Plan du module

1. Introduction aux systèmes à microprocesseurs
2. Les mémoires
3. Les microprocesseurs
4. Architecture d'un processeur RISC (Cortex M4)
5. **Programmation Assembleur Cortex M4**

Syntaxe assembleur ARM

Syntaxe:

label: opcode operand1, operand2,... ; Comments

- ***label:*** est optionnel, utilisé pour déterminer l'adresse de l'instruction
- ***opcode*** **:** appellation de l' instruction.
- Le nombre des opérandes dépend du type de l'instruction
- Généralement, le premier opérande désigne la destination de l'opération
- Après (;) ce sont des commentaires.

Exemple: (valeurs immédiates)

```
MOV   R0,   #0x12     ; Mettre  R0 = 0x12 (hexadecimal)
MOV   R1,   #'A'      ; Mettre R1 = ASCII character A
```

Jeu d'instructions Cortex M3/M4

L'ISA (Instruction Set Architecture) du Cortex M4 sera divisé en ces catégories d'instructions:

- instructions de transfert de données internes *(MOV, MOVW, MOVT, MOVN, ...)*
- instructions d'accès mémoire *(LDR, STR, LDRB, STRB, LDRH, …)*
- instructions arithmétiques *(ADD, SUB, MUL, SDIV, UDI, …)* et logiques *(AND, ORR, EOR, Complément à 1, Décalages et rotations,...)*
- instructions de comparaison *(CMP, CMN, TST, TEQ)*
- instructions de branchement *(B, BX, BL, BLX)*

Opérandes

➢ Un opérande est un registre ou une constante ou un autre paramètre spécifique

➢ Les instructions agissent sur les opérandes et généralement enregistre le résultat dans un registre de destination (mentionné généralement avant les opérandes)

➢ Dans certaines instructions on trouve **Operand 2** qui est un second opérande flexible:

✓ soient des constantes précédées par #

✓Soit des registres avec décalage possible: Rm {, shift}

shift est un décalage optionnel appliqué à Rm. Il peut etre:

ASR #n	Arithmetic shift right n bits, 1 ≤ n ≤ 32.
LSL #n	Logical shift left n bits, 1 ≤ n ≤ 31.
LSR #n	Logical shift right n bits, 1 ≤ n ≤ 32.
ROR #n	Rotate right n bits, 1 ≤ n ≤ 31.
RRX	Rotate right one bit, with extend.

Instructions de décalages et de rotations

Ces instructions déplacent d'un certain nombre de positions les bits d'un mot vers la gauche ou vers la droite.

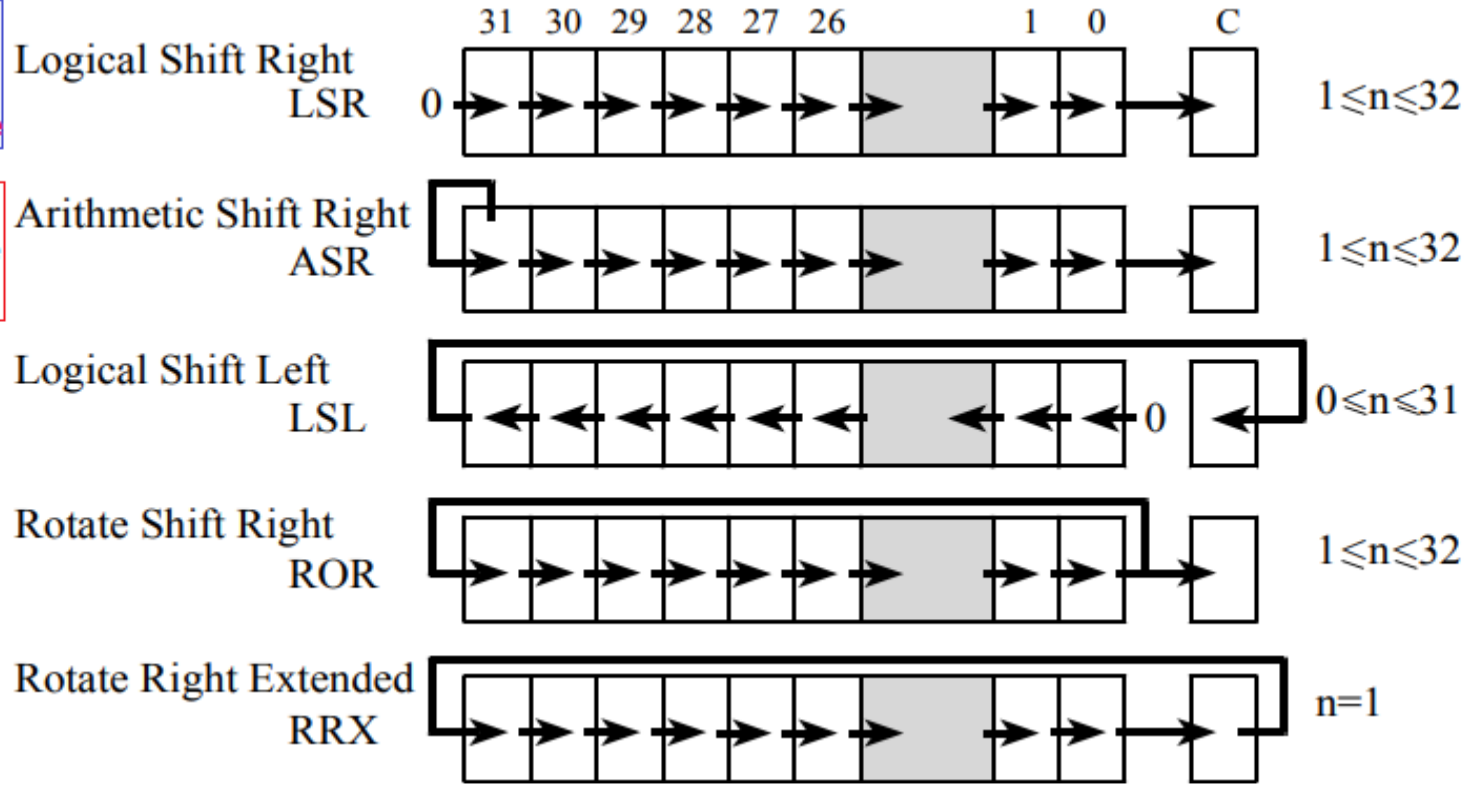
Dans les **décalages**, les **bits qui sont déplacés sont remplacés par des zéros**. Le ***carry flag contient le dernier bit sortant.***

Il y a les **décalages logiques** (opérations non signées) et les **décalages arithmétiques** (opérations signées).

Dans les rotations, les bits déplacés (i.e. les bits sortants) dans un sens sont **réinjectés** de l'autre coté du mot. Le ***carry flag contient le dernier bit sortant.***

Utiliser l'instruction **LSR** pour le décalage d'un nombre non signé

Utiliser l'instruction **ASR** pour le décalage d'un nombre signé



Instructions de décalages et de rotations

Syntaxe:

Instructions de décalage:

```
LSR{S}{cond}  Rd,  Rm,  Rs
LSR{S}{cond}  Rd,  Rm,  #sh

ASR{S}{cond}  Rd,  Rm,  Rs
ASR{S}{cond}  Rd,  Rm,  #sh

LSL{S}{cond}  Rd,  Rm,  Rs
LSL{S}{cond}  Rd,  Rm,  #sh
```

Instructions de rotation:

```
ROR{S}{cond}  Rd,  Rm,  Rs
ROR{S}{cond}  Rd,  Rm,  #sh

RRX{S}{cond}  Rd,  Rm
```

Exemples

```
LSL r0,r1,r2  ; pour (r0 = r1 ≪ r2[7:0]) Seul l'octet faible de r2 est utilisé
ASR r3,r4,r5  ; pour (r3 = r4 ≫ r5[7:0]) Seul l'octet faible de r5 est utilisé
```

Paramètres:

