Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

a.12 - Modes d'adressage

Classes T-2/I-2 // 2018-2019

Daniel Gachet | HEIA-FR/TIC

a.12 | 16.11.2018

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

Contenu

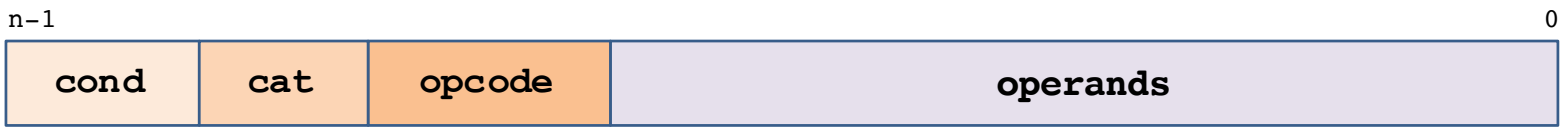
- Introduction
- Traitement de données
- Echange de données avec la mémoire
- Echange de données avec les coprocesseurs

Introduction - Format du jeu d'instructions

- Un jeu d'instructions basé sur une taille fixe permet un décodage simple et efficace des instructions par le µP
- Une exécution conditionnelle de chaque instruction permet d'éviter des branchements et ainsi d'augmenter les performances de traitement du µP
- Le jeu d'instructions d'un µP générique peut être classifié en 3 grandes catégories
  - Traitement de données dans les registres
  - Transfert de données entre mémoire et registres
  - Branchements

Introduction - Format du jeu d'instructions (II)

- Le format d'une instruction peut être composé de 4 champs principaux
  - La condition (cond)
  - La catégorie (cat)
  - Le code de l'instruction (opcode)
  - Les opérandes (operands)



Introduction - ARM principe de fonctionnement

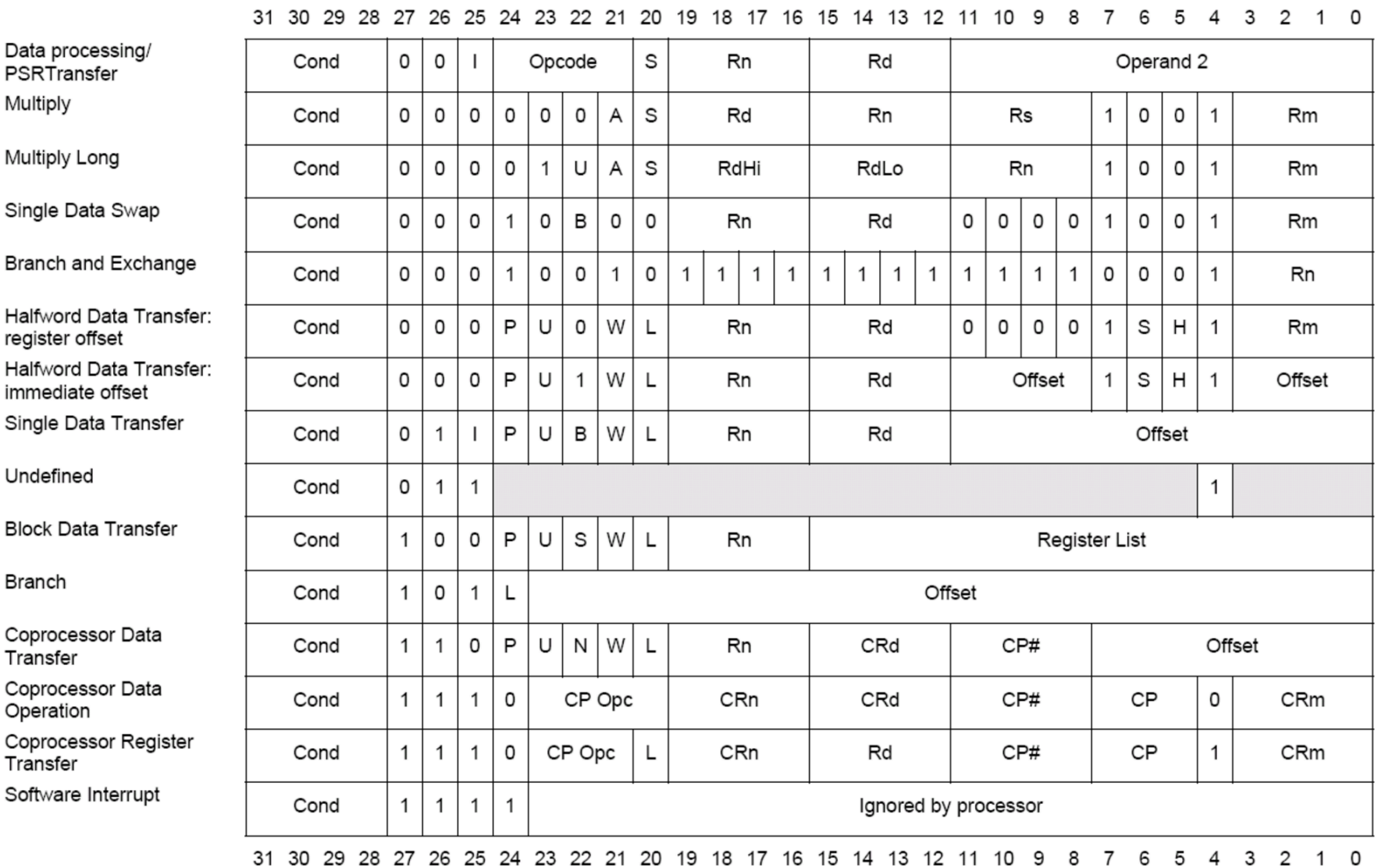
- Le µP ARM fonctionne sur le principe de « Load and Store »
- Pour effectuer des opérations sur des données stockées en mémoire, le µP ARM doit

read	charger les données de la mémoire dans ses registres internes
modify	effectuer les opérations souhaitées sur ces données
write	stocker à nouveau le résultat en mémoire

Introduction - Architecture ARM Cortex-A8

- Le µP ARM Cortex-A8 est un processeur RISC (Reduced Instruction Set Computer)
- Son jeu d'instructions est codé avec des mots d'une longueur fixe de 32 bits selon l'architecture ARMv7-A
- Toutes les instructions sont exécutés de façon conditionnelle
- Le µP dispose de
  - 13 registres à usage général (R0 à R12)
  - 3 registres spécialisés
    - R13/SP ↔ stack pointer
    - R14/LR ↔ link register
    - R15/PC ↔ program counter
  - 1 registre d'état (cpsr ↔ current program status register)
- Toutes les opérations arithmétiques et logiques sont exécutées sur des mots de 32 bits et ne peuvent être réalisées que dans les registres internes du µP, R0 à R15

