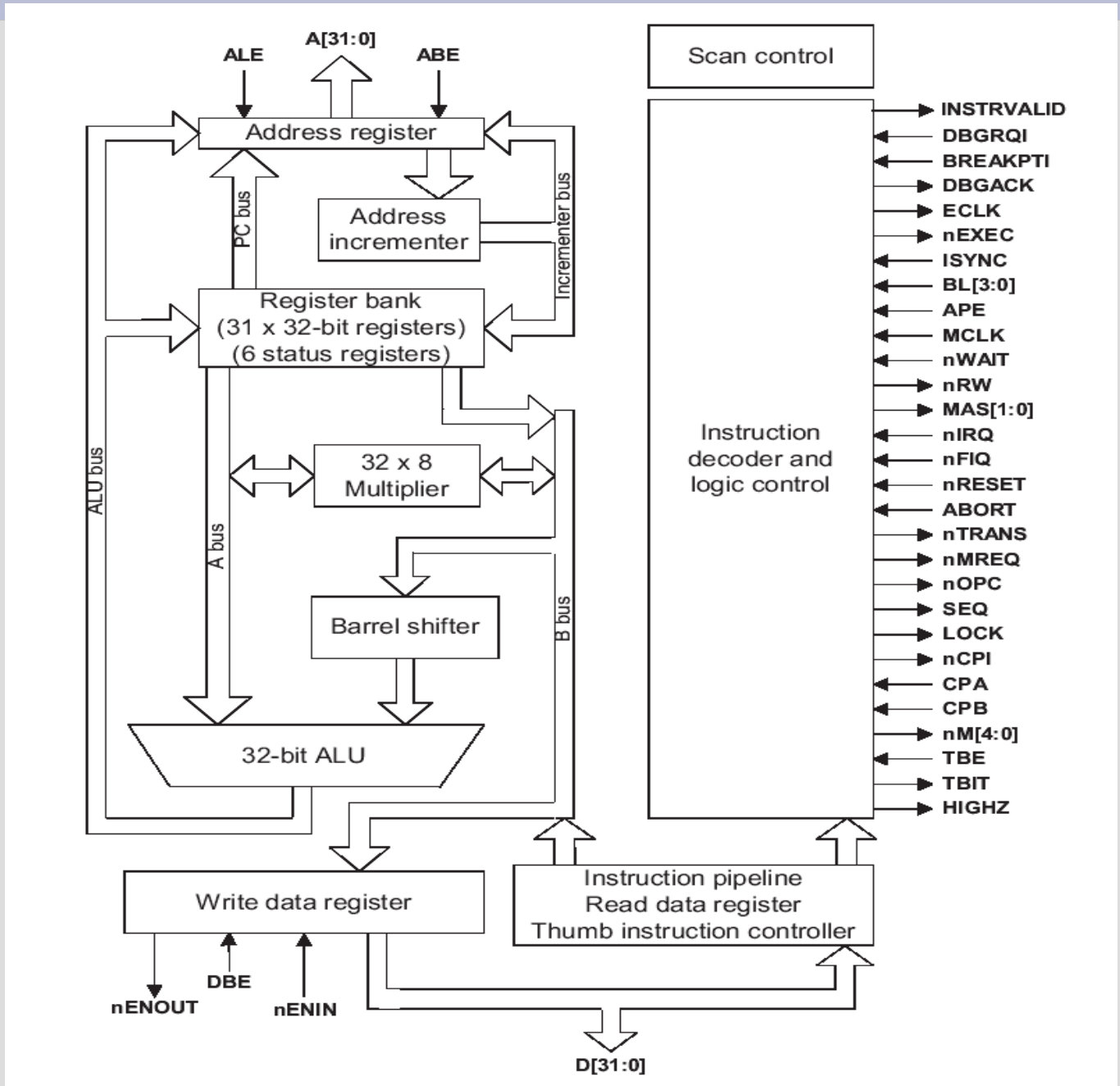
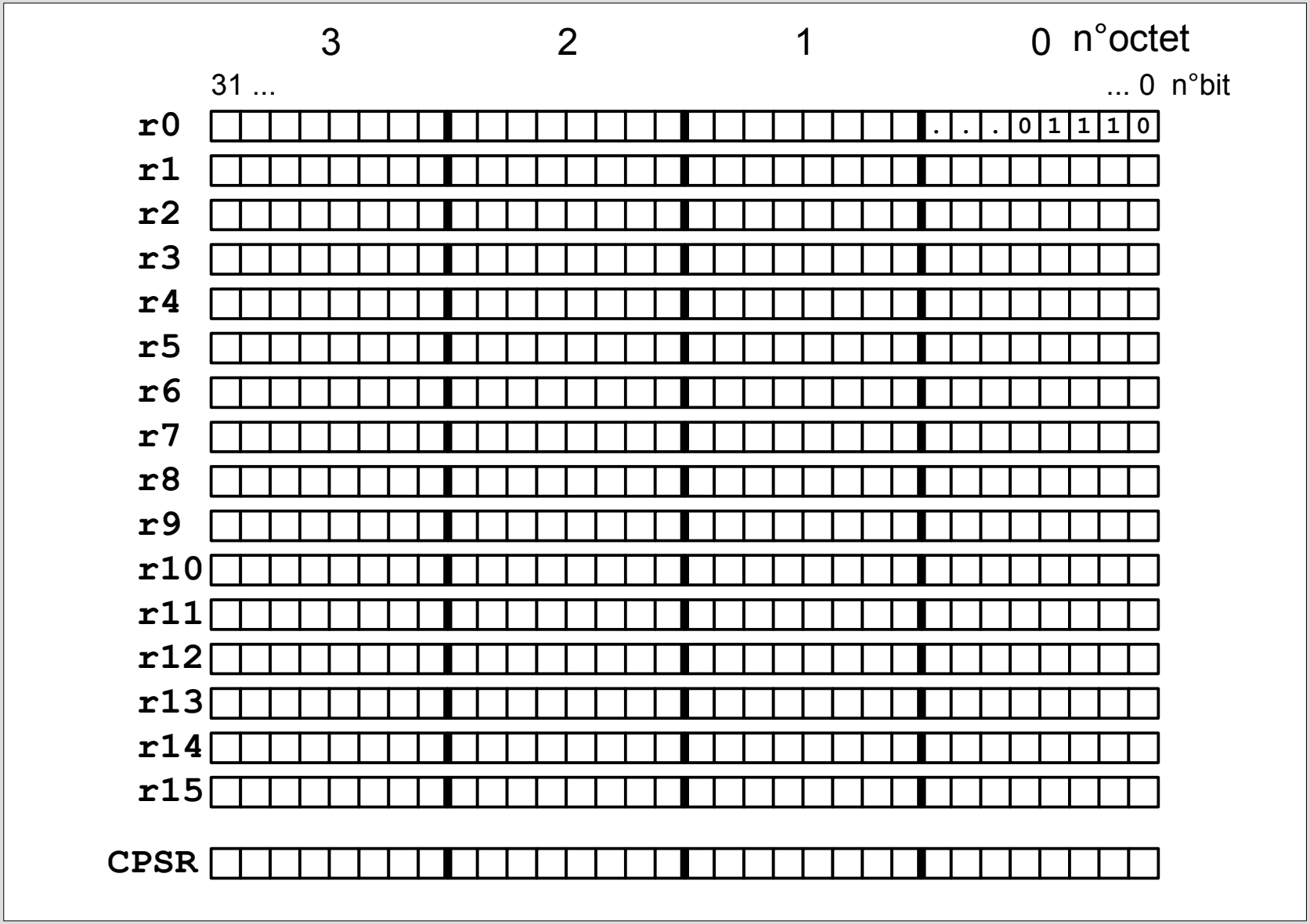
# Introduction à la programmation assembleur ARM

