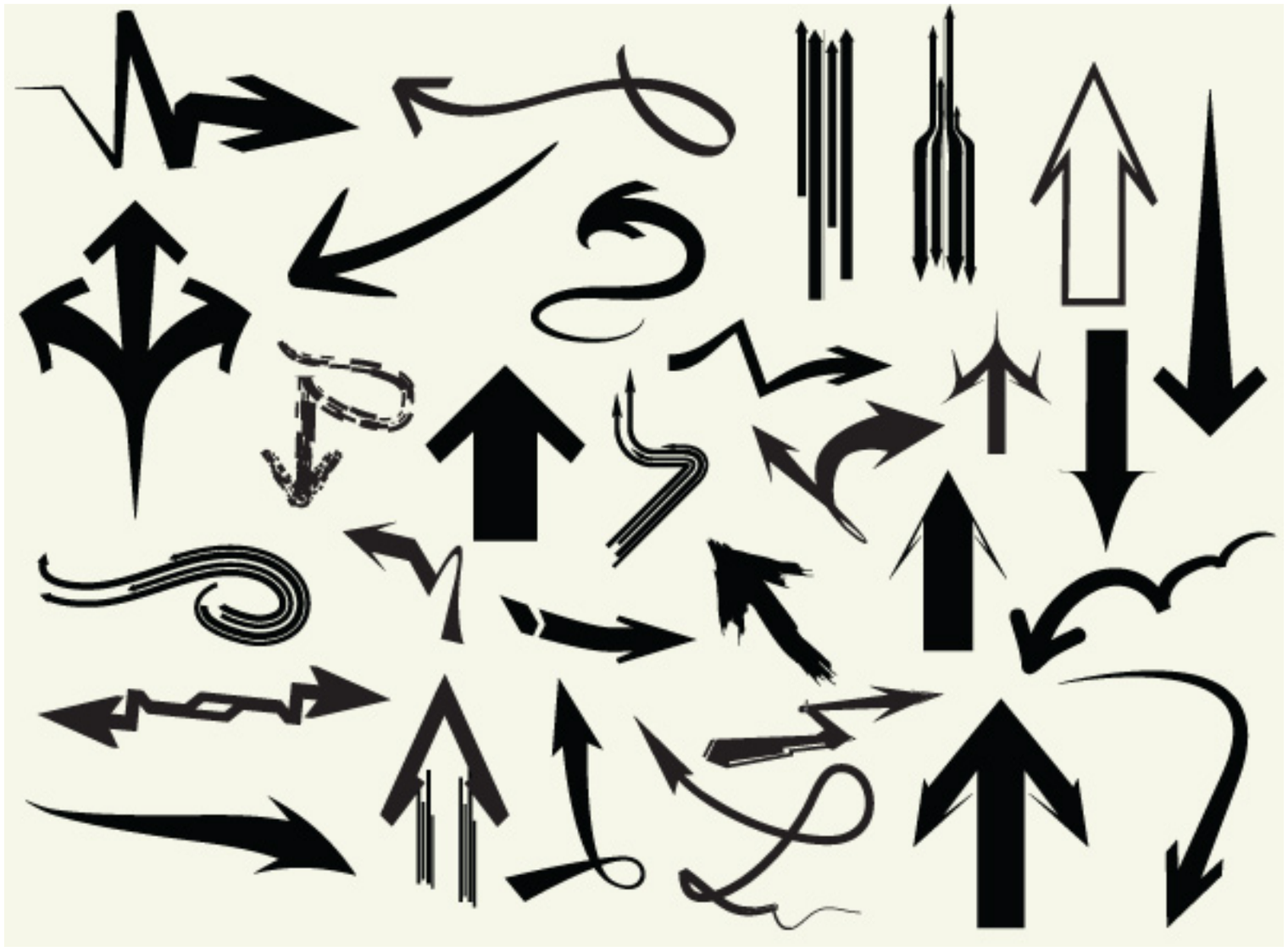
Introduction à l'assembleur ARM:

décalages et déplacements



GIF-1001 Ordinateurs: Structure et Applications, Hiver 2019

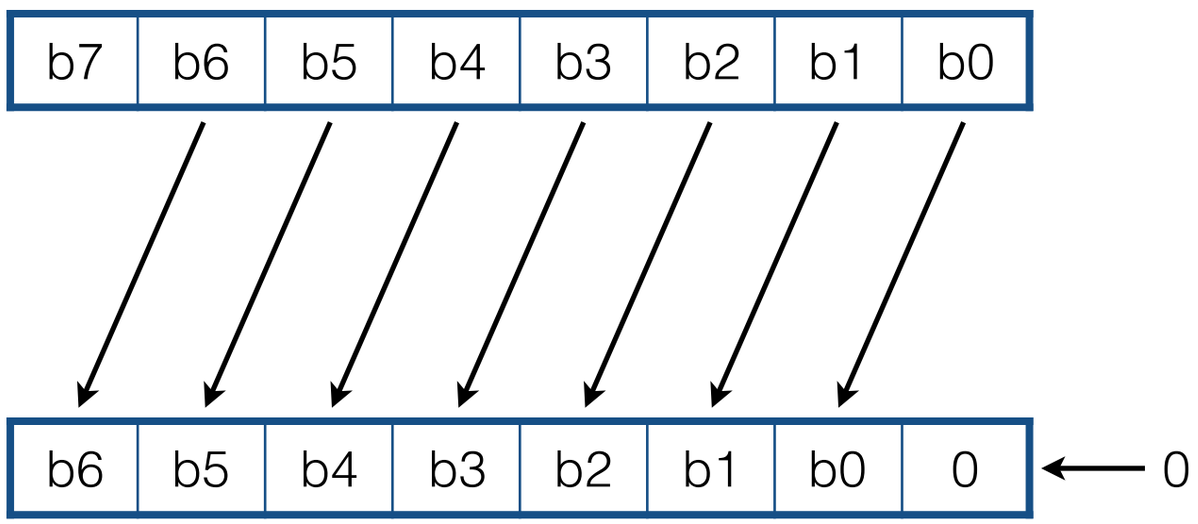
Jean-François Lalonde

image

Décalage de bits

- LSL (*Logical Shift Left*):
  - décale vers la gauche
  - mets des « 0 » à droite
  - équivaut à multiplier par 2