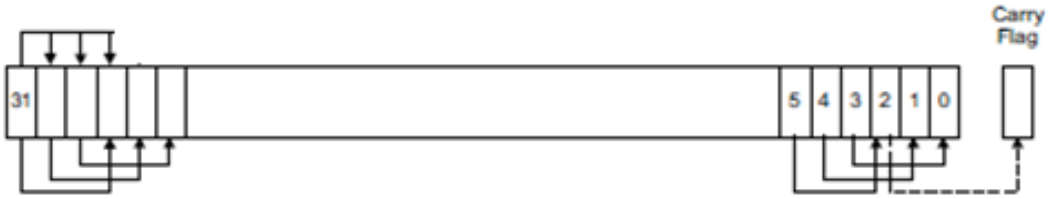
Nom	Description
S	Ce paramètre optionnel permet d'indiquer le suffixe. Si ce paramètre est spécifié, les drapeaux de condition sont mise à jour dans le résultat de l'opérande
cond	Ce paramètre optionnel permet d'indiquer le code de condition
Rd	Ce paramètre permet d'indiquer le registre de destination.
Rm	Ce paramètre permet d'indiquer le registre contenant le premier opérande. Cet opérande est décalé vers la droite.
Rs	Ce paramètre permet d'indiquer une valeur de décalage à appliquer à valeur du paramètre *Rm*. **Seul l'octet de poids le plus faible est utilisé**.
#sh	Ce paramètre permet d'indiquer une constante de décalage. Les valeurs doit être situé entre 1 et 31.

Instructions de décalages et de rotations

Dans certaines instructions on trouve les opérations de décalage et de rotation comme troisième paramètre

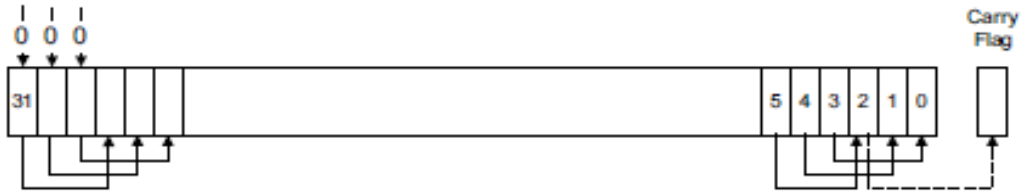
Exemples explicatifs

ASR#3



Si n ≥ 32 tous les bits auront la valeur du bit 31 de Rm (aussi le Carry flag si n ≥ 33)

LSR#3



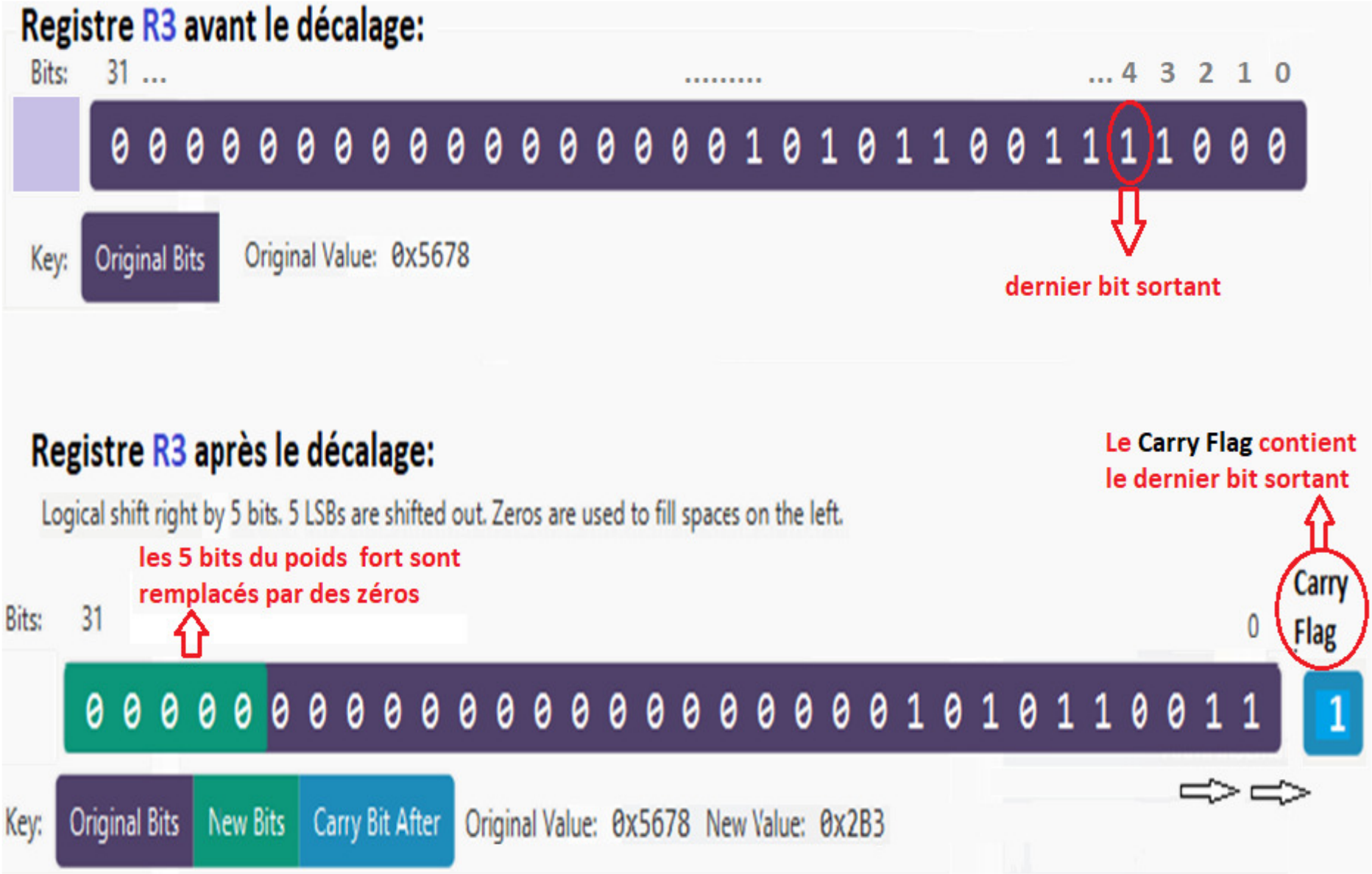
Si n ≥ 32 tous les bits auront la valeur 0 (aussi le Carry flag si n ≥ 33)

Application 1: Donner le contenu du registre R3 après avoir exécuter le code suivant :

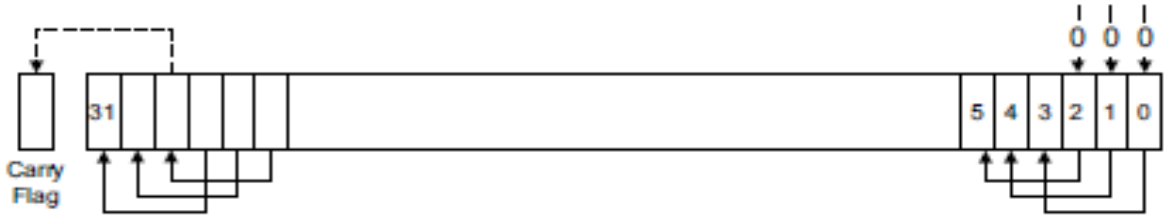
```
MOVW R1, #0x5678      ; Mettre  R1 = 5678 h
MOV  R3, R1           ; R3 = R1
MOV R3, R1 , LSR #5   ; Charger  R3 par la valeur
                      ; stockée dans R1 décalée par 5 bits à droite
                      ; le contenu de R1 ne va pas changer.
                      ; R3= (R1 >> 5)
```

Instructions de décalages et de rotations

Suite Application 1:



LSL#3



Si n ≥ 32 tous les bits auront la valeur 0 (aussi le Carry flag si n ≥ 33)

Application 2: Donner le contenu du registre R1 après avoir exécuter le code suivant :

```
MOVW R1, #0x8ACF   ; Mettre  R1 = 8ACF h

MOV R1, R1 , LSL #1
```

Avant la rotation, registre R1 = 0000 8ACF h:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1

Après la rotation, registre R1 = 0001 159E h:

