r1  
    BX     lr
```

# Changement de contexte

---

Chaque sous programme utilise les registres R0-R12 pour stocker diverses informations selon une organisation qui lui est propre.

L'ensemble des valeurs liées à cette organisation s'appelle un contexte.

Lors de l'appel et du retour de sous-programme, il y a changement de contexte.

# Changement de contexte

Deux approches possibles :

- La fonction appelante doit se débrouiller pour sauver et restaurer son contexte → si elle ne connaît pas bien le fonctionnement de la fonction appelée, elle risque de sauvegarder trop de registre (perte de temps et mémoire).
- La fonction appelée doit sauver et restaurer les valeurs des registres qu'elle modifie pour que l'appelant retrouve son contexte au retour.

Dans les deux cas la pile est utilisée pour sauvegarder et restaurer les valeurs.

# Changement de contexte

---

## Convention utilisée avec l'ARM :

- Les registres R0, R1, R2, R3 et R12 peuvent être modifiés par la fonction appelée. La fonction appelante les empile puis les dépile si il faut sauvegarder leurs valeurs.
- Les autres registres doivent être rendus par la fonction appelante dans leur état d'origine. C'est donc à elle de les sauvegarder avant de les modifier puis de les restaurer avant le retour.

# Changement de contexte

## Exemple

- Sécuriser le contexte au niveau de l'appelant

```
PUSH  {r0-r3, r12}  
BL    proc  
POP   {r0-r3, r12}
```

- Sécuriser le contexte au niveau de l'appelé

```
proc:  
    PUSH  {r4-r11, lr}  
    ...  
    POP   {r4-r11, lr}  
    BX    lr
```

# Variables locales

Les variables déclarées à l'intérieur d'un sous-programme correspondent à des emplacements sur la pile, on parle de variables locales.

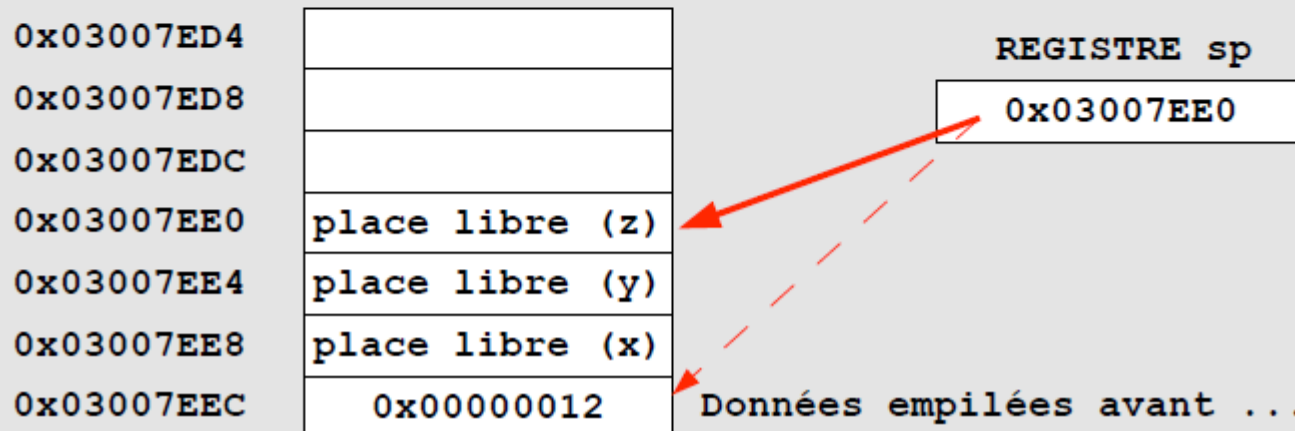
Pour réserver de l'espace d'une variable locale, il suffit de décrémenter SP de la taille de la variable.

Pour libérer l'espace des variables locales, il faut incrémenter SP à la fin du sous programme.

# Variables locales

```
void Procedure() {  
    int x,y,z; // variables automatiques  
    ...  
}
```

-> SUB sp,sp,#12



# Variables locales

Les variables locales sont accédées par adressage relatif à partir de SP.

Exemple écrire la valeur 0 dans x

```
MOV    r0, #0  
STR    r0, [sp, #8]
```

Attention : il faut connaître parfaitement l'organisation de la pile...