Introduction - Encodage du jeu d'instructions ARM



Modes d'adressage

- Les µP ARM connaissent différents modes d'adressage [1].
  Ceux-ci peuvent être regroupés en trois grandes catégories :
  - Traitement de données
  - Echange de données avec la mémoire
  - Echange de données avec les coprocesseurs

[1] 01_armv7_architecture_reference_manual.pdf

Traitement de données

- Le traitement des données sont décrits à l'aide de 3 opérandes

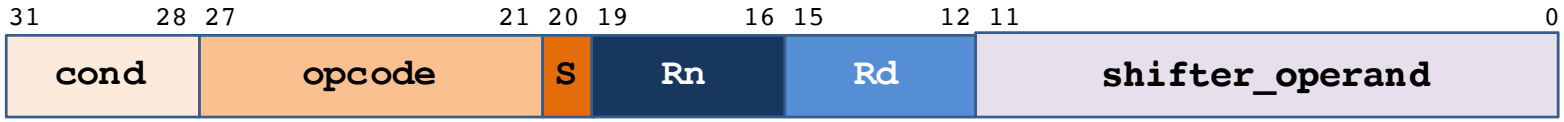
  <opérande_1> = <opérande_2> <opération> <opérande_3>

- La 3e opérande est appelée opérande de décalage *(shifter_operand)*
- Syntaxe

  <opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

  - <opcode>: opération (mov, add, sub,…)
  - {<cond>} : condition d'exécution, optionnel
  - {S} : mise à jour des fanions après exécution, optionnel
  - <Rd>: 1er opérande -> registre de destination
  - <Rn>: 2e opérande
  - <shifter_operand>: 3e opérande

- Format des instructions

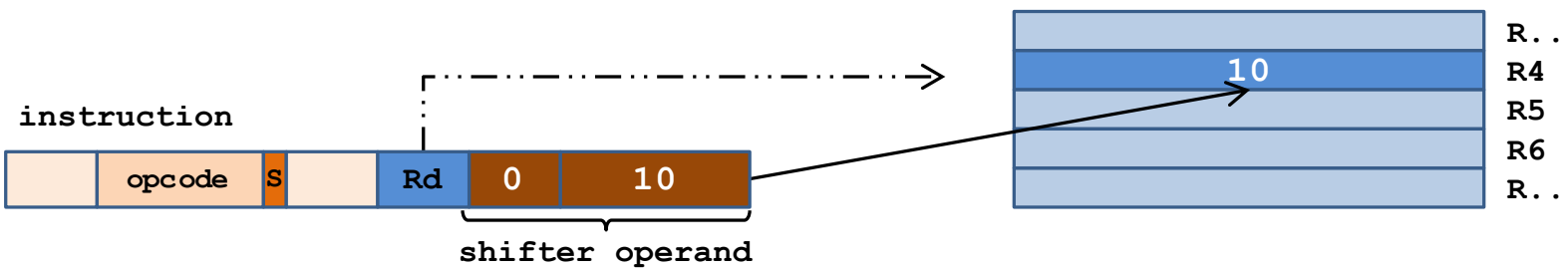


Opérande de décalage « Shifter Operand »

- L'opérande de décalage peut prendre différentes formes
  - Immédiate : `#<immediate>`
  - Registre : `<Rm>`
  - Décalage et rotation :
    - avec valeur immédiate : `<Rm>, LSL|LSR|ASR|ROR #<shift_immediate>`
    - par registre : `<Rm>, LSL¦LSR¦ASR¦ROR <Rs>`
    - rotation étendue : `<Rm>, RRX`

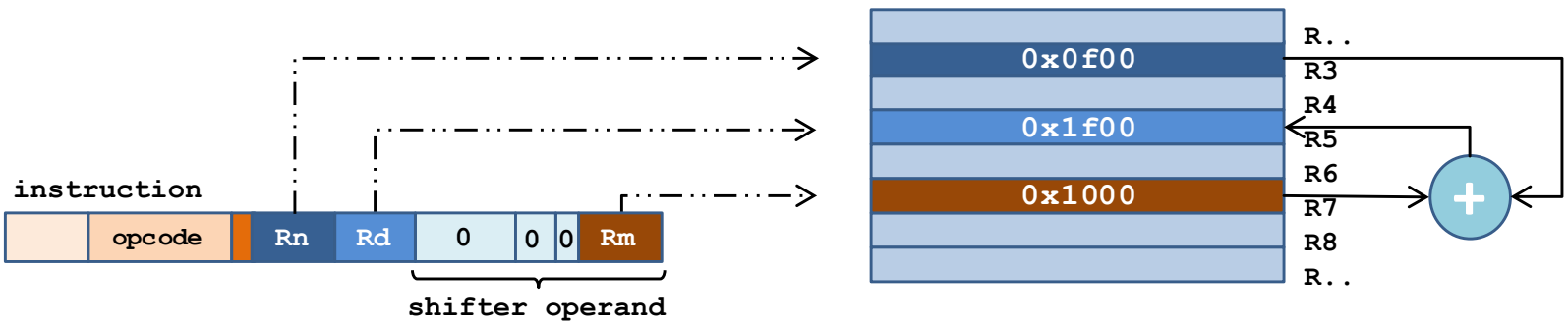
Opérande de décalage : immédiat

- La valeur de la 3e opérande *(shifter operand)* est une valeur numérique entière
- Elle est calculée en effectuant une rotation par la droite de la valeur immédiate, mot de 8 bits contenu dans l'instruction, dans un mot de 32 bits
- Exemple
  - `mov r4, #10`
  - Place dans le registre R4 la valeur décimale 10



Opérande de décalage : registre

- La valeur contenue dans le registre <Rm> est utilisée directement comme opérande par l'instruction de traitement de données
- Exemple
  - `add r5, r3, r7`
  - Le contenu du registre R3 est additionné au contenu du registre R7
    Le résultat est placé dans le registre R5



Opérande de décalage : décalage et rotation

- La valeur de la 3e opérande est obtenue par décalage ou par rotation de la valeur contenue dans un des registres du processeur
- Les 5 types de décalage et rotation sont

ASR	décalage arithmétique *(signed)* vers la droite
LSR	décalage logique *(unsigned)* vers la droite
LSL	décalage logique *(unsigned)* vers la gauche
ROR	rotation vers la droite
RRX	rotation étendue avec le carry vers la droite

- Le nombre de bits de décalage ou de rotation peut être spécifié soit
  - avec une valeur immédiate
  - par l'intermédiaire d'un registre

Opérande de décalage : ASR

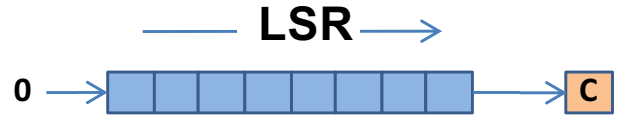
- Avec ASR, l'opérande de décalage est formé en décalant la valeur du registre <Rm> par la droite (→ division par 2 signée)
- Le bit du poids fort (Rm[31]) est inséré à la place du bit vacant
- Le dernier bit de poids faible décalé (Rm[0]) est, quant à lui, placé dans le Carry si une mise à jour des fanions est souhaitée (S=1)

- Exemple
  - `mov r5, r7, ASR #8`

  ou `asr r5, r7, #8`
  - Le contenu du registre R7 est décalé arithmétiquement vers la droite de 8 bits et le résultat est placé dans le registre R5.

