

# Deuxième Partie

1

## Plan

### Algèbre de BOOLE

Définition des variables et fonctions logiques

Les opérateurs de base et les portes logiques

Les lois fondamentales de l'algèbre de Boole

### Logique combinatoire

# Variables et fonctions booléennes

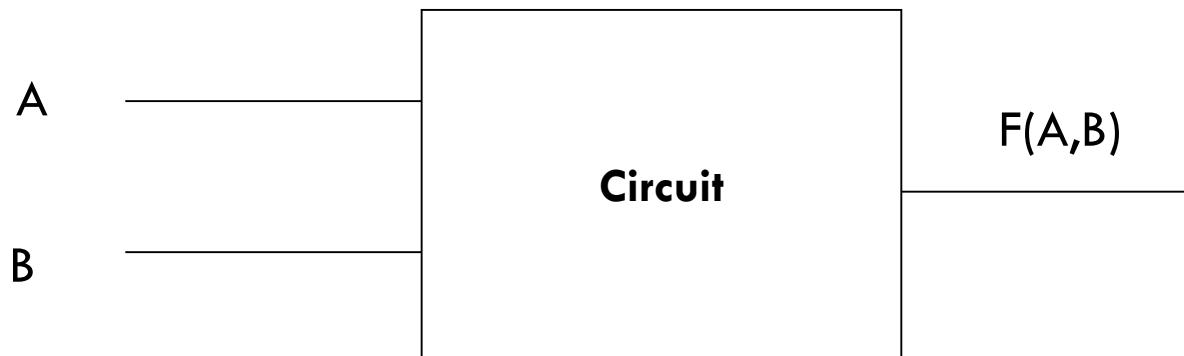
2

- l'algèbre de BOOLE est une structure mathématique qui traite l'ensemble des opérations formelles appliquées aux propositions du raisonnement logique qui seront représentés par des états logiques '0' ou '1'.
- Une variable est une grandeur (symbole) qui peut prendre que 0 ou 1.
- Une fonction logique est représentée par un groupe de variables logiques reliés par des opérations logiques.

# Variables et fonctions booléennes

3

- Les machines numériques sont constituées d'un ensemble de circuits électroniques.
- Chaque circuit fournit une fonction logique bien déterminée ( addition, comparaison ,....).



La fonction  $F(A,B)$  peut être : la somme de  $A$  et  $B$  , ou le résultat de la comparaison de  $A$  et  $B$  ou autre fonction

# Variables et fonctions booléennes

4

- Pour concevoir et réaliser ce circuit on doit avoir un modèle mathématique de la fonction réalisée par ce circuit .
- Ce modèle doit prendre en considération le système binaire.
- Le modèle mathématique utilisé est celui de Boole.
- George Boole est un mathématicien anglais ( 1815-1864).

# Variables booléennes

5

- Définition:
- Il a fait des travaux dont les quels les fonctions ( expressions ) sont constitués par des variables qui peuvent prendre les valeurs 'OUI' ou 'NON' .
- Ces travaux ont été utilisés pour faire l'étude des systèmes qui possèdent deux états s'exclut mutuellement :
  - ▣ Le système peut être uniquement dans deux états E1 et E2 tel que E1 est l'opposé de E2.
  - ▣ Le système ne peut pas être dans l'état E1 et E2 en même temps
- Ces travaux sont bien adaptés au Système binaire ( 0 et 1 ).

# Exemple de systèmes à deux états

6

- Un interrupteur est ouvert ou non ouvert ( fermé )
- Une lampe est allumée ou non allumée ( éteinte )
- Une porte est ouverte ou non ouverte ( fermée )

## □ Conclusion:

On peut utiliser les conventions suivantes :

OUI → VRAI ( true )  
NON → FAUX ( false )

OUI → 1 ( Niveau Haut )  
NON → 0 ( Niveau Bas )

# Définitions

7

- **Niveau logique** : Lorsque on fait l'étude d'un système logique il faut bien préciser le niveau du travail.

Niveau	Logique positive	Logique négative
H ( Hight ) haut	1	0
L ( Low ) bas	0	1

## Exemple :

Logique positive :

lampe allumée : 1

lampe éteinte : 0

Logique négative

lampe allumée : 0

lampe éteinte : 1

# Variable logique ( booléenne )

8

- Une variable logique ( booléenne ) est une variable qui peut prendre soit la valeur 0 ou 1 .
- Généralement elle est exprimée par un seul caractère alphabétique en majuscule ( A , B, S , ...)
- **Exemple :**
  - Une lampe : allumée     $L = 1$   
                  éteinte     $L = 0$



# Fonction logique

9

- C'est une fonction qui relie  $N$  variables logiques grâce à un ensemble d'opérateurs logiques de base.
- Dans l'Algèbre de Boole il existe trois opérateurs de base : ET , OU, NON .
- La valeur d'une fonction logique est égale à 1 ou 0 selon les valeurs des variables logiques.
- Si une fonction logique possède  $n$  variables logiques  $\rightarrow 2^n$  combinaisons  $\rightarrow$  la fonction possède  $2^n$  valeurs.
- Les  $2^n$  combinaisons sont représentées dans une table qui s'appelle table de vérité ( TV ).

# fonction logique

10

□ Exemple:

$$F(A, B, C) = \bar{A}.\bar{B}.C + \bar{A}.B.C + A.\bar{B}.C + A.B.C$$

La fonction F possède 3 variables  $\rightarrow 2^3$  combinaisons

$$F(0,0,0) = \bar{0}.\bar{0}.0 + \bar{0}.0.0 + 0.\bar{0}.0 + 0.0.0 = 0$$

$$F(0,0,1) = \bar{0}.\bar{0}.1 + \bar{0}.0.1 + 0.\bar{0}.1 + 0.0.1 = 1$$

$$F(0,1,0) = \bar{0}.\bar{1}.0 + \bar{0}.1.0 + 0.\bar{1}.0 + 0.1.0 = 0$$

$$F(0,1,1) = \bar{0}.\bar{1}.1 + \bar{0}.1.1 + 0.\bar{1}.1 + 0.1.1 = 1$$

$$F(1,0,0) = \bar{1}.\bar{0}.0 + \bar{1}.0.0 + 1.\bar{0}.0 + 1.0.0 = 0$$

$$F(1,0,1) = \bar{1}.\bar{0}.1 + \bar{1}.0.1 + 1.\bar{0}.1 + 1.0.1 = 1$$

$$F(1,1,0) = \bar{1}.\bar{1}.0 + \bar{1}.1.0 + 1.\bar{1}.0 + 1.1.0 = 0$$

$$F(1,1,1) = \bar{1}.\bar{1}.1 + \bar{1}.1.1 + 1.\bar{1}.1 + 1.1.1 = 1$$

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Une table de vérité

# Opérateurs logiques de base

11

## NON ( négation )

- **NON** : est un opérateur unaire ( une seule variable) qui à pour rôle d'**inverser** la valeur d'une variable .

$$F(A) = \text{Non } A = \overline{A}$$

A	$\overline{A}$
0	1
1	0



non

# Opérateurs logiques de base

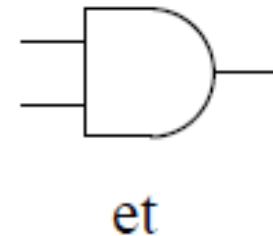
12

## ET ( AND )

Le ET est un opérateur binaire ( deux variables) , à pour rôle de réaliser le Produit logique entre deux variables booléennes.