# Chemins de données de l'ARM



# Assembleur ARM

- 17 registres de 32 bits (4 octets)



# Assembleur ARM

**Registres d'usage général : r0 à r12**

- On peut les utiliser librement pour faire
  - Des calculs entiers, des compteurs de boucles ...
  - Des pointeurs vers des adresses mémoire

- L'assembleur n'est pas typé : c'est au programmeur d'interpréter et d'utiliser de manière cohérente les contenus 32 bits des registres.
- La notation hexadécimale est privilégiée

# Assembleur ARM

**Registres particuliers**

- **r13** alias **sp** : stack pointer
  pointeur sur pile de données
- **r14** alias **lr** : link register
  contient l'adresse de retour à l'appelant
- **r15** alias **pc** : program counter
  c'est le compteur ordinal...

- Ils peuvent être manipulés par les instructions au même titre que les registres r0 à r12

# Assembleur ARM

**Registre CPSR** *Current Program Status Register*

- Il contient les **indicateurs de code condition** qui dépendent du résultat d'une opération
  - **N** (Negative) : bit indiquant un résultat négatif
  - **Z** (Zero) : bit indiquant un résultat nul
  - **C** (Carry) : bit indiquant une retenue sortante
  - **V** (oVerflow) : bit indiquant un débordement signé

- Le registre CPSR ne peut pas être manipulé par les instructions usuelles

# Assembleur ARM

- Exemple d'instruction : addition

```
ADD   r0,r1,r2 @ r0<-r1+r2
```

- Le premier élément indique l'**opération**
- Les registres r1 et r2 sont deux **opérandes**
- Le registre r0 est la **destination**
- Le `@` indique un commentaire (optionnel)

# Assembleur ARM

- Exemple d'instruction : incrémentation de 4

```
ADD   r0,r0,#4  @ r0<-r0+4
```

- En général le ou les registres opérandes ne sont pas modifiés par une instruction
- ( Sauf si le registre est aussi en destination )

# Assembleur ARM

- La documentation précise le format général :

```
ADD   rd,rn,<Oprnd2>
```

- Le premier opérande et la destination de l'addition sont nécessairement des registres
- La table *Operand 2* précise que le 2ème opérande est un registre, un registre décalé, ou une "valeur immédiate" préfixée par #
- Ces contraintes sont liées à l'architecture

# Assembleur ARM

- L'instruction de "mouvement" de donnée

```
MOV   rd,<Oprnd2>  @ rd <- Oprnd2
```

- Le registre destination prend la valeur du deuxième opérande
  - Lorsqu'il n'y a qu'un opérande on considère que c'est le "deuxième opérande"
  - Le "premier opérande" est implicitement absent

# Assembleur ARM

Exemples de mouvements de données

```
MOV     r3,#15      @ r3 <- 0x0000000F
MOV     r4,#0x2E    @ r4 <- 0x0000002E
MOV     r0,r3       @ r0 <- r3 = 0x0000000F
MOV     r1,r4       @ r1 <- r4 = 0x0000002E
```

- Contraintes sur les valeurs immédiates
  De 0 à 256  : toutes valeurs possibles
  De 256 à 1024 : multiples de 4
  De 1024 à 4096 : multiples de 16
  ...

# Assembleur ARM

- L'opérande valeur immédiate #<immed_8r>

  *Une constante 32 bits obtenue en faisant une rotation à droite une valeur 8 bits d'un nombre pair de bits.*

- En pratique la rotation est utilisée comme un décalage à gauche d'un nombre pair de bits,
- Donc valeurs de type [0 ... 255] * $2^{2n}$ $(0 \leq n < 12)$

# Assembleur ARM

Pourquoi ces contraintes sur les constantes ?

- Le format des instructions ARM est fixe : **Une instruction fait 4 octets (32 bits)**
- On ne peut donc faire rentrer à la fois le descriptif de l'opération et une constante de 32 bits dans un format fixe de 32 bits.