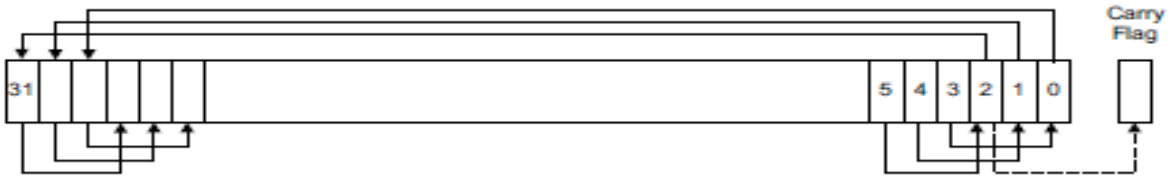
CF

0

←

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	0

ROR #3

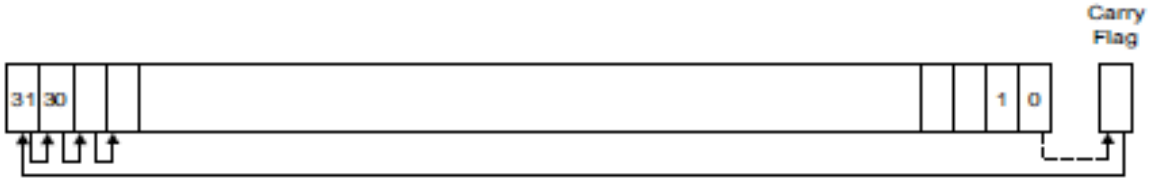


L'instruction **ROR #n** décale l'opérande **n** positions vers la droite et réinjecte par la gauche les **n** bits sortants. (Voir Application 3)

Si n =32 on retrouve Rm (aussi le Carry flag prend bit 31 de Rm)

Si n>32 c'est toujours le cas n=32

RRX



Cette instruction décale l'opérande d'une seule position vers la droite en passant par l'indicateur de retenue CF.

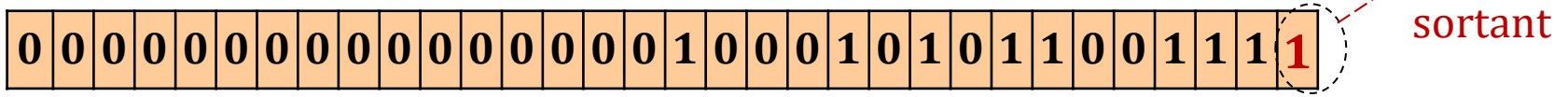
Le bit sortant par la droite est copié dans l'indicateur de retenue CF et **la valeur précédente de CF est réinjectée par la gauche**. (Voir Application 4)

Application 3: Donner le contenu du registre R2 après avoir exécuter le code suivant:

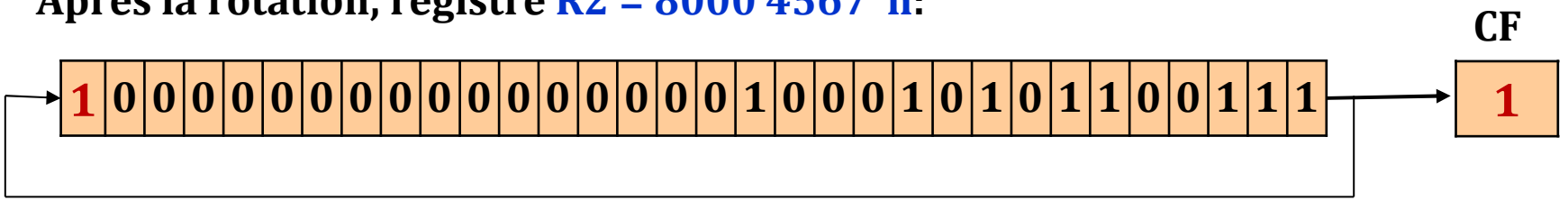
```
MOVW R1, #0x8ACF

MOV R2, R1 , ROR #1
```

L'instruction **ROR #1** décale l'opérande (dans ce cas la valeur binaire stockée dans le registre R1) d'une position vers la droite et réinjecte par la gauche le bit sortant. Le **résultat** est stocké dans le registre **R2**. ***Le contenu du registre R1 ne change pas***.

Avant et après la rotation, registre R1 = 0000 8ACF h:



Après la rotation, registre R2 = 8000 4567 h:



Application 4: Donner le contenu du registre R1 après avoir exécuter le code suivant:

```
MOVW R1, #0x8ACF
RRXS R1, R1
```

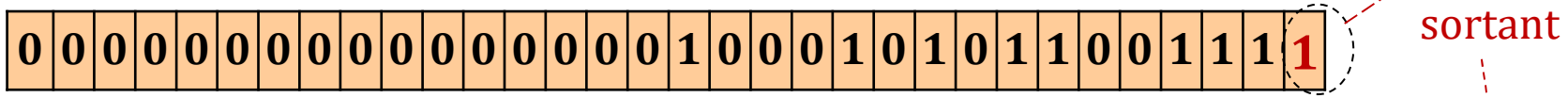
OU

```
MOVW R1, #0x8ACF
MOV R1, R1, RRX
```

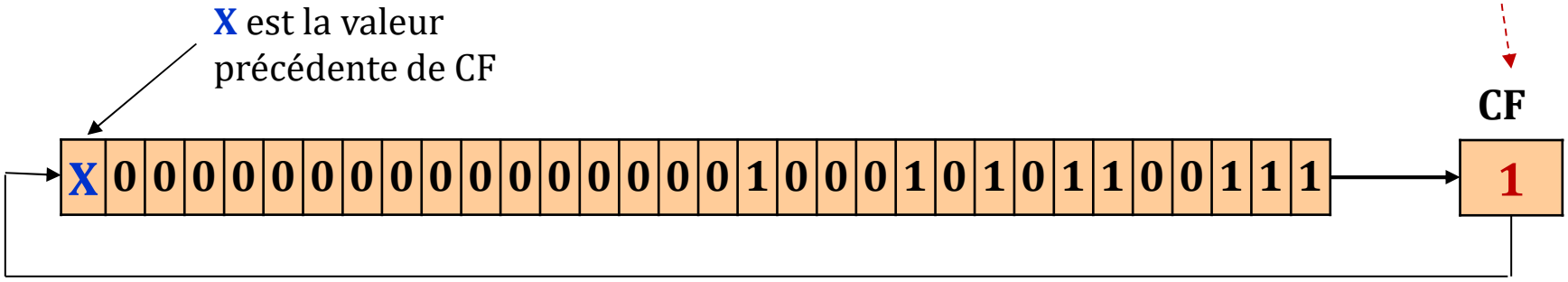
RRX décale l'opérande d'une seule position vers la droite.

Le bit sortant par la droite est copié dans l'indicateur de retenue CF et **la valeur précédente de CF est réinjectée par la gauche**.

Avant la rotation, registre R1 = 0000 8ACF h:



Après la rotation, registre R1 = X000 4567 h:



Les instructions de transfert de données internes

Elles permettent de:

- affecter une valeur immédiate à un registre
- transférer des données d'un registre vers un registre ;

Instructions: MOV ou MOVW ou MVN ou MOVT

Syntaxe:

MOV {S}{*cond*} ***Rd, Operand2***

MVN {S}{*cond*} ***Rd, Operand2***

MOV {*cond*} ***Rd, #imm8***

MOVW }{*cond*} ***Rd, #imm16***

MOVT }{*cond*} ***Rd, #imm16***

S: suffixe optionnel => mise à jour des flags concernés du PSR en tenant compte du résultat de l'opération (spécifiquement dans ce cas N et Z, V non affecté, C par l'opérande 2)

Cond: optionnel => opération réalisée sous une condition (voir suite de ce cours)

Rd: registre de destination.
Operand 2 : second opérande flexible
imm16: valeur immédiate sur 16 bits (de 0 à 65535)
imm8: valeur immédiate sur 8 bits (de 0 à 255)

Les instructions de transfert de données internes

- **Exemples:**

MOV R3, #0x34 ; instruction par défaut 16 bits qui charge R3 par la **valeur 8 bits** 0x34

MOVW R3, #0x1234 ; instruction 32 bits qui charge R3 par la **valeur 16 bits** 0x1234

MOVT R3, #0x1234 ; instruction 32 bits qui met les 16 bits **les plus significatifs** de
; R3 à la **valeur 16 bits** 0x1234

Pour charger R0 par la valeur immédiate **12345678**H (de taille 32 bits):

MOVW R0, #0x5678;

MOVT R0, #0x1234 ;

MOV R4, R0 ; instruction 16 bits qui copie la valeur de R0 dans R4

MVN R4, R0 ; copier le complément à 1 de la valeur de R0 dans R4

MOV R4, R0, LSL#2 ; copier dans R4 la valeur dans R0 qui subit un décalage logique à gauche de 2 bits.