Opérande de décalage : LSR

- Avec LSR, l'opérande de décalage est formé en décalant la valeur du registre <Rm> par la droite (→ division par 2 non-signée)
- Un 0 est inséré à la place du bit vacant
- Le dernier bit de poids faible décalé (Rm[0]) est, quant à lui, placé dans le Carry si une mise à jour des fanions est souhaitée (S=1)

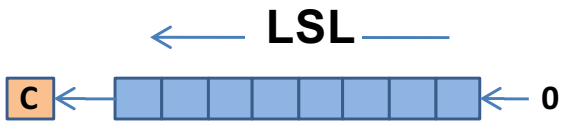


- Exemple
  - `mov r5, r7, LSR r6`

  ou `lsr r5, r7, r6`
  - Le contenu du registre R7 est décalé vers la droite par le nombre de bits spécifié dans le registre R6 et le résultat est placé dans le registre R5.

Opérande de décalage : LSL

- Avec LSL, l'opérande de décalage est formé en décalant la valeur du registre <Rm> vers la gauche (→ multiplication par 2)
- Un 0 est inséré à la place du bit vacant
- Le dernier bit de poids fort décalé (Rm[31]) est, quant à lui, placé dans le Carry si une mise à jour des fanions est souhaitée (S=1)



- Exemple
  - `orr r5, r4, r7, LSL r6`
  - Le contenu du registre R7 est décalé vers la gauche par le nombre de bits spécifié dans le registre R6, puis le contenu de R4 est additionné (addition logique bit à bit). Finalement, le résultat est placé dans le registre R5.

Opérande de décalage : ROR

- Avec ROR, l'opérande de décalage est formé en effectuant une rotation du contenu du registre <Rm> par la droite
- Le bit du poids faible (Rm[0]) est inséré à la place du bit de poids fort (Rm[31])
- Le dernier bit de poids faible décalé (Rm[0]) est, quant à lui, placé dans le Carry si une mise à jour des fanions est souhaitée (S=1)

- Exemple
  - `mov r5, r7, ROR #8`

  ou `ror r5, r7, #8`
  - Le contenu du registre R7 subit une rotation vers la droite de 8 bits et le résultat est placé dans le registre R5.

Opérande de décalage : RRX

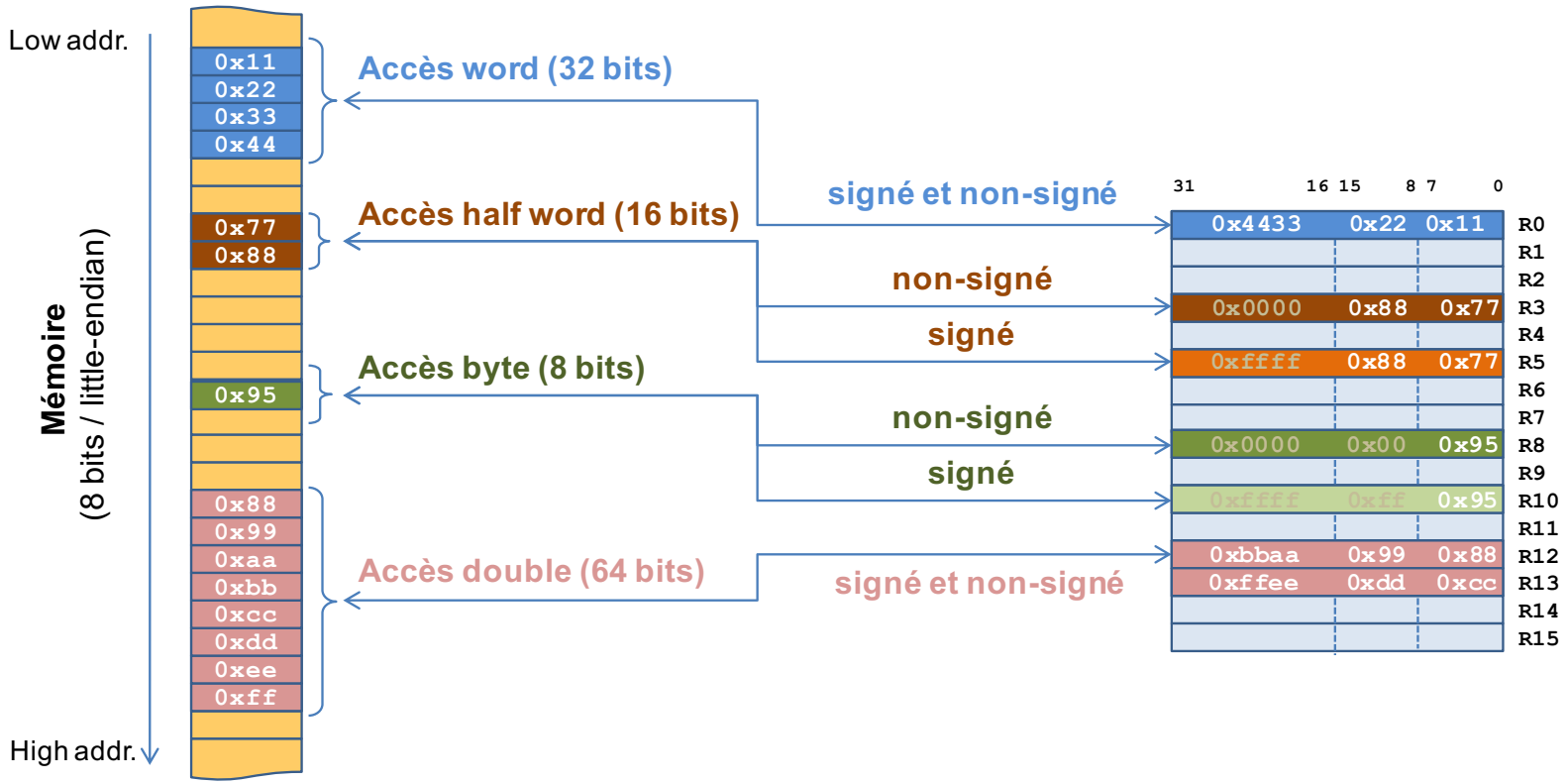
- Avec RRX, l'opérande de décalage est formé en effectuant une rotation de 1 bit du contenu du registre <Rm> par la droite
- Le Carry est inséré à la place du bit de poids fort (Rm[31])
- Le bit de poids faible (Rm[0]) est quant à lui placé dans le Carry si une mise à jour des fanions est souhaitée (S=1)

- Exemple
  - `movs r5, r7, RRX // C=1`

  ou `rrxs r5, r7`
  - Le contenu du registre R7 est subit une rotation vers la droite à travers le carry de 1 bit. Le carry est inséré dans le bit 31. Le bit 0 est placé dans le carry. Le résultat est placé dans le registre R5.

