Manipulation des données

Opérations arithmétiques et logiques

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

`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

Opérations de déplacement de données entre registres

```
OPE r_dest, r_s1
```

`MOV r0,r1`	→	`r0=r1`	Déplacement
`MVN r0,r1`	→	`r0=~r1`	Déplacement et négation

Opérations de décalage

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

`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

Opérations de comparaison

`OPE r_s1, r_s2`

`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.

La multiplication

`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

- 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

Transfert des données

Instructions pour transférer les données

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

- `LDR` : charger un registre avec une donnée en mémoire
- `STR` : enregistrer la valeur du registre en mémoire

`LDR/STR`	:	mots de 32 bits (`words`)
`LDRH/STRH`	:	mots de 16 bits (`half words`)
`LDRB/STRB`	:	mots de 16 bits (`byte`)

Généralement, les adresses **doivent** être alignées :

`LDR/STR`	:	4
`LDRH/STRH`	:	2
`LDRB/STRB`	:	quelconque

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]!` ($r_1 = r_1 + 8$ puis $r_0 = RAM[r_1]$)
- Adressage indirect avec déplacement et post-incrémentation
  `LDR r0,[r1],#8` ($r_0 = RAM[r_1]$ puis $r_1 = r_1 + 8$)

Transferts multiples

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

`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}`	$(RAM[r_0] = r_1)$
	$(RAM[r_0 + 4] = r_2)$
	$(RAM[r_0 + 8] = r_3)$

Il existe 4 suffixes possibles pour les instructions de transferts multiples :

- `IA` pour la post-incrémentation (*Increment After*)
- `IB` pour la pré-incrémentation (*Increment Before*)
- `DA` pour la post-décrémentation (*Decrement After*)
- `DB` pour la pré-décrémentation (*Decrement Before*)

Pour que la valeur du registre d'adresse soit modifiée il faut ajouter (**!**)

```
LDMIA r0!,{r1-r3}
```

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` (deprecated)

- pour les instructions `B` et `BL` l'adresse est stockée comme un immédiat qui représente un offset par rapport à la position actuelle :
  - l'offset est forcement limité
- pour l'instruction `BX` le mode (ARM/Thumb) est déterminé en fonction du bit de poids faible de l'adresse normalement non utilisé (*voir interworking*)

Exécution conditionnelle

`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

CMP r0,r1
BEQ adress : Branchement si r0=r1
BHI adress :Branchement si r0>r1

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Exemple

```
Start:
    MOV  r0,#0  @ mise zero de r0
    MOV  r2,#10 @ charger la valeur 10 dans r2
Loop:
    ADD  r0,r0,r2,LSL #1 @ r0=r0+2*r2
    SUBS r2,r2,#1        @ r2--
    BNE  Loop
    B    Start
```

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

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

Exemple: Si `MaVar` vaut 4, exécute une tâche:

exemple en C

```
if (MaVar == 4) {
  // exécute une tâche…
}

// le programme continue…
```

assembleur

```
LDR R0, MaVar      ; Met la valeur de MaVar dans R0
CMP R0, 4          ; Change les drapeaux comme R0-4

BNE PasEgal
; execute une tâche…

PasEgal
; le programme continue…
```