Les instructions de transfert de données internes

Syntaxe: **MOVS** ***Rd, Operand2***

MVNS ***Rd, Operand2***

S suffixe optionnel = mise à jour de PSR en tenant compte du résultat de l'opération (spécifiquement dans ce cas N et Z, V non affecté, C par l'opérande 2)
Rd: registre de destination.
Operand 2 : second opérande flexible

➢ Exemples:

MOVS R3, #0x34 ; charge R3 par la valeur 8 bits 0x34 mais avec mise à jour du PSR
MOVS R4, R0 ; copie la valeur de R0 dans R4 avec mise à jour du PSR
MVNS R2, #0xF ; écrire la valeur 0xFFFFFFF0 dans R2 avec Mise à jour de PSR

➢ Remarques:

✓ possibilité d'utilisation de PC dans une instruction MOV mais il faut que operand2 soit sans shift et ne pas spécifier le suffixe S
✓ Si Rd = PC dans une instruction MOV on ignore le bit 0 et on le met à zéro
✓ Rd de MOVT ne doit pas être PC
✓MOVT ne change pas PSR

Les instructions d'accès mémoire
[Instructions de chargement/rangement]

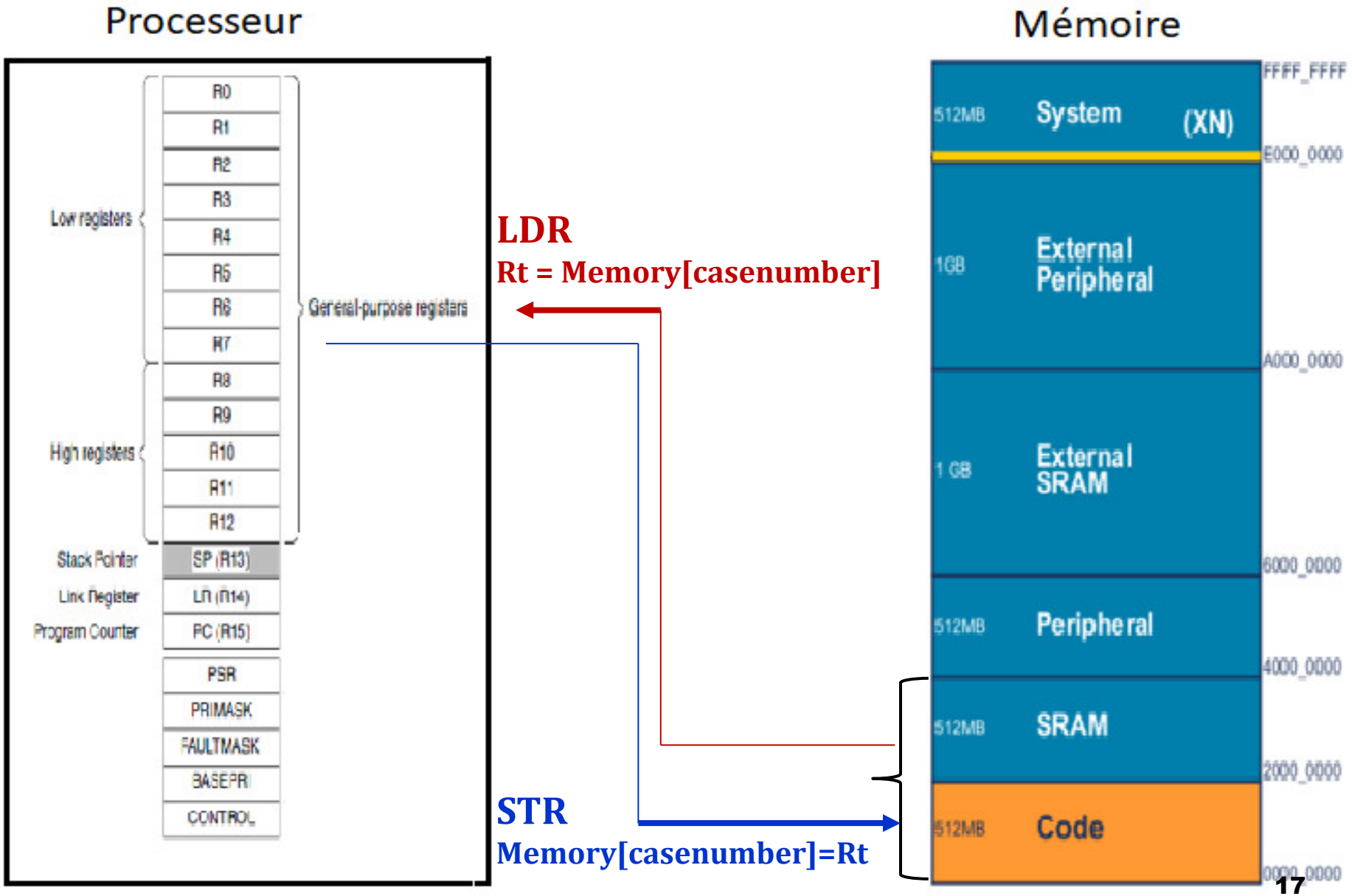
- **Modes d'adressage** des instruction à accès mémoire:
  - Adressage simple **Basé** (contenu dans un registre de base):
    - ✓ [<Rn>]
  - Adressage avec déplacement **pré-indexé:**
    - ✓ [<Rn>, #+/-<offset_12>]
    - ✓ [<Rn>, #+/-<Rm>]
    - ✓ [<Rn>, #+/-<Rm>, {LSL|LSR|ASR|ROR|RRX} #<shift_imm>]
  - Adressage avec déplacement **post-indexé**

  Crochet fermant juste après le registre de base Rn

    - ✓ [<Rn>] , #+/-<offset_12>
    - ✓ [<Rn>] , #+/-<Rm>
    - ✓ [<Rn>] , #+/-<Rm>, {LSL|LSR|ASR|ROR|RRX} #<shift_imm>

- Instructions de **chargement/rangement de**:
  - WORD (4 octets): **LDR** et **STR**
  - Byte (1octet): **LDRB** (Load Byte) et **STRB** (Store Byte):
  - Half-word (demi-mot): LDR**H** (Load Half-word) **et** STR**H** (Load Half-word)
- Instruction de **chargement/rangement signées** : LDR**SB** , STR**SB,** LDR**SH et** STR**SH**

Les instructions d'accès mémoire
[Instructions de chargement/rangement]