# Les instructions

## Instructions diverses

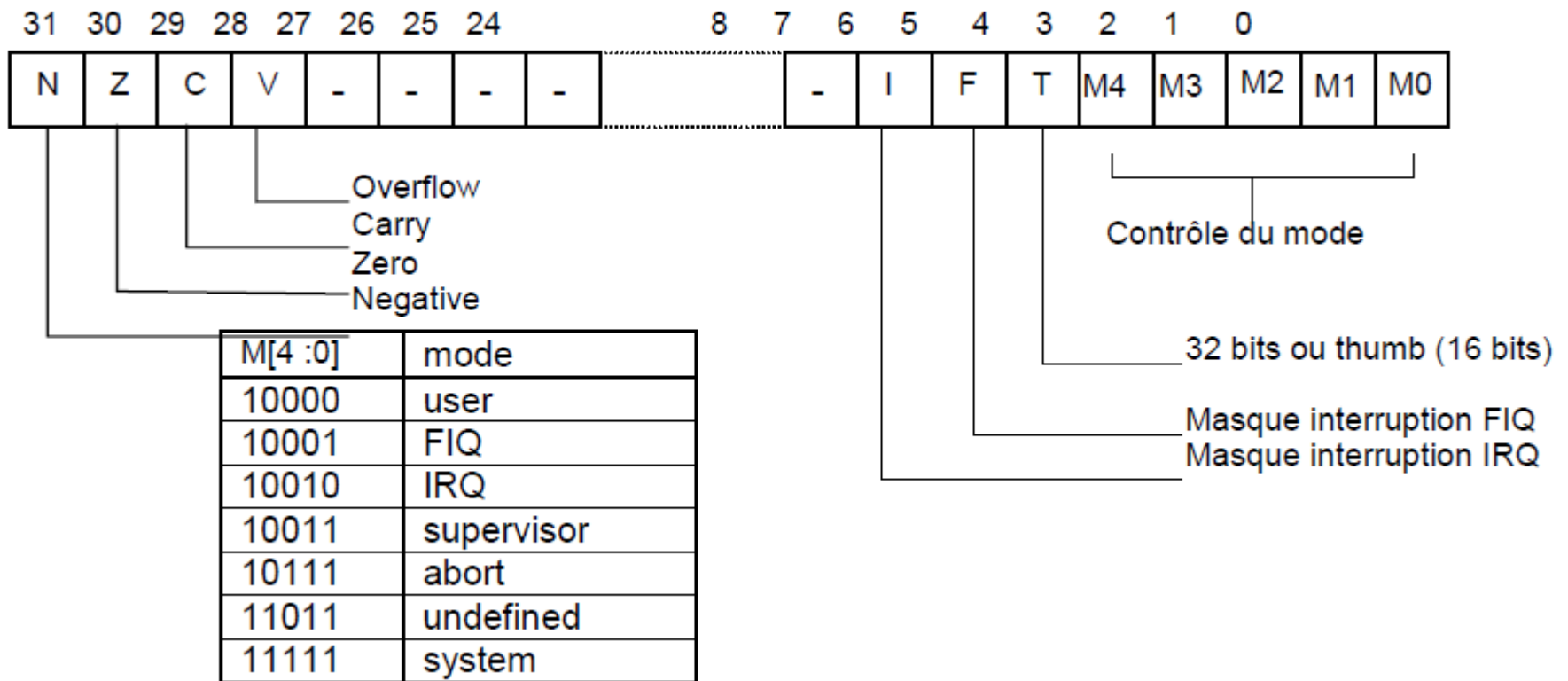
# Manipulation de CPSR et SPSR

- Deux instructions permettent l'accès aux registres CPSR et SPSR :
  - une lecture vers un registre général et
  - une écriture dans les registres d'états.

La lecture des registres d'états ne pose pas de problème, alors que l'écriture implique de posséder les droits à modifier certaines ressources.

# Manipulation de CPSR et SPSR

## Les fonctions des registres d'états



# Manipulation de CPSR et SPSR

La lecture des registres de status est effectuée par MRS (move status to register) :

MRS { cond } <Rd>, CPSR ; copie du registre CPSR dans  
; le registre <Rd>

MRS { cond } <Rd>, SPSR ; copie du registre SPSR dans  
; le registre <Rd>

Ces instructions sont utiles pour sauver CPSR et SPSR lorsque des procédures ou des interruptions sont imbriquées.

