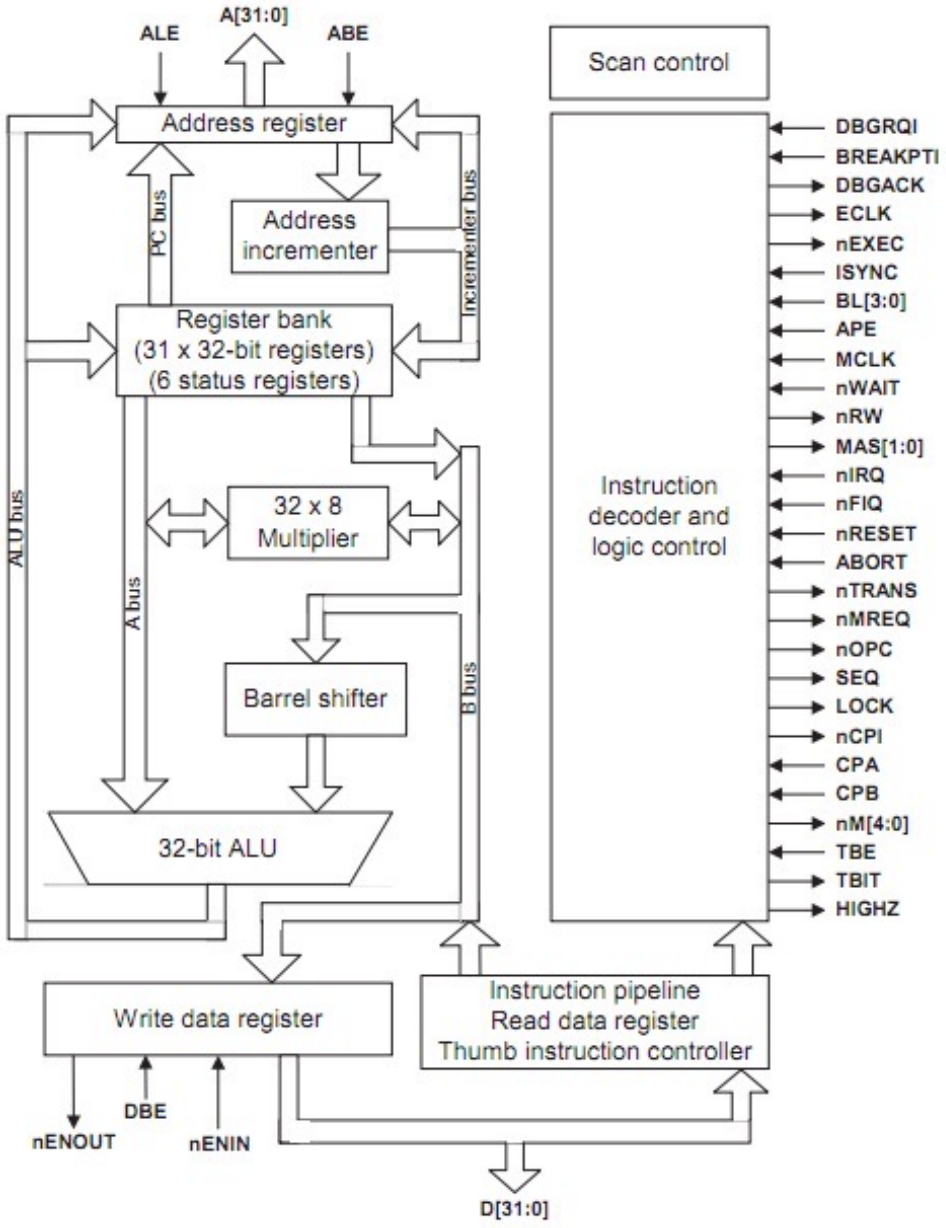
# Architecture ARM7TDMI

Yann DOUZE,

d’après les supports de cours de Sébastien Bilavarn

# Architecture ARM7 TDMI

- Processeur 32-bit
- UAL 32-bit
- File de registres
- Registre à décalage
- Multiplieur 32x8

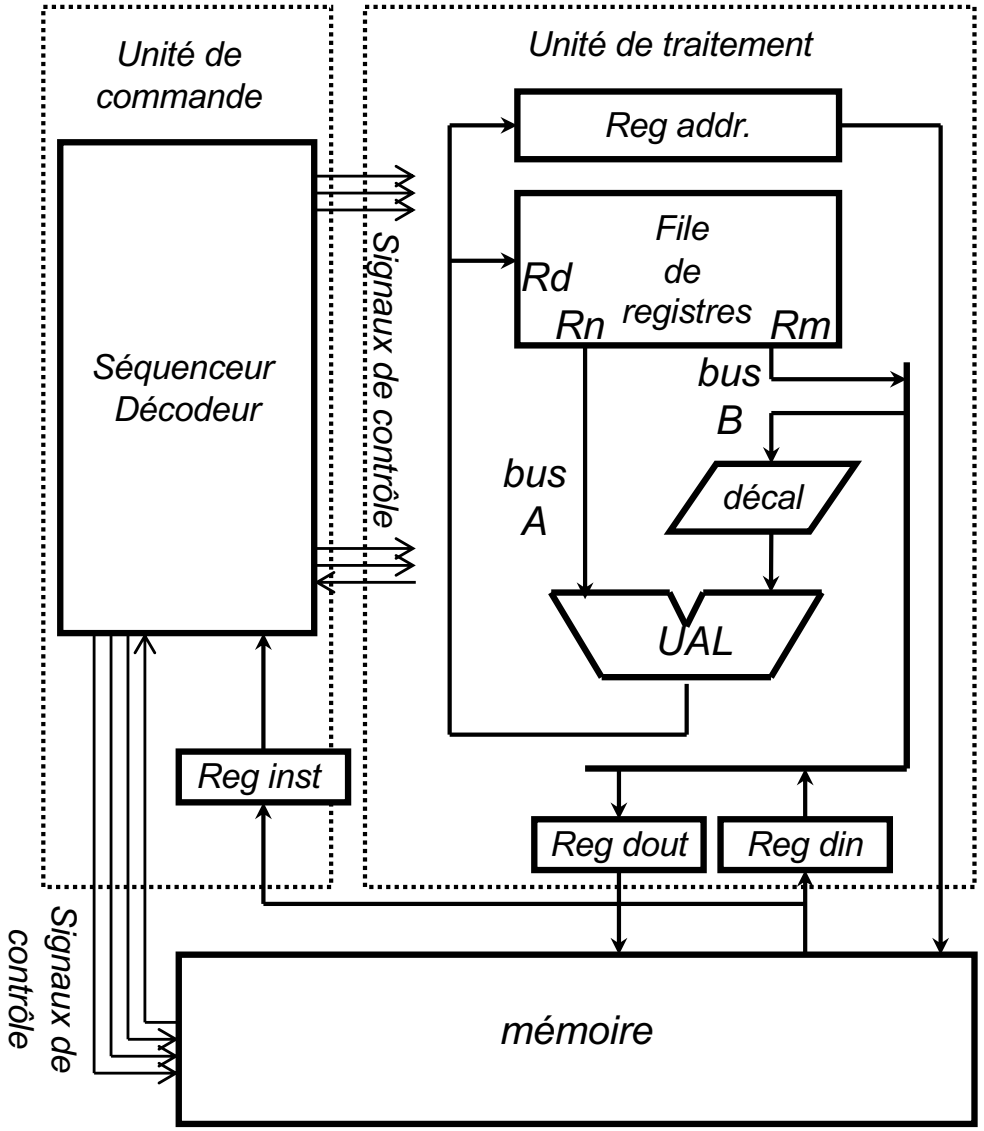


# Caractéristiques générales

- Architecture load-store
  - Les instructions ne traitent que des données en registre et placent les résultats en registre. Les seules opérations accédant à la mémoire sont celles qui copient une valeur mémoire vers un registre (load) et celles qui copient une valeur registre vers la mémoire (store).
- Format de codage fixe des instructions
  - Toutes les instructions sont codées sur 32 bits.
- Format 3 adresses des instructions de traitement
  - Deux registres opérandes et un registre résultat, qui peuvent être spécifiés indépendamment.
- Exécution conditionnelle
  - Chaque instruction peut s'exécuter conditionnellement
- Instructions spécifiques de transfert
  - Instructions performantes de transfert multiples mémoire – registre
- UAL + shift
  - Possibilité d'effectuer une opération Arithmétique ou Logique et un décalage en une instruction (1 cycle), la ou elles sont realisées par des instructions séparées sur la plupart des autres processeurs

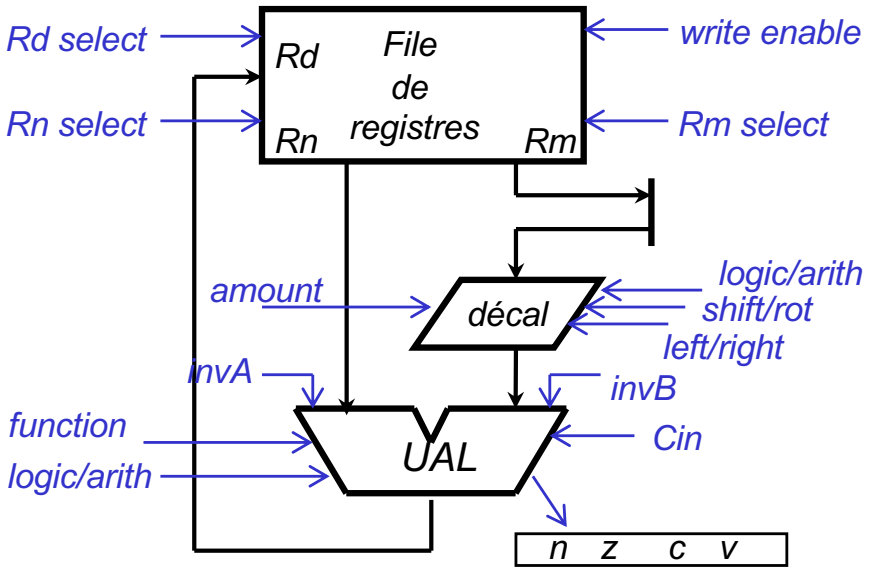
# Unité de traitement

- Unités fonctionnelles de l'UT:
  - File de registres
    - 16 registres pour permettre une manipulation souple des données et stockage des résultats de l'UAL
  - Unité Arithmétique et Logique
    - 2 opérandes : entrée A donnée provenant d'un registre, entrée B donnée reliée au décaleur
    - Résultat de l'UAL: renvoyé dans un registre
  - Registre à décalage
    - Opérations de décalage (1 décalage à gauche = x2, 1 décalage à droite = /2)
    - Opérations de rotation (décalage + ré-injection du bit perdu)
    - Associé à l'entrée B de l'UAL pour réaliser une instruction UAL + shift en 1 cycle

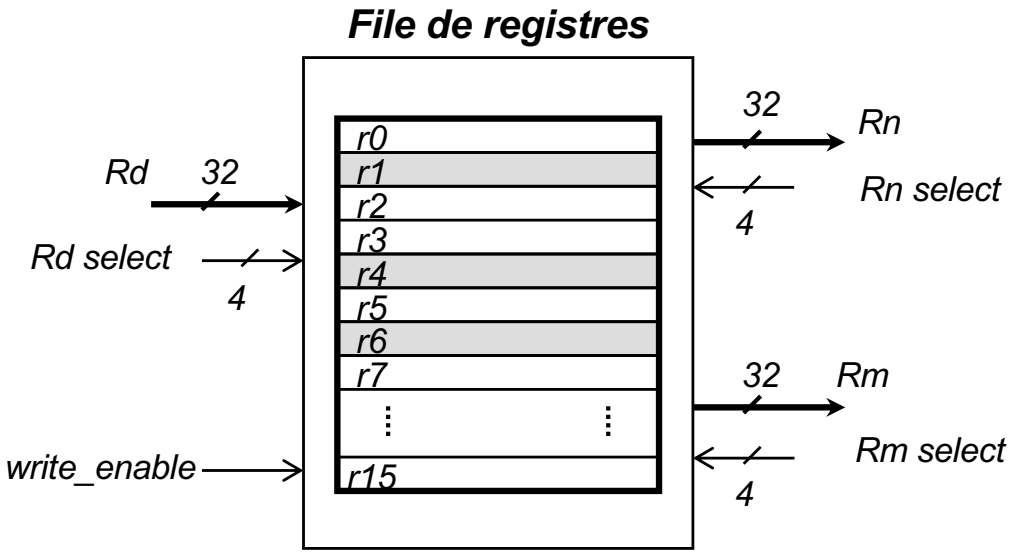


# Organisation de l'unité de traitement

- Signaux de commande de l'unité de traitement:
  - File de registres
    - Rd, Rn, Rm select : sélection du registre cible dans la file des 16 registres
    - write enable: activation de la file de registres
  - UAL
    - logic/arith+function: mode logique ou arthmétique, dans chaque mode choix de la fonction
    - InvA, invB: inversion des opérandes A et B (not)
    - Cin: injection du bit retenue C
    - Indicateurs: indiquent l'état de l'UAL après une opération (ex: retenue C)
  - Décaleur
    - shift/rot: opération de décalage ou rotation
    - logic/arith: opération logique ou arthmétique
    - Amount: nombre de bits de décalage/rotation