Echange de données avec la mémoire

- Les µP ARM fonctionnent sous le principe de « *Load and Store* ».
- Ce principe implique que pour traiter des données celles-ci doivent impérativement être chargées « *Load* » dans les registres du processeur, puis, après traitement, elles peuvent être à nouveau stockées « *Store* » dans la mémoire principale.



Echange de données - Les instructions...

- Syntaxe

  `LDR|STR{<cond>}{H|B|D|SH|SB} <Rd>, <addr_mode>`

  - `<LDR|STR>` : instruction (Load ou Store)
  - `{<cond>}` : condition d'exécution, optionnel
  - `{S}` : mise à jour des fanions après exécution, optionnel
  - `<Rd>` : registre de destination/source
  - `{H|B|D|SH|SB}` : taille et signe de la donnée, par défaut word
    H: halfword, B: byte, D: double
    SH: signed halfword, SB: signed byte
  - `<addr_mode>` : mode d'adressage

Addressing Mode - Mots de 32 bits et 8 bits non-signés

- L'opérande du mode d'adressage (offset/index) peut prendre les trois formes suivantes
  - Immédiat

```
    normal ldr|str{b} <Rd>, [<Rn>, #+/-<offset_12>]
 pré-indexé ldr|str{b} <Rd>, [<Rn>, #+/-<offset_12>]!
post-indexé ldr|str{b} <Rd>, [<Rn>], #+/-<offset_12>
```

  - Par registre

```
    normal ldr|str{b} <Rd>, [<Rn>, +/-<Rm>]
 pré-indexé ldr|str{b} <Rd>, [<Rn>, +/-<Rm>]!
post-indexé ldr|str{b} <Rd>, [<Rn>], +/-<Rm>
```

  - Par registre et décalage

```
    normal ldr|str{b} <Rd>, [<Rn>, +/-<Rm>, <shift> #<shift_imm>]
 pré-indexé ldr|str{b} <Rd>, [<Rn>, +/-<Rm>, <shift> #<shift_imm>]!
post-indexé ldr|str{b} <Rd>, [<Rn>], +/-<Rm>, <shift> #<shift_imm>
```

Addressing Mode - Mots de 16 bits, 8 bits signés et 64 bits

- L'opérande du mode d'adressage (offset/index) peut prendre les trois formes suivantes
  - Immédiat

```
    normal ldr|str{h|sh|sb|d} <Rd>, [<Rn>, #+/-<immediate_8>]
pré-indexé ldr|str{h|sh|sb|d} <Rd>, [<Rn>, #+/-<immediate_8>]!
post-indexé ldr|str{h|sh|sb|d} <Rd>, [<Rn>], #+/-<immediate_8>
```

  - Par registre

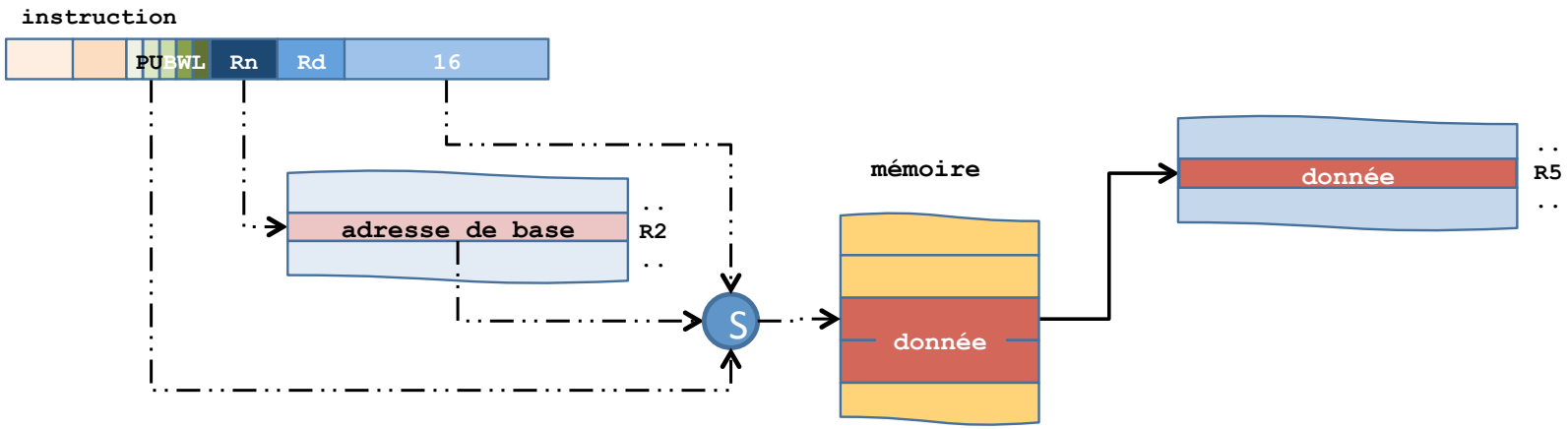
```
    normal ldr|str{h|sh|sb|d} <Rd>, [<Rn>, +/-<Rm>]
pré-indexé ldr|str{h|sh|sb|d} <Rd>, [<Rn>, +/-<Rm>]!
post-indexé ldr|str{h|sh|sb|d} <Rd>, [<Rn>], +/-<Rm>
```

Adressing Mode : Immediate offset/index

- L'adresse effective de la donnée en mémoire est calculée en additionnant ou soustrayant la valeur contenue dans le registre <Rn> avec la valeur de l'offset/index contenue dans l'instruction.
- Valeurs possibles pour
  - mots de 32-bits et 8-bits non-signés : 0 .. 4095
  - mots de 64-bits, 16-bits et 8-bits signés : 0 .. 255
- Un fois l'adresse effective calculée, la donnée placée en mémoire à cette même adresse est mise dans le registre <Rd> ou inversement.

Immediate offset/index - exemple...

- Exemple
  - `ldr r5, [r2, #16]`
  - Place la donnée (32 bits) contenue à l'adresse mémoire R2 + 16_{10} dans le registre R5.