# Manipulation de CPSR et SPSR

---

L'écriture dans CPSR et SPSR est soumise aux restrictions suivantes :

- en mode utilisateur (user mode) seuls les bits de conditions (N,Z,C,V) sont modifiables dans CPSR
- le registre SPSR est différent pour chaque mode. Par exemple, seul SPSR\_fiq est accessible si le processeur est en mode FIQ

# Manipulation de CPSR et SPSR

La forme générale est

MSR { cond } CPSR\_<fields> #<immediat8>

MSR { cond } CPSR\_<fields>, <Rm>

MSR { cond } SPSR\_<fields> #<immediat8>

MSR { cond } SPSR\_<fields>, <Rm>

Le champ <field>

- c : bit [7 :0]
- x : bit [15 :8]
- s : bit [23 :16]
- f : bit [31 :24]

# Manipulation de CPSR et SPSR

Seuls les bits de condition sont modifiables par une valeur immédiate.

Pour les autres champs, il faut passer par un registre.

## Exemples

MRS R0, CPSR ; lecture de CPSR

BIC R0, R0, #0x1F ; tous les bits du mode à 0

ORR R0, R0, #0x13 ; mode superviseur

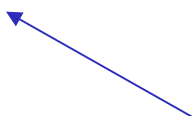
MSR CPSR\_c, R0 ; écriture dans le registre CPSR

# Interruptions logicielles

Il est possible de générer une exception par programme. Les exceptions étant traitées en mode superviseur, elle est souvent programmée pour passer en mode superviseur, mode dans lequel agissent les systèmes d'exploitation.

La forme de l'instruction est

SWI {cond } <24\_bit\_immediate>



Les 24 bits n'ont pas de sens pour l'instruction elle-même, mais permettent de passer des informations au superviseur qui peut alors les interpréter.



# Interruptions logicielles

L'exécution de SWI induit les actions :

- sauvegarde de l'adresse de l'instruction qui suit SWI dans R14\_svc
- sauvegarde de CPSR dans SPSR\_svc
- entrée dans le mode superviseur et désactivation de IRQ en écrivant la valeur 10011 dans CPSR [4 : 0] et 1 dans CPSR[7]
- PC ( R15) = [0x08] vecteur pour SWI qui contient l'adresse de la routine de traitement

# Interruptions logicielles

Le retour à l'instruction qui suit SWI s'effectue en chargeant PC avec R14\_svc et en rétablissant CPSR par copie de SPSR\_svc, par exemple :

MOVS R15, R14 ; registre destination R15 (PC)

L'exécution de MOVS provoque les actions :

- R15 := R14\_svc
- CPSR := SPSR\_svc

# Gestion des sémaphores

Les instructions SWP (swap) et SWPB ( swap byte) sont dédiées à la gestion des sémaphores.

Forme générale des instructions :

SWP { cond } <Rd>, <Rm>, [ <Rn> ]

SWP { cond } B <Rd>, <Rm>, [ <Rn> ]

Elles permettent un échange entre la mémoire et les registres en assurant un cycle lecture / écriture non interruptible. Le mode d'adressage registre indirect simple est le seul autorisé.

# Gestion des sémaphores

## Exemples

- SWP R1, R2, [R3] ; chargement de R1 avec le contenu de la ; mémoire pointée par R3 et stockage de ; R2 dans cette même position mémoire
- SWPB R1, R2, [R3] ; opération identique, mais avec un byte
- SWP R1, R1, [R3] ; échange de valeur entre le contenu de la ; mémoire pointée par R3 et le registre R1

# Instructions coprocesseurs

---

L'architecture ARM est prévue pour l'utilisation de coprocesseurs.

Leur implantation nécessite une interface relativement sophistiquée qui comprend des mécanismes de communication et des mécanismes de synchronisation.

Le processeur comprend donc des instructions spécialisées qui lui permettent de communiquer de manière logicielle avec les coprocesseurs.

# Instructions coprocesseurs

---

Le processeur ARM7TDMI-S comporte deux coprocesseurs internes :

- CP14 : Debug controller
- CP15 : System control for cache and MMU

Les instructions ARM destinées au contrôle des coprocesseurs sont de trois types :

- coprocessor data operations
- coprocessor data transfers
- coprocessor register transfers

# Instructions coprocesseurs

## Coprocessor data operations

- Il s'agit d'instructions génériques puisqu'elles se réfèrent au coprocesseur capable de les traiter. L'instanciation des instructions n'a de sens que pour un coprocesseur défini et présent dans le système.