- Le ET fait la conjonction entre deux variables.
- Le ET est défini par :  $F(A,B) = A \cdot B$

A	B	A . B
0	0	0
0	1	0
1	0	0
1	1	1



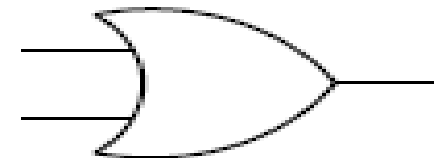
# Opérateurs logiques de base

13

## OU ( OR )

- Le OU est un opérateur binaire ( deux variables) , à pour rôle de réaliser la somme logique entre deux variables logiques.
- Le OU fait la disjonction entre deux variables.
- Le OU est défini par  $F(A,B) = A + B$  ( il ne faut pas confondre avec la somme arithmétique )

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1



ou

# Opérateurs logiques de base

14

NB:

- Dans la définition des opérateurs ET , OU , nous avons juste donner la définition de base avec deux variables logiques.
- L'opérateur ET peut réaliser le produit de plusieurs variables logique ( ex :  $A . B . C . D$  ).
- L'opérateur OU peut aussi réaliser la somme logique de plusieurs variables logiques ( ex :  $A + B + C + D$  ).
- Dans une expression on peut aussi utiliser les parenthèses.

# priorité des opérateurs

15

- Pour évaluer une expression logique ( fonction logique) :
  - ▣ on commence par évaluer les sous expressions entre les parenthèses.
  - ▣ puis le complément ( NON ) ,
  - ▣ en suite le produit logique ( ET )
  - ▣ enfin la somme logique ( OU)

## Exemple :

$$F(A,B,C) = (\overline{A \cdot B}) \cdot (C + B) + A \cdot \overline{B} \cdot C$$

si on veut calculer  $F(0,1,1)$  alors :

$$F(0,1,1) = (\overline{0 \cdot 1})(1 + 1) + 0 \cdot \overline{1} \cdot 1$$

$$F(0,1,1) = (\overline{0}) (1) + 0 \cdot 0 \cdot 1$$

$$F(0,1,1) = 1 \cdot 1 + 0 \cdot 0 \cdot 1$$

$$F(0,1,1) = 1 + 0$$

$$F(0,1,1) = 1$$

## Exercice :

Trouver la table de vérité de la fonction précédente ?

# priorité des opérateurs

16

Pour trouver la table de vérité , il faut trouver la valeur de la fonction F pour chaque combinaisons des trois variables A, B , C

3 variables  $\rightarrow 2^3 = 8$  combinaisons

$$F(A,B,C) = (\overline{A \cdot B}) \cdot (C + B) + A \cdot \overline{B} \cdot C$$

$$F(0,0,0) = (\overline{0 \cdot 0}) \cdot (0 + 0) + 0 \cdot \overline{0} \cdot 0 = 0$$

$$F(0,0,1) = (\overline{0 \cdot 0}) \cdot (1 + 0) + 0 \cdot \overline{0} \cdot 1 = 1$$

$$F(0,1,0) = (\overline{0 \cdot 1}) \cdot (0 + 1) + 0 \cdot \overline{1} \cdot 0 = 1$$

$$F(0,1,1) = (\overline{0 \cdot 1}) \cdot (1 + 1) + 0 \cdot \overline{1} \cdot 1 = 1$$

$$F(1,0,0) = (\overline{1 \cdot 0}) \cdot (0 + 0) + 1 \cdot \overline{0} \cdot 0 = 0$$

$$F(1,0,1) = (\overline{1 \cdot 0}) \cdot (1 + 0) + 1 \cdot \overline{0} \cdot 1 = 1$$

$$F(1,1,0) = (\overline{1 \cdot 1}) \cdot (0 + 1) + 1 \cdot \overline{1} \cdot 0 = 0$$

$$F(1,1,1) = (\overline{1 \cdot 1}) \cdot (1 + 1) + 1 \cdot \overline{1} \cdot 1 = 0$$

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0



# Lois fondamentales de l'Algèbre de Boole

17

## L'opérateur NON

$$\overline{\overline{A}} = A$$

$$\overline{A} + A = 1$$

$$\overline{A}.A = 0$$

## L'opérateur ET

$$(A.B).C = A.(B.C) = A.B.C \quad \text{Associativité}$$

$$A.B = B.A \quad \text{Commutativité}$$

$$A.A = A \quad \text{Idempotence}$$

$$A.1 = A \quad \text{Elément neutre}$$

$$A.0 = 0 \quad \text{Elément absorbant}$$

# Lois fondamentales de l'Algèbre de Boole

18

## L'opérateur OU

$(A + B) + C = A + (B + C) = A + B + C$	Associativité
$A + B = B + A$	Commutativité
$A + A = A$	Idempotence
$A + 0 = A$	Élément neutre
$A + 1 = 1$	Élément absorbant

## Distributivité

$A . (B + C) = (A . B) + (A . C)$  Distributivité du ET sur le OU

$A + (B . C) = (A + B).(A + C)$  Distributivité du OU sur le ET

# Lois fondamentales de l'Algèbre de Boole

19

## L'opérateur OU

### Autres relations utiles

$$A + (A \cdot B) = A$$

$$A \cdot (A + B) = A$$

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

$$A + \overline{A} \cdot B = A + B$$

# Dualité de l'algèbre de Boole

20

- Toute expression logique reste vraie si on remplace le ET par le OU , le OU par le ET , le 1 par 0 , le 0 par 1.
- Exemple :

$$A + 1 = 1 \rightarrow \overline{A} . 0 = 0$$

$$A + \overline{A} = 1 \rightarrow A . \overline{A} = 0$$

# Théorème de DE MORGANE

21

- La somme logique complimentée de deux variables est égale au produit des compléments des deux variables.

$$\overline{A + B} = \bar{A} . \bar{B}$$

- Le produit logique complimenté de deux variables est égale à la somme logique des compléments des deux variables.

