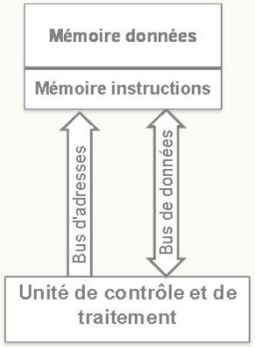
heig-vd
Haute Ecole d'Ingénierie
du Canton de Vaud



ARO2

Micro-architecture d'un processeur
La partie MEMORY ACCESS

Basé sur le cours du prof. E. Sanchez
et le cours ASP du prof. M.Starkier

Romuald Mosqueron

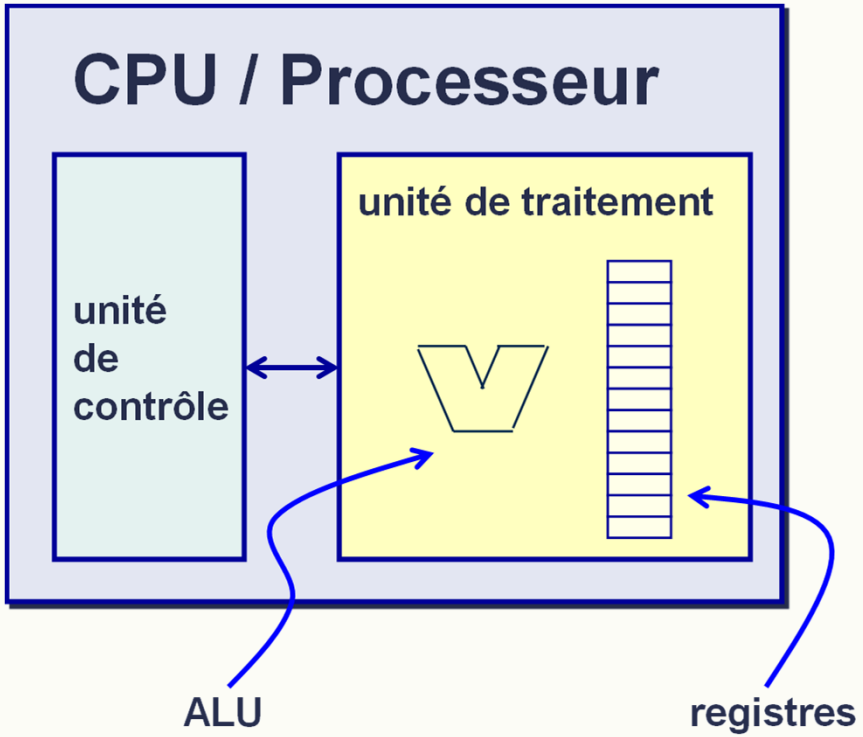
Le bloc MEMORY ACCESS

- Le bloc MEMORY ACCESS fait partie de l'unité de traitement
- Le bloc MEMORY ACCESS permet les accès à la mémoire de donnée en lecture et écriture
- Le bloc MEMORY ACCESS comporte une interface entre :
  - les bus internes du CPU et les bus d'adresse et de données de la mémoire
  - les signaux internes du CPU et les signaux de contrôle de la mémoire (lecture /écriture, formats ....)

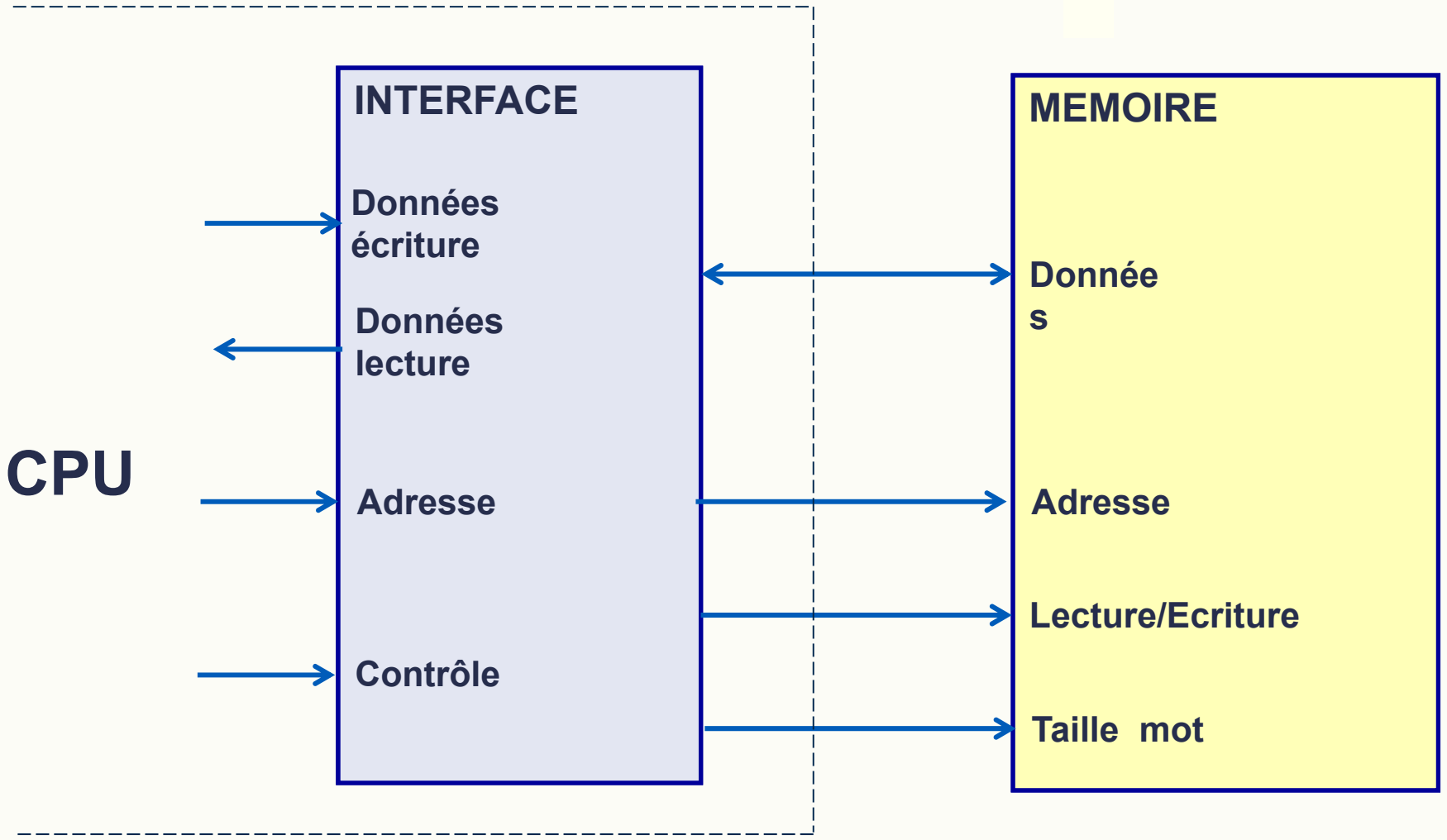
Le bloc MEMORY ACCESS

Rappel

Bloc diagramme d'un processeur avec l'unité de contrôle et l'unité de traitement



ARM :Bloc diagramme
Interface mémoire de donnée



ARM :Bloc diagramme
Interface mémoire de donnée

- Quand le processeur exécute une instruction de transfert de donnée vers la mémoire (écriture), le mécanisme de traitement fournit à l'interface:
  - la donnée à écrire dans la mémoire, provenant d'un registre, l'adresse de l'emplacement mémoire, calculée dans le CPU les signaux de contrôle indiquant entre autres la taille de la donnée (octet, mot de 16 bits, ...)