exemple en C

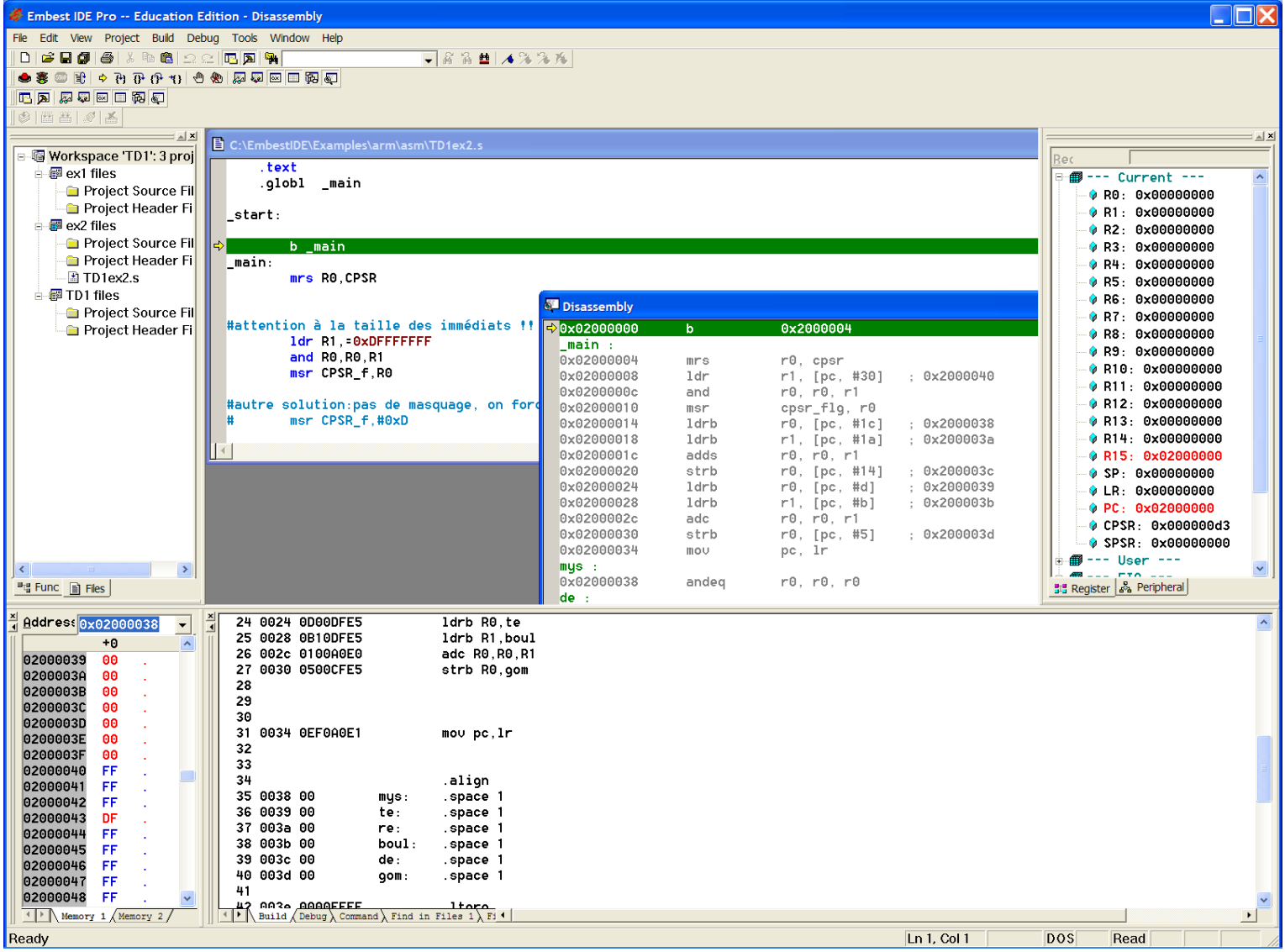
```
for (int i = 0; i < 5; ++i) {
  // tâche à l'intérieur de la boucle
}
// le programme continue…
```

assembleur

```
InitDeBoucle
    MOV R4,#5

DebutDeBoucle
    ; tâche à l'intérieur de la boucle
    SUBS R4, R4, #1      ; R4 = R4 - 1, change les drapeaux
    BNE DebutDeBoucle    ; Condition d'arrêt

; le programme continue…
```



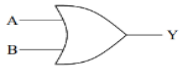
https://youtu.be/9xKp1RH1y1Y https://youtu.be/BFMc5Ba9_cQ