# Format des instructions ARM

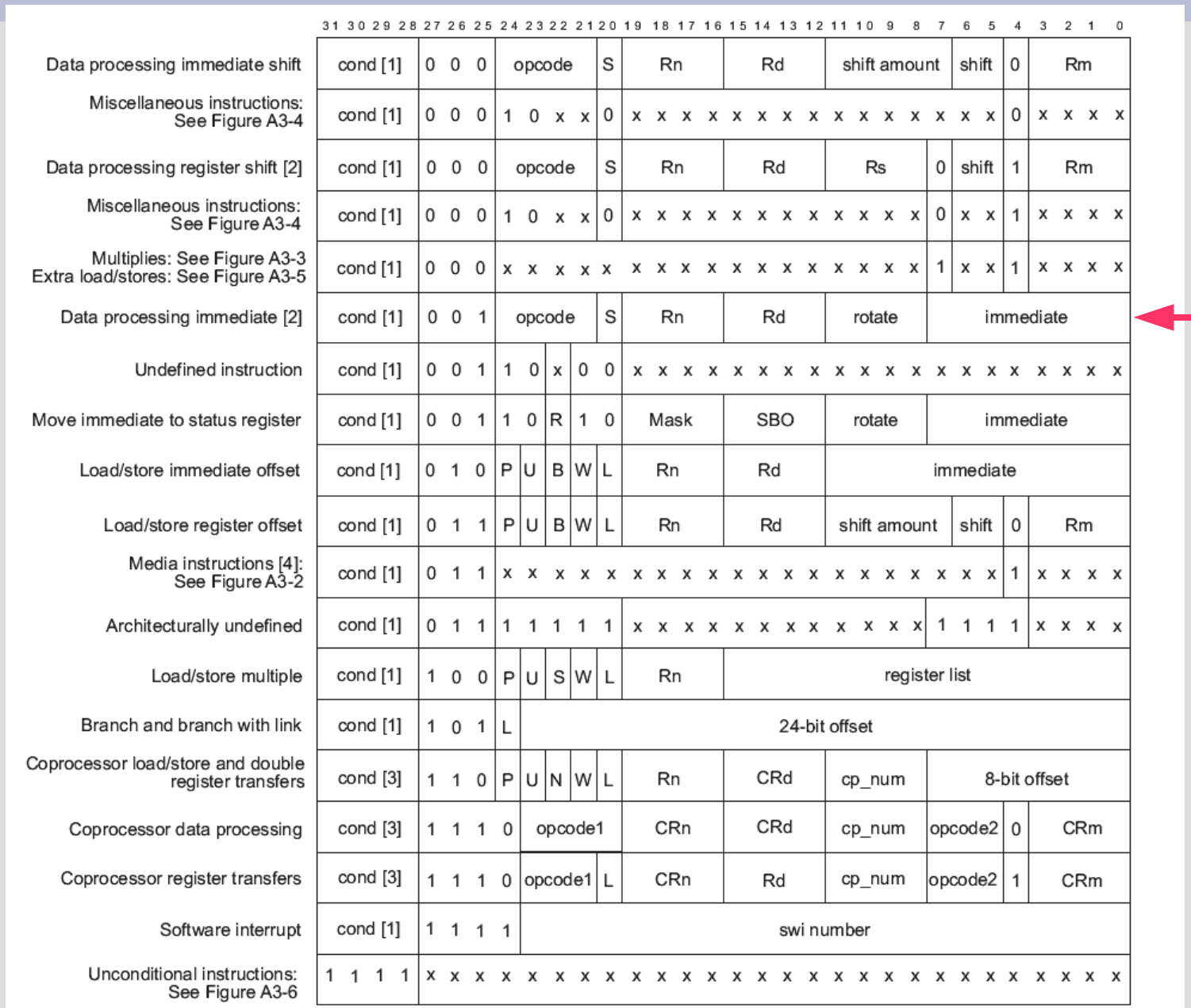



**Figure A3-1 ARM instruction set summary**

# Codage d'instruction

**Exemple de codage d'une instruction :**

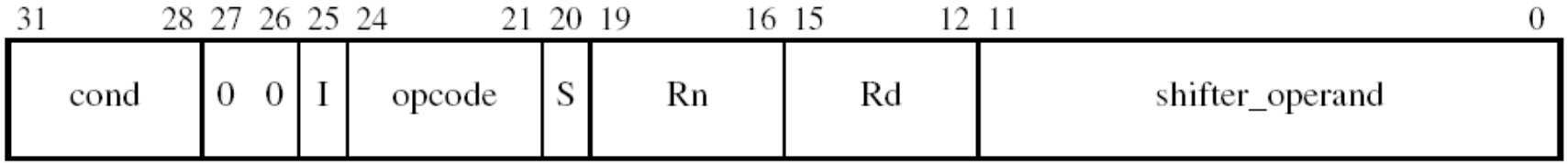
```
MOV r5,#7  @ Instruction Assembleur
```

```
 0b11100011101000000101000000000111
=0xE3A05007     ( code langage machine)
```

# Format instruction Data-processing

## A3.4.1 Instruction encoding

```
<opcode1>{<cond>}{S}  <Rd>, <shifter_operand>
<opcode1> := MOV | MVN
<opcode2>{<cond>}  <Rn>, <shifter_operand>
<opcode2> := CMP | CMN | TST | TEQ
<opcode3>{<cond>}{S}  <Rd>, <Rn>, <shifter_operand>
<opcode3> := ADD | SUB | RSB | ADC | SBC | RSC | AND | BIC | EOR | ORR
```




**I bit** Distinguishes between the immediate and register forms of <shifter_operand>.

**S bit** Signifies that the instruction updates the condition codes.

**Rn** Specifies the first source operand register.

**Rd** Specifies the destination register.

**shifter_operand** Specifies the second source operand. See *Addressing Mode 1 - Data-processing operands* on page A5-2 for details of the shifter operands.

# Conditions d'exécution

| Opcode [31:28] | Mnemonic extension | Meaning | Condition flag state |
|---|---|---|---|
| 0000 | EQ | Equal | Z set |
| 0001 | NE | Not equal | Z clear |
| 0010 | CS/HS | Carry set/unsigned higher or same | C set |
| 0011 | CC/LO | Carry clear/unsigned lower | C clear |
| 0100 | MI | Minus/negative | N set |
| 0101 | PL | Plus/positive or zero | N clear |
| 0110 | VS | Overflow | V set |
| 0111 | VC | No overflow | V clear |
| 1000 | HI | Unsigned higher | C set and Z clear |
| 1001 | LS | Unsigned lower or same | C clear or Z set |
| 1010 | GE | Signed greater than or equal | N set and V set, or N clear and V clear (N == V) |