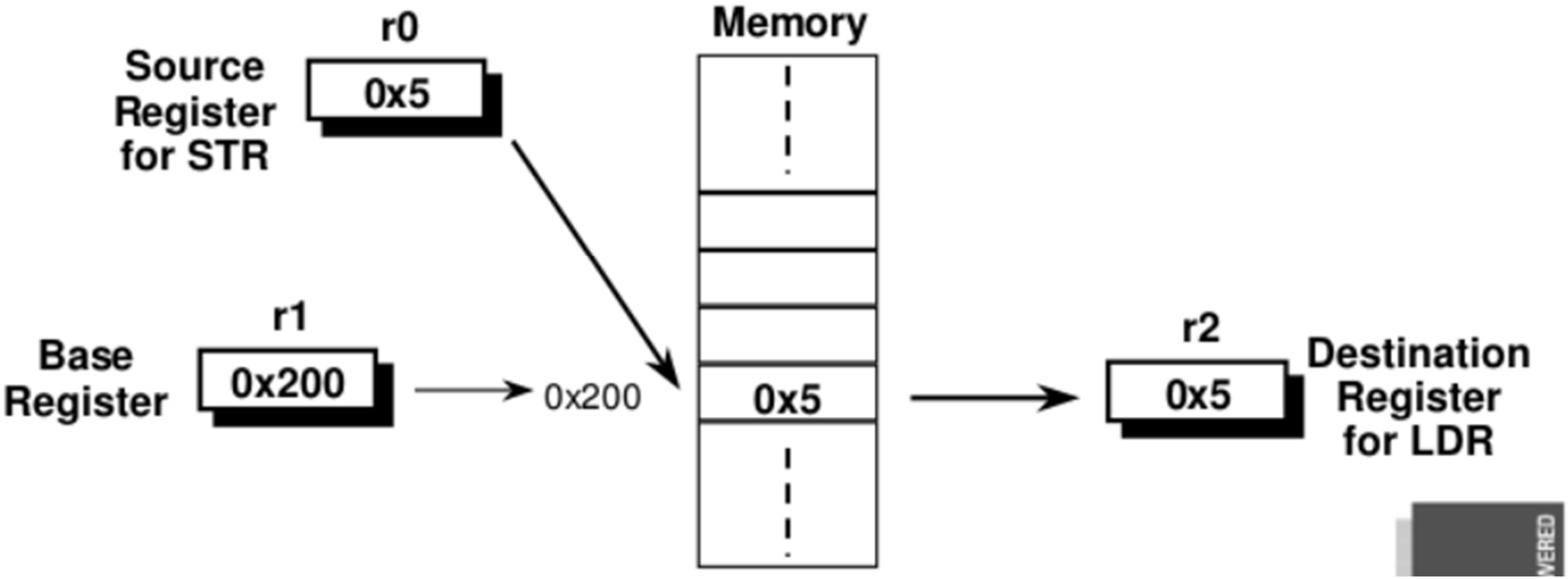


Les instructions d'accès mémoire
[Instructions de chargement/rangement]

Registre de base

* **The memory location to be accessed is held in a base register**

  - STR r0, [r1] ; Store contents of r0 to location pointed to
  ; by contents of r1.
  ; Memory[r1] = r0
  - LDR r2, [r1] ; Load r2 with contents of memory location
  ; pointed to by contents of r1.
  ; r2 = Memory[r1]

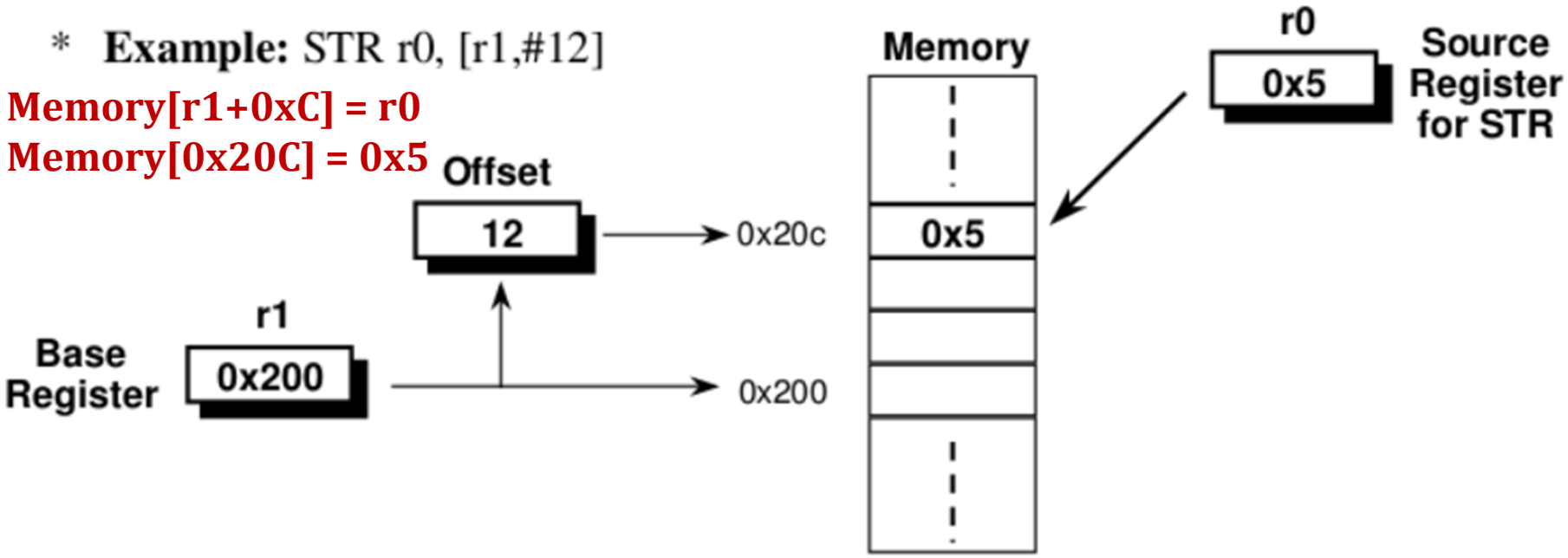


Les instructions d'accès mémoire
[Instructions de chargement/rangement]

pré-indexé

* **Example:** STR r0, [r1,#12]

; Memory[r1+0xC] = r0
; Memory[0x20C] = 0x5



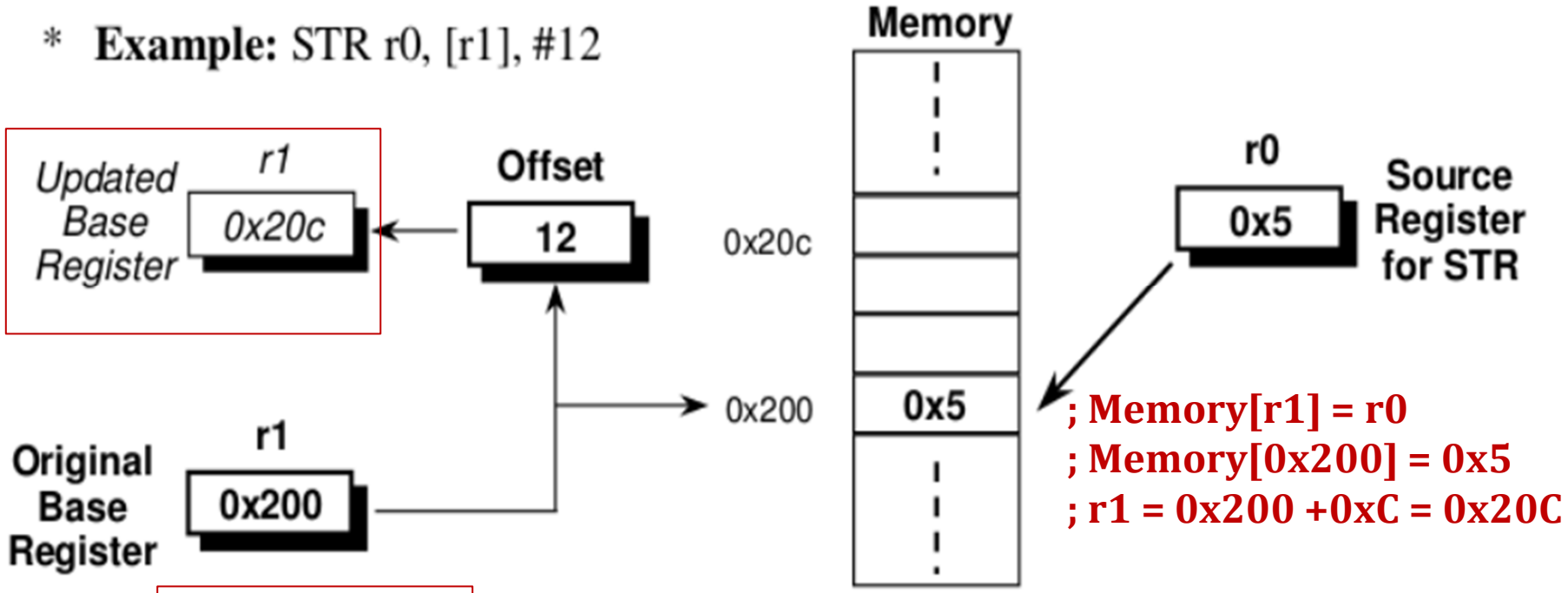
* **To store to location 0x1f4 instead use:** STR r0, [r1,#-12]
* **To auto-increment base pointer to 0x20c use:** STR r0, [r1, #12]!
* **If r2 contains 3, access 0x20c by multiplying this by 4:**
  - STR r0, [r1, r2, LSL #2]

Remarque: un décalage à droite revient à faire une division par 2 et un décalage à gauche, une multiplication par 2.