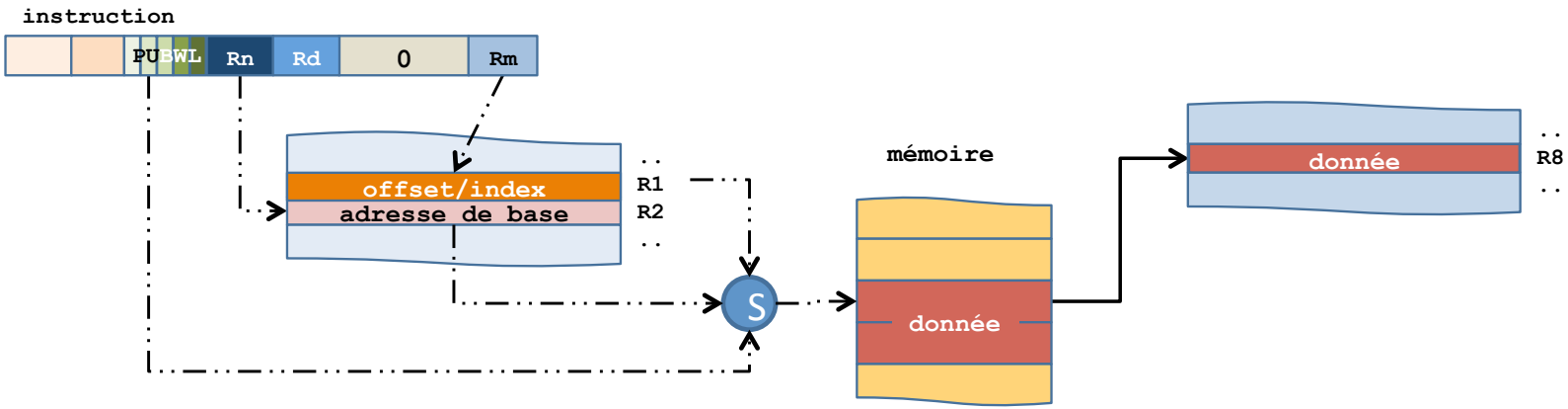


Adressing Mode : Register offset/index

- L'adresse effective de la donnée en mémoire est calculée en additionnant ou soustrayant la valeur contenue dans le registre <Rn> avec la valeur de l'offset/index contenue dans le registre <Rm>.
- Une fois, l'adresse effective calculée, la donnée placée en mémoire à cette même adresse est mise dans le registre <Rd> ou inversement.

Register offset/index - exemple...

- Exemple
  - `ldrb r8, [r2, -r1]`
  - Place la donnée (8 bits) contenue à l'adresse mémoire R2 – R1 dans le registre R8. Les 24 bits de poids fort sont mis à zéro.

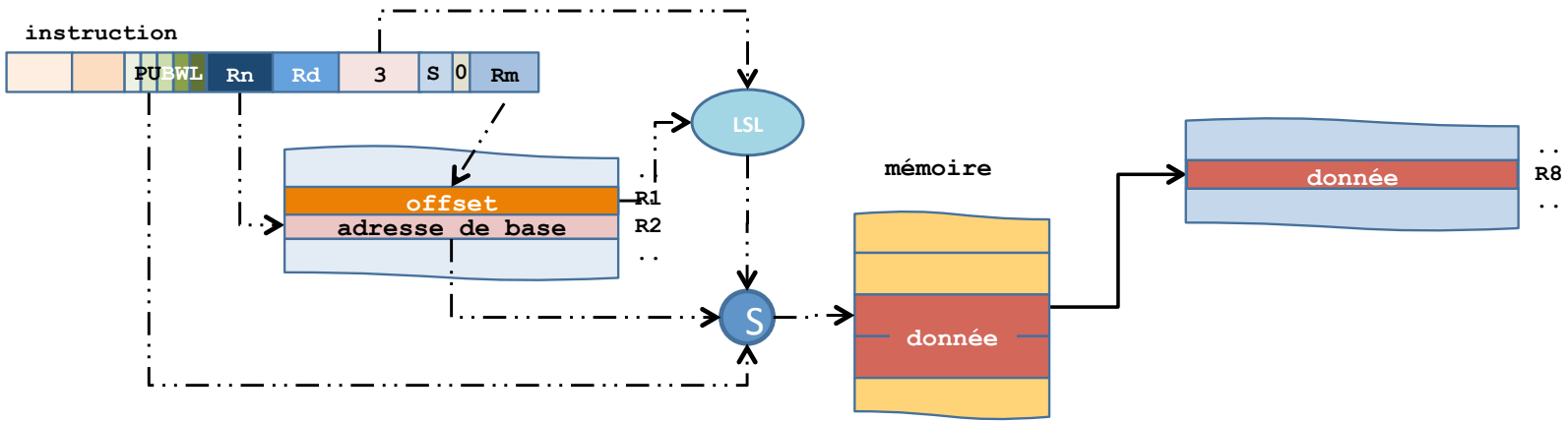


Adressing Mode : Scaled register offset/index

- L'adresse effective de la donnée en mémoire est calculée en additionnant ou soustrayant la valeur contenue dans le registre <Rn> avec la valeur de l'offset contenue dans le registre <Rm> après décalage ou rotation du nombre de bits spécifié par « shift_imm » codé dans l'instruction.
- Les 5 variantes *(identique à l'opérande de décalage)* sont
  - `[<Rn>, +/-<Rm>, LSL #<shift_imm>]` // shift_imm: 0..31
  - `[<Rn>, +/-<Rm>, LSR #<shift_imm>]` // shift_imm: 1..32
  - `[<Rn>, +/-<Rm>, ASR #<shift_imm>]` // shift_imm: 1..32
  - `[<Rn>, +/-<Rm>, ROR #<shift_imm>]` // shift_imm: 1..31
  - `[<Rn>, +/-<Rm>, RRX]`
- Un fois l'adresse effective calculée, la donnée placée en mémoire à cette même adresse est mise dans le registre <Rd> ou inversement.
- Ce mode d'adressage n'est possible que sur des données de 32-bits et de 8-bits non-signées !

Scaled register offset/index - exemple...

- Exemple
  - `ldrb r8, [r2, +r1, LSL #3]`
  - Place la donnée (8 bits) contenue à l'adresse mémoire R2 + (R1 « 3) dans le registre R8. Les 24 bits de poids fort sont mis à zéro.