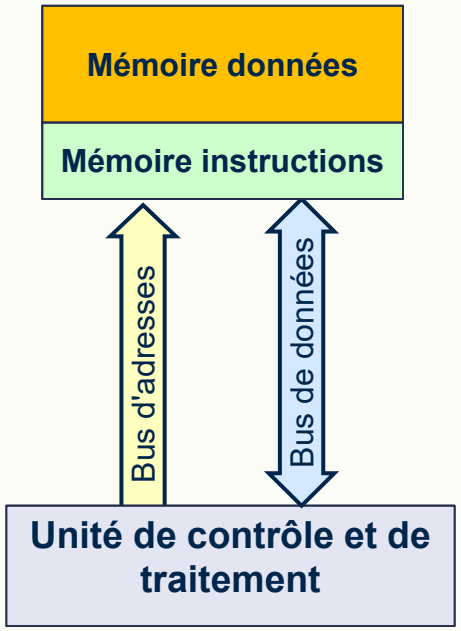
- Quand le processeur exécute une instruction de transfert de donnée à partir de la mémoire (lecture), le mécanisme de traitement fournit à l'interface:
  - l'adresse de l'emplacement mémoire, calculée dans le CPU les signaux de contrôle indiquant entre autres la taille de la donnée (octet, mot de 16 bits, ...) et reçoit la donnée lue dans la mémoire pour écriture dans un registre

Cours AR02

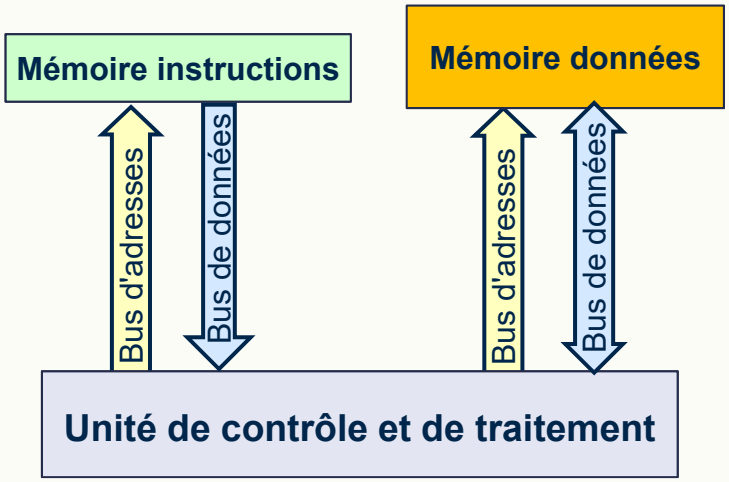
ARCHITECTURE ET MEMORY MAPPING

Architectures (rappel)

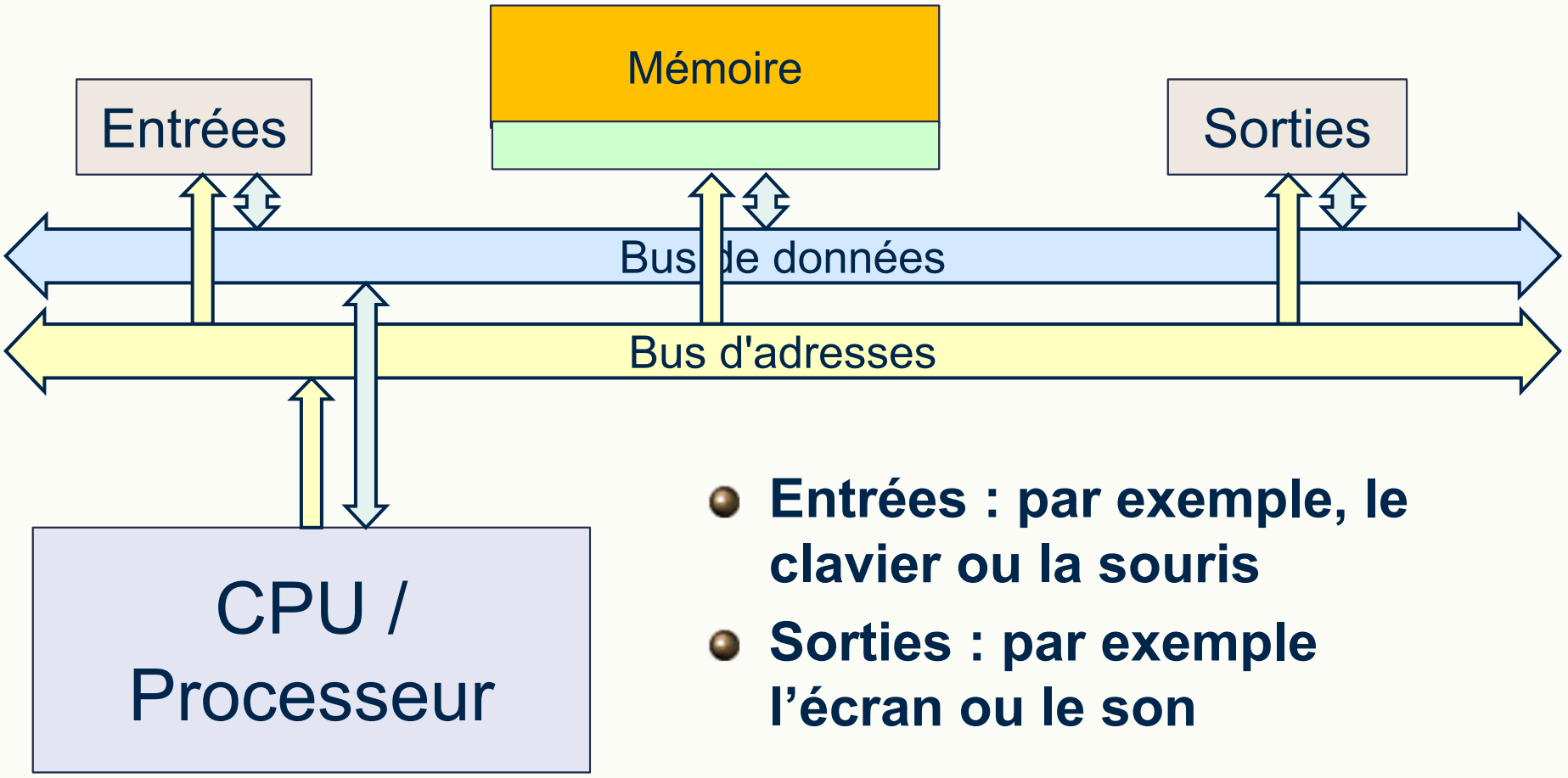
Von Neumann



Harvard



Architecture globale d'un ordinateur (rappel)