$$\overline{A . B} = \bar{A} + \bar{B}$$

# Généralisation du Théorème de DE MORGANE à N variables

22

$$\overline{A.B.C.....} = \overline{A} + \overline{B} + \overline{C} + .....$$

$$\overline{A + B + C + .....} = \overline{A}.\overline{B}.\overline{C}.....$$

# Autres opérateurs logiques

23

## OU exclusif ( XOR)

$$F(A, B) = A \oplus B$$

$$A \oplus B = \overline{A}.B + A.\overline{B}$$

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

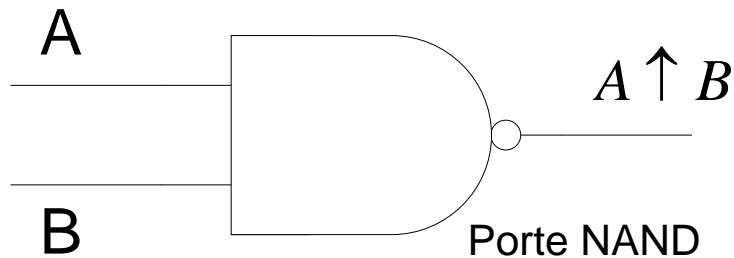
# Autres opérateurs logiques

24

## NAND ( NON ET )

$$F(A, B) = \overline{A \cdot B}$$

$$F(A, B) = A \uparrow B$$



A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0



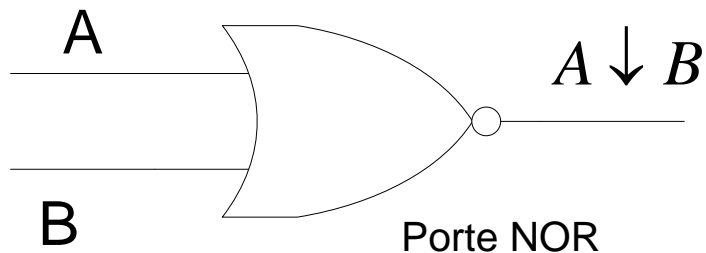
# Autres opérateurs logiques

25

## □ NOR ( NON OU )

$$F(A, B) = \overline{A + B}$$

$$F(A, B) = A \downarrow B$$



A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

# Combinaison des opérateurs

26

- En utilisant les NAND et les NOR on peut exprimer n'importe quelle expression ( fonction logique).
- Pour cela , Il suffit d'exprimer les opérateurs de base ( NON , ET , OU ) avec des NAND et des NOR.

# Réalisation des opérateurs de base avec des NOR

27

$$\overline{A} = \overline{A + A} = A \downarrow A$$

$$A + B = \overline{\overline{A + B}} = \overline{A \downarrow B} = (A \downarrow B) \downarrow (A \downarrow B)$$

$$A.B = \overline{\overline{A.B}} = \overline{\overline{A} + \overline{B}} = \overline{A} \downarrow \overline{B} = (A \downarrow A) \downarrow (B \downarrow B)$$

# Exercice

28

- Exprimer le NON , ET , OU en utilisant des NAND ?

# Propriétés des opérateurs NAND et NOR

29

$$A \uparrow 0 = 1$$

$$A \uparrow 1 = \overline{A}$$

$$A \uparrow B = B \uparrow A$$

$$(A \uparrow B) \uparrow C \neq A \uparrow (B \uparrow C)$$

$$A \downarrow 0 = \overline{A}$$

$$A \downarrow 1 = 0$$

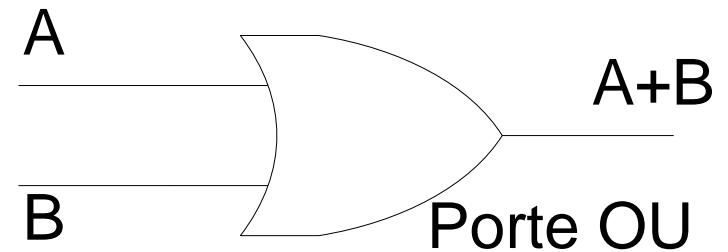
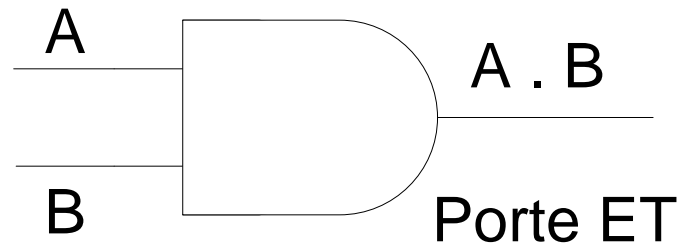
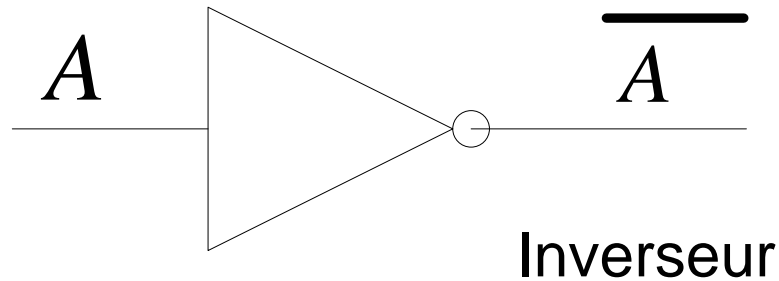
$$A \downarrow B = B \downarrow A$$

$$(A \downarrow B) \downarrow C \neq A \downarrow (B \downarrow C)$$

# Portes logiques

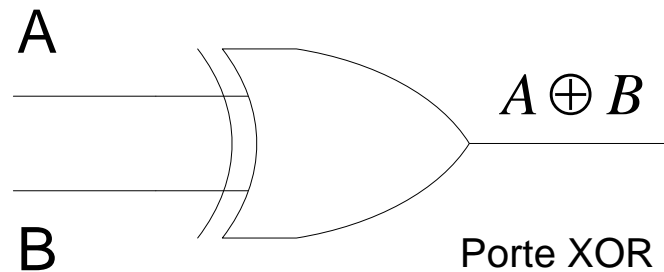
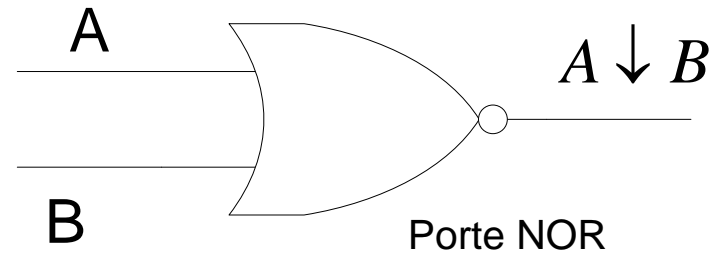
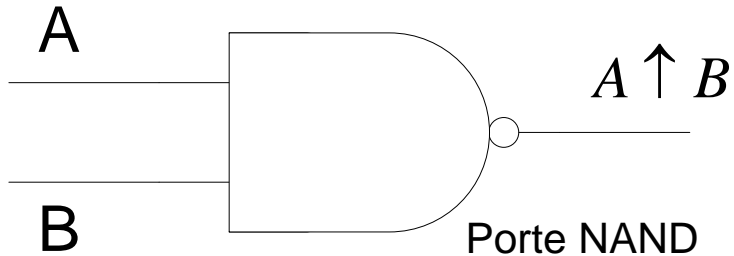
30

Une porte logique est un circuit électronique élémentaire qui permet de réaliser la fonction d'un opérateur logique de base .