# File de registres




- 16 registres utilisateurs r0, …, r15
- Notation pour l'appel aux instructions:
  - INST Rd, Rn, Rm
    - Format 3 adresses (2 opérandes + résultat)
    - *Rn* registre opérande 1, relié au port A (32 bits)
    - *Rm* registre opérande 2, relié au port B (32 bits) par l'intermédiaire du décaleur → possibilité de rotation/décalage sur entrée B
    - *Rd* registre de destination (32 bits)
    - Select 4 bits pour le choix du registre parmi les 16 disponibles

# File de registres

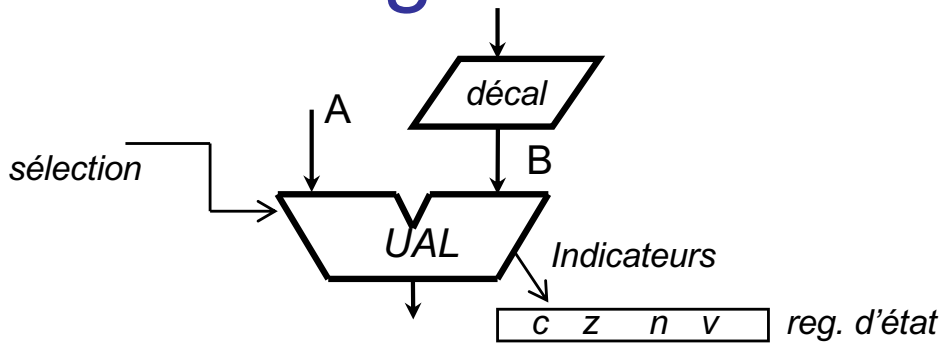
- Instructions de mouvements de données entre registres
  - MOV (Move), MVN (Move not)
    - MOV Rd, *#literal*
    - MOV Rd, Rn
    - MOV Rd, Rm, *shift*

    - *Mouvements de données entre registres, ou d'une constante vers registre, uniquement .*

    - *#literal*: valeur immédiate (constante)
    - *shift*: le deuxième opérande peut être sujet à un décalage

  - Exemples
    - `MOV r3, #2` `@ r3 ← 2`
    - `MOV r3, r4` `@ r3 ← r4`
    - `MOV r3, r4, LSL #2` `@ r3 ← r4<<2`

# UAL et registre à décalage




- Effectue des opérations arithmétiques et logiques
  - (ADD, SUB, AND, OR, …)
- Associée au registre à décalage en entrée B
- Le registre d'état (SR) fournit des indications sur les résultats d'opération:
  - C: Carry
    - bit indicateur de dépassement pour une opération arithmétique (ou de décalage)
  - Z: Zero
    - bit indicateur de résultat nul de l'UAL
  - N: Négatif
    - bit indicateur de résultat négatif de l'UAL
  - V: Débordement (oVerflow)
    - bit indicateur de dépassement de capacité du résultat de l'UAL (modification du bit de signe)

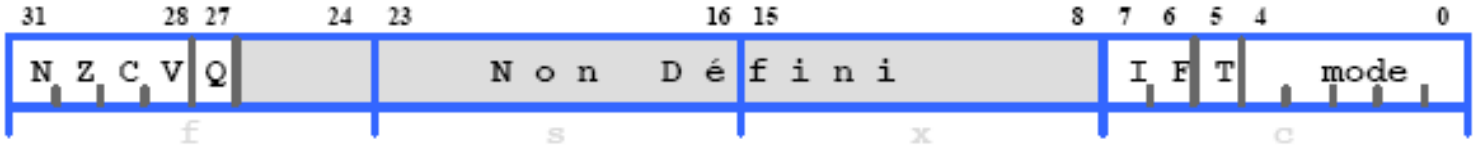
# UAL et registre à décalage

- Exemple sur 4 bits : 1 0 1 0 + 1 0 0 1 = (1) 0 0 1 1
  - Résultat de l'UAL: 0 0 1 1
  - C = 1
  - Z = 0, le résultat est différent de 0 0 0 0
  - N = 0, 0 0 1 1 est un nombre positif car le bit de signe (4ème bit) = 0
  - V = 1, car 1 0 1 0 et 1 0 0 1 sont des nombre négatifs et le résultat est positif

  - les bits C, V, N sont examinés ou ignorés en fonction de l'interprétation des nombres
    - Si les nombres sont en représentation non-signée, C est utile, V et N inutiles
    - Si les nombre sont en représentation signée, C est inutile, V et N sont utiles
    - Ces indicateurs sont positionnés par l'UAL, le programmeur les utilise ou non en fonction de ses besoins (par exemple pour effectuer une addition sur 8 bits à partir de l'addition 4 bits dans le cas de l'exemple)

# Le registre d'état

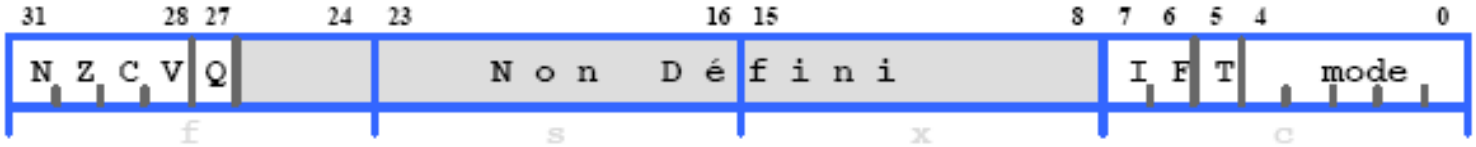
- CPSR: Current Program Status Register
- Contient les indicateurs Z (zero), N (negative), C (carry), V (overflow)
- Donne des informations sur le résultat d'une opération arithmétique ou d'une comparaison
- Permet à une instruction de s'exécuter ou non en fonction de ce résultat (exécution conditionnelle)
- Permet de conserver la retenue pour effectuer des opérations sur plus de 32-bit



# Le registre d'état

- Indicateurs conditionnels
  - Z résultat nul
  - N résultat négatif
  - C retenue
  - V Débordement
- Q débordement
  - Indique un type de débordement particulier (arithmétique saturée)
- Zone non définie

- Validation des interruptions
  - I=1 dévalide IRQ
  - F=1 dévalide FIQ
- Mode thumb
  - T=0 mode ARM (32 bits)
  - T=1 mode thumb (16 bits)
- Indicateur de mode
  - Indiquent le mode actif: système, IRQ, FIQ, utilisateur…



# UAL et registre à décalage

- Mnémonique pour décalage (*shift*)
  - LSL #*n* : Logical Shift Left (0 ≤ n ≤ 31)
  - LSR #*n* : Logical Shift Right (1 ≤ n ≤ 32)
  - ASR #*n* : Arithmetic Shift Right (1 ≤ n ≤ 32)
  - ROR #*n* : Rotate Right (1 ≤ n ≤ 31)
  - RRX: Rotate Right Extended (ROR étendu de 1 bit avec le bit de retenue C du registre d'état)

    - Décalage (>>, <<): déplacement du mot vers la droite ou vers la gauche. Les positions libérées sont remplies avec des zéros (logique) ou avec le bit de signe (arithmétique).
      - Ex1: 0111, LSR #1 → 0011, LSL #1 → 1110
      - Ex2: 1011, ASR #1 → 1101, ASL #1 → 0110 pas d'extension de signe possible, identique a LSL #1
    - Rotation: décalage circulaire, les bits perdus sont réinjectés
      - Ex: 0111, ROR #1 → 1011
      - Avec retenue: rotation effectuée sur un bit de plus (C), ex (0) 0111 RRX → (1) 0011

# UAL et registre à décalage

- Ex: l'instruction d'addition
  - ADD Rd, Rn, *#literal*
  - ADD Rd, Rn, Rm
  - ADD Rd, Rn, Rm, *shift*

- Exemples
  - `ADD r3, r2, #1` `@ r3 ← r2 + 1`
    - Opérande 2 (Rm) = constante
  - `ADD r3, r2, r5` `@ r3 ← r2 + r5`
  - `ADD r3, r2, r5, LSL #2` `@ r3 ← r2 + (r5<<2)`
    - opérande 2 (Rm) = opérande décalé.
    - Multiplie par 4 la valeur de r5, ajoute la valeur de r2 et stocke le résultat dans r3

# UAL et registre à décalage

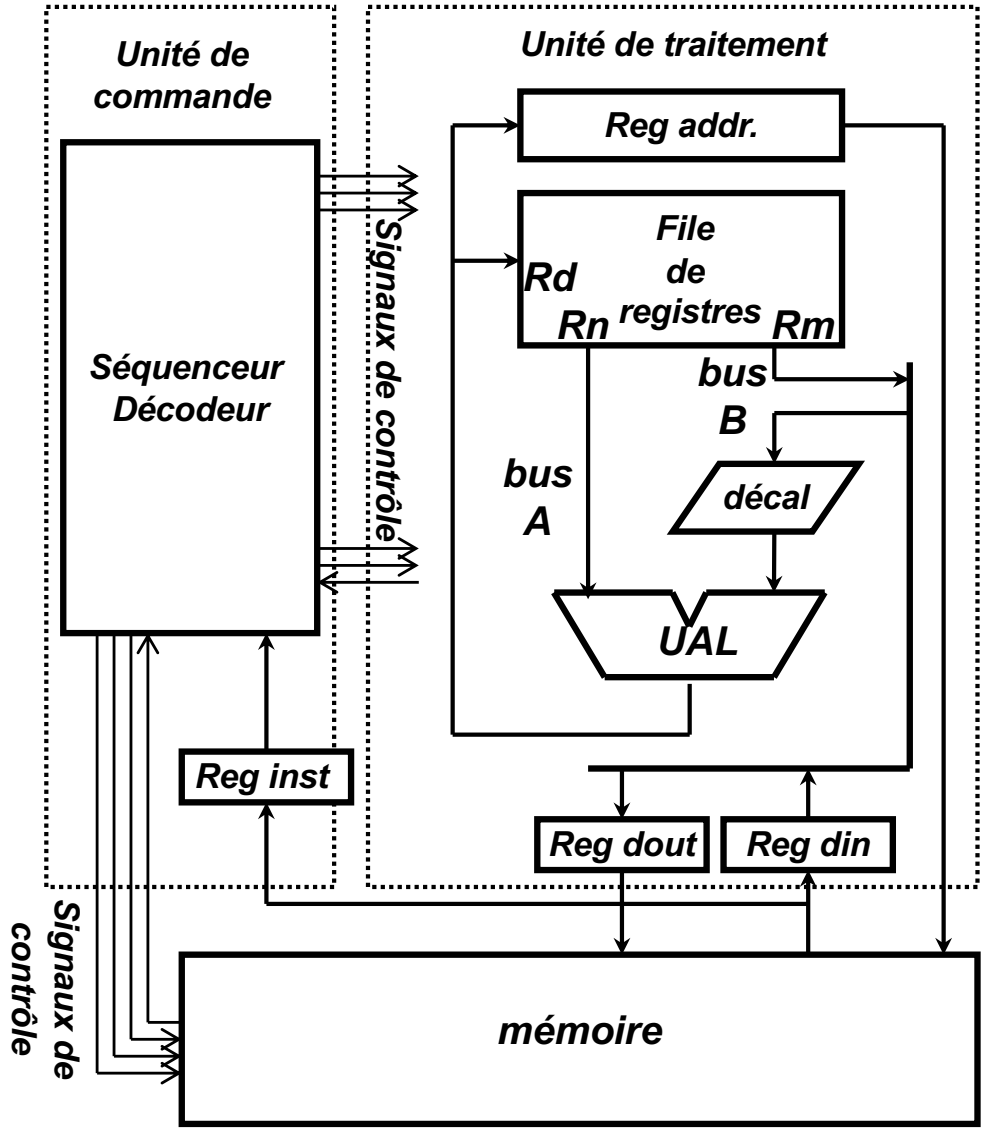
- Exercice

- Ecrire une séquence qui multiplie par 10 la valeur de r5 et stocke le résultat dans r6
  - 10xr5 = 8xr5 + 2xr5
  - 1) multiplier par 8 = 3 décalages à gauche
  - `MOV r1, r5, LSL #3` `@ r1 ← (r5<<3)`
  - 2) multiplier par 2 = 1 décalage à gauche
  - `MOV r2, r5, LSL #1` `@ r2 ← (r5<<1)`
  - 2) ajouter:
  - `ADD r6, r1, r2` `@ r6 ← r1 + r2`