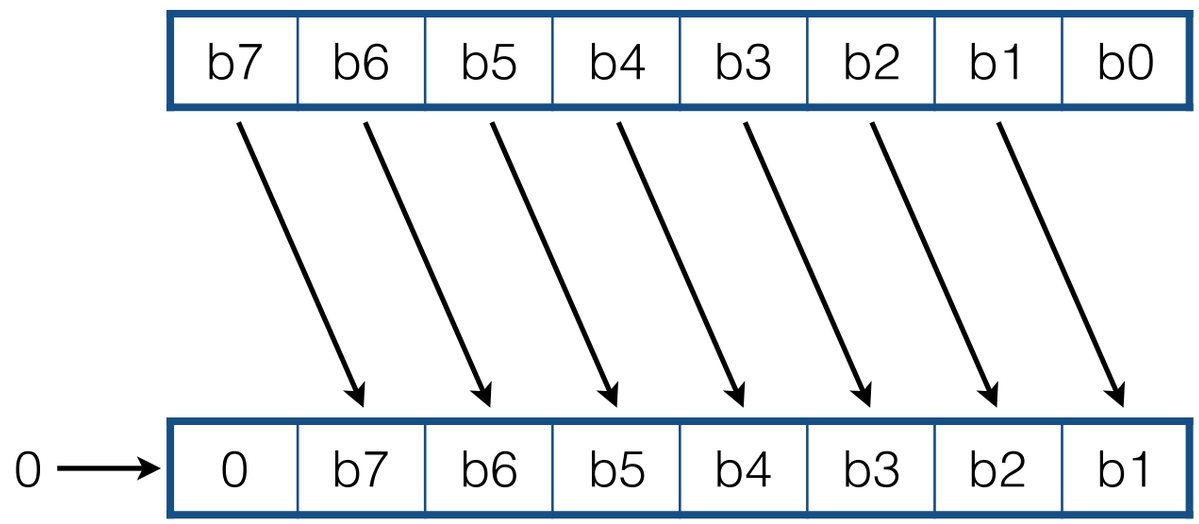
```
LSL R0, R0, #1
```



Décalage de bits

- LSR (*Logical Shift Right*):
  - décale vers la droite
  - mets des « 0 » à gauche
  - équivaut à diviser un **nombre non-signé** par 2

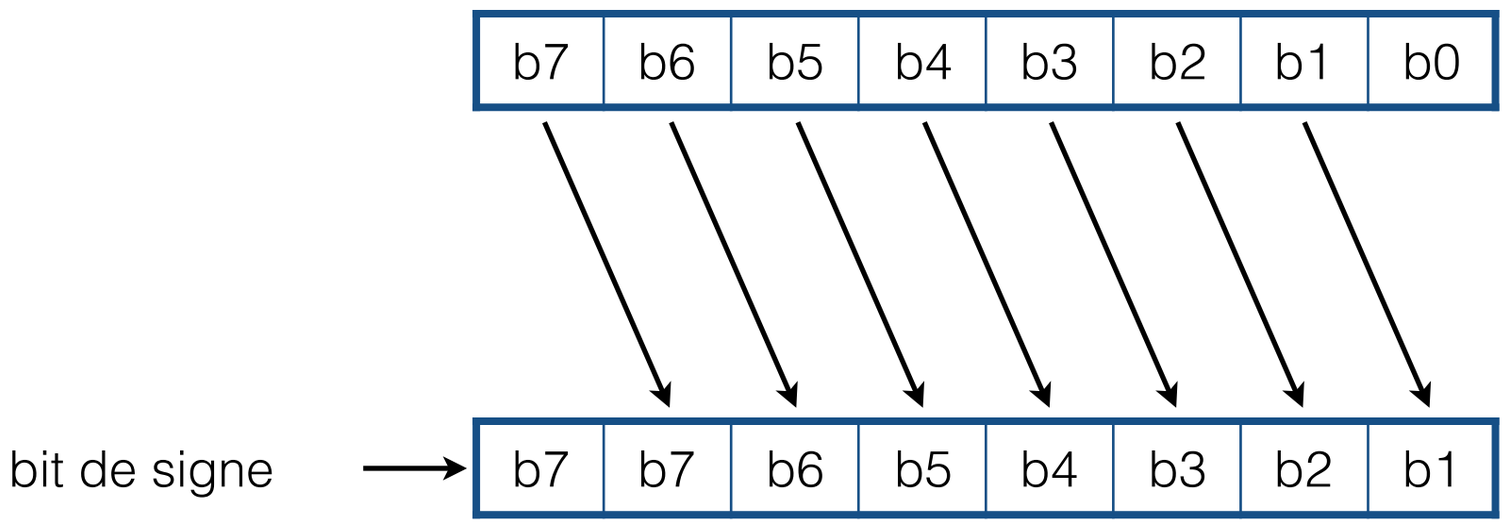
```
LSR R0, R0, #1
```



Décalage de bits

- ASR (*Arithmetic Shift Right*):
  - décale vers la droite
  - mets **le bit de signe** à gauche
  - équivaut à diviser un **nombre signé** par 2

```
ASR R0, R0, #1
```



Déplacement de Données: MOV

- L'instruction

```
MOV Rn Op1
```

  met l'opérande de type 1 `Op1` dans le registre `Rn`

- Opérande de type 1:
  - Constante: toujours précédée du symbole #
  - Registre
  - Registre décalé
    - Le décalage est fait avant l'opération

