# CHAPITRE 3 : JEU D'INSTRUCTIONS ARMV7

## 3.1 Langage Assembleur

Un processeur interprète des instructions "numériques" (généralement codées en binaire), c'est ce qui est appelé *langage machine*.

L'assembleur est un langage de programmation bas niveau,

- Lisible par l'homme (représentation textuelle),
- équivalent au langage machine (1 ↔ 1).

Il est spécifique à chaque processeur et lié à son architecture.

## 3.2 Modèle du programmeur

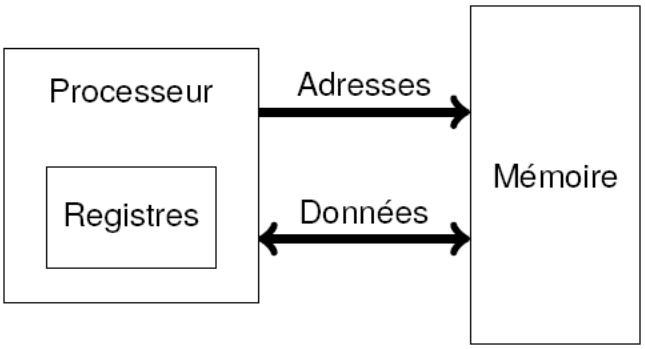



Figure 3.1 architecture ARM

- Espace mémoire extérieur au processeur ;
- Des registres internes au processeur ;
- Instructions pour le transfert entre les registres et la mémoire ;
- Opérations sur le contenu des registres ;
- L'espace mémoire est commun aux instructions et aux données ;
- Les périphériques font aussi partie de l'espace mémoire.

Pour rendre le code plus compact et permettre des réductions de coûts, des jeux d'instructions simplifiés ont été ajoutés :

Les instructions font :

- 32 bits (mode ARM) ;
- 16 bits (mode THUMB) ;
- mixte 32/16 bits (mode THUMB 2).

## 3.3 Instructions ARM/Thumb

On peut classer les instructions en trois grandes catégories :

1. Traitement et manipulation des données :

- Arithmétiques et logiques ;
- Tests et comparaisons.

2. Transfert de données depuis et vers la mémoire ;

3. Contrôle de flot ;

- Branchements

### 3.3.1 Opérations arithmétiques et logiques

Opération sur 3 registres

```
OPE  r_dest, r_s1, r_s2
```

*Exemples*

`AND r0,r1,r2` pour $(r_0 = r_1 \& r_2)$
`ADD r5,r1,r5` pour $(r_5 = r_1 + r_5)$

*Les instructions*

| | | |
|---|---|---|
| `ADD r0,r1,r2` | → `r0=r1+r2` | Addition |
| `ADC r0,r1,r2` | → `r0=r1+r2+C` | Addition avec retenue |
| `SUB r0,r1,r2` | → `r0=r1-r2` | Soustraction |
| `SBC r0,r1,r2` | → `r0=r1-r2-C+1` | Soustraction avec retenue |
| `RSB r0,r1,r2` | → `r0=r2-r1` | Soustraction inversée |
| `RSC r0,r1,r2` | → `r0=r2-r1-C+1` | Soustraction inversée avec retenue |
| `AND r0,r1,r2` | → `r0=r1&r2` | Et binaire |
| `ORR r0,r1,r2` | → `r0=r1\|r2` | Ou binaire |
| `EOR r0,r1,r2` | → `r0=r1^r2` | Ou exclusif binaire |
| `BIC r0,r1,r2` | → `r0=r1&~r2` | Met à `0` les bits de r1 indiqués par r2 |

### 3.3.2 Opérations de déplacement de données entre registres

Opération sur 2 registres

```
OPE r_dest, r_s1
```

*Exemples*

`MOV r0,r1` pour $(r_0 = r_1)$
`MOV pc,lr` pour $(pc = lr)$
`MVN r0,r1` pour $(r_0 = \sim r_1)$

### 3.3.3 Opérations de décalage

Opérations de décalage

```
OPE r_dest, r_s, r_m
```

*Exemples*

`LSL r0,r1,r2` pour $(r_0 = r_1 \ll r_2[7{:}0])$
`ASR r3,r4,r5` pour $(r_3 = r_4 \gg r_5[7{:}0])$

Seul l'octet de poids faible de r_m est utilisé.