Simplification	$A + (A \bullet B) = A$ $A \bullet (A + B) = A$ $(A + B) \bullet (A + \overline{B}) = A$ $A + (\overline{A} \bullet B) = A + B$
Théorème De Morgan	$\overline{A \bullet B \bullet C \bullet \ldots} = \overline{A} + \overline{B} + \overline{C} + \ldots$ $\overline{A + B + C + \ldots} = \overline{A} \bullet \overline{B} \bullet \overline{C} \bullet \ldots$
OU exclusif	$A \oplus B = (A + B) \bullet \overline{(A \bullet B)}$ $A \oplus B = (A \bullet \overline{B}) + (B \bullet \overline{A})$ $A \oplus B = \overline{(A \bullet B) + (\overline{A} \bullet \overline{B})}$ $A \oplus B = (A + B) \bullet (\overline{A} + \overline{B})$

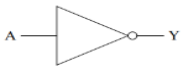
OU	$(A + B) + C = A + (B + C) = A + B + C$ $A + B = B + A$ $A + A = A$ $A + 0 = A$ $A + 1 = 1$	Associativité Commutativité Idempotence Elément neutre Elément absorbant
ET	$(A \cdot B) \cdot C = A \cdot (B \cdot C) = A \cdot B \cdot C$ $A \cdot B = B \cdot A$ $A \cdot A = A$ $A \cdot 1 = A$ $A \cdot 0 = 0$	Associativité Commutativité Idempotence Elément neutre Elément absorbant
Distributivité	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ $A + (B \cdot C) = (A + B) \cdot (A + C)$	
NON	$\overline{\overline{A}} = A$ $\overline{A} + A = 1$ $\overline{A} \cdot A = 0$	

AND (ET)

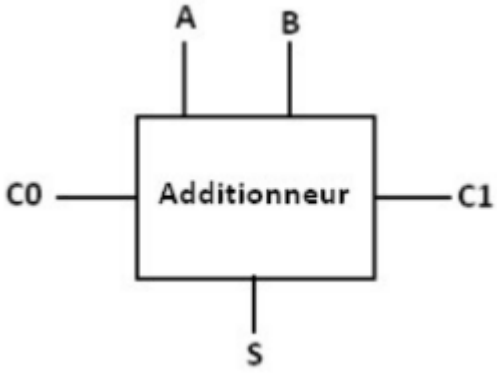
OR (OU)



NOT (NON)



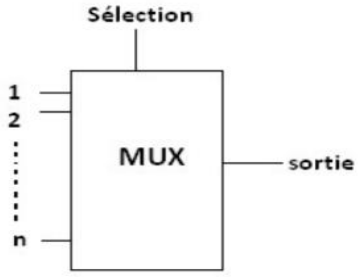
(XOR) OU exclusif



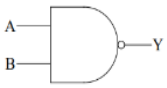
Quelques circuits combinat

Le multiplexeur : (MUX) circuit 1 parmi 2^n

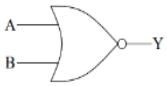
- Il est composé de 2^n entrées, 1 sortie, et n lignes de sélection (lignes de commande, lignes d'adresse)
- Il permet de faire la liaison entre 1 entrée parmi $m=2^n$ et la sortie en fonction des n lignes de sélection
- Il permet de véhiculer sur un seul support les informations provenant de plusieurs sources en sélectionnant une entrée parmi les entrées



NAND (NON ET)

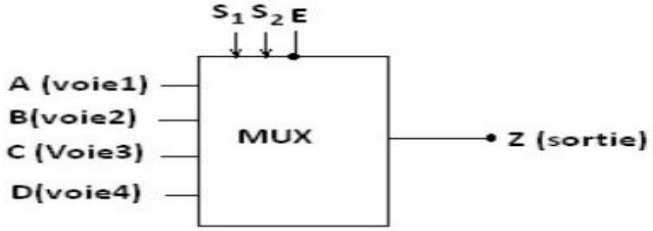


NOR (NON OU)



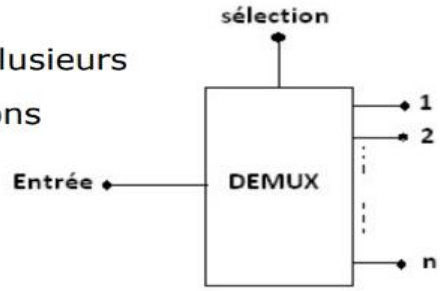
Exemple d'un multiplexeur à quatre voies