  - Plus efficace:
  - `MOV r1, r5, LSL #3` `@ r1 ← (r5<<3)`
  - `ADD r6, r1, r5, LSL #1` `@ r6 ← r1 + (r5<<1)`

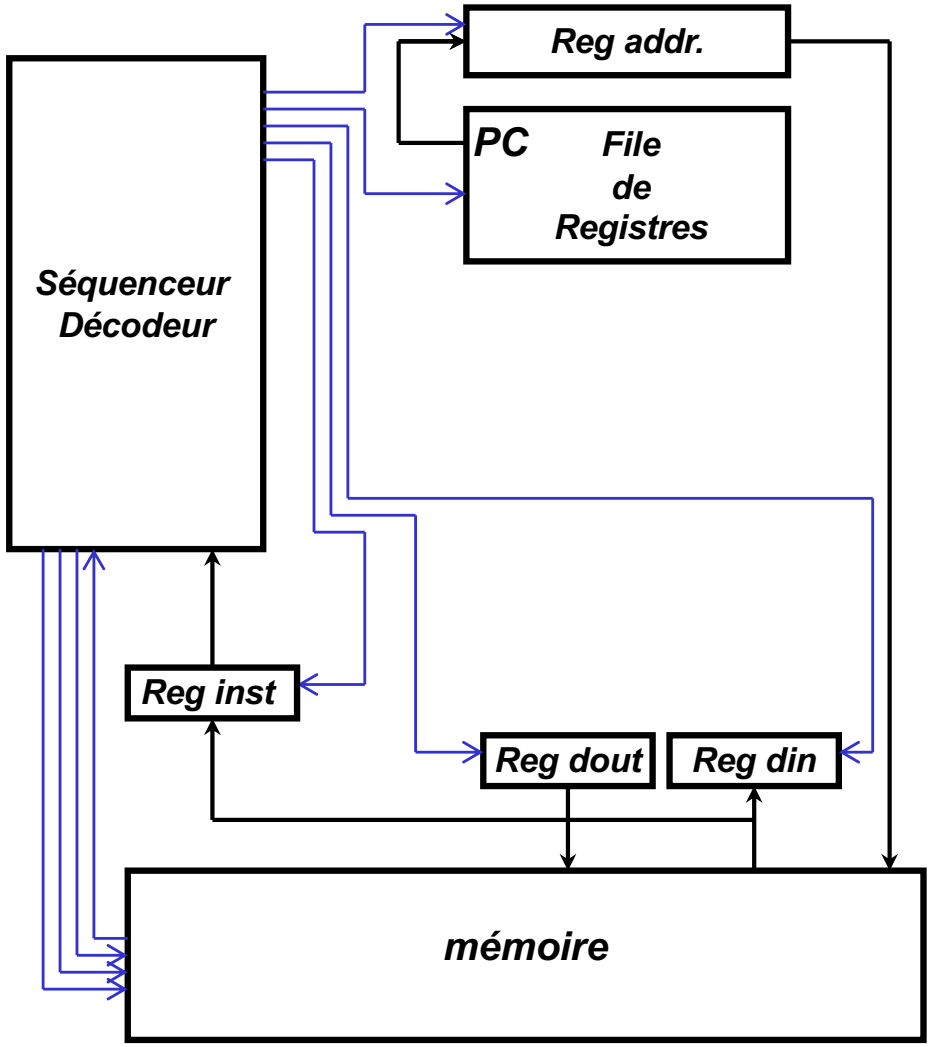
# Rôle de l'unité de commande

- Contrôle des accès mémoire
  - Pour l'accès aux instructions/données
    - Positionne les signaux nécessaires pour un accès aux instructions/données en mémoire
- Décodage des instructions
  - Interprétation des instructions
    - Processus de transformation d'une instruction en signaux de commande
- Contrôle de l'unité de traitement
  - Pour l'exécution d'une instruction
    - Positionne les signaux nécessaires pour l'exécution d'une instruction



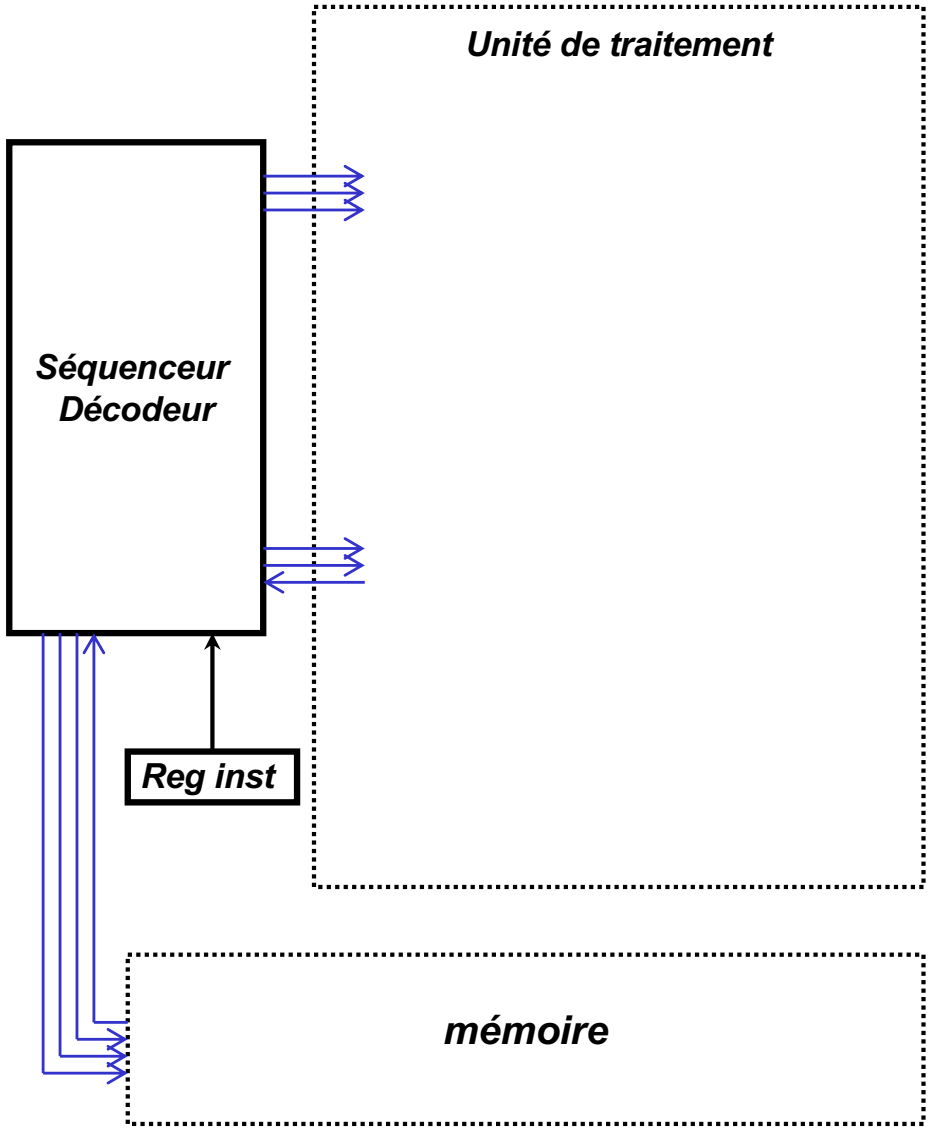
# Rôle de l'unité de commande

- **Contrôle des accès mémoire**
  - **Accès aux instructions**
    - **Pour l'exécution du programme, pendant la phase de recherche d'une instruction en mémoire**
    - **Registre d'adresses: contient l'adresse de l'instruction à exécuter**
    - **Registre d'instructions: l'instruction correspondante est renvoyée dans le registre d'instruction pour décodage**
  - **Accès aux données**
    - **Pendant l'exécution d'une instruction load/store.**
    - **Registres de données (Reg din, Reg dout)**
  - **Génération des signaux**
    - **positionne tous les signaux de contrôle mémoire nécessaire (nMREQ, nRW, MAS, registres)**



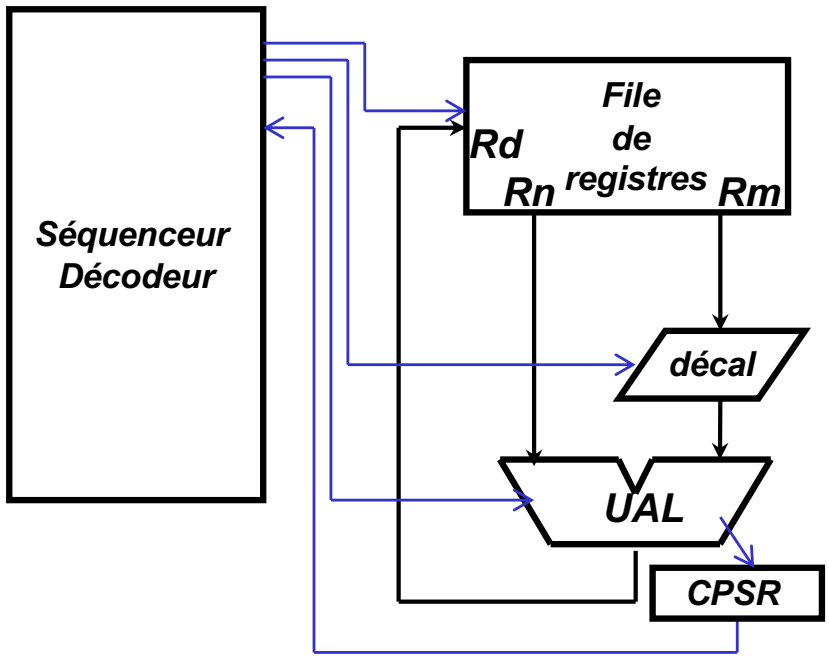
# Rôle de l'unité de commande

- **Décodage des instructions**
  - **Convertir une instruction en signaux de commande**
    - **Le décodage consiste à transformer la représentation binaire 32 bits d'une instruction en signaux synchronisés**
    - **Pour le séquencement de l'unité de traitement**
    - **Pour le séquencement de la mémoire**
  - **Processus complexe qui dépend de nombreux paramètres**
    - **Type de l'instruction (traitement, transfert, etc)**
    - **Opérandes (valeur immédiate, registre, registre avec décalage)**
    - **Bits du registre d'état pour évaluation d'une condition d'exécution (ex: branchement si egal)**
    - **Mode d'adressage**
    - **Type de données**



# Rôle de l'unité de commande

- **Contrôle de l'unité de traitement**
  - **Circulation des données dans l'unité de traitement**
    - **Sélection des registres dans la file de registres pour chaque opérande (Rn, Rm) et pour le résultat (Rd)**
  - **Sélection des opérations**
    - **Décalage (arithmétique, logique, rotation, nombre de décalages)**
    - **UAL (arithmétique, logique, fonction)**
  - **Exécution conditionnelle des instructions**
    - **Registre d'état (Current Program Status Register)**



# Cycle instruction du ARM7

- 1. Phase de recherche ("fetch")
  - Recherche de l'instruction en mémoire
- 2. Phase d'analyse ("decode")
  - Décodage de l'instruction: analyse de la représentation binaire 32 bit de l'instruction et de ses opérandes pour mise en œuvre des signaux par l'unité de commande
  - Pour les phases suivantes, 2 cas possibles selon qu'il s'agit d'une instruction de traitement (sollicite l'unité de traitement) ou d'une instruction d'accès mémoire (sollicite les unités de traitement et mémoire)

Pour une instruction de traitement de données:

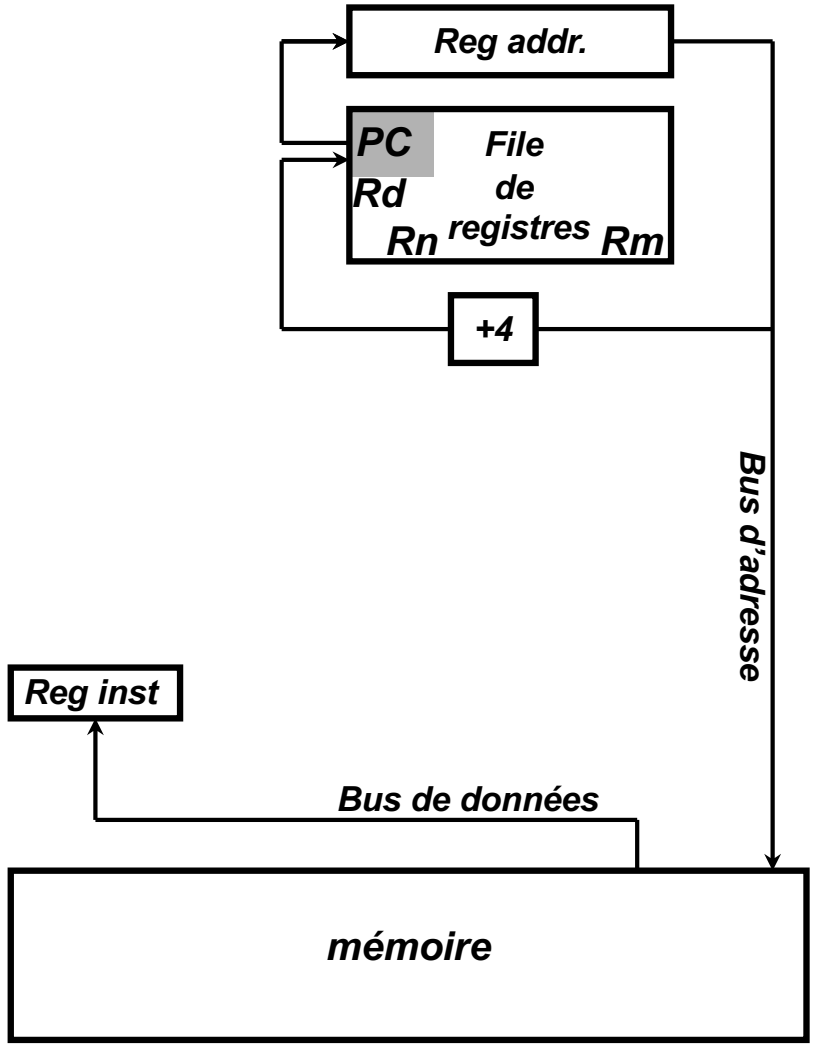
- 3. Phase d'exécution ("execute")
  - Circulation des données, sélection des opérations, exécution conditionnelle

Pour une instruction d'accès mémoire:

- 3. Phase de calcul d'adresse ("address calc")
  - L'adresse est calculée par l'UAL et stockée dans le registre d'adresse
- 4. Phase d'accès mémoire ("data transfer")
  - L'adresse est envoyée à la mémoire qui lit/écrit la donnée

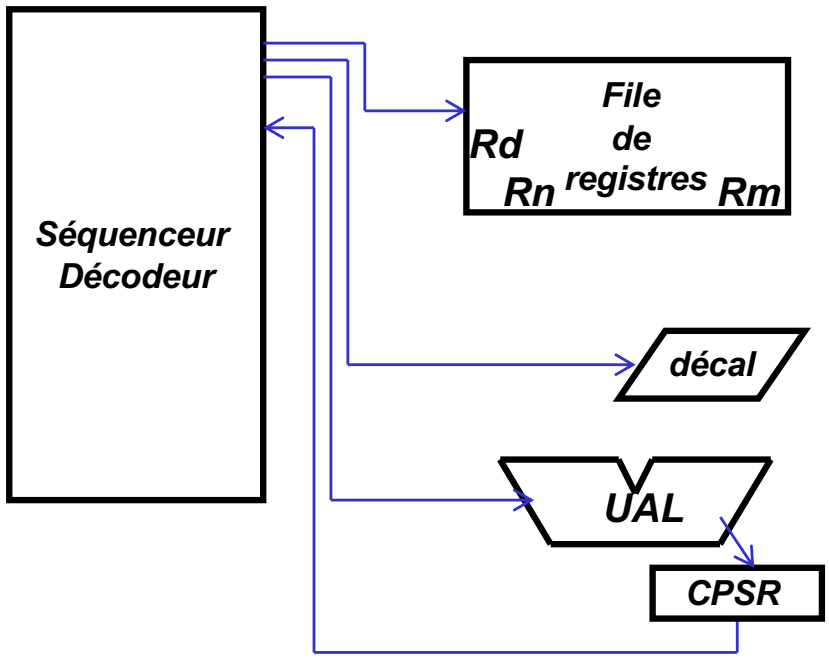
# Cycle d'instruction

- Phase de recherche ("fetch")
  - Chargement du registre d'adresse
    - L'adresse de l'instruction à exécuter est contenue dans le registre PC (Program Counter)
    - Cette adresse est copiée dans le registre d'adresse
  - Lecture mémoire
    - L'adresse est placée sur le bus d'adresse
    - On calcule l'adresse de l'instruction suivante (PC+4)
  - Chargement du registre d'instructions
    - L'instruction est disponible sur le bus de données et copiée dans le registre d'instruction
    - PC est mis à jour (adresse + 4)



# Cycle d'instruction

- Phase d'analyse ("decode")
  - Lecture du registre d'instruction
    - Le mot de 32 bits représentant l'instruction est extrait du registre d'instruction
    - L'instruction est décodée pour déterminer la séquence d'action à effectuer et avec quelles données.
  - Prise en compte du registre d'état
    - Instructions conditionnelles
  - Sélection opérandes source / destination
    - Sélection opérandes source (valeur immédiate ou sélection des registres Rn et Rm)
    - Résultat (sélection de Rd)
  - Sélection des opérations à effectuer
    - UAL (logic/arith, function, invA, invB, Cin)
    - Décaleur (logic/arith, shift/rot, left/right, amount)



# Cycle instruction du ARM7

- 1. Phase de recherche ("fetch")
- 2. Phase d'analyse ("decode")

Pour une instruction de traitement de données:

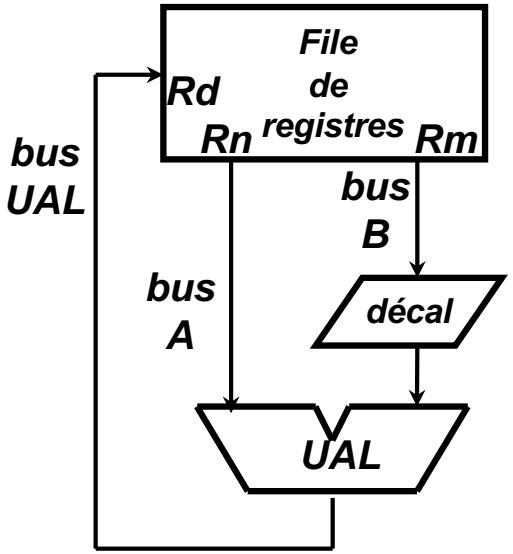
- 3. Phase d'exécution ("execute")

Pour une instruction d'accès mémoire:

- 3. Phase de calcul d'adresse ("address calc")
- 4. Phase d'accès mémoire ("data transfer")

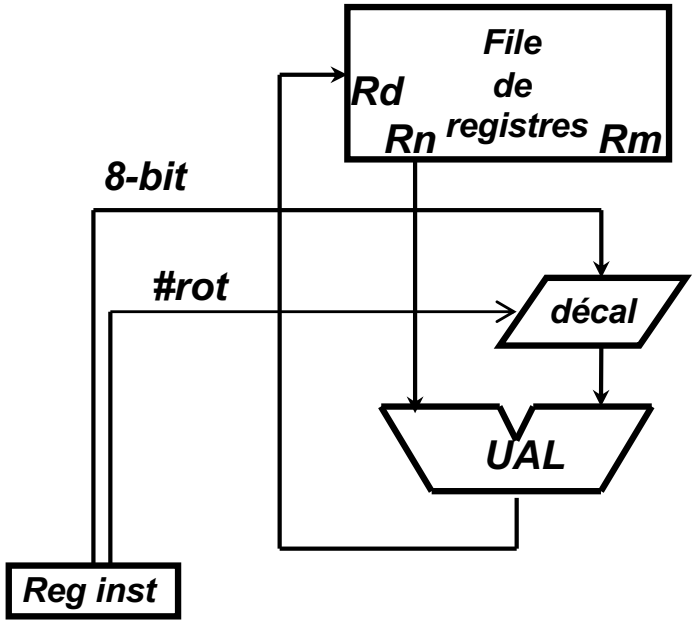
# Cycle d'instruction

- Instruction de traitement: phase d'exécution
  - Cas d'une instruction de type ADD Rd, Rn, Rm, *shift*
    - Les opérandes sont lus dans la file de registres. Rn est présenté sur la première entrée de l'UAL (bus A). Rm est présenté sur l'entrée du décaleur (bus B).
    - Une opération de décalage est réalisée ainsi qu'une opération arithmétique ou logique.
    - Le résultat est renvoyé dans le registre de destination (Rd)



# Cycle d'instruction

- Instruction de traitement: phase d'exécution
  - Cas d'une instruction de type ADD Rd, Rn, *#litteral*
    - Rn est lu dans la file de registres
    - *#litteral* est obtenu à partir de l''instruction (il est codé dans l'appel à l'instruction sur 8 bits+rotation)
    - Calcul de l'opération arithmétique ou logique.
    - Le résultat est renvoyé dans le registre de destination (Rd)



# Cycle instruction du ARM7

- 1. Phase de recherche ("fetch")
- 2. Phase d'analyse ("decode")

Pour une instruction de traitement de données:

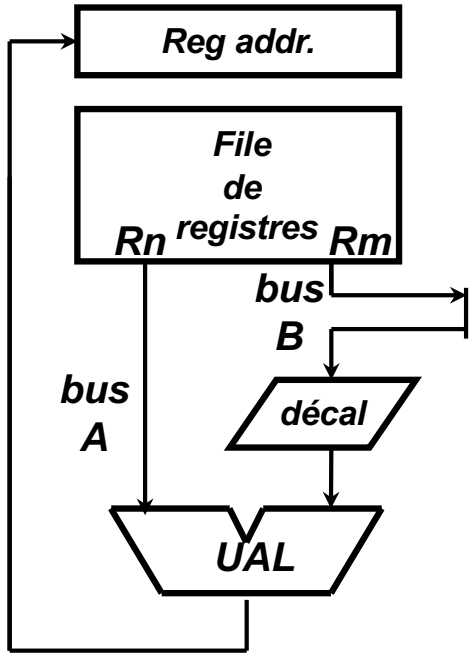
- 3. Phase d'exécution ("execute")

Pour une instruction d'accès mémoire:

- 3. Phase de calcul d'adresse ("address calc")
- 4. Phase d'accès mémoire ("data transfer")

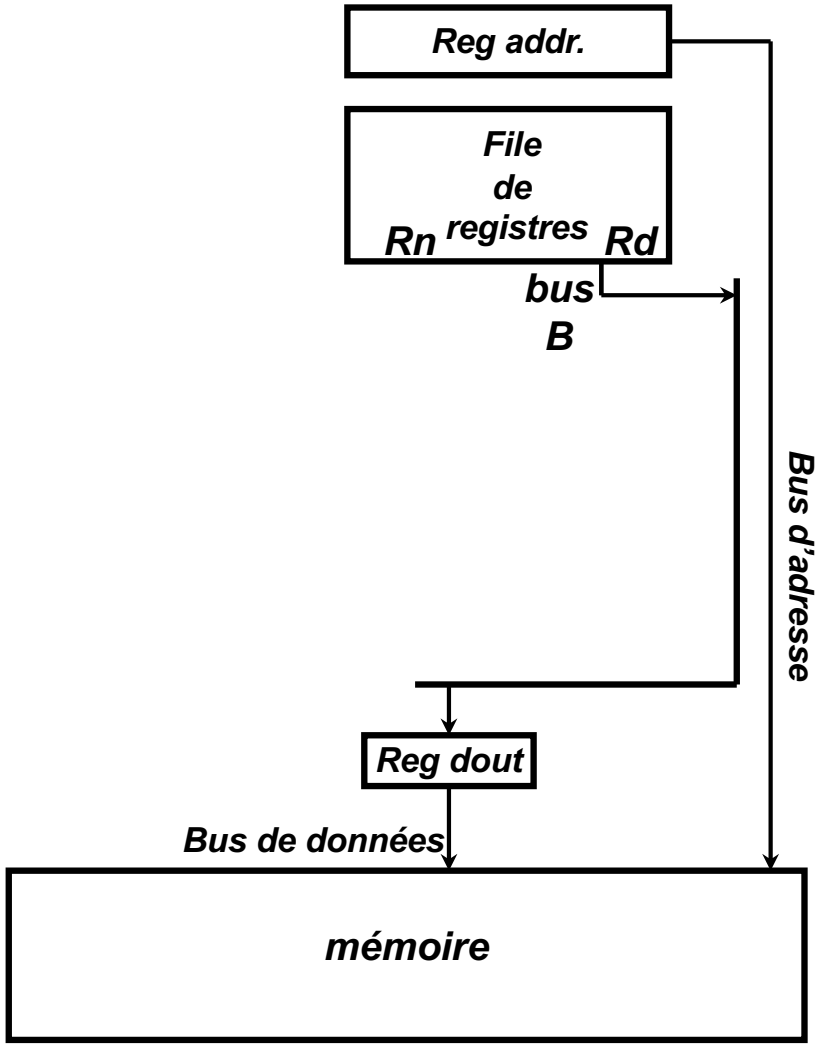
# Cycle d'instruction

- Cas d'une instruction de type STR Rd, [Rn, +/-Rm, *shift*]
  - Phase de calcul d'adresse
    - Le calcul d'adresse est effectué par l'UAL. Le résultat du calcul est renvoyé dans le registre d'adresse.