# Portes logiques

31



Remarque :

- Les portes ET , OU , NAND , NOR peuvent avoir plus que deux entrées
- Il n'existe pas de OU exclusif à plus de deux entrées

# Schéma d'un circuit logique (Logigramme)

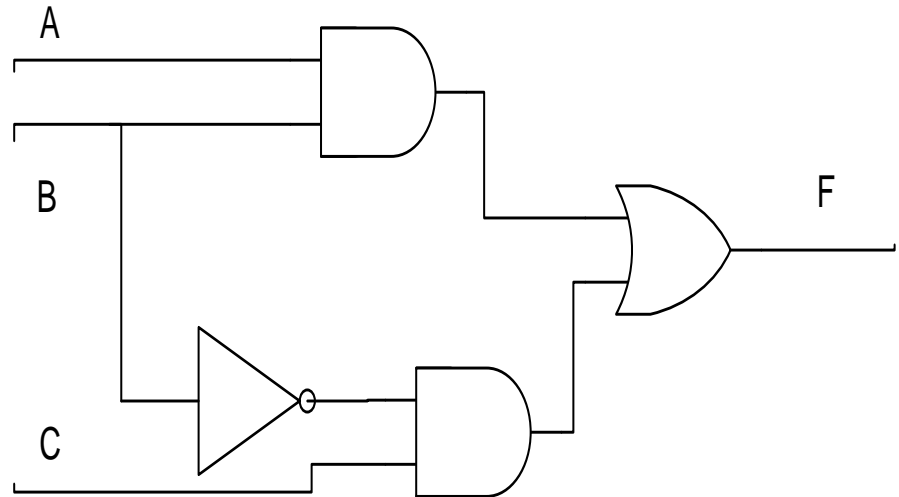
32

C'est la traduction de la fonction logique en un schéma électronique.

Le principe consiste à remplacer chaque opérateur logique par la porte logique qui lui correspond.

## Exemple 1

$$F(A, B, C) = A.B + \bar{B}.C$$



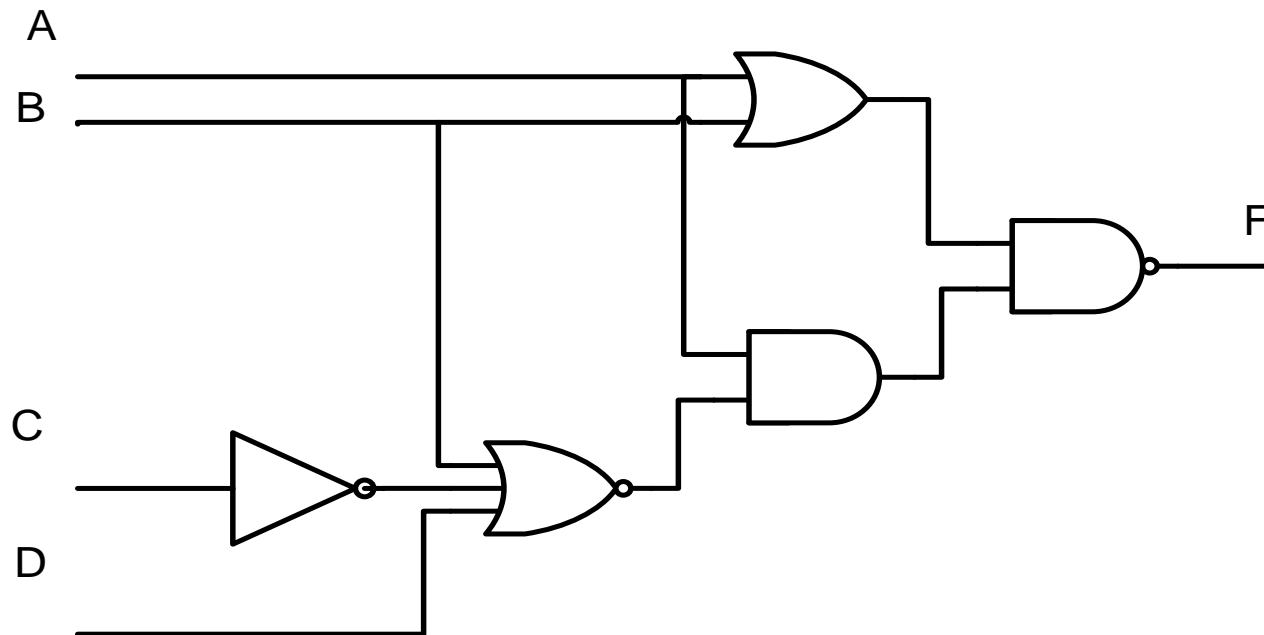


# Schéma d'un circuit logique (Logigramme)

33

## Exemple 2

$$F(A, B, C, D) = (A + B) . (\overline{B + \overline{C} + D}) . A$$



# Exercices

34

## □ Exercice 1

Donner le logigramme des fonctions suivantes :

$$F(A, B) = \overline{A}.B + A.\overline{B}$$

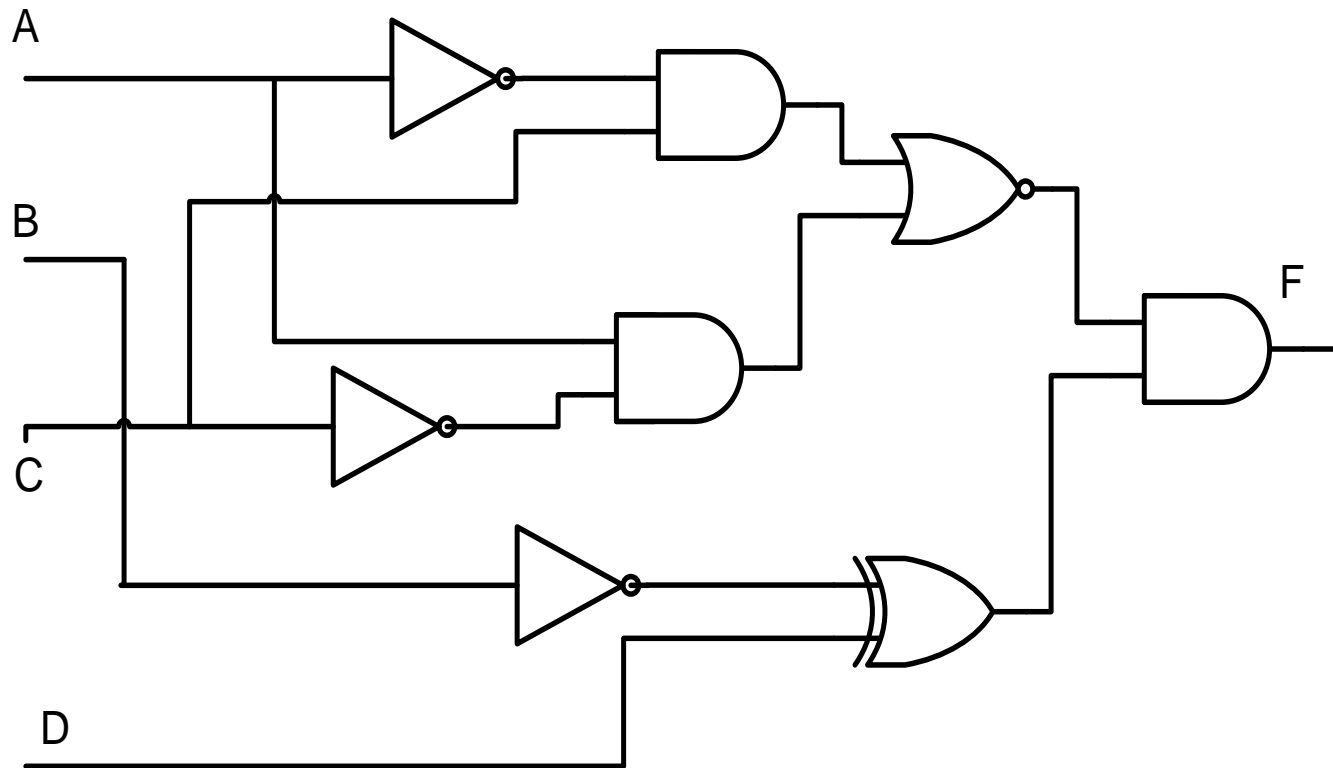
$$F(A, B, C) = (A + B).(\overline{A} + C).(B + \overline{C})$$