- **A, B, C** et **D** Constituent les voies d'entrées
- **Z** constitue la sortie
- **E** est l'entrée de validation
- S_1 et S_2 pour la sélection



Le démultiplexeur : (DEMUX)

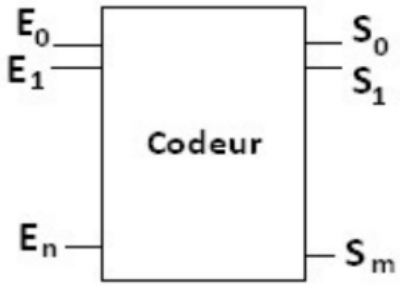
- Circuit à n+1 entrées et 2^n sorties
- n entrées, appelées entrées d'adressage, et une entrée, appelée l'entrée donnée
- Il a pour rôle de repartir sur plusieurs lignes parallèles les informations provenant en série d'une seule ligne



1, 2, ... n sont les sorties

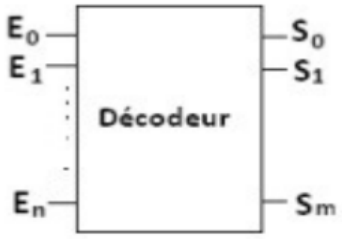
Le codeur

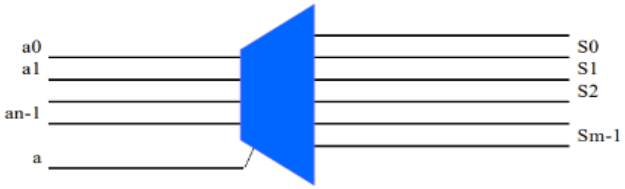
- Un **codeur** est un circuit logique qui a pour rôle de convertir un code (décimal) en un code binaire
- Il possède en général 2^m entrées dont une seule qui doit être active à la fois, et m sorties



Le décodeur

- Il a n lignes d'entrée (lignes d'adresse) et 2^n lignes de sortie
- En fonction des lignes d'adresse, on va activer l'une des 2^n sorties
- On peut avoir une entrée supplémentaire a, on obtient donc un décodeur à validation





Le comparateur

- Il permet de faire la comparaison de deux mots de n bits
- Cas simple : 1 si A=B
  0 si A≠B
- On a l'égalité des deux mots si tous les bits sont égaux

$$A_i=B_i \Rightarrow A_i \oplus B_i = 0$$
$$A_i \neq B_i \Rightarrow A_i \oplus B_i = 1$$

- On utilise ces deux relations pour construire le comparateur

$$A=B \Rightarrow \forall i\ A_i=B_i \Rightarrow \sum A_i \oplus B_i = 0$$
$$A=B \Leftrightarrow \text{NON}\ (\sum A_i \oplus B_i)$$

Codage des entiers relatifs

Binaire signé

Sur n bits :

- 1 bit de signe (bit de poids fort : n-1)
  - 1 pour un nombre négatif
  - 0 pour un nombre positif
- (n-1) bits pour coder la valeur absolue

Exemples :

-8 : 10001000 (bit de signe | valeur absolue)

8 : 00001000 (bit de signe | valeur absolue)

Problème :

2 codes pour le zéro : 10000000 et 00000000

Codage des nombres

Cette méthode de traduction des parties fractionnaires en binaire peut conduire à des problèmes d'arrondi

Exemple : 0.777 à coder en binaire sur un octet

```
0.777 * 2 =  1.554
0.554 * 2 =  1.108
0.108 * 2 =  0.216
0.216 * 2 =  0.432
0.432 * 2 =  0.864
0.864 * 2 =  1.728
0.728 * 2 =  1.456
0.456 * 2 =  0.912
```

0.11000110

0.11000110=0.5+0.25+0.015625+0.0078125=0.7734375

Problème :