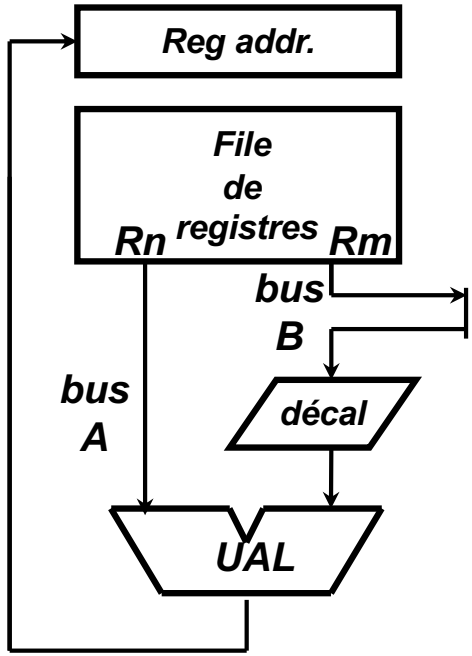

# Cycle d'instruction

- Cas d'une instruction de type STR Rd, [Rn, +/-Rm, *shift*]
  - Phase d'accès
    - L'adresse d'écriture contenue dans le registre d'adresse est envoyée sur le bus d'adresse
    - La valeur du registre à copier (Rd) en mémoire est copiée dans le registre Reg dout
    - La donnée est envoyée sur le bus de données et écrite en mémoire



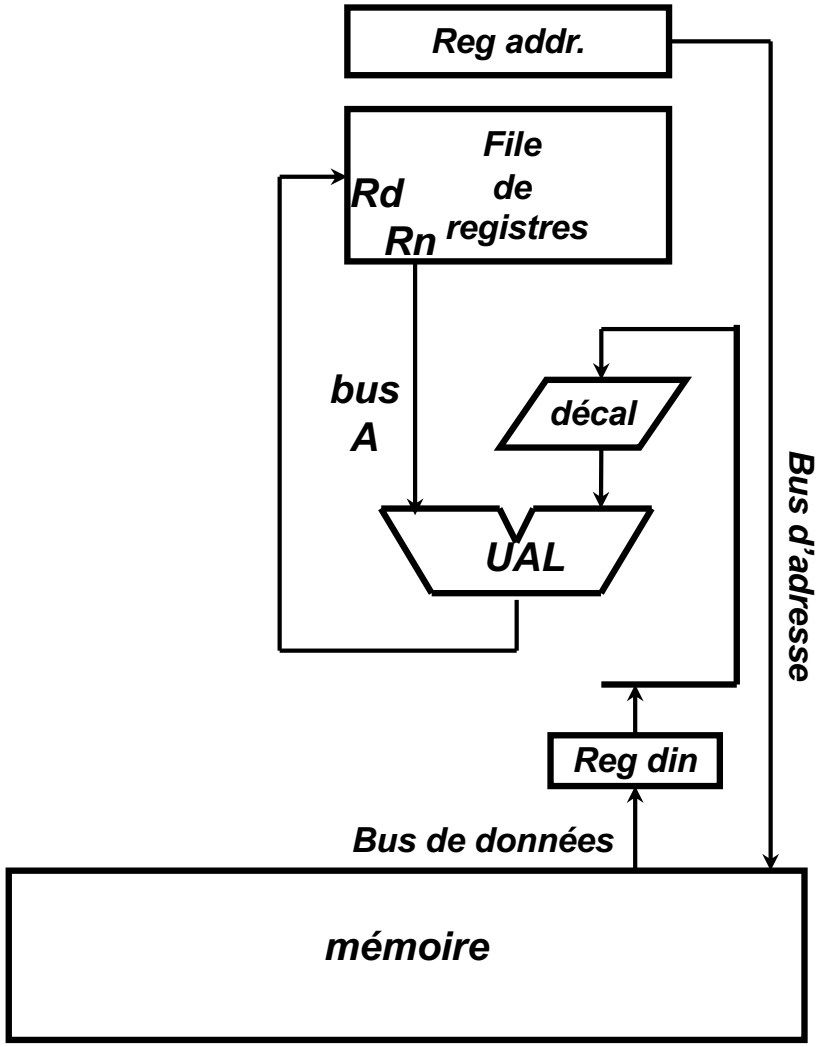
# Cycle d'instruction

- Cas d'une instruction de type LDR Rd, [Rn, +/-Rm, *shift*]
  - Phase de calcul d'adresse
    - Le calcul d'adresse est effectué par l'UAL. Le résultat du calcul est renvoyé dans le registre d'adresse.



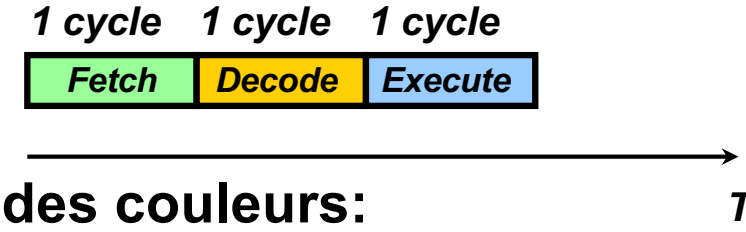

# Cycle d'instruction

- Cas d'une instruction de type LDR Rd, [Rn, +/-Rm, *shift*]
  - Phase d'accès
    - L'adresse de lecture est envoyée sur le bus d'adresse
    - La donnée est lue en mémoire
    - La donnée est envoyée sur le bus de données
    - La donnée est acheminée vers le registre de destination (Rd)



# Exécution pipeline

- Sur l'ARM7, les phases "fetch", "decode" et "execute" prennent chacune un cycle d'horloge
- Chaque instruction de traitement s'exécute donc en trois cycles d'horloge




**Convention des couleurs:**

- **Vert: utilisation de la mémoire (fetch lit les instructions en mémoire)**
- **Orange: utilisation du décodeur/séquenceur (analyse des instructions)**
- **Bleu: utilisation de l'unité de traitement (exécution d'une instruction de traitement)**

# Exécution pipeline

- Combien de temps pour exécuter la séquence d'instructions suivante ?

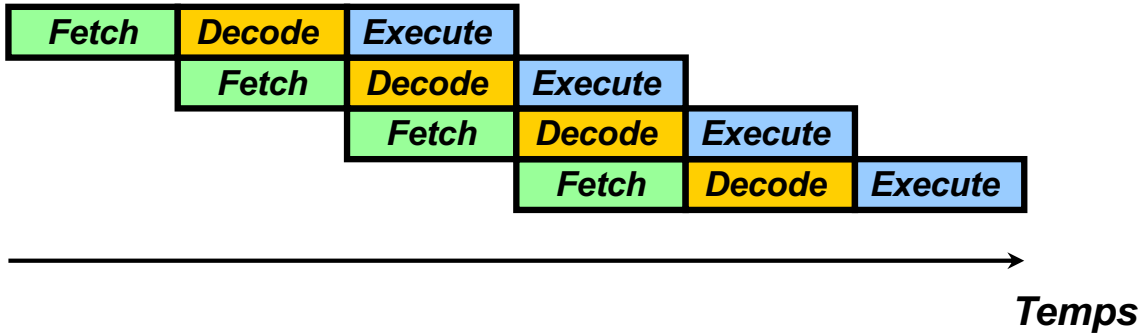
```
MOV r5, r3, LSL #1
ADD r5, r5, r3, LSL #3
MOV r7, r5, LSL #1
ADD r5, r5, r3, LSL #3
```

- 4 instructions x 3 = 12 cycles d'horloge ?

# Exécution pipeline

- Réponse: 6 cycles d'horloge !!

```
MOV r5, r3, LSL #1
ADD r5, r5, r3, LSL #3
MOV r7, r5, LSL #1
ADD r5, r5, r3, LSL #3
```

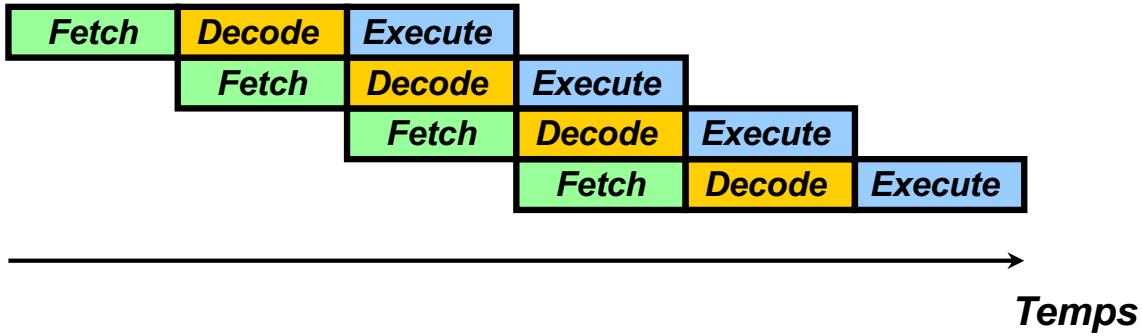



- **Le pipeline permet de recouvrir les phases d'exécution**
  - **La division en phases indépendantes (fetch, decode, execute) permet de démarrer l'exécution d'une instruction avant que la précédente ne soit terminée**
  - **On peut démarrer la phase fetch dès la phase decode de l'instr précédente**
  - **On peut démarrer la phase decode dès la phase execute de l'instr précédente**

# Exécution pipeline

- Réponse: 6 cycles d'horloge !!

```
MOV r5, r3, LSL #1
ADD r5, r5, r3, LSL #3
MOV r7, r5, LSL #1
ADD r5, r5, r3, LSL #3
```

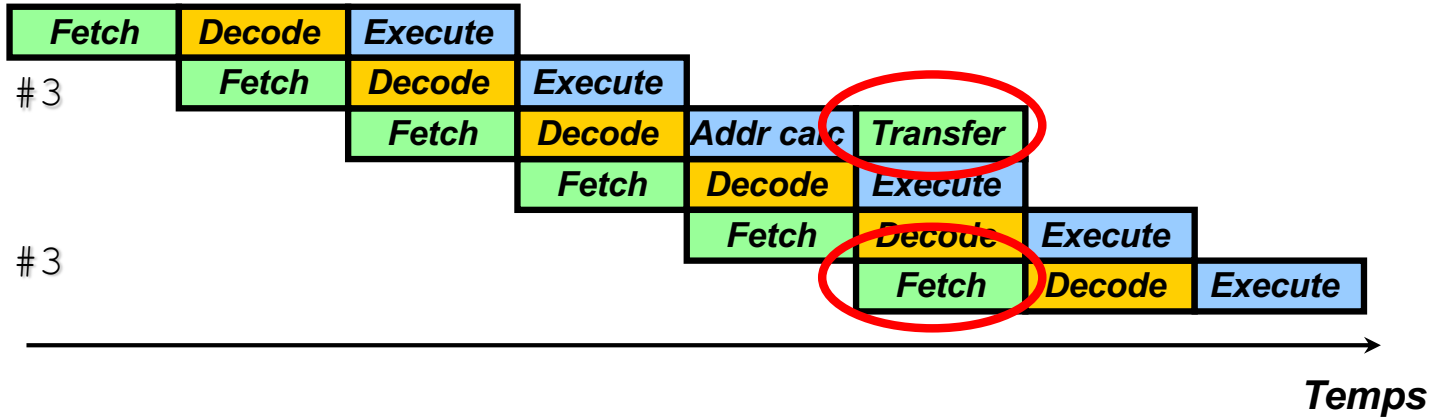



- **On peut exécuter une instruction par cycle d'horloge**
  - **A partir du cycle 3, une phase d'exécution se termine à chaque cycle d'horloge**
  - **On utilise au mieux les ressources: 3 phases s'exécutent parfois simultanément**
  - **Condition: ne pas avoir 2 fois la même couleur dans le même cycle!!**

# Exécution pipeline

- **Ralentissement du pipeline: cas des instructions d'accès mémoire (1)**

```
MOV r5, r3, LSL #1
ADD r5, r5, r3, LSL #3
LDR r8, [r2, #12]
MOV r7, r8, LSL #1
ADD r7, r7, r8, LSL #3
SUB r0, r7, r5
```