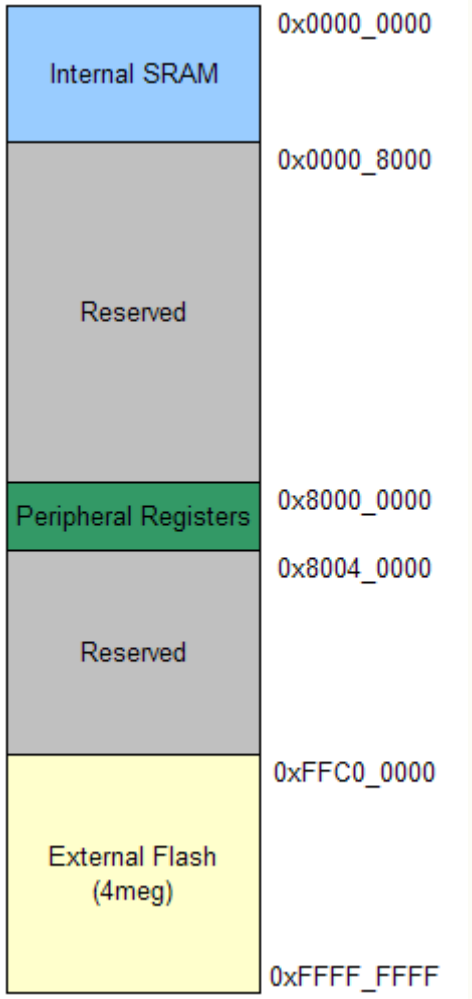
- **Entrées : par exemple, le clavier ou la souris**
- **Sorties : par exemple l'écran ou le son**

Adressage mémoire
Memory mapping

- L'espace mémoire total adressable par un processeur avec un bus d'adresse de n bits est 2^n bytes
- Les mémoire et les périphériques occupent des espaces définis à des adresses précises : ceci s'appelle le *memory mapping*
- L'adresse de début et la taille des espaces mémoire sont définis par un décodage d'adresse effectué par comparaison des bits de poids forts

Exemple de *memory map*

- L'espace d'adressage total est
  2^{32} = 4GB 0x00000000 – 0xFFFFFFFF
- Chaque zone mémoire occupe un espace d'adressage:
  - Internal SRAM 32KB 0x00000000 - 0x00007FFF
  - External Flash 4MB 0xFFC00000 – 0xFFFFFFFF
- Les registres des périphériques occupent l'espace d'adressage:
  - 256KB 0x80000000 – 0x8003FFFF



Exemple

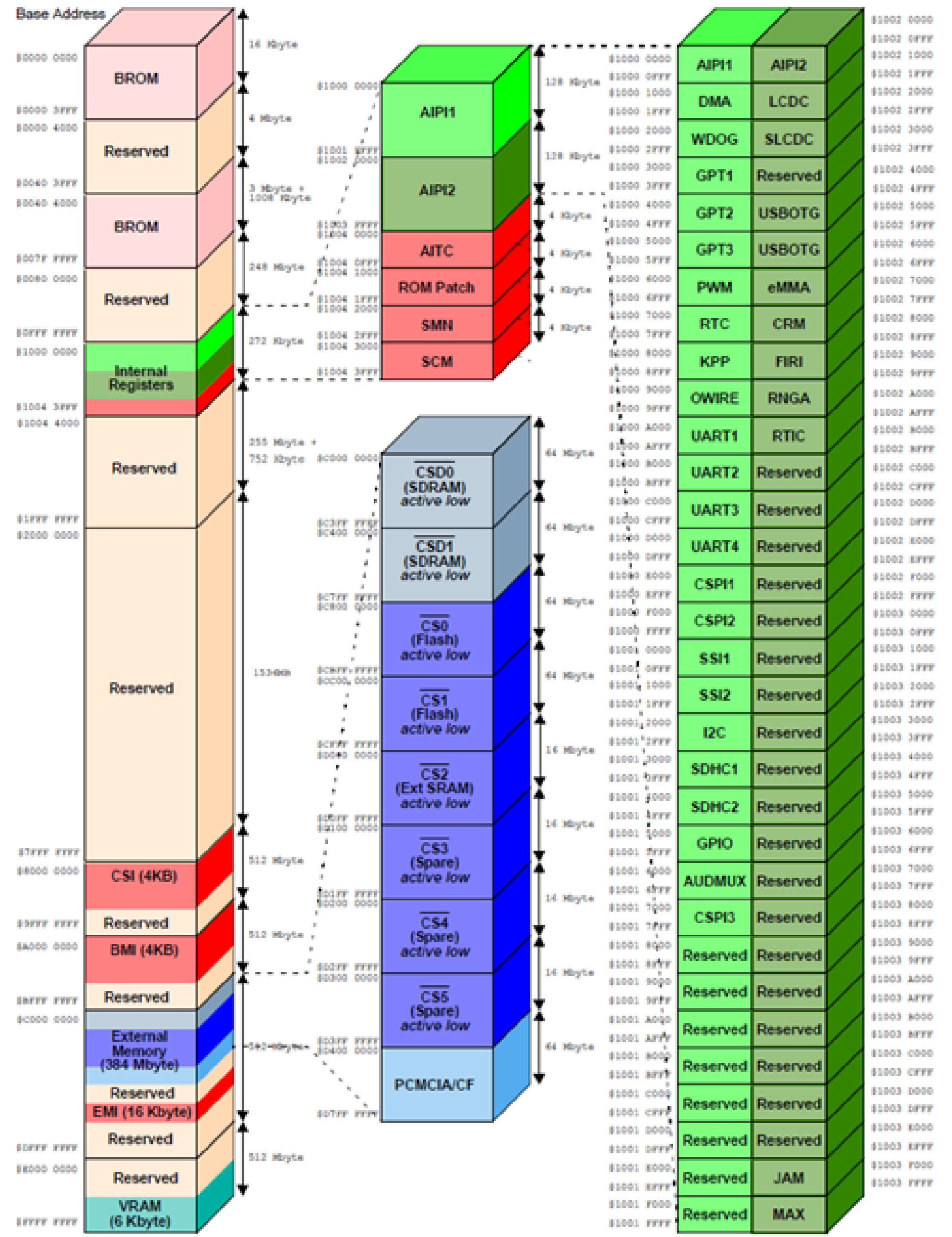
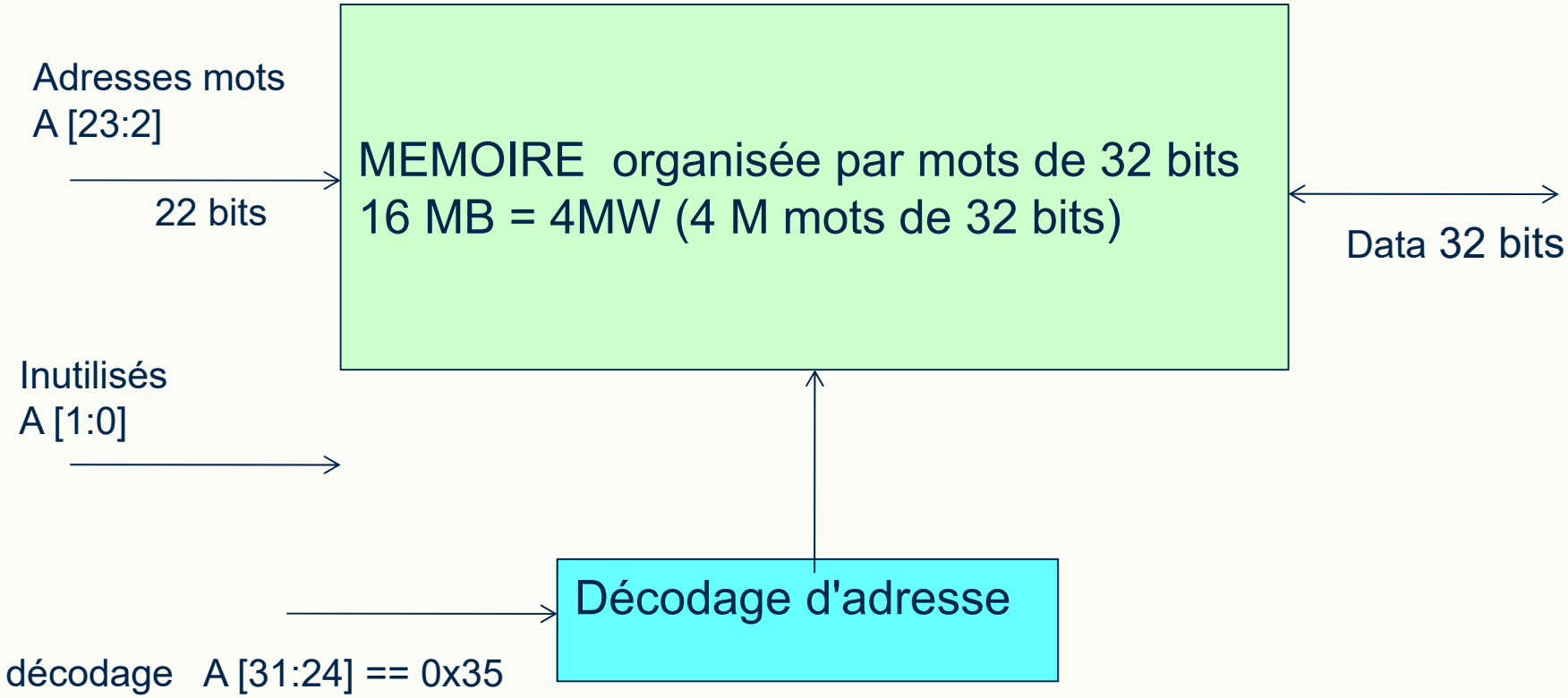