| 1011 | LT | Signed less than | N set and V clear, or N clear and V set (N != V) |
| 1100 | GT | Signed greater than | Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V) |
| 1101 | LE | Signed less than or equal | Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V) |
| 1110 | AL | Always (unconditional) | - |

# Codage d'instruction: condition

```
MOV r5,#7
```

- Exécution inconditionnelles : Always
  ->1110
- Puis spécification de famille d'opération
  ( Data-processing : MOV, ADD, CMP ...)
  ->111000
- Puis indicateur d'opérande immédiat (I)
  ->1110001

# Codes d'opérations : Opcode

Table A3-2 Data-processing instructions

| Opcode | Mnemonic | Operation | Action |
|---|---|---|---|
| 0000 | AND | Logical AND | Rd := Rn AND shifter_operand |
| 0001 | EOR | Logical Exclusive OR | Rd := Rn EOR shifter_operand |
| 0010 | SUB | Subtract | Rd := Rn - shifter_operand |
| 0011 | RSB | Reverse Subtract | Rd := shifter_operand - Rn |
| 0100 | ADD | Add | Rd := Rn + shifter_operand |
| 0101 | ADC | Add with Carry | Rd := Rn + shifter_operand + Carry Flag |
| 0110 | SBC | Subtract with Carry | Rd := Rn - shifter_operand - NOT(Carry Flag) |
| 0111 | RSC | Reverse Subtract with Carry | Rd := shifter_operand - Rn - NOT(Carry Flag) |
| 1000 | TST | Test | Update flags after Rn AND shifter_operand |
| 1001 | TEQ | Test Equivalence | Update flags after Rn EOR shifter_operand |
| 1010 | CMP | Compare | Update flags after Rn - shifter_operand |
| 1011 | CMN | Compare Negated | Update flags after Rn + shifter_operand |
| 1100 | ORR | Logical (inclusive) OR | Rd := Rn OR shifter_operand |
| 1101 | MOV | Move | Rd := shifter_operand (no first operand) |
| 1110 | BIC | Bit Clear | Rd := Rn AND NOT(shifter_operand) |
| 1111 | MVN | Move Not | Rd := NOT shifter_operand (no first operand) |