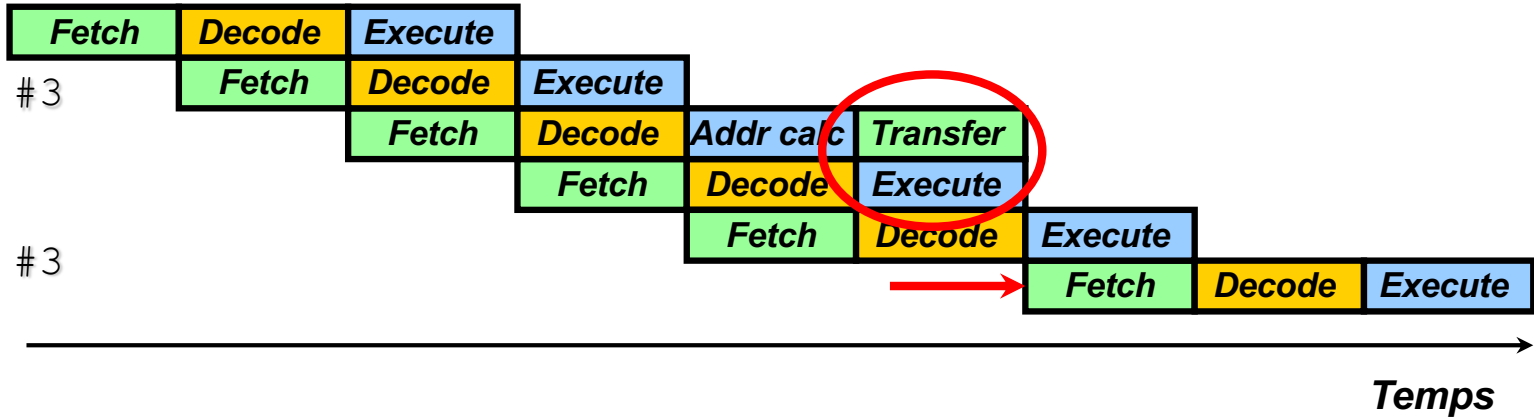

**On insère une instruction d'accès mémoire (LDR r8, [r2, #12])**

- **La phase addr calc utilise l'unité de traitement (bleu)**
- **La phase de transfert utilise la mémoire (vert)**
- **Les deux étapes transfert (LDR) et fetch (SUB) ne peuvent pas utiliser la mémoire simultanément**

# Exécution pipeline

- **Ralentissement du pipeline: cas des instructions d'accès mémoire (2)**

```
MOV r5, r3, LSL #1
ADD r5, r5, r3, LSL #3
LDR r8, [r2, #12]
MOV r7, r8, LSL #1
ADD r7, r7, r8, LSL #3
SUB r0, r7, r5
```



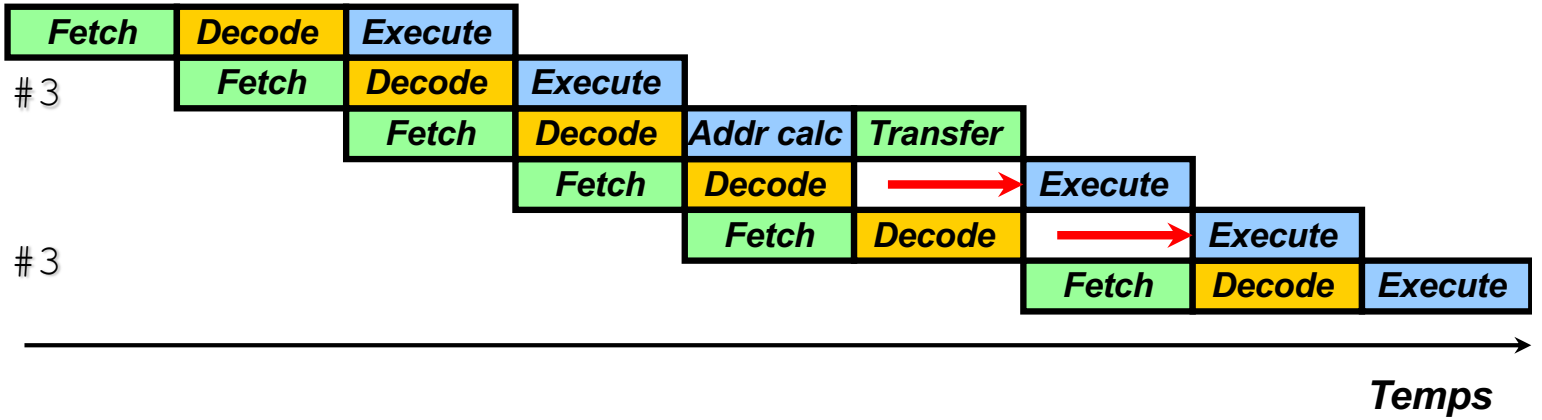

**Solution: décaler la dernière instruction de 1 cycle**

- **Décaler les phases de l'instruction SUB de 1 cycle**
- **MAIS: l'instruction suivant LDR (MOV) a besoin de la valeur du registre r8 qui n'est pas disponible (transfert non terminé), elle ne peut donc pas s'exécuter.**
- **Il faut à nouveau décaler les phases d'exécution de l'instruction MOV, et celle de l'instruction ADD**

# Exécution pipeline

- **Ralentissement du pipeline: cas des instructions d'accès mémoire (3)**

```
MOV r5, r3, LSL #1
ADD r5, r5, r3, LSL #3
LDR r8, [r2, #12]
MOV r7, r8, LSL #1
ADD r7, r7, r8, LSL #3
SUB r0, r7, r5
```

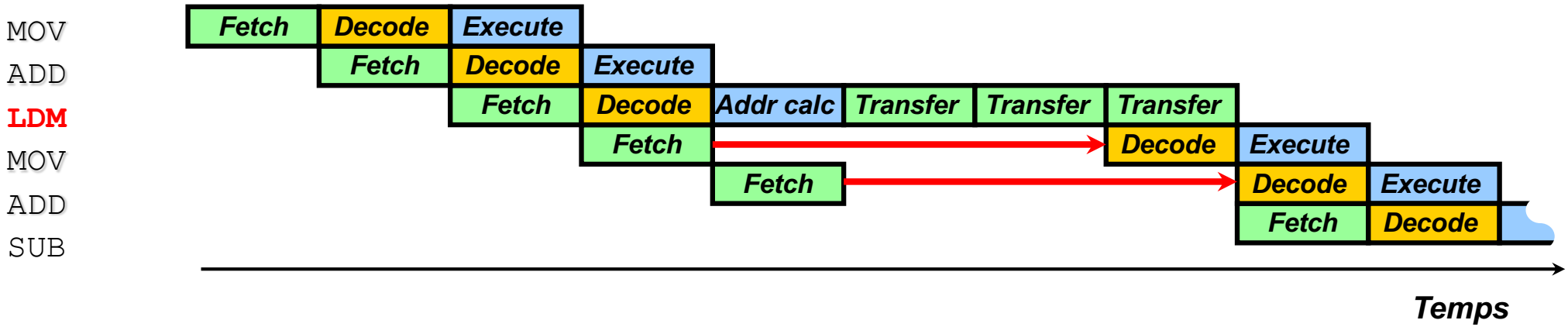



**Solution: décaler les phases d'exécution des 2 avant dernières instructions**

- **Ralentissement du pipeline: le cycle de l'instruction en cours est arrêté jusqu'à ce que la donnée soit disponible (géré par le processeur)**
- **Apparition d'interruptions ("bulles") dans le pipeline**
- **Le pipeline implique un surcoût en complexité pour sa mise en œuvre**

# Exécution pipeline

- **Ralentissement du pipeline: cas des instructions d'accès mémoire multiples**



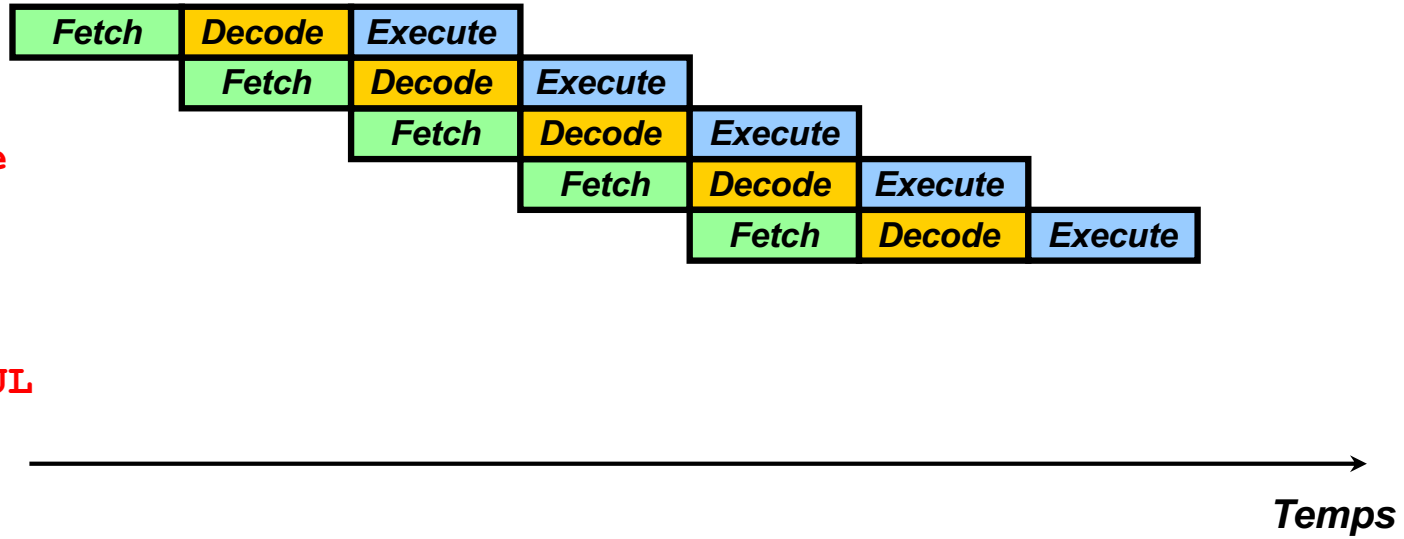

**Autres cas de ralentissement du pipeline**

- **L'exécution est ralentie par la durée de la phase de transfert**

# Exécution pipeline

- **Ralentissement du pipeline: cas des instructions de branchement (1)**

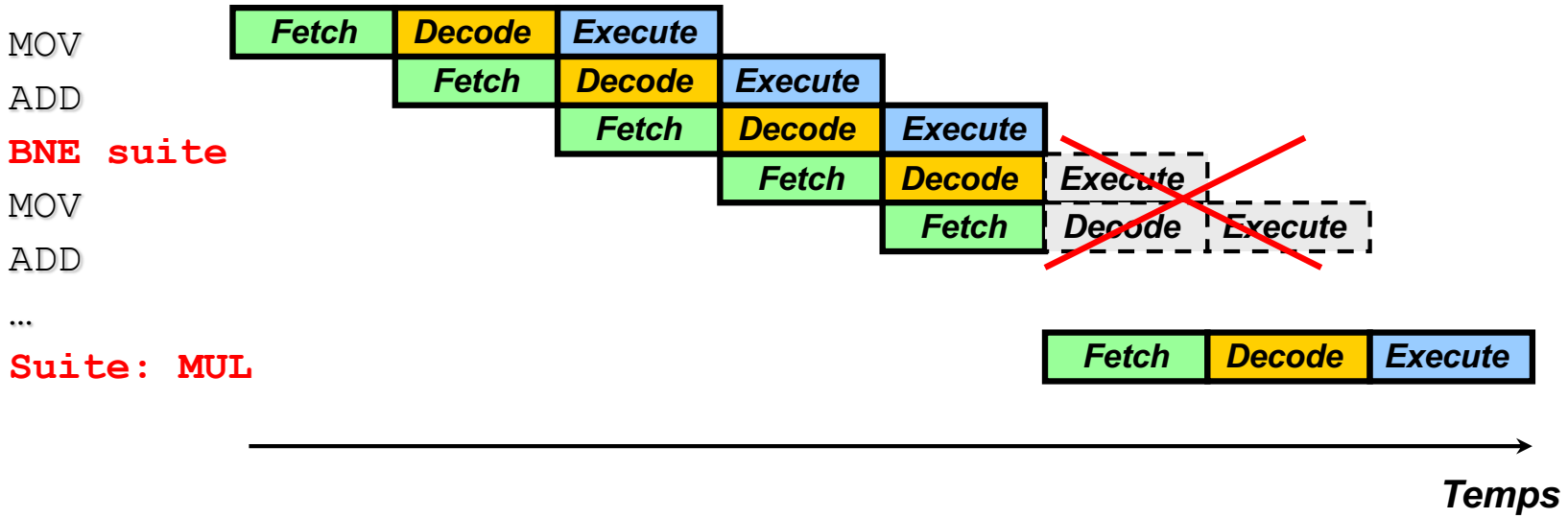
```
MOV
ADD
BNE suite
MOV
ADD
…
Suite: MUL
```