$$F(A, B, C) = (\overline{A}.B).(C + B) + A.\overline{B}.C$$

# Exercices

35

□ **Exercice 2 :** Donner l'équation de F ?





# Chapitre 2

37

- Définition textuelle d'une fonction logique,
- table de vérité,
- formes algébriques,
- simplification algébrique,
- table de Karnaugh

# Définition textuelle d'une fonction logique

38

- Généralement la définition du fonctionnement d'un système est donnée sous un format textuelle .
- Pour faire l'étude et la réalisation d'un tel système on doit avoir son modèle mathématique (fonction logique).
- Donc il faut tirer ( déduire ) la fonction logique a partir de la description textuelle.

# définition textuelle du fonctionnement d'un système

39

- Une serrure de sécurité s'ouvre en fonction de trois clés. Le fonctionnement de la serrure est définie comme suit :
  - ▣ La serrure est ouverte si au moins deux clés sont utilisées.
  - ▣ La serrure reste fermée dans les autres cas .
- Donner le schéma du circuit qui permet de contrôler l'ouverture de la serrure ?

# Étapes de conception et de réalisation d'un circuit numérique

40

Pour faire l'étude et la réalisation d'un circuit il faut suivre les étapes suivantes :

1. Il faut bien comprendre le fonctionnement du système.
2. Il faut définir les variables d'entrée.
3. Il faut définir les variables de sortie.
4. Etablir la table de vérité.
5. Ecrire les équations algébriques des sorties ( à partir de la table de vérité ).
6. Effectuer des simplifications ( algébrique ou par Karnaugh).
7. Faire le schéma avec un minimum de portes logiques.



# Étapes de conception et de réalisation d'un circuit numérique

41

- Si on reprend l'exemple de la serrure :
  - ▣ Le système possède trois entrées : chaque entrée représente une clé.
  - ▣ On va correspondre à chaque clé une variable logique: clé 1  $\rightarrow$  A , la clé 2  $\rightarrow$  B , la clé 3  $\rightarrow$  C
    - Si la clé 1 est utilisée alors la variable  $A=1$  sinon  $A=0$
    - Si la clé 2 est utilisée alors la variable  $B=1$  sinon  $B=0$
    - Si la clé 3 est utilisée alors la variable  $C=1$  sinon  $C=0$
  - ▣ Le système possède une seule sortie qui correspond à l'état de la serrure ( ouverte ou fermé ).
  - ▣ On va correspondre une variable S pour designer la sortie :
    - $S=1$  si la serrure est ouverte ,
    - $S=0$  si elle est fermée

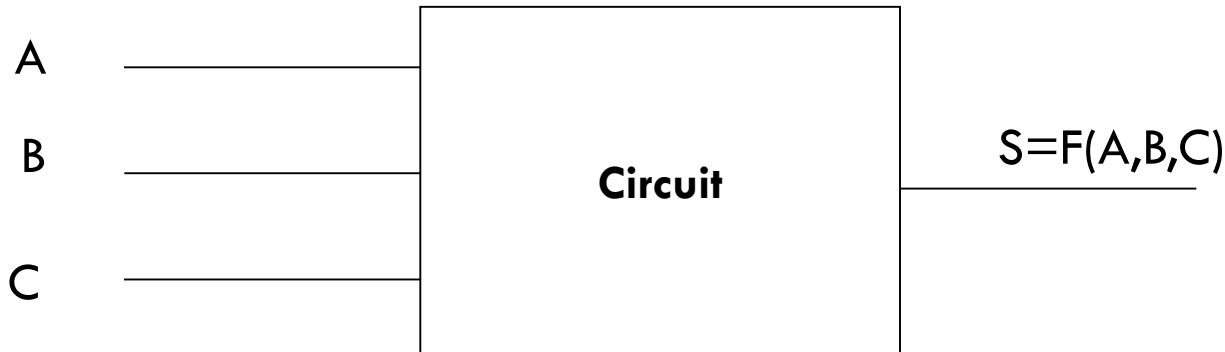
# Étapes de conception et de réalisation d'un circuit numérique

42

$$S = F(A, B, C)$$

$F(A, B, C) = 1$  si au moins deux clés sont introduites

$F(A, B, C) = 0$  si non .



## Remarque :

Il est important de préciser aussi le niveau logique avec lequel on travaille ( logique positive ou négative ).

# Table de vérité

43

- Si une fonction logique possède  $N$  variables logiques  $\rightarrow$  ? combinaisons  $\rightarrow$  la fonction possède ? valeurs.
- Les  $2^n$  combinaisons sont représentées dans une table qui s'appelle table de vérité.

# Table de vérité

44

A	B	C	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

→  $A + B + C$  : max terme

→  $A + B + \bar{C}$  : max terme

→  $A + \bar{B} + C$  : max terme

→  $\bar{A} . B . C$  : min terme

→  $\bar{A} + B + C$  : max terme

→  $A . \bar{B} . C$  : min terme

→  $A . B . \bar{C}$  : min terme

→  $A . B . C$  : min terme

# Extraction de la fonction logique à partir de la T.V

45

F = somme min termes

$$F(A, B, C) = \bar{A} . B . C + A . \bar{B} . C + A . B . \bar{C} + A . B . C$$

F = produit des max termes

$$F(A, B, C) = (A + B + C) (A + B + \bar{C})(A + \bar{B} + C) (\bar{A} + B + C)$$

# Forme canonique d'une fonction logique

46

- On appelle forme canonique d'une fonction la forme où chaque terme de la fonction comportent toutes les variables.
- Exemple :

$$F(A, B, C) = AB\bar{C} + A\bar{C}B + \bar{A}BC$$

Il existent plusieurs formes canoniques : les plus utilisées sont la première et la deuxième forme .