# Codage d'instruction: Opcode

**`MOV r5,#7`**

- C'est une opération Move
  **`->1110001`**`1101`
- Puis indicateur "pas de mise à jour des codes conditions" (pas de S en suffixe)
  **`->111000111010`**

# Codage d'instruction: Registres

```
MOV r5,#7
```

- MOV : pas de 1er opérande (Rn à 0)
  `->1110001110100000`
- Registre destination R5 (Rd code pour 5)
  `->1110001110100000101`

# Codage d'instruction: immed_8r

**`MOV r5,#7`**

- Format immédiat : constante 8 bits et rotation
  **`constante = 0b00000111`**
- Pas besoin de déplacer la constante 8 bits pour obtenir la valeur immédiate (rotation 0)
  `->11100011101000000101`**`0000`**

- Constante 8 bits
  `->11100011101000000101000`**`00000111`**

# Codage d'instruction: résultat

```
MOV   r5,#7

0b11100011101000000101000000000111
=0xE3A05007
```

# Assembleur ARM

- Opérations d'addition et de soustraction

```
ADD r0,r1,r2  @ r0 <- r1+r2
SUB r0,r1,r2  @ r0 <- r1-r2
RSB r0,r1,r2  @ r0 <- r2-r1
```

- Opérations de multiplication

```
MUL r0,r1,r2     @ r0 <- r1*r2
MLA r0,r1,r2,r3  @ r0 <- r1*r2+r3
```

- Pas d'instruction de division !

# Assembleur ARM

- Opérations logiques "bit à bit"

```
AND r0,r1,r2  @ ET     r0=r1&r2;
ORR r0,r1,r2  @ OU     r0=r1|r2;
EOR r0,r1,r2  @ OUEx.  r0=r1^r2;
BIC r0,r1,r2  @ ETNon  r0=r1&(~r1);
MVN r0,r2     @ NON    r0=~r2;
```

- A droite : code C équivalent

# Assembleur ARM

- Exemples d'opérations logiques

```
@ LDR rd,=val : charger constante quelconque
LDR r1,=0b1111000011110000
LDR r2,=0b0000000011111111

AND r3,r1,r2
ORR r4,r1,r2
EOR r5,r1,r2

@ opérande1          r1=0b1111000011110000
@ opérande2          r2=0b0000000011111111
@ résultat ET        r3=0b0000000011110000
@ résultat OU        r4=0b1111000011111111
@ résultat OUEx.     r5=0b1111000000001111
```

# Assembleur ARM

- Opérations de décalages et rotations

```
MOV r0,r2,LSL #3    @ r0=r2<<3;
MOV r0,r2,LSR #5    @ r0=r2>>5;(1)
MOV r0,r2,ASR #5    @ r0=r2>>5;(2)
MOV r0,r2,ROR #6    @ rotation ...
```

- (1) décalage à droite non signé
- (2) décalage à droite signé

# Assembleur ARM

- Opérations de décalages et rotations peuvent se faire sur le 2ème opérande

```
@                   r0=r1+(r2<<3);
@ équivaut à r0=r1+(r2*8);

ADD r0,r1,r2,LSL #3

@                   r0=r1-(r2>>2);
@ équivaut à r0=r1-(r2/4);

SUB r0,r1,r2,LSR #2
```

# Assembleur ARM

- La possibilité d'effectuer rotations et décalage sur le deuxième opérande **avant** d'effectuer une opération arithmétique ou logique vient de la présence d'un "décaleur à barillet" sur le chemin du bus B en amont de l'UAL.
- La possibilité concerne donc aussi les constantes ( pour données immédiates ) transitant avec l'instruction par le registre d'instruction.