*Les instructions*

| | | |
|---|---|---|
| LSL | → | Décalage logique vers la gauche |
| LSR | → | Décalage logique vers la droite |
| ASL | → | Décalage arithmétique vers la gauche |
| ASR | → | Décalage arithmétique vers la droite |
| ROR | → | Décalage circulaire vers la droite |

**Modification des indicateur du PSR**

Par défaut, les opérations arithmétiques et logiques ne modifient pas les indicateurs (N,Z,C,V) du PSR. Il faut ajouter le suffixe "S" au mnémonique de l'instruction.

*Exemples*

```
ADDS r0,r1,r2
ANDS r0,r1,r2
MOVS r0,r1
```

### 3.3.4 Opérations de comparaison

Opération sur 2 registres

```
OPE r_s1, r_s2
```

*Exemples*

`CMP r0,r1` pour $(psr \leftarrow r_0 - r_1)$
`TEQ r0,r1` pour $(psr \leftarrow r_0 \oplus r_1)$

*Les instructions*

| | | | |
|---|---|---|---|
| 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 |

Ces instructions ne modifient que les bits (N,Z,C,V) du PSR, le résultat n'est pas gardé.

### 3.3.5 Opérandes immédiats

Un des opérandes source peut être un immédiat.

- Un immédiat est une valeur constante qui est encodée dans une partie de l'instruction.

On doit les précéder du symbole '#'. Il peut être décimal, hexadécimal (0x) ou octal (0).

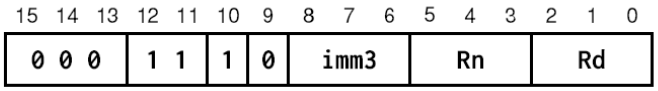
*Exemples*

```
MOV r0,#0x20
CMP r0,#32
ADD r0,r1,#1
```

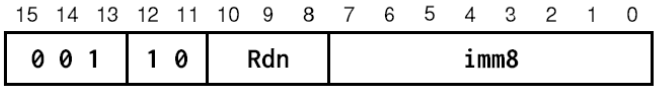
**Opérandes immédiats en mode thumb-16bits**

Ça dépend de l'instruction. Par exemple :

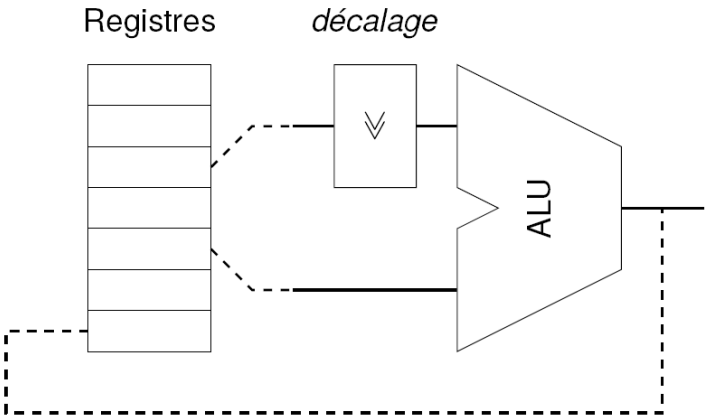
- `ADDS Rd,Rn,#imm3` (source et destination différentes)




- `ADDS Rdn,#imm8` (même source et destination)




### 3.3.6 Combiner une opération avec un décalage (ALU + Barrel shifter)




*Exemples*

```
ADD r0,r1,r2,LSL #4        (r0 = r1 + r2 × 16)
ADD r0,r1,r2,LSL r3        (r0 = r1 + r2 × 2^r3)
```

### 3.3.7 Les multiplications

Les opérandes ne peuvent être que des registres.