- Exemples:

```
MOV R0, #1234          ; R0 ← 1234
MOV R0, R1             ; R0 ← R1
MOV R0, R1, ASR #2     ; R0 ← R1 / 4
```

Accès Mémoire: Load/Store

- Les accès à la mémoire se font avec deux instructions:
  - `LDR` (*LoaD Register*) lit la mémoire et met la valeur lue dans un registre.
  - `STR` (*STore Register*) met la valeur d'un registre dans la mémoire.
- Ces instructions ont le format

```
LDR Rd, Op2
STR Rs, Op2
```

- `Rd` et `Rs` décrivent le registre de destination ou de source
- `Op2` est une opérande de type 2

Opérande de type 2

- Symbolise tous les modes d'adressage du microprocesseur: toutes les façons permises pour désigner une adresse de la mémoire.

- Se découpe ainsi

```
LDR Rd, [Rb, Offset]
```

  - `Rb` est le registre de base
  - `Offset` est une opérande de type 1

```
LDR R0, [R2]               ; R0 ← Mémoire[R2]
LDR R0, [R2, #4]           ; R0 ← Mémoire[R2 + 4]
LDR R0, [R2, R3]           ; R0 ← Mémoire[R2 + R3]
LDR R0, [R1, R2, LSL #2] ; R0 ← Mémoire[R1 + (R2 * 4)]
```

- Pour calculer l'adresse, on additionne Rb et Offset

Variantes de LDR/STR

- Pour faciliter les accès aux tableaux, on peut modifier Rb:
  - avant le calcul d'accès mémoire (pre-indexing)
    - symbole « ! »

```
LDR R0, [R1, #4]!     ; R1 ← R1 + 4, suivi de R0 ← Memoire[R1]
STR R0, [R1, #4]!     ; R1 ← R1 + 4, suivi de Memoire[R1] ← R0
```

  - après le calcul d'accès à la mémoire (post-indexing).
    - en dehors des [].

```
LDR R0, [R1], #4      ; R0 ← Memoire[R1], suivi de R1 ← R1 + 4
STR R0, [R1], #4      ; Memoire[R1] ← R0, suivi de R1 ← R1 + 4
```

Démonstration

(Tableaux avec adressage)

Récapitulation: MOV vs LDR/STR

- MOV: déplacements **entre des registres seulement**

```
MOV R0, #0xFF          ; R0 ← 0xFF
MOV R0, R1             ; R0 ← R1
MOV R0, R1, ASR #2     ; R0 ← (R1 / 4)
```

- LDR/STR: déplacements **entre le CPU et la mémoire**

```
LDR R0, [R1]           ; R0 ← Memoire[R1]
LDR R0, [R1, #4]       ; R0 ← Memoire[R1 + 4]
LDR R0, [R1, R2]       ; R0 ← Memoire[R1 + R2]
LDR R0, [R1], #4       ; R0 ← Memoire[R1], R1 ← R1 + 4
```

```
STR R0, [R1]           ; Memoire[R1] ← R0
STR R0, [R1, #4]       ; Memoire[R1 + 4] ← R0
STR R0, [R1, R2]       ; Memoire[R1 + R2] ← R0
STR R0, [R1], #4       ; Memoire[R1] ← R0, R1 ← R1 + 4
```

Accès mémoire avec PC

- On peut aussi se servir de PC pour accéder à la mémoire

```
LDR Rd, [PC, #16]  ; Rd ← Memoire[PC + 16]
```

- On se rappelle:
  - PC contient l'adresse de l'instruction courante **+ 8**
    - PC est « en avance »: il pointe 2 instructions plus loin.
  - Donc, dans l'exemple ci-haut, si l'instruction courante est à l'adresse 0x80, nous aurons

```
0x80     LDR Rd, [PC, #16]  ; Rd ← Memoire[(0x80+8) + 16]
```

Démonstration

(Adressage en ARM)