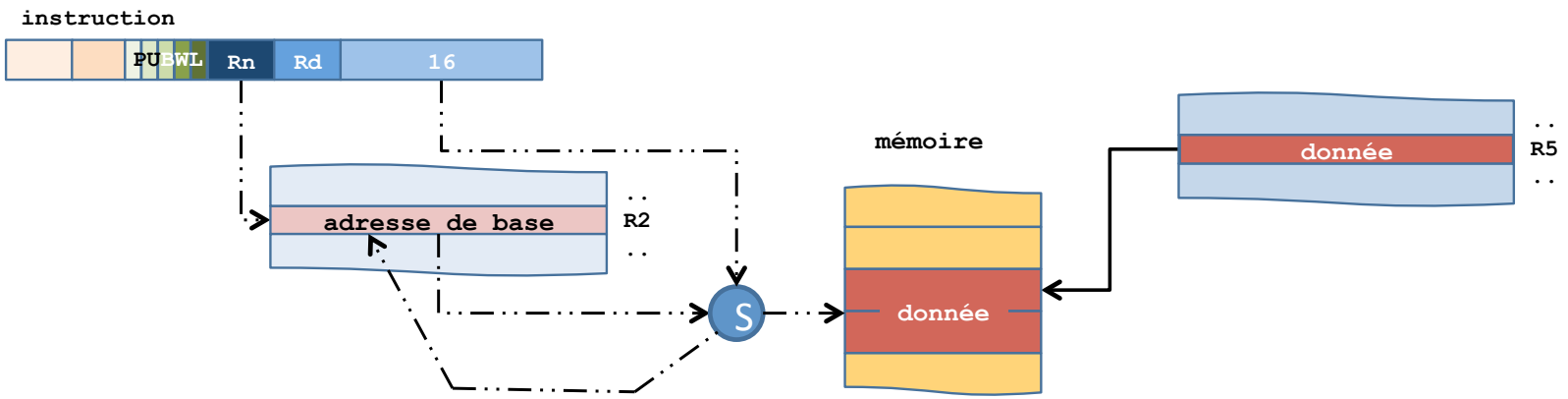


Adressing Mode : Pre-indexed

- Le mode d'adressage « pré-indexé » permet de stocker dans le registre <Rn> le résultat du calcul de l'adresse effective de la donnée en mémoire selon l'une des 3 méthodes précédentes.
- Ce n'est qu'après le calcul de l'adresse effective que la donnée est accédée.
- Ce mode permet d'implémenter très simplement des boucles d'itération.

Pre-indexed - exemple...

- Exemple
  - `strh r5, [r2, #10]!`
  - Calcule l'adresse effective de la donnée en mémoire en additionnant le contenu de R2 à 10_{10} et place le résultat dans R2.
  - La donnée (16 bits) contenue dans le registre R5 (16 bits de poids faible) est ensuite placée à l'adresse mémoire contenue dans R2.

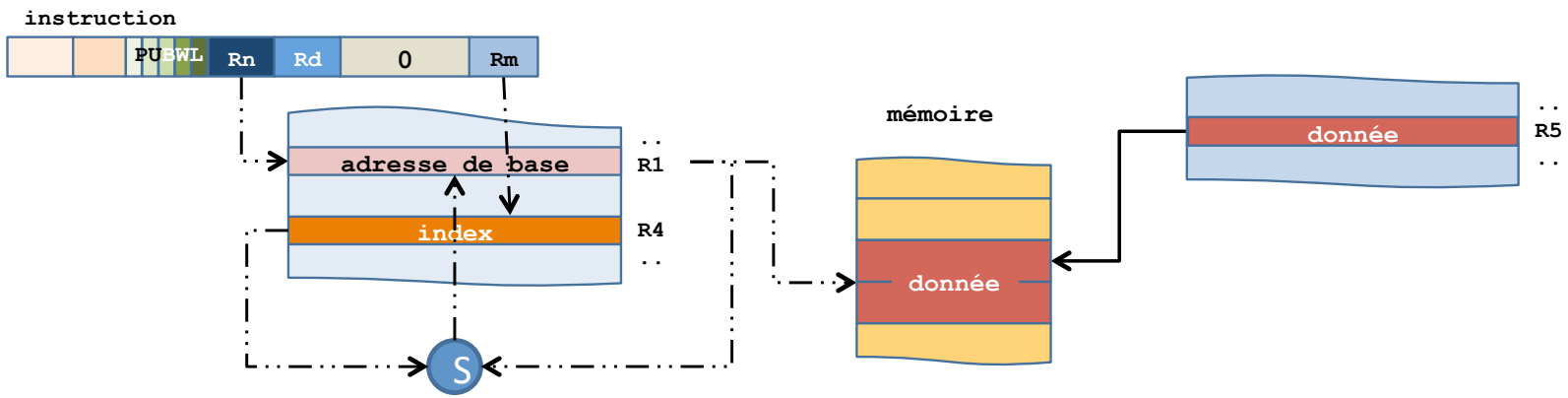


Adressing Mode : Post-indexed

- Le mode d'adressage « post-indexé » permet de stocker dans le registre <Rn> le résultat du calcul de l'adresse effective de la donnée en mémoire selon l'une des 3 premières méthodes d'adressage.
- Le calcul de la nouvelle adresse effective n'est exécuté qu'après avoir accédé la donnée stockée en mémoire.
- Ce mode permet d'implémenter très simplement des boucles d'itération.

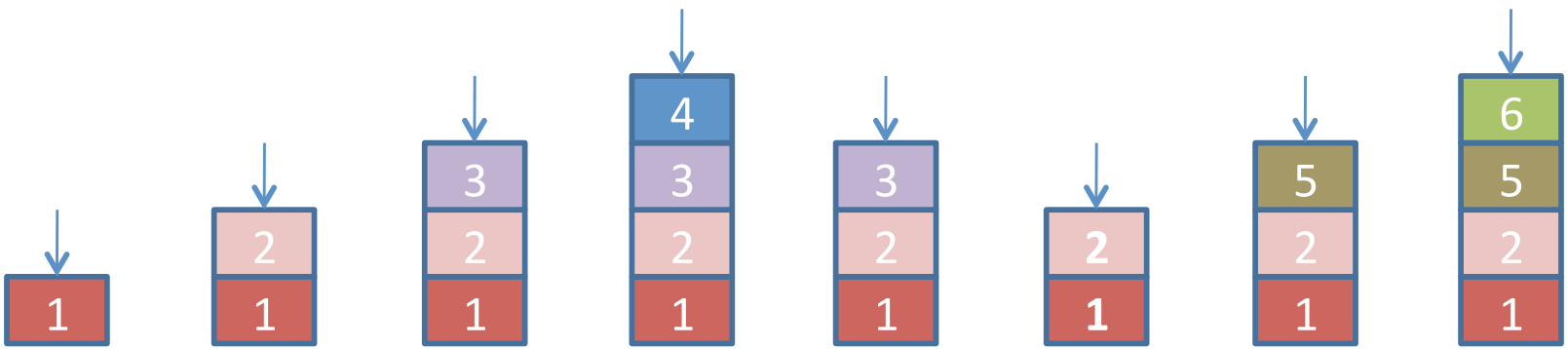
Post-indexed - exemple...