Les instructions d'accès mémoire
[Instructions de chargement/rangement]

post-indexé

* **Example:** STR r0, [r1], #12



* **To auto-increment the base register to location 0x1f4 instead use:**
  - STR r0, [r1], #-12
* **If r2 contains 3, auto-incremenet base register to 0x20c by multiplying this by 4:**
  - STR r0, [r1], r2, LSL #2

; Memory[r1] = r0
; r1 = r1+(r2*2^2) = 0x2000+$(3*2^2)_{10}$
; r1 = 0x2000+0xC = 0x200C

Les instructions d'accès mémoire
[Instructions de chargement/rangement]

Récapitulation:

- Instructions de **chargement/rangement de WORD** (4 octets=4cases mémoires):

✓ **LDR** (Load Word): **Syntaxe:** **LDR**{<cond>} **<Rt>, <Mode d'adressage>**

✓ **STR** (Store Byte): **Syntaxe:** **STR**{<cond>} **<Rt>, <Mode d'adressage>**

(**ATTENTION** ordre opérandes, **destination est la mémoire**, dernier opérande)

Exemple: Tracer le code suivant et préciser le mode d'@ de chaque inst LDR/STR

```
LDR R9, [R0]          ; Adressage basé (@basé). R9 = Memory[R0]
STR R7,[R1]           ; (@basé). R1 est le registre de base Memory[R1]= R7
STR R7,[R1,#8]        ; (@Pré-indexé). @deRangement = @base + déplacement(offset)
MOVW R0, 0x2100       ; R0 = 2100 h
LDR R3, [R0,#0x10]    ; (@Pré-indexé). R3 = Memory[R0+10h] = Memory[2110h]
                      ; Accéder à (@2100+10) h et Chargement dans R3
                      ; Après l'exécution de cette instruction R0 vaut toujours 2100 h
STR R3, [R0,#0x24]!   ; (@Pré-indexé). Memory[0x2124]= R3 et R0 = 2124 h
                      ; (adresse de rangement = (@basé + déplacement simple)
                ; Avec sauvegarde de l'@ de rangement dans le registre de base R0 (auto-increment)
LDR R3, [R0, R2, LSL #2]  ; (@Pré-indexé). R3 = Memory[R0+(R2 << 2)]
MOVW R1, 0x3200
STR R3, [R1],#0x24    ; (@Post-indexé). (adresse de rangement = 0x3200)
                      ; Memory[R1]= R3 ➔ Memory[0x3200]= R3
                      ; R1 = 0x3224 (auto-increment)
```

Les instructions d'accès mémoire
[Instructions de chargement/rangement]

Récapitulation:

- Instructions de chargement/rangement de Half-WORD (4 octets):
  - ✓ **Syntaxe: LDRH** {<cond>} **<Rt>, <Mode d'adressage>**

    Remplissage du reste du registre par des 0

    Syntaxe: STRH {<cond>} **<Rt>, <Mode d'adressage>**
- Instructions de chargement/rangement de Byte (1 octet):
  - ✓ **Syntaxe: LDRB** {<cond>} **<Rt>, <Mode d'adressage>**

    Remplissage du reste du registre par des 0
  - ✓ **Syntaxe: STRB** {<cond>} **<Rt>, <Mode d'adressage>**
- Instructions de chargement/rangement de Signed Byte/SignedHalf-WORD
  - ✓ **Syntaxe: LDRSB** {<cond>} **<Rt>, <Mode d'adressage>**

    Remplissage du reste du registre par le bit du signe
  - ✓ **Syntaxe: STRSB** {<cond>} **<Rt>, <Mode d'adressage>**
  - ✓ **Syntaxe: LDRSH** {<cond>} **<Rt>, <Mode d'adressage>**

    Remplissage du reste du registre par le bit du signe
  - ✓ **Syntaxe: STRSH** {<cond>} **<Rt>, <Mode d'adressage>**

Les instructions d'accès mémoire
[Instructions de chargement/rangement]

Exemple2: Tracer le code suivant et préciser le mode d'@ de chaque inst LDR/STR

LDR**B** R0, [R1, #0x3] ; (@Pré-indexé). **R0 = Memory[R1+0x3]**

; **lecture d'un octet** à partir de l'@ (R1+0x3) et stockage dans R0

; i.e. lecture d'une seule case mémoire

LDRB R6, [R2] ; (@basé). **R6 = Memory[R2]**

LDR R3, [R0, R2, LSL #2] ; (@Pré-indexé). **R3 = Memory[R0+(R2 << 2)]**

LDR R0, [R1], #-4 ; (@Post-indexé). **R0 = Memory[R1-4]**

; R1 = R1 - 4

STR**H** R5, [R0,R7] ; (@Pré-indexé). **Memory[R0+R7]= R5**

;Ecriture dans deux cases mémoire

LDR**H** R0, [R1, #10] **!** ; (@Pré-indexé). **R0 = Memory[R1+10]**

; R1 = R1 + 10 (pré-indexé avec auto-increment lorsqu'il y a « ! »)

; **lecture d'un demi-mot** à partir de l'@ (R1+10) et stockage dans R0

Les instructions arithmétiques et logiques

Les instructions arithmétiques de base sont l'addition, la soustraction, la multiplication et la division qui incluent diverses variantes.

Syntaxe

op{S} {Rd} Rn, Operand2
op {Rd} Rn, #imm12

op est l'une de ces opérations:

ADD: Add ADC: Add with Carry

SUB: Subtract. SBC: Subtract with Carry RSB: Reverse Subtract

MUL: Multiply MLA: muliplication et accumulation

MLS: multiplication et soustraction

UMULL: multiplication 64 bits non signée

SMULL: multiplication 64 bits signée

UDIV: division non signée SDIV: division signée

AND: ET logique ORR: OU logique EOR: XOR logique

ORN: OR avec complément à 1 du deuxième opérande

BIC: ET avec complément à 1 du deuxième opérande

Les instructions arithmétiques et logiques

S suffixe optionnel = mise à jour du PSR en tenant compte du résultat de l'opération
Rd: registre de destination. S'il est omis Rn sera le registre de destination
Rn: registre premier opérande
Operand 2 : second opérande flexible
imm12: valeur immédiate sur 12 bits (de 0 à 4095)

Exemples:

ADD R0, #1 ; Use 16-bit Thumb instruction by default for smaller size (R0 =R0+1)

ADD.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide) (R0 =R0+1)

ADD R0, R0, R1 ; R0 = R0 + R1

ADDS R0, R0,#0x12 ; R0 = R0+12H avec mise à jour PSR

ADDS R0, #1 ; Use 16-bit Thumb instruction by default for smaller size (update PSR)

ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide) (update PSR)

ADC R0,R1,R2 ; R0 = R1 + R2 + PSR (Carry)

AND R1,R1,R2 ; R1 = R1 AND R2

ANDS R1,R1,R2 ; R1 = R1 AND R2 (update PSR)

Les instructions arithmétiques et logiques

MUL r0,r1,r2 ; multiplication r0 = r1*r2

UMULL r0, r1,r2,r3 ; multiplication 64 bits non signée {r1,r0} = r2*r3

SMULL r0,r1,r2,r3 ; multiplication 64 bits signée {r1,r0} = r2*r3

MLA r0,r1,r2,r3 ; multiplication et accumulation r0 = r1+r2*r3

MLS r0,r1,r2,r3 ; multiplication et accumulation r0 = r1-r2*r3

Les instructions de comparaison

- Ces instructions permettent de comparer deux opérandes
- Un opérande est un registre ou une constante ou un autre paramètre spécifique
- Ces instructions **ne modifient que les bits (N,Z,C,V)** du PSR.