# Première forme canonique

47

- Première forme canonique (forme disjonctive):  
somme de produits
- C'est la somme des min termes.
- Une disjonction de conjonctions.
- Exemple :

$$F(A, B, C) = \bar{A} . B . C + A . \bar{B} . C + A . B . \bar{C} + A . B . C$$

Cette forme est la forme la plus utilisée.

# Deuxième forme canonique

48

- Deuxième forme canonique (conjonctive):  
produit de sommes
- Le produit des max termes
- Conjonction de disjonctions
- Exemple :

$$F(A, B, C) = (A + B + C) (A + B + \bar{C})(A + \bar{B} + C) (\bar{A} + B + C)$$

La première et la deuxième forme canonique sont équivalentes .



## □ Remarque 1

- On peut toujours ramener n'importe quelle fonction logique à l'une des formes canoniques.
- Cela revient à rajouter les variables manquants dans les termes qui ne contiennent pas toutes les variables ( les termes non canoniques ).
- Cela est possible en utilisant les règles de l'algèbre de Boole :
  - ▣ Multiplier un terme avec une expression qui vaut 1
  - ▣ Additionner à un terme avec une expression qui vaut 0
  - ▣ Par la suite faire la distribution

## □ Example

$$1. F(A, B) = A + B$$

$$= A (B + \bar{B}) + B (A + \bar{A})$$

$$= AB + A\bar{B} + AB + \bar{A}B$$

$$= AB + A\bar{B} + \bar{A}B$$

$$2. F(A, B, C) = AB + C$$

$$= AB(C + \bar{C}) + C(A + \bar{A})$$

$$= ABC + AB\bar{C} + AC + \bar{A}C$$

$$= ABC + AB\bar{C} + AC(B + \bar{B}) + \bar{A}C(B + \bar{B})$$

$$= ABC + AB\bar{C} + ABC + A\bar{B}C + \bar{A}BC + \bar{A}\bar{B}C$$

$$= ABC + AB\bar{C} + A\bar{B}C + \bar{A}B\bar{C} + \bar{A}\bar{B}C$$

## □ Remarque 2

- Il existe une autre représentation des formes canoniques d'une fonction , cette représentation est appelée forme numérique.
- R : pour indiquer la forme disjonctive
- P : pour indiquer la forme conjonctive.

**Exemple :** si on prend une fonction avec 3 variables

$$R(2,4,6) = \sum (2,4,6) = R(010,100,110) = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$$

$$\begin{aligned} P(0,1,3,5,7) &= \prod (0,1,3,5,7) = P(000,001,011,101,111) \\ &= (A + B + C)(A + B + \bar{C})(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{C})(\bar{A} + \bar{B} + \bar{C}) \end{aligned}$$

### □ Remarque 3 : déterminer F

A	B	C	F	Non(F)
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

$$\bar{F} = \bar{A}.\bar{B}.\bar{C} + \bar{A}.\bar{B}.C + \bar{A}.B.\bar{C} + A.\bar{B}.\bar{C}$$

# Exercices

53

## □ Exercice 1

Déterminer la première , la deuxième forme canonique et la fonction inverse à partir de la Table de Vérité suivante ?  
Tracer le logigramme de la fonction ?

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

# Exercices

54

## □ Exercice 2

Faire le même travail avec la Table de Vérité suivante :

A	B	C	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

# Exercices

55

## □ Exercice 3

Un jury composé de 4 membres pose une question à un joueur, qui à son tour donne une réponse. Chaque membre du jury positionne son interrupteur à " 1 " lorsqu'il estime que la réponse donnée par le joueur est juste (avis favorable ) et à " 0 " dans le cas contraire (avis défavorable ). On traite la réponse de telle façon à positionner :

- Une variable succès ( $S=1$ ) lorsque la décision de la majorité des membres de jury est favorable,
- une variable Échec ( $E=1$ ) lorsque la décision de la majorité des membres de jury est défavorable
- et une variable Égalité ( $N=1$ ) lorsqu'il y a autant d'avis favorables que d'avis défavorables.

## Question :

- a./ Déduire une table de vérité pour le problème,
- b./ Donner les équations de  $S$ ,  $E$ ,
- c./ En déduire l'équation de  $N$ ,

# Simplification des fonctions logiques

56

- L'objectif de la simplification des fonctions logiques est de :
  - ▣ réduire le nombre de termes dans une fonction
  - ▣ et de réduire le nombre de variables dans un terme
- Cela afin de réduire le nombre de portes logiques utilisées → réduire le coût du circuit
- Plusieurs méthodes existent pour la simplification :
  - ▣ La Méthode algébrique
  - ▣ Les Méthodes graphiques : ( ex : table de karnaugh )
  - ▣ Les méthodes programmables



# Méthode algébrique

57

- Le principe consiste à appliquer les règles de l'algèbre de Boole afin d'éliminer des variables ou des termes.
- Mais il n'y a pas une démarche bien spécifique.
- Voici quelques règles les plus utilisées :

$$A \cdot B + \overline{A} \cdot B = B$$

$$A + A \cdot B = A$$

$$A + \overline{A} \cdot B = A + B$$

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

$$A \cdot (A + B) = A$$

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

# Règles de simplification de la méthode algébrique

58

- Règles 1 : regrouper des termes à l'aide des règles précédentes
- Exemple

$$\begin{aligned}ABC + AB\bar{C} + A\bar{B}CD &= AB(C + \bar{C}) + A\bar{B}CD \\&= AB + A\bar{B}CD \\&= A(B + \bar{B}(CD)) \\&= A(B + CD) \\&= AB + ACD\end{aligned}$$

# Règles de simplification de la méthode algébrique

59

- Règles 2 : Rajouter un terme déjà existant à une expression
- Exemple :

$$A B C + \bar{A} B C + A \bar{B} C + A B \bar{C} =$$

$$A B C + \bar{A} B C + A B C + A \bar{B} C + A B C + A B \bar{C} =$$
$$B C + \quad \quad \quad A C \quad \quad + \quad A B$$

# Règles de simplification de la méthode algébrique

60

- Règles 3 : il est possible de supprimer un terme superflu ( un terme en plus ), c'est-à-dire déjà inclus dans la réunion des autres termes.
- Exemple 1 :

$$\begin{aligned} F(A, B, C) &= A B + \overline{B} C + A C = A B + \overline{B} C + A C ( B + \overline{B} ) \\ &= A B + \overline{B} C + A C B + A \overline{B} C \\ &= A B ( 1 + C ) + \overline{B} C ( 1 + A ) \\ &= A B + \overline{B} C \end{aligned}$$

# Règles de simplification de la méthode algébrique

61

- **Exemple 2** : il existe aussi la forme conjonctive du terme superflu

$$\begin{aligned} F(A, B, C) &= (A + B) \cdot (\overline{B} + C) \cdot (A + C) \\ &= (A + B) \cdot (\overline{B} + C) \cdot (A + C + B \cdot \overline{B}) \\ &= (A + B) \cdot (\overline{B} + C) \cdot (A + C + B) \cdot (A + C + \overline{B}) \\ &= (A + B) \cdot (A + C + B) \cdot (\overline{B} + C) \cdot (A + C + \overline{B}) \\ &= (A + B) \cdot (\overline{B} + C) \end{aligned}$$