- Exemple
  - `strh r5, [r1], r4`
  - La donnée (16 bits) contenue dans le registre R5 (16 bits de poids faible) est placée à l'adresse mémoire contenue dans R1.
  - Ensuite, on calcule l'adresse effective de la donnée en mémoire en additionnant le contenu de R4 à R1 et place le résultat dans R1.



Echange multiple de données

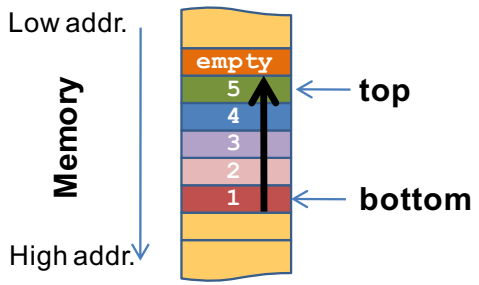
- La sauvegarde et la restitution d'une partie ou de l'ensemble des registres du processeur sont des tâches très courantes.
- Ces opérations sont très intéressantes lors d'appel de sous-routines pour conserver l'état des registres de la routine appelante. Elles sont spécialement utilisées par les langages de programmation évolués. On parle alors de pile (« stack »).
- La pile (« stack ») est simplement une liste linéaire d'éléments dont l'accès se fait en ajoutant ou en enlevant des éléments.



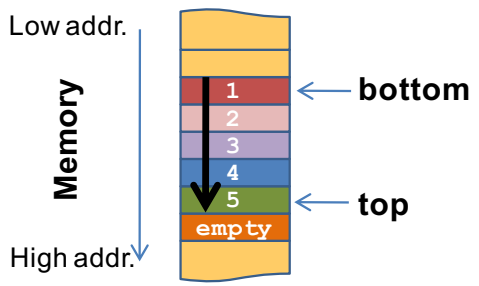
Représentation sous forme de pile (stack)

Echange multiple de données (II)

- Les données peuvent être stockées sur ces piles de deux façons différentes
  - Pile descendante (descending stack)
    Avec ce modèle, la pile croît vers le bas de la mémoire, c.à.d. les adresses mémoires sont décrémentées

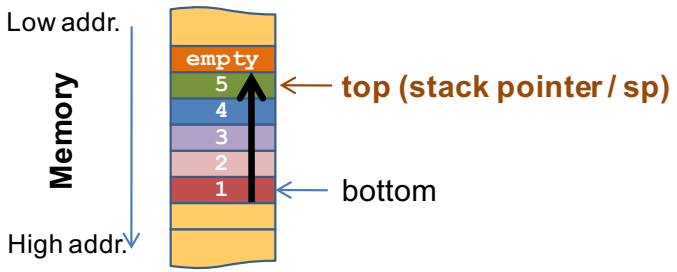


  - Pile ascendante (ascending stack)
    Dans ce modèle, la pile croît vers le haut de la mémoire, c.à.d. les adresses mémoires sont incrémentées

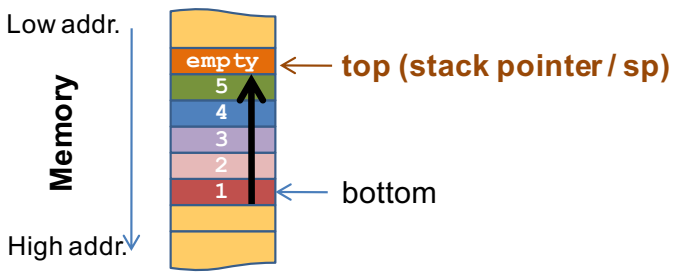


Echange multiple de données (III)

- Les µP ARM proposent deux types de piles
  - Pile pleine (full stack)
    Le pointeur de pile (top) indique la position mémoire de la dernière valeur stockée



  - Pile vide (empty stack)
    Le pointeur de pile (top) indique la position mémoire de la première place non utilisée



Echange multiple de données (IV)

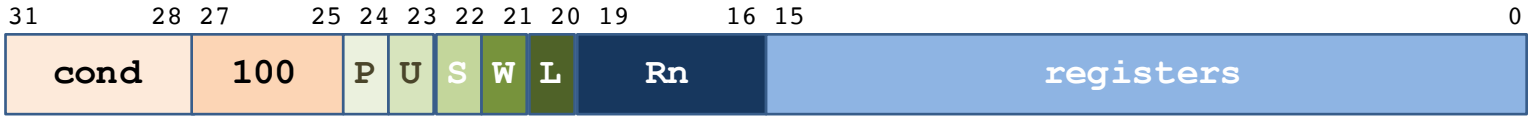
- Sur la base de ces différents modèles, les µP ARM fournissent un jeu d'instructions permettant d'effectuer ces opérations et d'échanger le contenu des registres avec celui de la mémoire.
- Syntaxe

  `LDM|STM{<cond>}<addr_mode> <Rn>{!}, <registers>{^}`

  - `LDM|STM :` instruction (load multiple ou store multiple)
  - `{<cond>} :` condition d'exécution, optionnel
  - `<Rn> :` registre utilisé comme pointeur de pile (sp)
  - `<addr_mode> :` mode d'adressage
  - `<registers> :` liste des registres à sauver/récupérer
  - `{!} :` mise à jour du registre <Rn> après exécution
  - `{^} :` sauvegarde/restitution des registres utilisateurs
- Le contenu du registre avec le plus petit numéro est placé à l'adresse la plus basse en mémoire, tandis que celui du registre avec le plus grand numéro est placé à l'adresse la plus haute.

Echange multiple de données (V)

- Encodage des opérandes dans l'instruction.