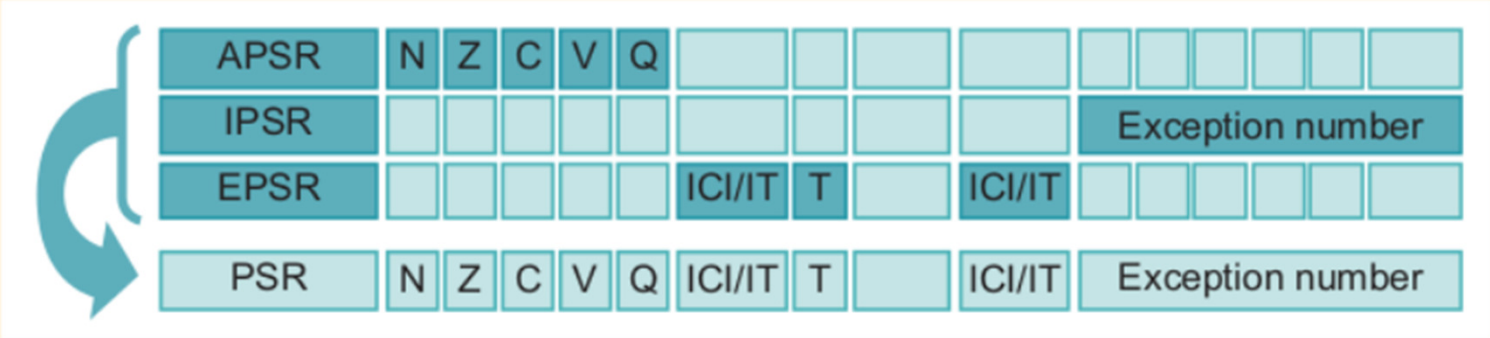
  Le résultat de l'opération **n'est pas gardé.**

Exemples:

`CMP r0,r1`	→	`psr ⇐ r0-r1`	Comparer
`CMN r0,r1`	→	`psr ⇐ r0+r1`	Comparer à l'inverse
`TST r0,r1`	→	`psr ⇐ r0&r1`	Tester les bits indiqués par r1
`TEQ r0,r1`	→	`psr ⇐ r0^r1`	Tester l'égalité bit à bit

Exécution conditionnelle

Rappel: les flags (drapeaux) générés par l'ALU sont mémorisés dans le APSR :



N : Le résultat de l'ALU est négatif (bit 31 égal `a 1)
Z : Le résultat de l'ALU est égal `a zéro (32 bits égaux `a 0) ;
C : Retenue généré l'ALU dans le cas des instructions arithmétiques et le décaleur (shifter) dans le cas des instructions logiques
V : Dépassement de capacité dans le cas d'une opération arithmétique signée
Q: Indique que le résultat a saturé (exemple: minimums et maximums atteints)

Exécution conditionnelle

Les drapeaux APSR sont très utilisées dans les architectures ARM :

- Il possible pour la plupart des instructions de préciser leur action sur les **drapeaux (suffixe S)**.

- Les **branchements** peuvent être **conditionnés** par la **valeur des flags** ce qui est très commun.

- Une particularité importante des architectures ARM: l'exécution de presque toutes les instructions **peut être conditionnée** par la **valeur des drapeaux.** L'objectif de cette utilisation de prédicat est de réduire le nombre de branchements.

Exécution conditionnelle

➢ L'exécution des instructions peut être rendue conditionnelle en rajoutant les suffixes suivants:

EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS/HS	Carry set/unsigned higher or same	C == 1
CC/LO	Carry clear/unsigned lower	C == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z == 0 and N == V
LE	Signed less than or equal	Z == 1 or N != V

Exemple:

`CMP r0,r1`	comparer r_0 à r_1
`SUBGE r0,r0,r1`	si $r_0 \geq r_1$ alors $r_0 = r_0 - r_1$
`SUBLT r0,r1,r0`	si $r_0 < r_1$ alors $r_0 = r_1 - r_0$

Instructions de branchement

➢ Ces instructions permettent de modifier le compteur de programme Pc (registre R15) en donnant une valeur d'adresse contenue dans un registre ou remplacée par un label déclaré précédemment dans un programme.

➢ **B** adresse ; l'adresse est généralement remplacée par un label (**Etiquette**)

➢ **BX** registre ; aller à l'adresse pointée **par le registre**

➢ **BL** adresse ; (identique à B adresse) et **BLX** registre (identique à BX registre) seulement avec le branchement on sauvegarde l'adresse suivant BL(X) dans le registre LR (registre R14)

➢ Pour revenir d'un branchement BL(X) il suffit de remettre LR dans Pc par l'instruction: BX lr

➢On utilise aussi BL pour exécuter une fonction en l'appelant par son label:

BL Nom_fonction

Et le retour de la fonction par: **BX lr**

➢ Comme les autres instructions on peut rendre ces instructions exécutées sous condition en ajoutant le suffixe adéquat

Exemple:

```
SUBS r0, r1, r2     ; r0= r1 - r2
BEQ Etiquette       ; aller à Etiquette si le résultat de l'opération
                    ; précédente est nul (i.e. r0= r1 - r2=0)
```

Exemple d'exécution conditionnelle

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

Assembleur: Directives d'assemblage

- Pour faciliter la programmation, en plus des instructions, il existe des outils permettent de :
  - Définir des :
    - ✓ des étiquettes (labels),
    - ✓ constantes,
    - ✓ variables,
    - ✓ tables
    - ✓ Symboles
- Des macros–fonctions
- On appelle cela des **directives d'assemblage**

Directives d'assemblage: Constantes et variables

➢ L'assembleur nous permet de **donner un nom à des adresses mémoires**: ce sont les constantes et les variables

- **Constante** : ne change pas
  - ✓ Déclarer une constante (réserve de l'espace en **Flash EPROM**)
- **Variable** : peut changer
  - ✓ Déclarer une variable (réserve de l'espace en mémoire **SRAM**)

➢Définir des constantes:

Syntaxe:

Nom_constante EQU valeur

Exemple:

```
ENABLE EQU 0x1
MOV    R1,  #ENABLE     ; Move immediate data to register
```

Directives d'assemblage: Constantes et variables

➢**Définir des variables:** lorsqu'on définit une variable elle aura automatiquement une adresse dans la mémoire de données (SRAM).

Syntaxe: **Nom_variable DCB valeur** ; varibales byte-size

Nom_variable DCW valeur ; variables Half-word-size

Nom_variable DCD valeur ; variables word-size

Nom_variable DCQ valeur ; Double-word-size

Exemple: **X DCB 0x7**

Y DCW 0x1149

Nombre DCD 0x12345678

Directives d'assemblage: SPACE et FILL

- La directive SPACE réserve un bloc mémoire rempli par des zéros.
- La directive FILL réserve un bloc mémoire à remplir par la valeur donnée.

Syntaxe: {*label*} **SPACE** *expr*

{*label*} **FILL** *expr*{,*value*{,*valuesize*}}

Exemple:

```
AREA  Données, DATA, READWRITE
   Tab1   SPACE   255          ; defines 255 bytes of zeroed store
   Tab2   FILL    50,0xAB,1    ; defines 50 bytes containing 0xAB
```

label: une étiquette facultative.

expr: le nombre d'octets à remplir ou à mettre à zéro.

value: la valeur à affecter aux octets réservés. la valeur est facultative et si elle est omise, elle vaut 0.

valuesize*:** la taille, **en octets**, de ***Value. Elle peut être 1, 2 ou 4. Elle est facultatif et si elle est omisse, elle vaut 1.