- Leur forme générale est

CDP { cond } <CP#>, <Cop1>, CRd, CRn, CRm { , <Cop2> }

<CP#> est le numéro du coprocesseur (au max 15 coprocesseurs)

<Cop1>, CRd, CRn, CRm, <Cop2> sont des champs définis par le codage des instructions du coprocesseur

# Instructions coprocesseurs

---

- Si l'instruction est lancée par le processeur et qu'aucun coprocesseur ne répond une exception est générée (instruction indéfinie) et le contrôle est rendu au processeur.

## Coprocessor data transfers

- Le processeur peut transférer des données directement dans les registres d'un coprocesseur ou transférer le contenu des registres du coprocesseur en mémoire. Les éléments de ces instructions dépendent du coprocesseur.



# Instructions coprocesseurs

## – Deux formes disponibles

- avec un pré index :

LDC |STC{cond} {L} <CP#>, CRd, [Rn, <offset>] { ! }

- avec un post index :

LDC |STC{cond} {L} <CP#>, CRd, [Rn], <offset>

CP# : numéro du coprocesseur

CRd : registre coprocesseur source / destination

Rn : registre d'adresse de base

<offset> : offset de 8 bits

L : transfert long (dépendant du coprocesseur)

# Instructions coprocesseurs

## Exemple

STCEQL p2, c3, [ R5, #24] !

; transfert, si Z = 1, le contenu du registre c3 du coprocesseur  
; #2 à l'adresse mémoire égale au contenu de R5 + 24, mise à  
; jour de R5, le transfert de type long ( dépend du  
; coprocesseur)

# Instructions coprocesseurs

---

## Coprocessor register transfers

- Le processeur et le coprocesseur peuvent échanger des informations directement de registre à registre.
- L'exemple classique est le transfert d'un entier converti sur 32 bits du coprocesseur qui a effectué des calculs vers le processeur pour la suite du processus de traitement.

# Instructions coprocesseurs

Forme des instructions disponibles :

MCR|MRC{cond} CP#, <Cop1>, Rd, CRn, Crm{,<Cop2>}

CP# : numéro du coprocesseur

CRn et Crm : registres source (coprocesseur)

Rd : registre destination (processeur)

Cop1 et Cop2 : champs pour le coprocesseur (dépendant du coprocesseur)

Exemple

MRCEQ p3, 9, R3, c5, c6, 2

; Si Z=1, le coprocesseur p3 exécute l'opération 9 de type 2, sur c5 et c6 du coprocesseur, le résultat est transféré sur R3 du processeur.

# Les instructions

## Le codage des instructions

# Principe de codage

---

MOV r5,#7 ; Instruction Assembleur

→ 0xE3A05007 ( code langage machine)

# Principe de codage

Data processing immediate	cond	0	0	1	op				S	Rn				Rd		rotate		immediate															
Data processing immediate shift	cond	0	0	0	opcode				S	Rn				Rd		shift immediate			shift	0	Rm												
Data processing register shift	cond	0	0	0	opcode				S	Rn				Rd		Rs		0	shift	1	Rm												
Multiply	cond	0	0	0	0	0	0	A	S	Rd				Rn				Rs		1	0	0	1	Rm									
Multiply long	cond	0	0	0	0	1	U	A	S	RdHi				RdLo				Rn		1	0	0	1	Rm									
Move from status register	cond	0	0	0	1	0	R	0	0	SBO				Rd				SBZ															
Move immediate to status register	cond	0	0	1	1	0	R	1	0	Mask				SBO				rotate		immediate													
Move register to status register	cond	0	0	0	1	0	R	1	0	Mask				SBO				SBZ					0	Rm									
Branch/exchange instruction set	cond	0	0	0	1	0	0	1	0	SBO				SBO				SBO		0	0	0	1	Rm									
Load/store immediate offset	cond	0	1	0	P	U	B	W	L	Rn				Rd				immediate															
Load/store register offset	cond	0	1	1	P	U	B	W	L	Rn				Rd				shift immediate			shift	0	Rm										
Load/store halfword/signed byte	cond	0	0	0	P	U	1	W	L	Rn				Rd				High offset		1	S	H	1	Low offset									