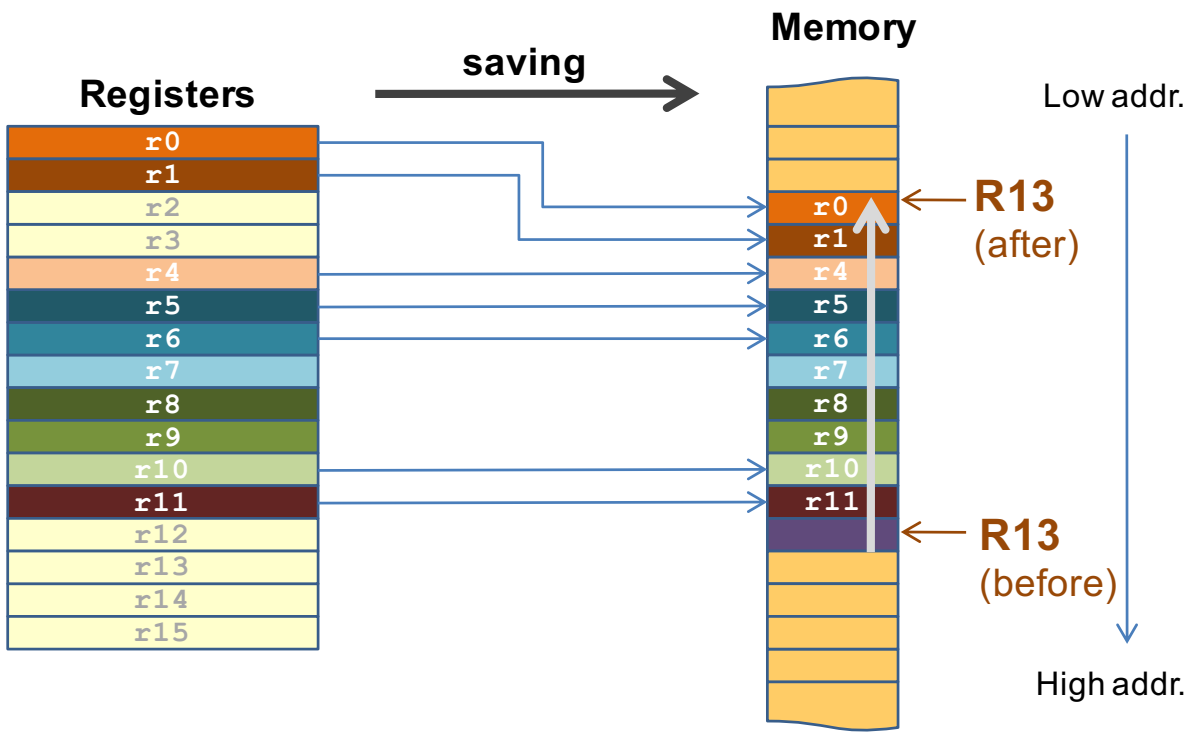


- Mode d'adressage sous leur forme générique
  - IA increment after *(sauvegarder les registres dans une structure)*
  - IB increment before
  - DA decrement after
  - DB decrement before
- Mode d'adressage sous leur forme générique
  - FA full ascending
  - FD full descending *(C/C++ stack convention)*
  - EA empty ascending
  - ED empty descending

Sauvegarde des registres - exemple...

- Exemple
  - `stmfd r13!, {r0,r1,r4-r11}`

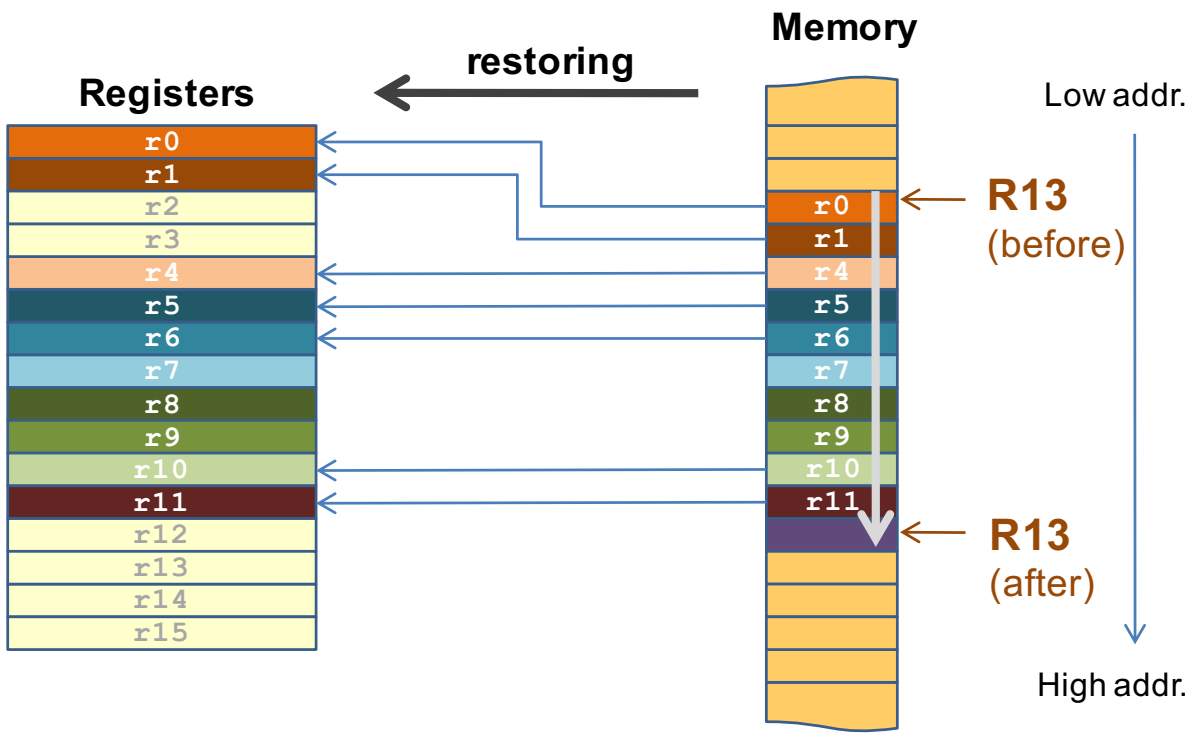
  ou `push {r0,r1,r4-r11}`
  - Le contenu des registres R0, R1 et R4 à R11 est sauvé dans la mémoire à l'adresse spécifiée par le registre R13/SP (stack pointer).
  - R13/SP sera mis à jour après l'exécution de l'opération (décrémenté).
  - Méthode utilisée : full descending stack



Restauration des registres - exemple...

- Exemple
  - `ldmfd r13!, {r0,r1,r4-r11}`

  ou `pop {r0,r1,r4-r11}`
  - Le contenu des registres R0, R1 et R4 à R11 est restauré depuis la mémoire à l'adresse spécifiée par le registre R13/SP (stack pointer).
  - R13/SP sera mis à jour après l'exécution de l'opération (incrémenté).
  - Méthode utilisée : full descending stack



Echange de données avec les coprocesseurs

- Les µP ARM sont équipés de coprocesseurs pour effectuer des tâches spécifiques, p.ex. gestionnaire de la mémoire (MMU), de la cache, etc.
- Pour l'échange de données avec les coprocesseurs, le µP ARM implémente un jeu d'instructions avec la syntaxe suivante
- Syntaxe

```
<opcode>{<cond>}{L} <coproc>, <CRd>, <addr_mode>
```

  - `<opcode>` : opération (cdp, ldc, mcr, …)
  - `{<cond>}` : condition d'exécution, optionnel
  - `{L}` : Load / Store
  - `<coproc>` : nom du coprocesseur
  - `<CRd >` : registre du coprocesseur
  - `<addr_mode >` : mode d'adressage