Figure 3-2. I.MX21 Physical Memory Map (4 Gbyte)

Décodage d'adresse d'une mémoire

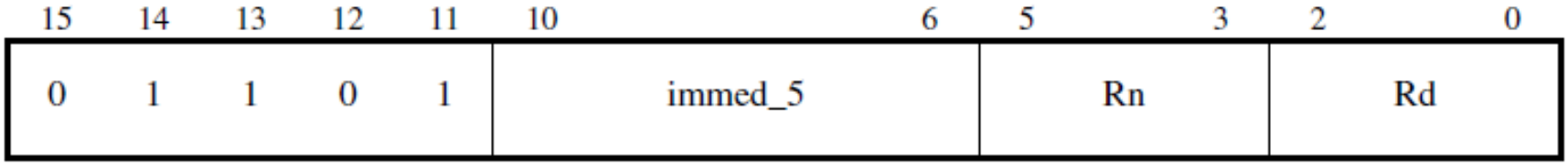
Adresse de la mémoire : 0x35000000
Bus d'adresse 32 bits



Cours AR02

LECTURE ET ÉCRITURE DES DONNÉES

Lecture mémoire 32 bits:
Instructions ARM LDR

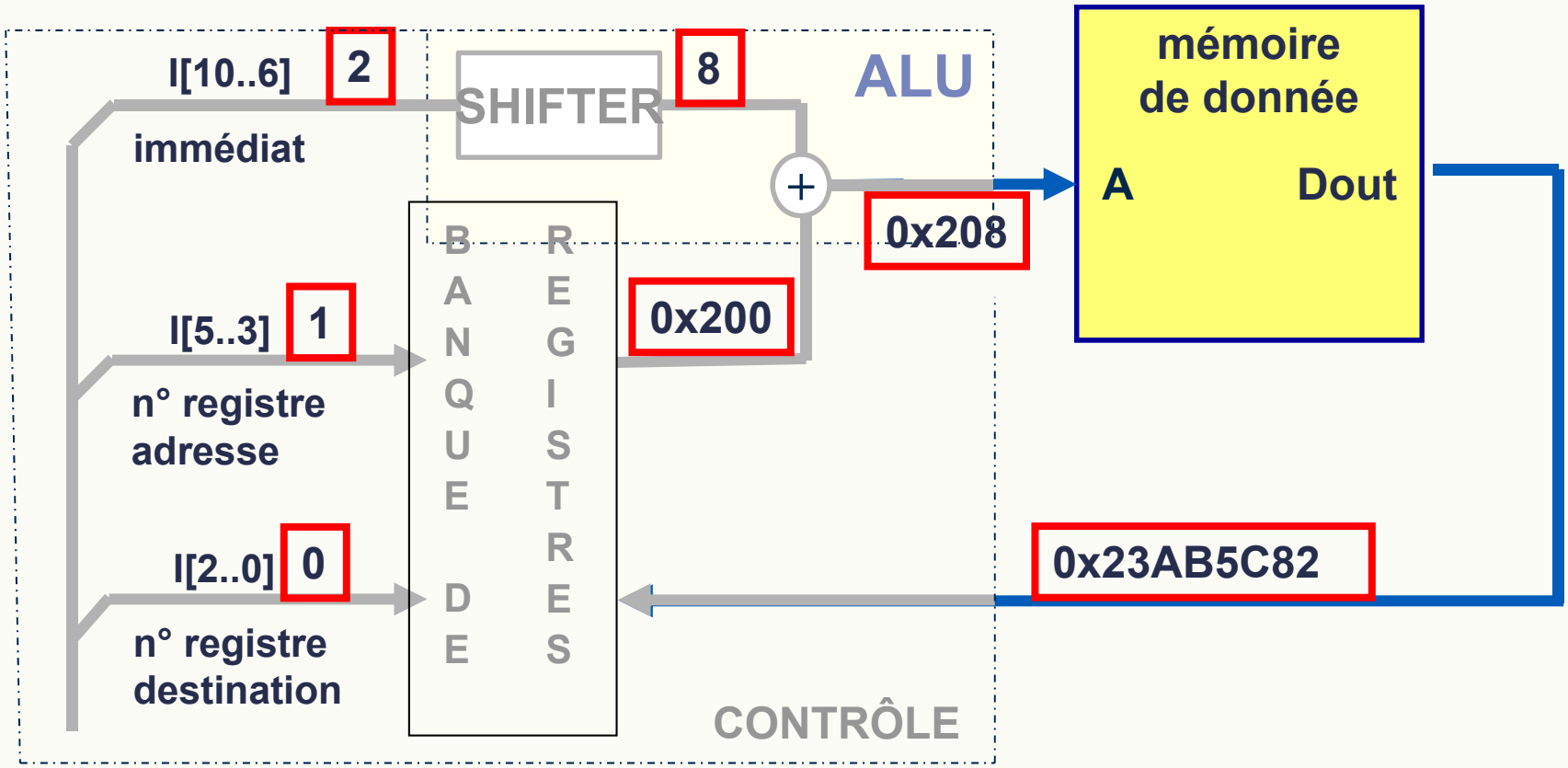


- LDR signifie Load to Register
- syntaxe : LDR <Rd>, [<Rn>, #<immed_5> * 4]
- charge dans le registre Rd la donnée (32 bits) en mémoire (lecture mémoire)
- l'adresse mémoire est calculée en ajoutant la valeur dans Rn à l'offset immed_5 (5 bits non signé) multiplié par 4
- Rd ← M[Rn+immed_5*4]

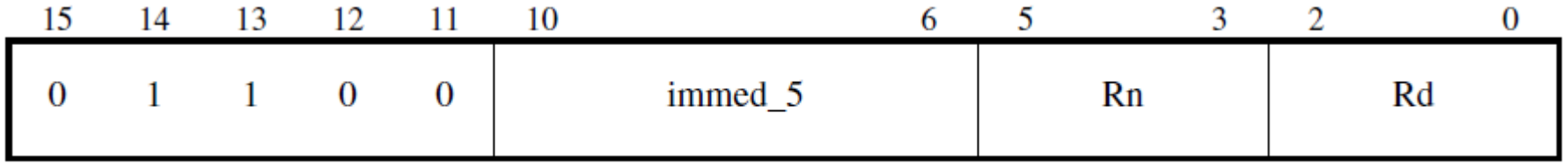
⚠ l'offset permet d'adresser les éléments d'un tableau

Lecture d'une donnée

- Exemple ARM : M[0x208] = 0x23AB5C82, r1 = 0x200
- LDR r0, [r1,#2*4]



Ecriture mémoire 32 bits:
Instructions ARM STR



- STR signifie Store from Register
- syntaxe : STR<Rd>, [<Rn>, #<immed_5> * 4]