  - **Si la condition de branchement (BNE) est fausse, on poursuit la séquence d'instruction (MOV, ADD). Il n'y a pas d'interruption du pipeline.**

# Exécution pipeline

- **Ralentissement du pipeline: cas des instructions de branchement (2)**




  - **Si la condition de branchement (BNE) est vraie, on annule l'exécution des deux instructions suivantes, on poursuit à l'étiquette `Suite` (réinitialisation du pipeline).**
  - **C'est la tâche du programmeur d'écrire du code assembleur qui utilise efficacement le pipeline pour optimiser l'exécution d'un programme**
  - **Par exemple en utilisant l'exécution conditionnelle au lieu d'instructions de branchement (quand c'est possible), ou en ré-agençant les instructions pour éviter les ralentissement dus aux données non disponibles.**

# Faille Meltdown et Spectre

- https://www.macg.co/materiel/2018/01/meltdown-et-spectre-tout-savoir-sur-les-failles-historiques-des-processeurs-100954
- https://www.ovh.com/fr/blog/failles-de-securite-spectre-meltdown-explication-3-failles-mesures-correctives-public-averti/
- https://meltdownattack.com/meltdown.pdf

# Codage binaire d'une instruction

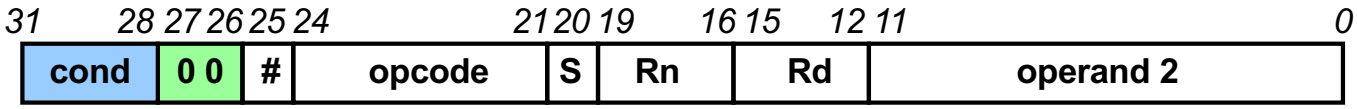
- Une instruction représente une opération élémentaire du processeur
  - L'ensemble des instructions disponibles s'appelle le jeu d'instruction
  - Chaque instruction définit un traitement précis opéré par la machine
- En assembleur, l'instruction s'écrit sous la forme d'un mot-clé suivi de ses opérandes.
  - On utilise des mnémoniques pour des raisons de commodité (faciliter l'écriture de programmes assembleur)
  - Le processus d'assemblage transforme ensuite la séquence de mnémoniques instructions en séquence de codes binaires interprétable par la machine.
- Pour le processeur, une instruction est codée par un mot de 32 bits (codage binaire).
  - Chaque du bit de l'instruction 32 bit a un rôle précis

# Codage binaire d'une instruction

- Une instruction de traitement est composée des champs
  - Registre résultat
    - Rd: R0…R15
  - Premier opérande (toujours un registre)
    - Rn: R0…R15
  - Deuxième opérande
    - Rm: R0…R15
    - Rm: R0…r15 avec décalage (log/arith/rot, quantité)
    - Une valeur immédiate (constante)
  - Opération à réaliser par l'UAL
    - Opcode
  - Affecte le registre d'état CPSR
    - Suffixe S
  - Condition
    - Exécution conditionnelle (EQ, LE, etc.)

# Codage binaire

- Instructions de traitement (1)




- *Cond*: l'instruction est exécutée si le registre d'état CPSR vérifie la condition spécifiée

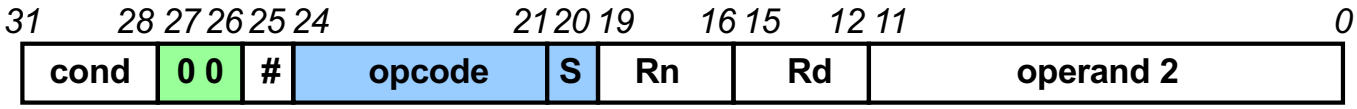
| Asm | Cond |
|---|---|
| EQ | 0000 |
| NE | 0001 |
| CS/HS | 0010 |
| CC/LO | 0011 |
| MI | 0100 |
| PL | 0101 |

| Asm | Cond |
|---|---|
| VS | 0110 |
| VC | 0111 |
| HI | 1000 |
| LS | 1001 |
| GE | 1010 |
| LT | 1011 |

| Asm | Cond |
|---|---|
| GT | 1100 |
| LE | 1101 |
| AL | 1110 |
| NV | 1111 |

# Codage binaire

- Instructions de traitement (2)




- *S = 1* : affecte CPSR
- *Opcode* : code de l'opération à effectuer

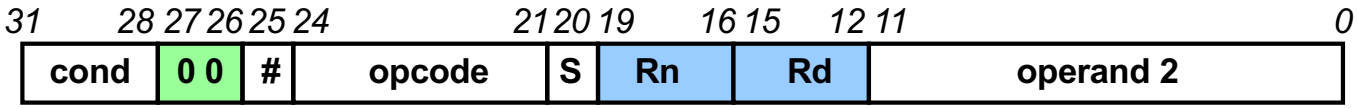
| Asm | Opcode |
|---|---|
| AND | 0000 |
| EOR | 0001 |
| SUB | 0010 |
| RSB | 0011 |
| ADD | 0100 |
| ADC | 0101 |

| Asm | Opcode |
|---|---|
| SBC | 0110 |
| RSC | 0111 |
| TST | 1000 |
| TEQ | 1001 |
| CMP | 1010 |
| CMN | 1011 |

| Asm | Opcode |
|---|---|
| ORR | 1100 |
| MOV | 1101 |
| BIC | 1110 |
| MVN | 1111 |

# Codage binaire

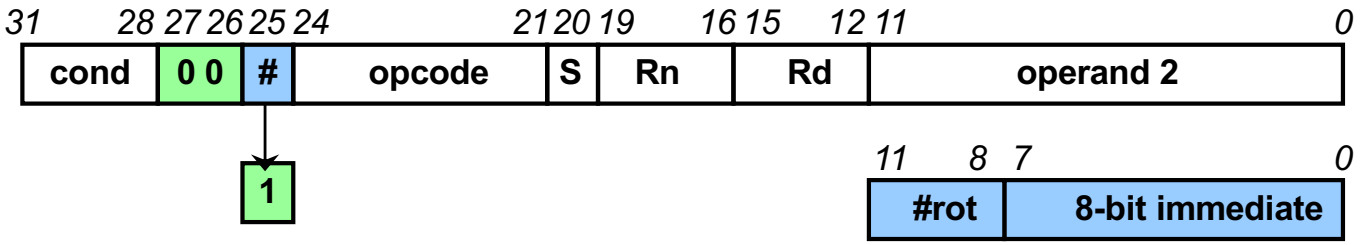
- Instructions de traitement (3)




- *Rd* → numéro du registre de destinations
  - 4 bits pour sélection parmi les 16 registres possibles
- *Rn* → numéro du registre qui sert de premier opérande
  - 4 bits pour sélection parmi les 16 registres possibles
- *operand2* → relié à l'entrée B
  - possibilité de rotation/décalage

# Codage binaire

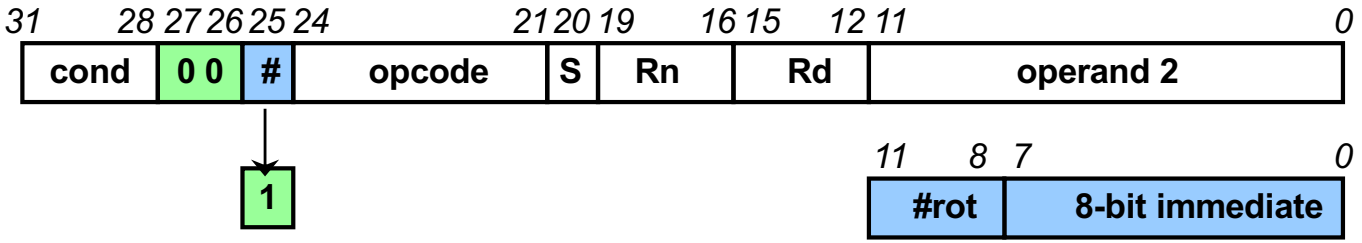
- Instructions de traitement (4)




- Si le bit # est à 1, *operand2* est une valeur littérale sur 32 bits.
  - Une instruction est codée sur 32 bits, il n'est donc pas possible de coder n'importe quelle valeur littérale 32 bits.
  - Cette valeur est construite par rotation vers la droite d'une valeur sur 8 bits.

# Codage binaire

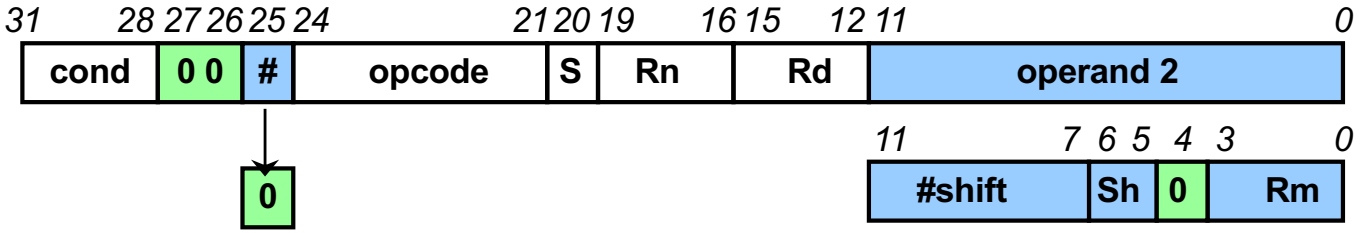
- Instructions de traitement (5)




- Cette valeur est construite par rotation vers la droite d'une valeur sur 8 bits.

# Codage binaire

- Instructions de traitement (6)




| Asm | n | Sh | #shift |
|---|---|---|---|
| *rien* | - | 00 | 0 |
| LSL #n | 0 à 31 | 00 | n |
| LSR #n | 1 à 32 | 01 | n si n<32, 0 si n=32 |
| ASR #n | 1 à 32 | 10 | n si n<31, 0 si n=32 |
| ROR #n | 1 à 31 | 11 | n |
| RRX | - | 11 | 0 |

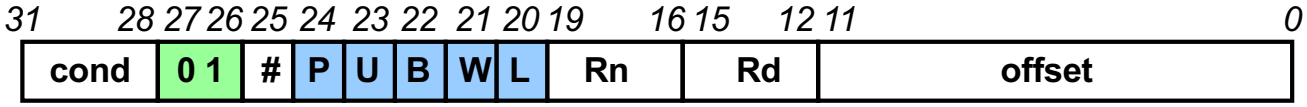
- Si le bit # est à 0, *operand2* est un registre (*Rm*) sur lequel on applique (ou non) une rotation/décalage
  - Le nombre de bits de décalage est soit
    - Un litéral #shift (5 bits 32 valeurs)
    - Le contenu d'un registre Rs (4 bits pour sélection parmi R0…R15)

# Codage binaire d'une instruction

- Instructions de transfert
  - Lecture/écriture
  - Données accédées
    - Mots, demi-mots, octets
    - Signées/non signées
  - Mode d'accès (pré/post indexé, +/- offset, write back)
  - Transfert simple/multiple
  - Registre source/destination, liste de registres
  - Registre de base
  - Offset
  - Condition
    - Exécution conditionnelle d'un transfert

# Codage binaire

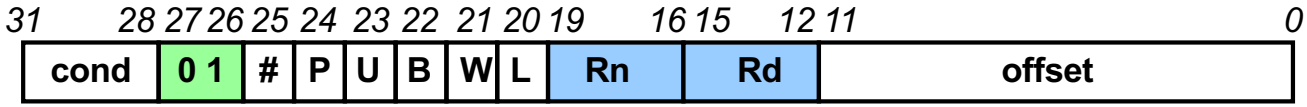
- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)




- *P* : pre/post index
  - 1 pré-indexé, 0 post-indexé
- *U* : up/down
  - 1 + offset, 0 -offset
- *B* : byte/word
  - M1 accès 8 bits, 0 accès 32 bits
- *W* : write-back
  - Si P=1, W=1 adressage pré-indexé automatique
- *L* : load/store
  - 1 load, 0 store