Directives d'assemblage: Etiquette

La forme générale d'une instruction est:

{<Étiquette>}

[<instruction ou directive>] {**;** <commentaire>}

- Les lignes ne contenant que des commentaires ou étiquettes ne sont pas comptées.
- Les étiquettes (labels) seront remplacées par l'adresse de l'instruction qui suit.

Exemple:

```
Start     ; « Start » est une étiquette qui va être remplacer
          ; par l'@ de l'instruction [MOV r0, #0]
      MOV r0, #0      ; mise à zéro de r0
      MOV r2, #10     ; charger la valeur 10 dans r2
Loop  ; «  Loop » est une étiquette qui va être remplacer
          ;  par l'@ de l'instruction [ADD r0, r0, r2, LSL #1]
      ADD r0, r0, r2, LSL #1 ; r0=r0+2*r2
      SUBS r2, r2,#1 ; r2--
BNE Loop ; Si r2 != 0 l'exécution va reprendre à partir de
              ;l'instruction [ADD r0, r0, r2, LSL #1]
B Start   ;Branchement non conditionnel à l'étiquette « Start »
```

LDR pseudo-instruction

LDR est une instruction utilisée pour **charger à partir de la mémoire.** Seulement, on peut l'utiliser comme **pseudo-instruction** semblable à MOV pour **charger un registre** par soit:

- **Une valeur constante immédiates.** Cela est important lorsque la valeur immédiate ne peut pas être chargée dans le registre par l'instruction **MOV car elle est de taille 32 bits.**

  Syntaxe : **LDR**{*cond*} Rd, **= imme32**
  LDR{*cond*} Rd, = imme8
  LDR{*cond*} Rd, = imme16

  Exemples : **LDR R1, = 0x12345678**
  LDR R3,=0x1ff

- **Une adresse**

  Exemple :

```
MY_NUMBER        DCD    0x12345678
LDR R3, =MY_NUMBER ; Get the memory address value of MY_NUMBER
```

LDR pseudo-instruction

Exercice : Ecrire un programme qui permet de déclarer une variable X de taille 16 bits et une constante Y. Charger le registre R1 avec la valeur de X et le registre R2 avec la valeur de Y.

```
     AREA donnees, DATA, READWRITE
X DCB 0x7      ; Les varibles doivent être déclarer dans une zone mémoire ReadWrite
               ; (dans la mémoire des données SRAM)

     AREA |.text|, CODE, READONLY

Y EQU 0x9      ; Les constantes doivent être déclarer dans une zone mémoire ReadOnly
               ; (dans la mémoire CODE flashEPROM)
 ENTRY
    EXPORT __main
__main
    LDR R1, = X      ; Charger l'adresse de la variable X dans R1
    LDRB R1, [R1]    ; Charger la valeur de variable X dans R1
                     ; R1= 0000 0007H
    LDR R2, = Y      ; R2= 0000 0009H

    NOP
    END
```

Ces deux instructions **sont nécessaires pour charger la valeur d'une variable dans un registre**. Pour charger l'@ nous avons utilisé LDR, mais pour charger la valeur, nous avons utilisé LDR**B** par ce que la taille de la variable X est de 16 bits.

Y est une constante, donc on peut affecter sa valeur à R2 en utilisant LDR pseudo-instruction

LDR pseudo-instruction

Exercice : Ecrire un programme qui permet de déclarer un tableau **tab** de taille 100 entiers sur 8 bits initialisés par défaut à 0 et une variable result sur 32 bits

• remplit le tableau avec les valeurs de 1 à 100

Le tableau doit être déclarer dans une zone mémoire ReadWrite (dans **la mémoire des données SRAM**

```
        AREA donnees, DATA, READWRITE
tab space 100 ; initialiser 100 cases à 0 pointée par l'étiquette tab

    AREA |.text|, CODE, READONLY
    LDR R0,=tab     ;R0 contient l'@ de début du tableau
    LDR R1,=100
    LDR R2,=0
; on va remplir le tableau
loop
    ADD R2,R2,#1
    STRSB R2, [R0],#1     ;Memory[R0]= R2, R0=R0+1 pour pointer sur
                          ;l'élément suivant du tableau
    CMP R1, R2 ;lorsque R2=100 on terminé le remplissage des 100 éléments
    BNE loop
; R2 contient 100 (taille du tableau) à la sortie de la boucle
```

LDR pseudo-instruction

Exemple: Tracer le code suivant et préciser le mode d'@ de chaque inst LDR/STR

LDR R0, =0x20005000 ; **Chargement de R0.** R0 =0x2000 5000. LDR pseudo-inst
LDR R1, =0x20005004 ; **Chargement de R1.** R1=0x2000 5004. LDR pseudo-inst
LDR R2, =0x20005008 ; **Chargement de R2.** R2 =0x2000 5008. LDR pseudo-inst
LDR R3, =0x2000500C ; **Chargement de R3.** R3 =0x2000 500C. LDR pseudo-inst

MOV R4, #0x2 ; **Chargement de R4.** R4 =0x2
MOV R5,#0x8 ; **Chargement de R5.** R5 =0x8
MOV R6,#0xAA ; **Chargement de R6.** R6 =0xAA
MOV R7,#0xCC ; **Chargement de R7.** R7 =0xCC

STR R6,[R2] ; (@basé). **Memory[R2]=** R6 ➔ **Memory[**0x2000 5008**]= 0xAA**
LDR R10, [R2],R5, LSR #1 ; (@ Post-indexé). **R10=**Memory[R2] **= Memory[**0x2000 5008**]= 0xAA**
; **R2 = R2+(R5 >> 1)** = 2000 5008 h+ 4 h = 2000 500C h
STR R7,[R1,#8] ; (@ Pré-indexé). **Memory[R1+8]=** R7 ➔ **Memory[**0x2000 500C**]= 0xCC**

LDR R8, [R2] ; (@basé). **R8=** Memory[R2] = **Memory[**0x2000 500C**] = 0xCC**
STR R2, [R0,R4, LSL #3] ; (@ Pré-indexé). **Memory[R0+(R<<3)]=** R2
; **Memory[**0x2000 5000+0x10**]= 0x2000 500C**
; **Memory[**0x2000 5010**]= 0x2000 500C**
LDR R9, [R0,#16] ; (@ Pré-indexé). **R9 =** Memory[R0+16]= Memory[R0+10h]
R9= Memory[0x2000 5000+10h] **= 2000 500C h**
STR R5,[R3,#-4] ;(@ Pré-indexé). **Memory[R3 – 4h]=** R5
; Memory[2000 500C h – 4h]= 8h ➔ **Memory[**0x2000 5008**]= 0x8**

LDR pseudo-instruction

Trace d'exécution de l'exemple (code exécuté avec visUAL)

R0	0x20005000
R1	0x20005004
R2	0x2000500C
R3	0x2000500C
R4	0x2
R5	0x8
R6	0xAA
R7	0xCC
R8	0xCC
R9	0x2000500C
R10	0xAA

Start address: 0x20005000 End address: 0x20005013

Word Address	Byte 3	Byte 2	Byte 1	Byte 0	Word Value
0x20005000	0x0	0x0	0x0	0x0	0x0
0x20005004	0x0	0x0	0x0	0x0	0x0
0x20005008	0x0	0x0	0x0	0x8	0x8
0x2000500C	0x0	0x0	0x0	0xCC	0xCC
0x20005010	0x20	0x0	0x50	0xC	0x2000500C

Exercice

;Une mémoire SRAM dispose de 20 bits d'@,
; unité d'@ 2 octets.
;Ecrire un programme assembleur qui permet de:
; -- Ranger la valeur 2000 0100 h dans R4
; -- Calculer la capacité mémoire en Ko en utilisant les instructions logiques de décalage
; -- Ranger le résultat à l'adresse contenu dans R4.

```
LDR   R4, =0x20000100    ; Chargement de R4
MOV  R1,#0x1             ; lnitialization de R1 à 1 avant de faire la multiplication Logique
                         ; (i.e. multiplication par décalage à gauche)
MOV  R2, R1 ,LSL #11     ; la capacité de la mémoire = 2^10 * 2 = 2^11
STR   R2, [R4]           ; stockage du résultat dans Mémoire[R4]
```