On a deux codes pour zéro : 00000000 et 11111111

Codage des réels : la norme IEEE-754

Représentation des réels en virgule flottante

Codage sous la forme d'un triplet évalué sur base binaire :

signe, valeur ou **mantisse** et **exposant**

Exemple : (en décimal)

$5487{,}643 = (-1)^{s} * 54876430 * 10^{-4}$ mantisse=54876430 exposant=-4 signe=0

$5487{,}643 = (-1)^{s} * 00054876 * 10^{-1}$ mantisse=00054876 exposant=-1 signe=0

Forme normalisée :

Le premier chiffre (de poids fort) doit être différent de 0

Codage des entiers relatifs

Codage biaisé

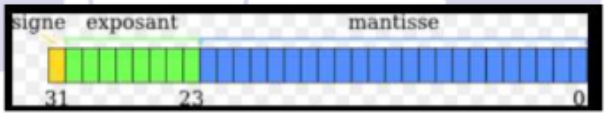
Codage binaire du nombre augmenté de $2^{n-1}-1$

n est le nombre de bits du codage (127 pour un octet)

Exemples : sur 4 bits (n = 4) ➔ $2^{n-1}-1 = 2^{4-1}-1 = 7$

-7 : -7 + 7 = 0 ➔ 0000
-4 : -4 + 7 = 3 ➔ 0011
0 : 0 + 7 = 7 ➔ 0111
1 : 1 + 7 = 8 ➔ 1000
6 : 6 + 7 = 13 ➔ 1011
8 : 8 + 7 = 15 ➔ 1111

Codage des réels : la norme IEEE-754



On admet deux représentations :

- **La simple précision, sur 4 octets**

  23 bits de mantisse

  8 bits d'exposant : codage biaisé

  Valeur($[S,E,M]_{flt}$) = $(-1)^S * 2^{(exposant)}$ *mantisse

  exposant=E-127 mantisse=1,M_{22} ….. M_0

  ➔ précision sur 7 chiffres

  ➔ la plus grande valeur absolue: 10^{38}

- **La double précision, sur 8 octets**

  52 bits de mantisse

  11 bits d'exposant

Opérations sur les chaînes de bits

b. Opérations de décalage :

Décalage à droite (DECD):

- Les bits de rang *n* à 1 sont décalés d'une position vers la droite;
- Le bit de rang *n* du résultat est mis à 0.
- DECD(1101) = 0110

Décalage à gauche (DECG):

- Opération inverse de DECD
- DECG(1101) = 1010

Différentes formulations du XOR:

• Y=A ⊕ B est égal à 1 si et seulement si A = 1 ou B = 1 mais pas simultanément :

$$A \oplus B = (A + B) \cdot \overline{(A \cdot B)}$$

• Y=A ⊕ B égal à 1 si A = 1 et B = 0 ou si B = 1 et A = 0. Soit :

$$A \oplus B = (A \cdot \overline{B}) + (B \cdot \overline{A})$$

• Si A et B sont égaux à 1 ou si A et B sont égaux à 0 :

$$A \oplus B = \overline{(A \cdot B) + (\overline{A} \cdot \overline{B})}$$

• Y=A ⊕ B correspond à un détecteur d'égalité (le complément de Y vaut 1 en cas d'égalité):

$$A \oplus B = (A + B) \cdot (\overline{A} + \overline{B})$$

Tableaux de Karnaugh

LES ETATS INDIFFERENTS

Présence d'états indifférents

Quand certaines combinaisons des variables sont **sans effet** sur la valeur de la fonction de sortie, on dit que ce sont des **états indifférents**.
Cela peut être aussi des combinaisons impossibles physiquement (capteur haut et bas sur un store par exemple).

On note ces états par **une croix** dans le tableau de Karnaugh et on les utilise **partiellement** ou **totalement** pour simplifier la fonction de sortie.

Attention comme il est possible d'obtenir l'équation simplifiée soit à partir du regroupement des 1 ou des 0, ces deux formes ne sont plus complémentaires car certains états indifférents figurent implicitement dans les deux formes.

- Toutes les combinaisons des variables sont représentées par une case :

➔ **Pour *n* variables on a un tableau de 2^n cases**