Load/store halfword/signed byte	cond	0	0	0	P	U	0	W	L	Rn				Rd				SBZ		1	S	H	1	Rm									
Swap/swap byte	cond	0	0	0	1	0	B	0	0	Rn				Rd				SBZ		1	0	0	1	Rm									
Load/store multiple	cond	1	0	0	P	U	S	W	L	Rn				Register list																			
Coprocessor data processing	cond	1	1	1	0	op1				CRn				CRd				cp_num			op2		0	CRm									
Coprocessor register transfers	cond	1	1	1	0	op1			L	CRn				Rd				cp_num			op2		1	CRm									
Coprocessor load and store	cond	1	1	0	P	U	N	W	L	Rn				CRd				cp_num			8_bit_offset												
Branch and branch with link	cond	1	0	1	L	24_bit_offset																											
Software interrupt	cond	1	1	1	1	swi_number																											
Undefined	cond	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x	
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

# Principe de codage

cond	0 0 1	opcode	S	Rn	Rd	rotate	immediate
------	-------	--------	---	----	----	--------	-----------

Les différents champs sont

- **cond** (bit 31 – bit 28 ) : condition d'exécution de l'instruction testée à partir des bits NZCV du registre de status
- **opcode** (bit 27- bit 21 ) : code de l'opération **ADDS** (bit 20) : si S= 1 le résultat modifie le registre de status
- **Rn** (bit 19 – bit 16) : no du registre source 1
- **Rd** (bit 15 – bit 11) : no du registre destination
- **rotate** (bit 10 – bit 8 ) : déplacement éventuel de la constante
- **immédiate** (bit 7 – bit 0 ) : constante



# Principe de codage

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-

# Principe de codage

MOV r5,#7

Exécution inconditionnelles : Always (pas de préfixe)

→ 1110

Puis spécification de famille d'opération (Data-processing : MOV, ADD, CMP ...)

→ 111000

Puis indicateur d'opérande immédiat (I)

→ 1110001

# Principe de codage

Opcode	Mnemonic	Operation	Action
0000	AND	Logical AND	$Rd := Rn \text{ AND shifter\_operand}$
0001	EOR	Logical Exclusive OR	$Rd := Rn \text{ EOR shifter\_operand}$
0010	SUB	Subtract	$Rd := Rn - \text{shifter\_operand}$
0011	RSB	Reverse Subtract	$Rd := \text{shifter\_operand} - Rn$
0100	ADD	Add	$Rd := Rn + \text{shifter\_operand}$
0101	ADC	Add with Carry	$Rd := Rn + \text{shifter\_operand} + \text{Carry Flag}$
0110	SBC	Subtract with Carry	$Rd := Rn - \text{shifter\_operand} - \text{NOT}(\text{Carry Flag})$
0111	RSC	Reverse Subtract with Carry	$Rd := \text{shifter\_operand} - Rn - \text{NOT}(\text{Carry Flag})$
1000	TST	Test	Update flags after $Rn \text{ AND shifter\_operand}$
1001	TEQ	Test Equivalence	Update flags after $Rn \text{ EOR shifter\_operand}$
1010	CMP	Compare	Update flags after $Rn - \text{shifter\_operand}$
1011	CMN	Compare Negated	Update flags after $Rn + \text{shifter\_operand}$
1100	ORR	Logical (inclusive) OR	$Rd := Rn \text{ OR shifter\_operand}$
1101	MOV	Move	$Rd := \text{shifter\_operand (no first operand)}$
1110	BIC	Bit Clear	$Rd := Rn \text{ AND NOT}(\text{shifter\_operand})$
1111	MVN	Move Not	$Rd := \text{NOT shifter\_operand (no first operand)}$

# Principe de codage

MOV r5,#7

C'est une opération Move

→ 11100011101

Pas de mise à jour des codes conditions (pas de S)

→ 111000111010

MOV : pas de 1er opérande (Rn a 0)

→ 1110001110100000

Registre destination R5 (Rd code pour 5)

→ 11100011101000000101

# Principe de codage

MOV r5,#7

Format immédiat : constante 8 bits et rotation

Pas de déplacement (rotation 0)

→ 111000111010000001010000

Constante 8 bits : 0b00000111

→ 11100011101000000101000000000111

0b11100011101000000101000000000111

0xE3A05007