# Règles de simplification de la méthode algébrique

62

- Règles 4 : il est préférable de simplifier la forme canonique ayant le nombre de termes minimum.
- Exemple :

$$F(A, B, C) = R(2, 3, 4, 5, 6, 7)$$

$$\begin{aligned}\overline{F(A, B, C)} &= R(0, 1) = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C \\ &= \overline{A} \cdot \overline{B} (\overline{C} + C) \\ &= \overline{A} \cdot \overline{B} = \overline{\overline{A} + B}\end{aligned}$$

$$F(A, B, C) = \overline{\overline{\overline{F(A, B, C)}}} = \overline{\overline{\overline{A} + B}} = A + B$$

# Exercice

63

Démontrer la proposition suivante :

$$A.B + B.C + A.C + A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + \overline{A}.\overline{B}.C = A + B + C$$

Donner la forme simplifiée de la fonction suivante :

$$F(A, B, C, D) = \overline{A}BCD + A\overline{B}CD + AB\overline{C}D + ABC\overline{D} + ABCD$$

# Simplification par la table de Karnaugh

64

## □ Les termes adjacents

Examinons l'expression suivante :

$$A.B + A.\overline{B}$$

- Les deux termes possèdent les même variables. La seule différence est l'état de la variable B qui change.
- Si on applique les règles de simplification on obtient :

$$AB + A\overline{B} = A(B + \overline{B}) = A$$

- Ces termes sont dites adjacents.



# Simplification par la table de Karnaugh

65

## □ Exemple de termes adjacents

Ces termes sont adjacents

$$A.B + \overline{A}.B = B$$

$$A.B.C + A.\overline{B}.C = A.C$$

$$A.B.C.D + A.B.\overline{C}.D = A.B.D$$

Ces termes ne sont pas adjacents

$$A.B + \overline{A}.\overline{B}$$

$$A.B.C + A.\overline{B}.\overline{C}$$

$$A.B.C.D + \overline{A}.\overline{B}.\overline{C}.D$$

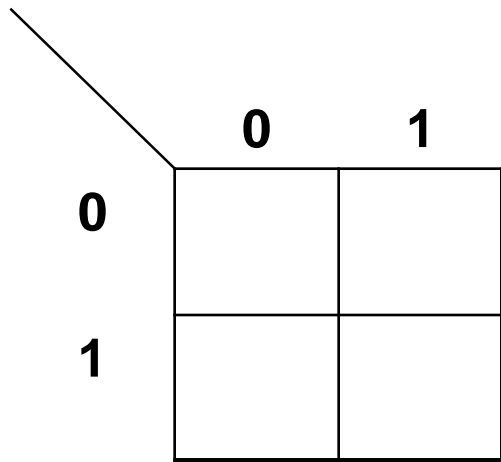
# Description de la table de karnaugh

66

- La méthode de Karnaugh se base sur la règle précédente.
- La méthode consiste à mettre en évidence par une méthode graphique (un tableaux ) tous les termes qui sont adjacents (qui ne diffèrent que par l'état d'une seule variable).
- La méthode peut s'appliquer aux fonctions logiques de 2,3,4,5 et 6 variables.
- Un tableau de Karnaugh comportent  $2^n$  cases ( N est le nombre de variables ).