# Accès mémoire

L'ARM est une architecture **LOAD/STORE**

- Des instructions spécifiques sont nécessaires pour accéder à la mémoire, c.à.d échanger des données entre mémoire et registres

**load : mémoire → registre**
**store : registre → mémoire**

# Accès mémoire

- Deux instructions spécifiques pour la mémoire :
- r0 prend la valeur du mot mémoire pointé par r1

```
LDR   r0,[r1]
```

- Le mot mémoire pointé par r1 prend la valeur r0

```
STR   r0,[r1]
```

# Accès mémoire

Quelques variantes :

- Pour Lire/Ecrire des demi-mots

```
LDRH    r0,[r1]
STRH    r0,[r1]
```

- Pour Lire/Ecrire des octets

```
LDRB    r0,[r1]
STRB    r0,[r1]
```

- Pour Lire/Ecrire plusieurs mots

```
LDM     r1,{r0,r1,r5-r8}
STM     r1,{r0,r1,r5-r8}
```

# Accès mémoire

On peut aussi "déplacer" l'adresse

- r0 prend la valeur du mot en mémoire à l'adresse r1 – 16  (16 octets avant)

```
LDR   r0,[r1,#-16] @ r1 non modifié
```

- Le mot en mémoire à l'adresse r1 + 560  prend la valeur de r0

```
STR   r0,[r1,#560] @ r1 non modifié
```

# Accès mémoire

- Il existe de nombreuses options pour accéder à des adresses "déplacées" par rapport au registre servant de pointeur de référence, avec ou sans modification du pointeur lui même
- Nous détaillerons ces possibilités plus tard... (Adressing Mode 2 / Adressing Mode 3)

- Impossible d'accéder directement à une adresse absolue : il faut passer par un registre pointeur

# Directives d'assemblage...

- Un fichier source assembleur n'est pas composé que d'instructions en langage assembleur
- Les **directives d'assemblage** sont des commandes ou des indications données au programme d'assemblage pour inclure des fichiers, définir des constantes, réserver des espaces en mémoire (données)
- Les directives d'assemblage sont toutes préfixées par un point

# Directives d'assemblage...

- Inclure un fichier

```
.INCLUDE "fichier.s"
```

- Définir une constante symbolique
( équivalent au #define du C)

```
.EQU    MACONSTANTE, 0x0200
```

# Directives d'assemblage...

- Réserver des emplacement pour des données (correspond aux variables statiques du C)

```
.BYTE   56,'a',-12,255
.HWORD  0xB200,0
.WORD   0x06000000
.FILL   8,2,0x0102
.ASCIZ  "une chaine"
.ALIGN
```

# Etiquettes

- Le programme d'assemblage décide des emplacements mémoire des données réservées et des instructions

- Il faut un système de référence symbolique pour désigner les emplacements mémoire
- Les étiquettes (*labels*) sont synonymes de l'adresse correspondant à leur emplacement
- Ce sont des identifiants **uniques** postfixés par un deux-point

# Etiquettes

- Exemple de définition d'une étiquette :

```
MonEtiquette:
        .WORD 0x89ABCDEF
        .WORD 0xFFFFFFFF
```

- Si le 1er mot réservé est placé en mémoire par l'assemblage à l'adresse **0x03001A00** alors MonEtiquette est synonyme de **0x03001A00**
- Le second mot est à l'adresse **0x03001A04** car le placement se fait à des adresse consécutives

# Etiquettes

- Problème pour initialiser un registre pointeur avec une adresse absolue donnée par une étiquette :

```
MOV r1,#MonEtiquette  @ Marche ?
LDR r0,[r1]
```

- En général l'adresse n'est pas une valeur immédiate de type **#<immed_8r>**

# Accès mémoire et LDR

- On peut quand même accéder à la variable si elle n'est pas "trop loin" en utilisant le registre pc comme adresse de départ :

```
   .WORD 0x89ABCDEF
DebutCode:
   ADD r3,r3,#1   @ du code entre .word et ldr
   SUB r4,r4,#1   @ du code
   LDR r0,[pc,#-(8+12)]
```

- Le registre pc est toujours 2 instructions "en avant" donc décalé de +8 par rapport à l'instruction en cours.