- écrit en mémoire la donnée contenue dans le registre Rd
- l'adresse mémoire est calculée en ajoutant la valeur dans Rn à l'offset immed_5 (5 bits non signé) multiplié par 4
- M[Rn+immed_5*4] ← Rd

⚠ l'offset permet d'adresser les éléments d'un tableau

Ecriture ou lecture autres formats
Instructions ARM LDRH/STRH/LDRB/STRB

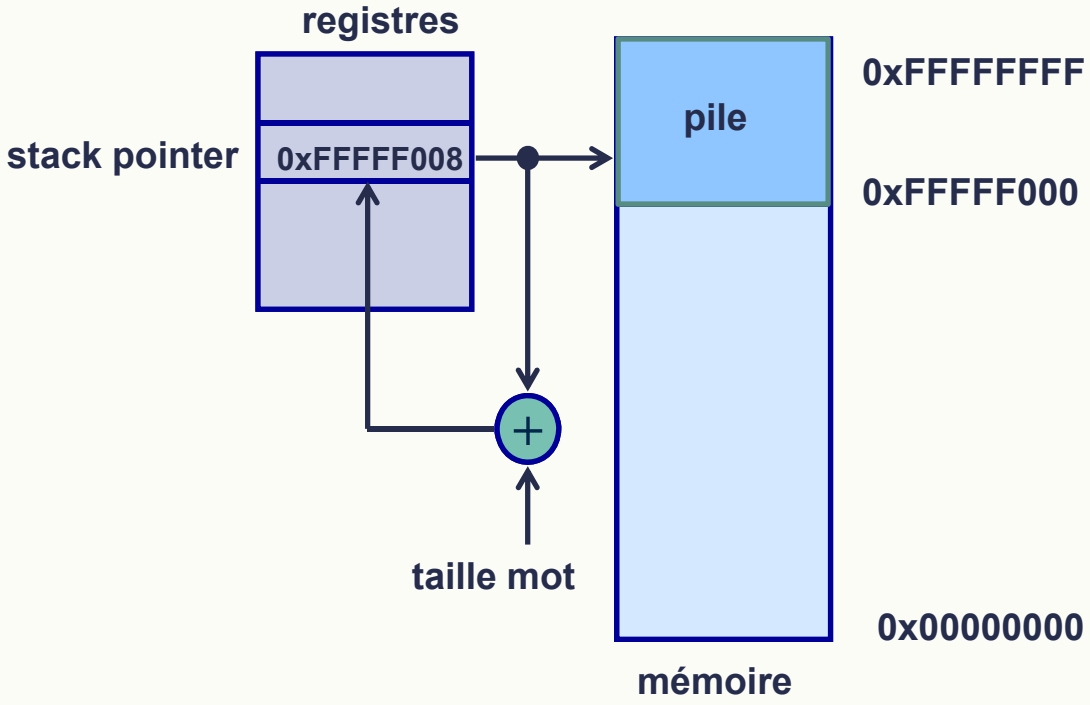
- Ecriture /Lecture mots de 16 bits
  - LDRH <Rd>, [<Rn>, #<immed_5> * 2]
  - STRH <Rd>, [<Rn>, #<immed_5> * 2]
- Ecriture /Lecture octets (8 bits)
  - LDRB <Rd>, [<Rn>, #<immed_5>]
  - STRB <Rd>, [<Rn>, #<immed_5>]
- Ces instructions permettent d'adresser les éléments d'un tableau de mots de 16 bits ou 8 bits

Cours AR02

PILE

Pile (stack)- Rappel

- Pile => zone mémoire réservée
- LIFO: Last Input , First Output
- Registre dédié => stack pointer
  - Pointe sur le bas de la pile à l'initialisation



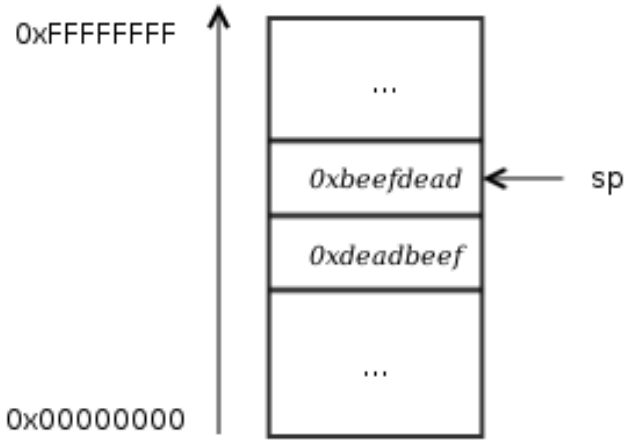
Pile (stack)