En fonction des versions de l'architecture :

- Toutes les instructions ne sont pas disponibles sur toutes les architectures.
- La retenue "C" et le dépassement "V" n'ont pas toujours le même comportement.

**Les instructions**

```
MUL r0,r1,r2       → r0=r*r2          multiplication
MLA r0,r1,r2,r3    → r0=r1+r2*r3      mult. et accumulation
```

| | | |
|---|---|---|
| `MLS r0,r1,r2,r3` | → `r0=r1-r2*r3` | mult. soustraction |
| `UMULL r0,r1,r2,r3` | → `{r1,r0}=r2*r3` | mult. 64bits non signée |
| `SMULL r0,r1,r2,r3` | → `{r1,r0}=r2*r3` | mult. 64bits signée |
| `UMLAL r0,r1,r2,r3` | → `{r1,r0}+=r2*r3` | MAC 64bits non signée |
| `SMLAL r0,r1,r2,r3` | → `{r1,r0}+=r2*r3` | MAC 64bits signée |

### 3.3.8 Instructions pour transférer les données

Deux instructions de transfert de données entre la mémoire et les registres.

1. LDR : charger un registre avec une donnée en mémoire
2. STR : enregistrer la valeur du registre en mémoire

*Exemples*

| | |
|---|---|
| `LDR r0,[r1]` | $(r_0 = RAM[r_1])$ |
| `STR r0,[r1]` | $(RAM[r_1] = r_0)$ |

### 3.4 Modes d'adressage

- **Adressage indirect**

  `LDR r0,[r1]` $(r_0 = RAM[r_1])$

- **Adressage indirect avec déplacement (offset)**

  `LDR r0,[r1,#8]` $(r_0 = RAM[r_1 + 8])$

  `LDR r0,[r1,r2]` $(r_0 = RAM[r_1 + r_2])$

- **Adressage indirect avec déplacement et pré-incrémentation**

  `LDR r0,[r1,#8]!` $(r_1 = r_1 + 8 \text{ puis } r_0 = RAM[r_1])$

- **Adressage indirect avec déplacement et post-incrémentation**

  `LDR r0,[r1],#8` $(r_0 = RAM[r_1] \text{ puis } r_1 = r_1 + 8)$

### 3.5 Transferts multiples

En plus des instructions LDR et STR le jeu d'instruction ARM propose les instructions LDM et STM pour les transferts multiples.

*Exemples*

| | |
|---|---|
| `LDMIA r0,{r1,r2,r3}` | $(r_1 = RAM[r_0])$ |
| | $(r_2 = RAM[r_0 + 4])$ |
| | $(r_3 = RAM[r_0 + 8])$ |
| `STMIA r0,{r1-r3}` | $(RAM[r_0] = r_1)$ |
| | $(RAM[r_0 + 4] = r_2)$ |
| | $(RAM[r_0 + 8] = r_3)$ |

IA pour la post-incrémentation (Increment After)

### 3.6 Branchements

Il existe deux instructions de branchement :

B adresse Aller à l'adresse

BX registre Aller à l'adresse pointée par le registre et éventuellement changer de mode (ARM/THUMB interworking)

Ces instructions modifient le compteur programme "pc" (r15) BL(X) Pour sauvegarder l'adresse de retour dans "lr" (r14)

L'adresse de retour est celle de l'instruction suivant le BL.

Pour revenir d'un branchement BL il suffit de remettre lr dans pc

- BX lr
- MOV pc, lr

### 3.7 Exécution conditionnelle des instructions

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

| | | |
|---|---|---|
| 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 |

*Exemples*

| | |
|---|---|
| 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 $r_0 = r_1 - r_0$ |
| SUBS r0,r1,r2 | $r_0 = r_1 - r_2$ |
| BEQ address | aller à adresse si le résultat est nul |