# Accès mémoire et LDR

- Cette solution en **adressage relatif** fonctionne mais n'est pas pratique : il faut compter les éléments en mémoire entre la donnée et l'instruction load.
- Le programme d'assemblage peut automatiser cette tâche :

```
MaVar:      .WORD   0x89ABCDEF
DebutCode:
   ADD r3,r3,#1
   SUB r4,r4,#1
   LDR r0,MaVar
```

# Accès mémoire et LDR

```
LDR r0,MaVar  @ Charge la valeur
```

- Il s'agit ici d'une **pseudo-instruction**
- Le programme d'assemblage transforme cette ligne du code source assembleur en une instruction réelle différente ou plus complexe...
- En réalité on aura l'instruction :

```
LDR r0,[pc,#-24]
```

# Accès mémoire et LDR

- Le programme d'assemblage nous assiste en prenant en charge la tâche ingrate de calculer des adresses relatives
- Attention il faut quand même comprendre ce qui se passe...

```
MaVar:          .WORD   0x89ABCDEF
                .FILL   1050,4
DebutCode:
  LDR r0,MaVar     @ Ne marche pas !
                   -> dépasse <immed_12>
```

# Mémoire, LDR et constantes

- Une autre utilisation de LDR très utile correspond également à une pseudo-instruction

- Permet (enfin !) de charger des valeurs constantes arbitraires dans un registre

```
LDR r0,=0x21

LDR r0,=0xAE274F21
```

# Mémoire, LDR et constantes

- Si valeur indiquée compatible avec <immed_8r>

```
LDR r0,=0x21    →    MOV r0,#0x21
```

- Sinon

```
LDR r0,=0xAE274F21
```

```
LDR r0,[pc,#0x???]
...
...    @ Fin des instructions
  .WORD 0xAE274F21
```

Déplacement relatif

# Mémoire, LDR et constantes

- Avec les pseudo-instructions le programme d'assemblage ne se contente pas uniquement d'une transformation syntaxique
- Pour des séquences de code longues (plus de 1000 instructions) il est nécessaire de prévoir des emplacements mémoire réservés pour stocker ces constantes
- C'est la directive **`.LTORG`** qui permet de le faire
- Pour simplifier, la mini-librairie du kit propose un mot clé POOL qui est utilisable au milieu du code (à n'utiliser qu'en cas de problème)

# Mémoire, STR

- STR reconnaît aussi un format type pseudo-instruction qui permet d'écrire directement une valeur dans un emplacement mémoire indiqué par une étiquette

```
MaVar:      .WORD  0
DebutCode:
   LDR r0,=0x76543210
   STR r0,MaVar    @ ro -> MaVar
   ...
```

- Mêmes contraintes que **`LDR r0,MaVar`**

# Mémoire, contraintes

- Pour accéder à des mots il est impératif d'utiliser des adresses multiples de 4
- Ceci est lié à l'accès par bus 32 bits : un mot non **aligné** serait à cheval sur 2 accès mémoire
- Les demi-mots nécessitent des adresses paires
- Les octets peuvent être accédés sans contrainte d'alignement d'adresse

# Mémoire, contraintes

- Lors d'une réservation de mémoire on peut désaligner la succession des emplacements quand on utilise des demi-mots ou des octets
- La directive .ALIGN oblige l'assembleur à compléter avec des octets (inutilisés) pour retrouver une adresse multiple de 4

```
Mes3Chars:  .BYTE   'a','b','c'
            .ALIGN
MonEntier:  .WORD   0xC00FC00F
```