- Ecriture / lecture en mode auto-incrémenté

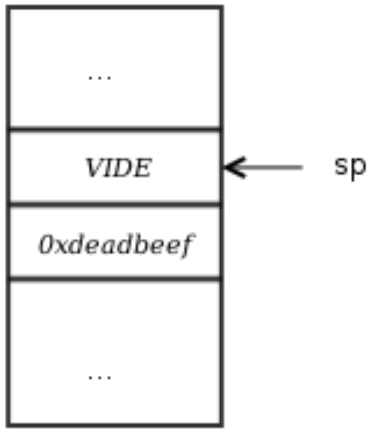
pile ascendante

- 1er mode (pile ascendante vide)
  - Push : écriture avec post-incrémentation
  - Pop : lecture avec pré-décrémentation
- 2ème mode (pile ascendante pleine)
  - Push : écriture avec pré-incrémentation
  - Pop : lecture avec post-décrémentation

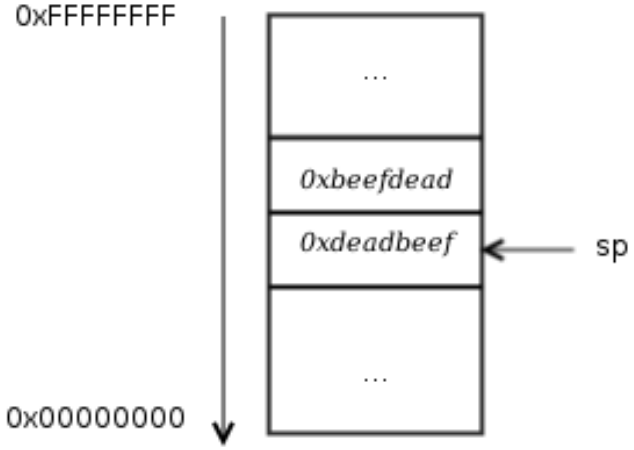
même principe pour pile descendante



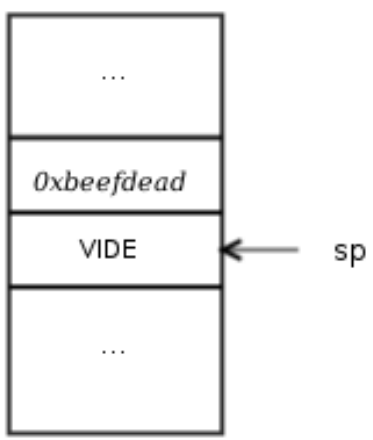
Pile ascendante



Pile ascendante vide



Pile descendante



Pile descendante vide

Pile (stack)

- ++i pré-incrémentation

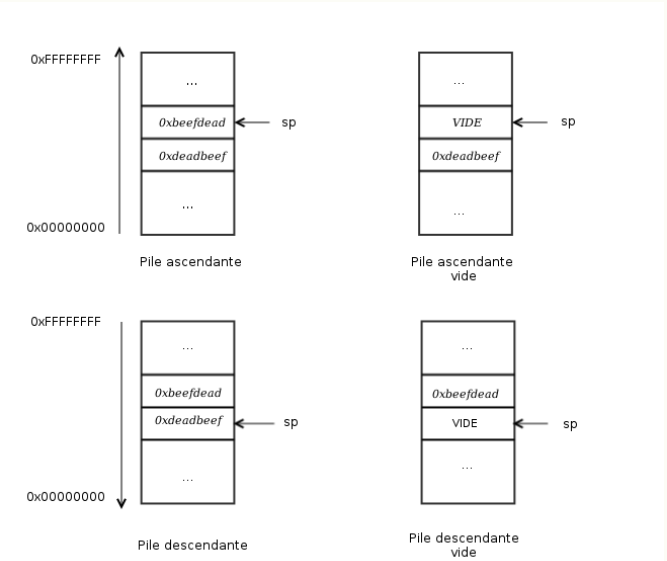
Incrémentation de SP

Puis écriture dans le registre à l'adresse SP

- i++ post incrémentation

Ecriture dans le registre à l'adresse SP

Puis incrémentation de SP



Pile (stack)