# Codage binaire

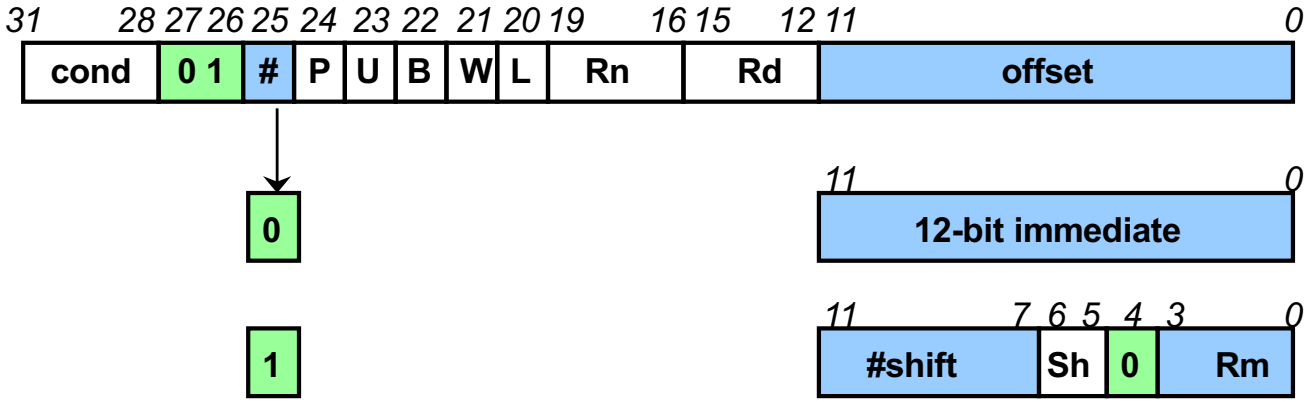
- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)




- *Rd* → registre source (si L=0, store) ou destination (si L=1, load)
- *Rn* → registre de base

# Codage binaire

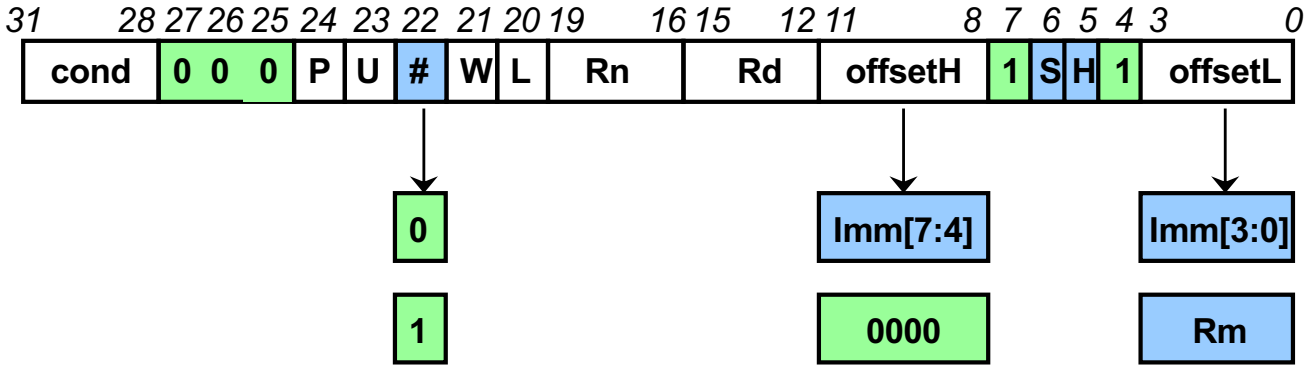
- Instructions de transfert de mots ou d'octets non-signés (LDR, STR, LDRB, STRB)




- Offset: soit un littéral non signé sur 12 bits, soit un registre d'index (*Rm*) éventuellement décalé sur un nombre constant de bits (*#shift*)

# Codage binaire

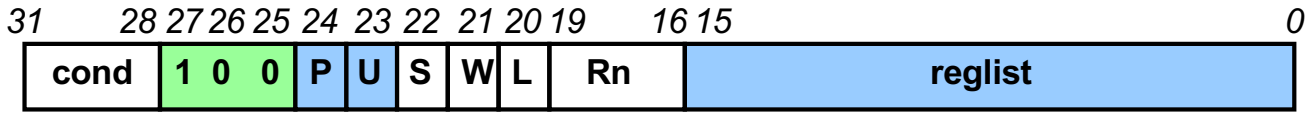
- Instructions de transfert de demi-mots ou d'octets signés (LDRH, STRH, LDRSH, STRSH, LDRSB, STRSB)




- S : signed
  - 1 nombre signé, 0 non signé
- H : half-word/byte
  - 1 accès 16 bits, 0 accès 8 bits
- Offset : soit un litéral sur 8 bits (*Imm*), soit un registre d'index (*Rm*) non décalé

# Codage binaire

- Instructions de transfert multiple (LDM*mode*, STM*mode*)




| mode | U | P |
|---|---|---|
| DA | 0 | 0 |
| DB | 0 | 1 |
| IA | 1 | 0 |
| IB | 1 | 1 |

- Chaque bit de reglist contrôle le transfert d'un registre: si reglist[i] = 1, alors le registre *ri* sera transféré

# Programme

- Pour nous, un programme est une séquence d'instructions disposées l'une après l'autre dans un fichier texte et repérées (éventuellement) par des labels
- Pour le processeur, un programme est
  - Une succession de mots de 32 bits
  - … conformes à la convention de codage des instructions pour le processeur considéré
  - … rangés en mémoire à des adresses contigües
  - Une instruction est repérée par son adresse mémoire

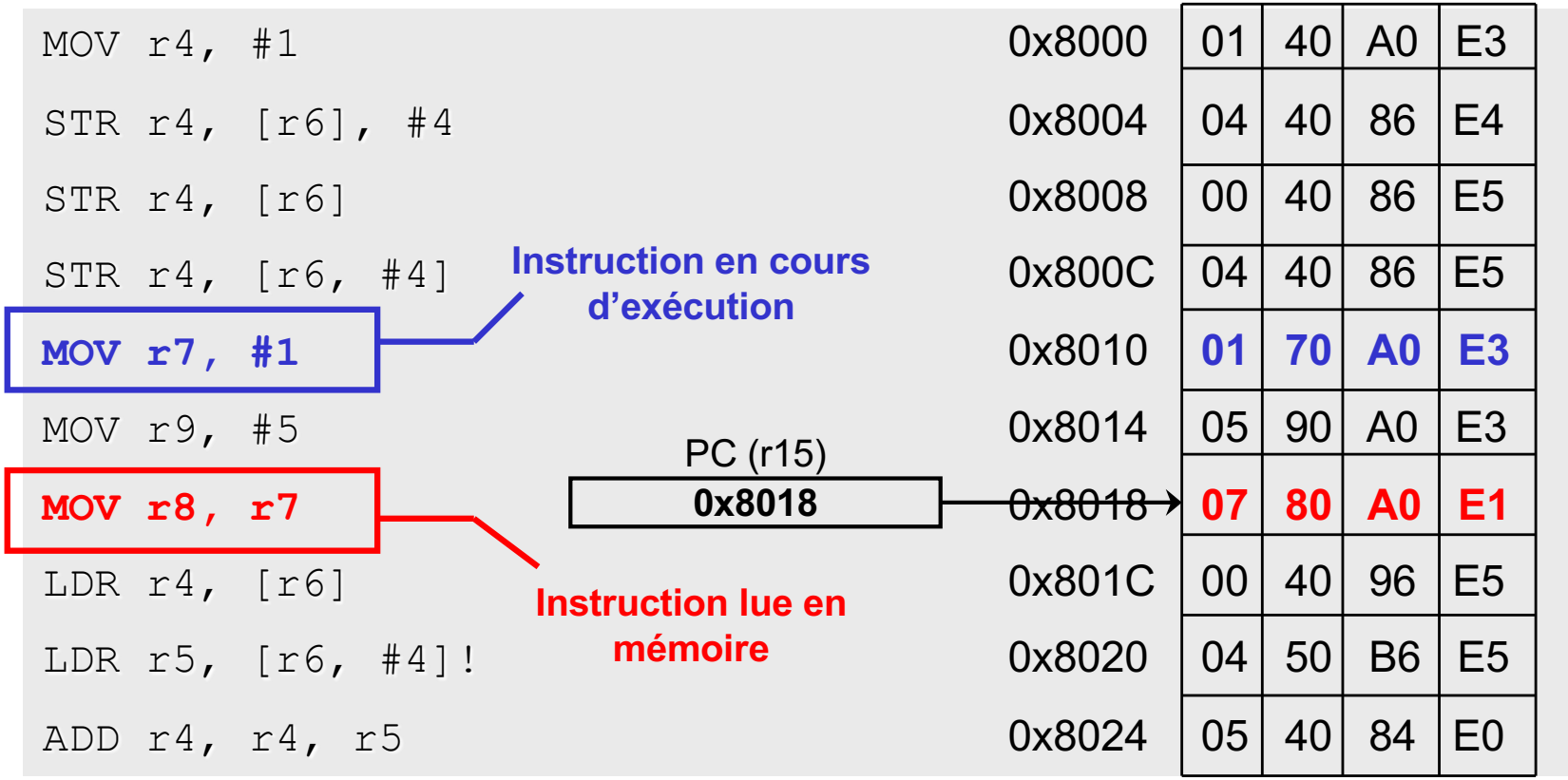
# Programme

- Exemple: programme assembleur ARM implanté à l'adresse 0x8000 pour une organisation little-endian

| | | Adresses croissantes → | | | |
|---|---|---|---|---|---|
| `MOV r4, #1` | 0x8000 | 01 | 40 | A0 | E3 |
| `STR r4, [r6], #4` | 0x8004 | 04 | 40 | 86 | E4 |
| `STR r4, [r6]` | 0x8008 | 00 | 40 | 86 | E5 |
| `STR r4, [r6, #4]` | 0x800C | 04 | 40 | 86 | E5 |
| `MOV r7, #1` | 0x8010 | 01 | 70 | A0 | E3 |
| `MOV r9, #5` | 0x8014 | 05 | 90 | A0 | E3 |
| `MOV r8, r7` | 0x8018 | 07 | 80 | A0 | E1 |
| `LDR r4, [r6]` | 0x801C | 00 | 40 | 96 | E5 |
| `LDR r5, [r6, #4]!` | 0x8020 | 04 | 50 | B6 | E5 |
| `ADD r4, r4, r5` | 0x8024 | 05 | 40 | 84 | E0 |
| | | Poids faible | | | Poids fort |

# Programme

- Le "compteur programme"
  - Registre r15 également appelé PC (Program Counter)
  - Contient l'adresse de la prochaine instruction à lire en mémoire
  - Évolue de 4 en 4 au fil de l'exécution
  - En avance de deux instructions sur l'instruction suivante



# Instructions de branchement relatif

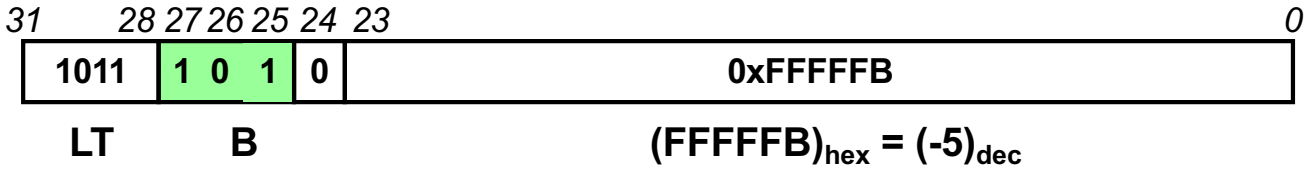
- Branch (B)
- Branch and Link (BL)


- Offset : déplacement signé sur 24 bits
- L : Link (0 branch, 1 branch and link)
- Effets:
  - B: PC ← PC + offset
  - BL: r14 ← PC – 4; PC = PC + offset
  - r14 : Link Register (contient l'adresse de retour)

# Instructions de branchement relatif

- Exemple traduction d'une boucle *for*
- Utilisation *de labels* en langage d'assemblage: le déplacement est calculé automatiquement par l'assembleur

```
                MOV     r4, #0          @ tmp=0
                MOV     r5, #0          @ i=0
loop:           ADD     r4, r4, r5      @ tmp+=i
                ADD     r5, r5, #1      @ i++
                CMP     r5, #5
                BLT     loop            @ i<5 : réitérer
                …
```




**Représentation binaire: BAFFFFFB**

# Instructions de branchement relatif

- Exemple traduction d'une boucle *for*
- Utilisation *d'adresses* en langage d'assemblage: le déplacement est calculé automatiquement par l'assembleur

```
          0x8000        MOV     r4, #0          @ tmp=0
          0x8004        MOV     r5, #0          @ i=0
(Loop:)   0x8008        ADD     r4, r4, r5      @ tmp+=i
          0x800C        ADD     r5, r5, #1      @ i++
          0x8010        CMP     r5, #5
          0x8014        BLT     0x8008 (Loop)   @ i<5 : réitérer
          0x8018        …
```