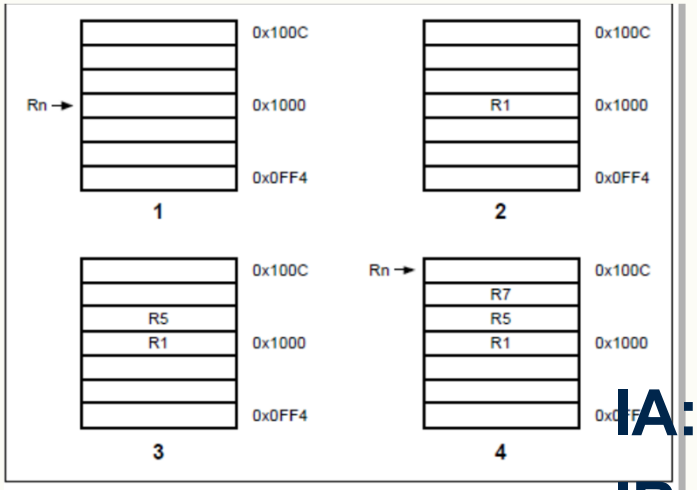


Figure 4-19: Post-increment addressing

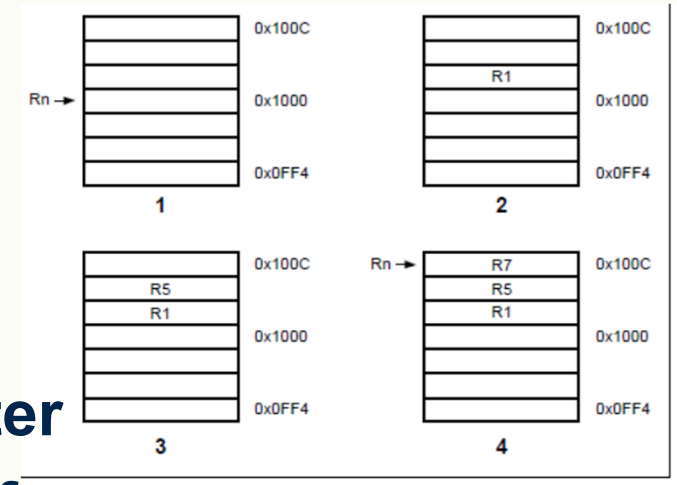


Figure 4-20: Pre-increment addressing

IA: Increment After

IB: Increment Before

DA: Decrement After

DB: Decrement Before

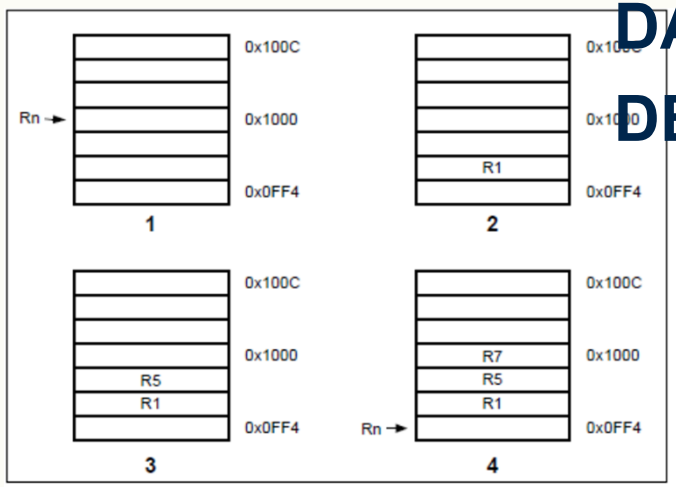


Figure 4-21: Post-decrement addressing

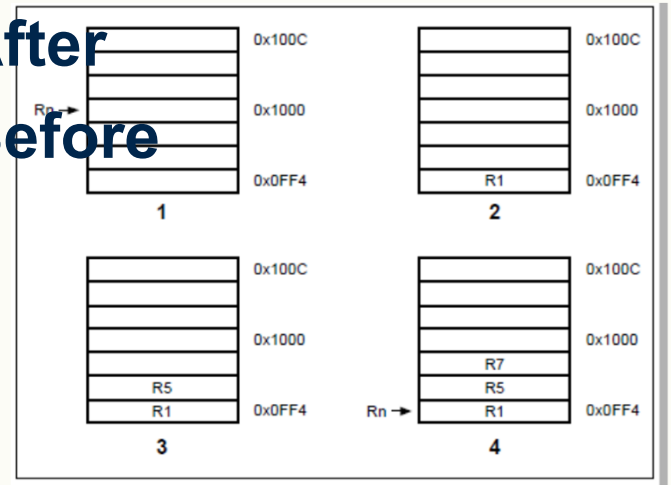
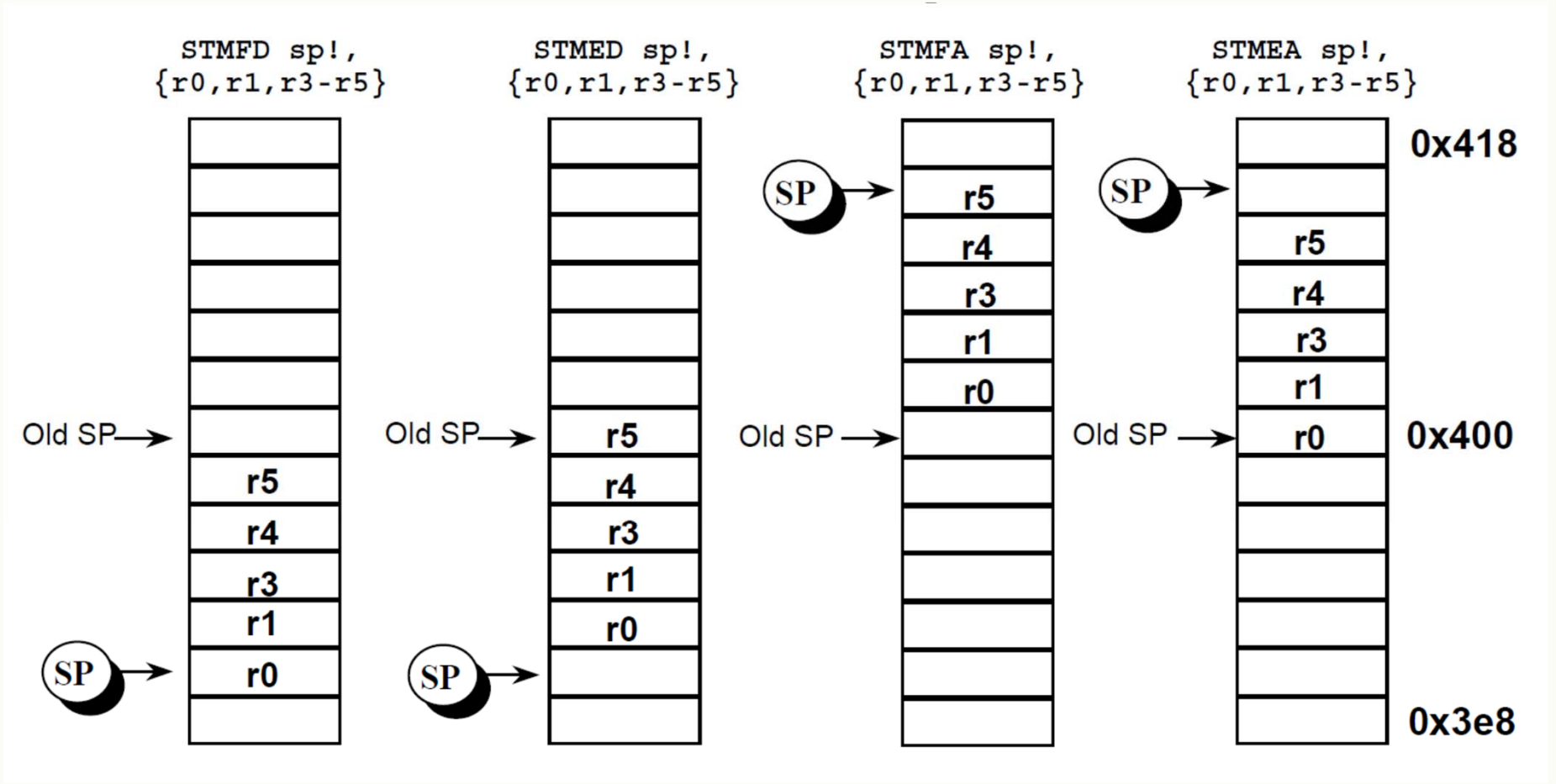


Figure 4-22: Pre-decrement addressing

Pile (stack)


STMFD sp!, {r0,r1,r3-r5}
STMED sp!, {r0,r1,r3-r5}
STMFA sp!, {r0,r1,r3-r5}
STMEA sp!, {r0,r1,r3-r5}
Old SP
SP
r5
r4
r3
r1
r0
0x418
0x400
0x3e8

Pile (stack)

- The value of the stack pointer can either:
  - Point to the last occupied address (Full stack)
    - and so needs pre-decrementing (ie before the push)
  - Point to the next occupied address (Empty stack)
    - and so needs post-decrementing (ie after the push)
- The stack type to be used is given by the postfix to the instruction:
  - STMFD (STMDB) / LDMFD (LDMIA) : Full Descending stack
  - STMFA (STMIB) / LDMFA (LDMDA): Full Ascending stack.
  - STMED (STMDA) / LDMED (LDMIB): Empty Descending stack
  - STMEA (STMIA) / LDMEA (LDMDB): Empty Ascending stack

Note: ARM Compiler will always use a Full descending stack.

Ecriture et lecture dans la pile

- Pile ascendante : les données sont écrites en incrémentant les adresses
- Pile descendante : les données sont écrites en décrémentant les adresses

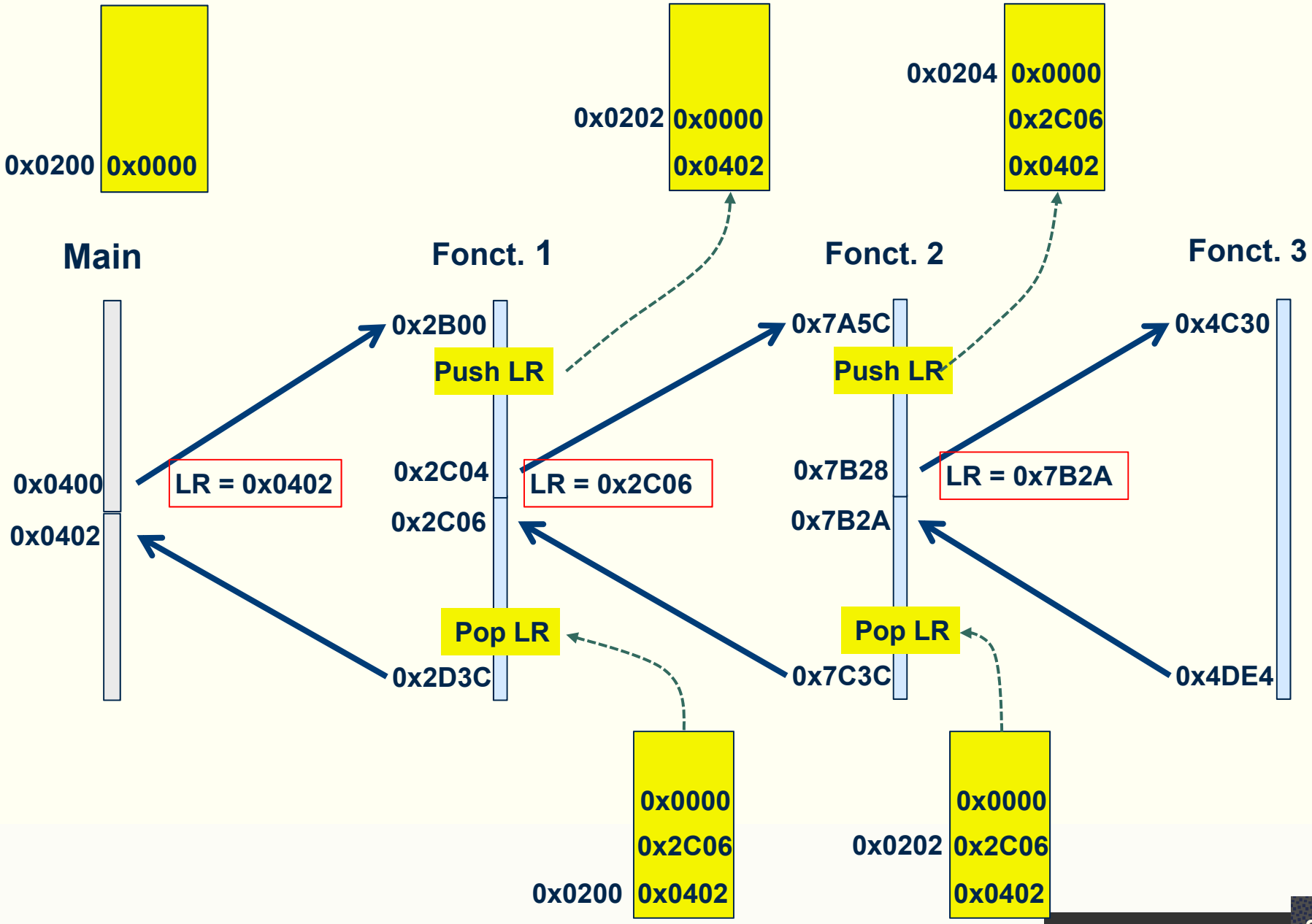
- Au démarrage, le pointeur de pile (SP) est initialisé avec l'adresse du bas de la pile (pile ascendante), ou l'adresse du haut de la pile (pile descendante)

- Push / Pop : écrire / lire une donnée dans la pile avec incrémentation ou décrémentation du SP

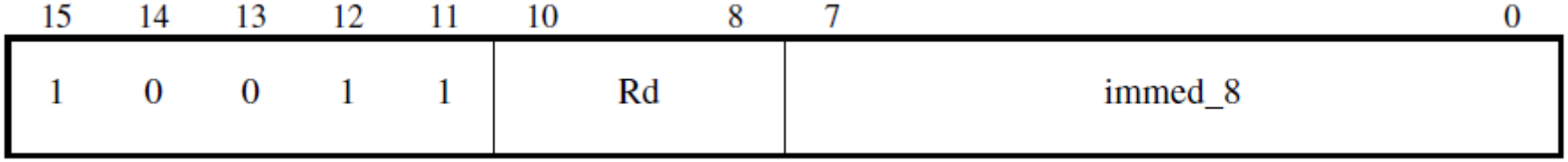
Utilisation de la pile

- Stockage des adresses de retour pour :
  - appels de fonctions
  - interruptions
- Sauvegarde de registres
  - changement de contexte (fonctions, interruptions)
- Stockage de variables locales
  - en C : variables déclarées dans le corps d'une fonction

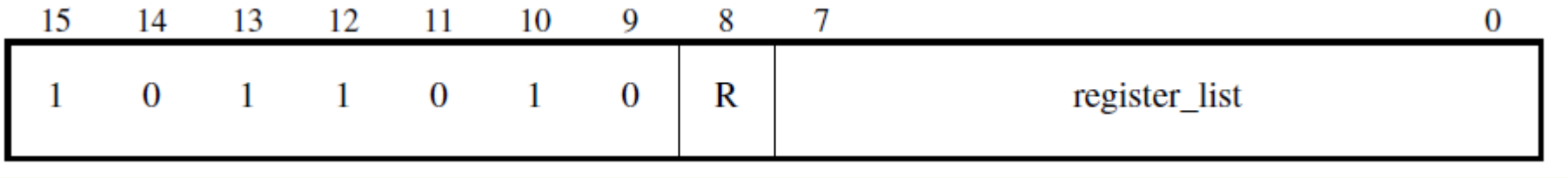
Pile: Exemple d'appels de fonction (instructions 16 bits)



Ecriture / lecture dans la pile:
Instructions ARM LDR, STR, PUSH, POP



- LDR <Rd>, [SP, #<immed_8> * 4]
- Rd ← M[SP+immed_8*4]



- PUSH <registers> écrit les valeurs de plusieurs registres dans la pile
- Liste de registres de R0 à R7 (un bit par registre) et R pour le Link Register