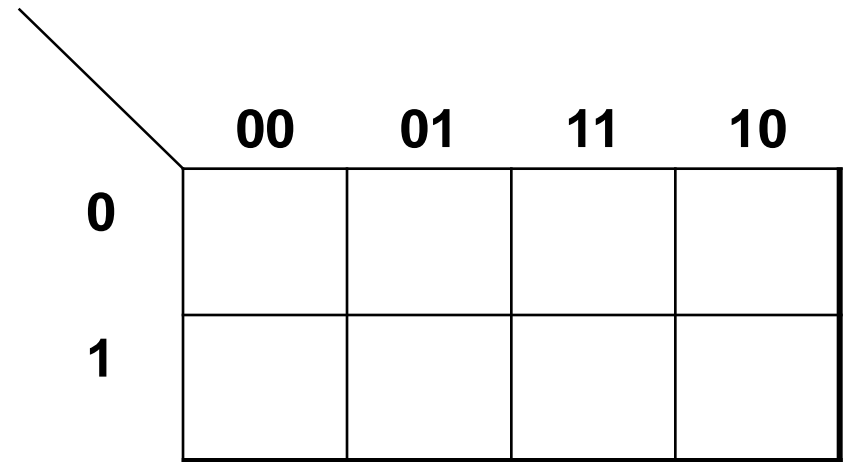
# Description de la table de karnaugh

67



	0	1
0		
1		

**Tableau à 2 variables**

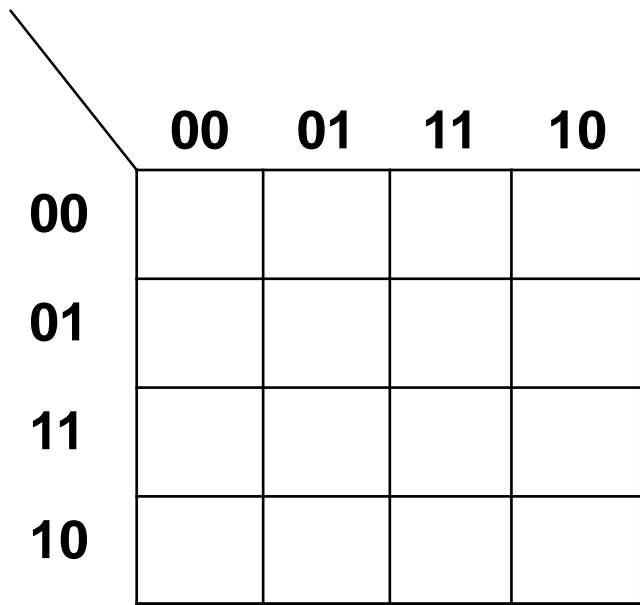


	00	01	11	10
0				
1				

**Tableaux à 3 variables**

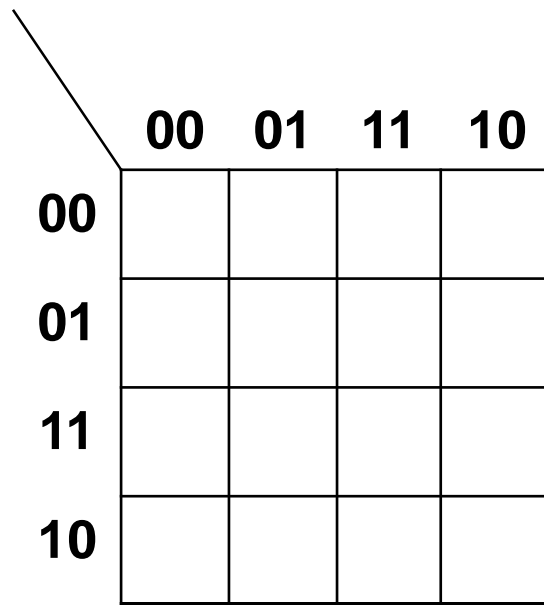
# Description de la table de karnaugh

68



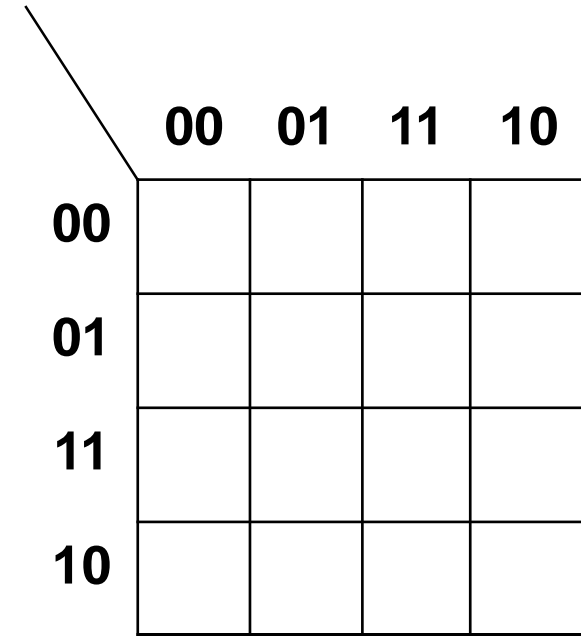
	00	01	11	10
00				
01				
11				
10				

**Tableau à 4 variables**



	00	01	11	10
00				
01				
11				
10				

**U = 0**



	00	01	11	10
00				
01				
11				
10				

**U = 1**

**Tableau à 5 variables**

- Dans un tableau de karnaugh , chaque case possède un certain nombre de cases adjacentes.

	00	01	11	10
0				
1				

Les trois cases bleues sont des cases adjacentes à la case rouge

	00	01	11	10
00				
01				
11				
10				

# Passage de la table de vérité à la table de Karnaugh

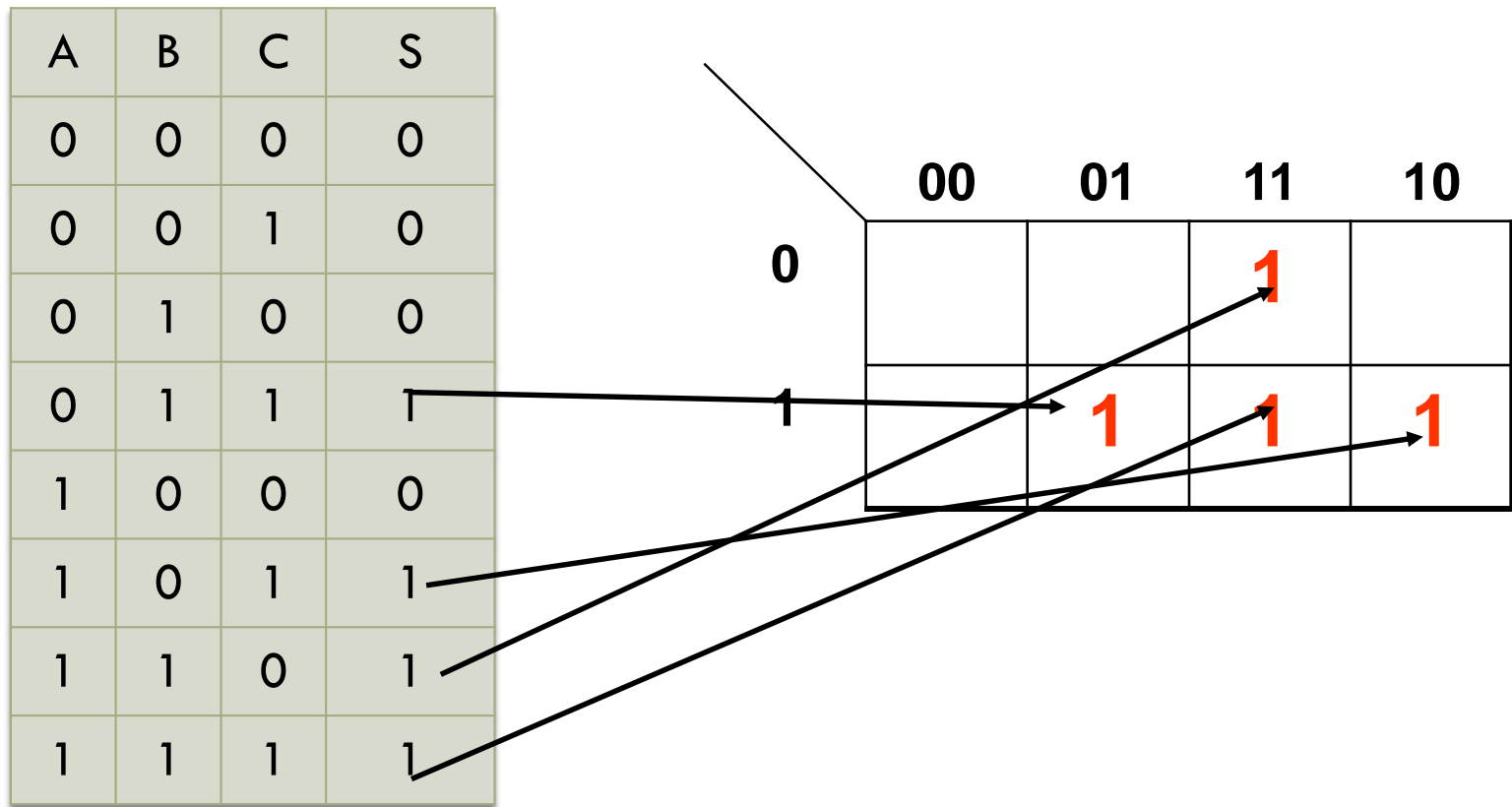
70

- Pour chaque combinaisons qui représente un min terme lui correspond une case dans le tableau qui doit être mise à 1
- Pour chaque combinaisons qui représente un max terme lui correspond une case dans le tableau qui doit être mise à 0 .
- Lorsque on remplis le tableau , on doit soit prendre les min terme ou les max terme

# Passage de la table de vérité à la table de Karnaugh

71

□ Exemple :



# Passage de la table de vérité à la table de Karnaugh

72

- Si la fonction logique est donnée sous la première forme canonique ( disjonctive), alors sa représentation est directe : pour chaque terme lui correspond une seule case qui doit être mise à 1.
- Si la fonction logique est donnée sous la deuxième forme canonique ( conjonctive), alors sa représentation est directe : pour chaque terme lui correspond une seule case qui doit être mise à 0 .



# Passage de la table de vérité à la table de Karnaugh

73

## □ Exemple

$$F1(A, B, C) = \sum (1, 2, 5, 7)$$

	00	01	11	10
0		1		
1	1		1	1

$$F2(A, B, C) = \prod (0, 2, 3, 6)$$

	00	01	11	10
0	0	0		0
1		0		

# Méthode de simplification (Exemple : 3 variables )

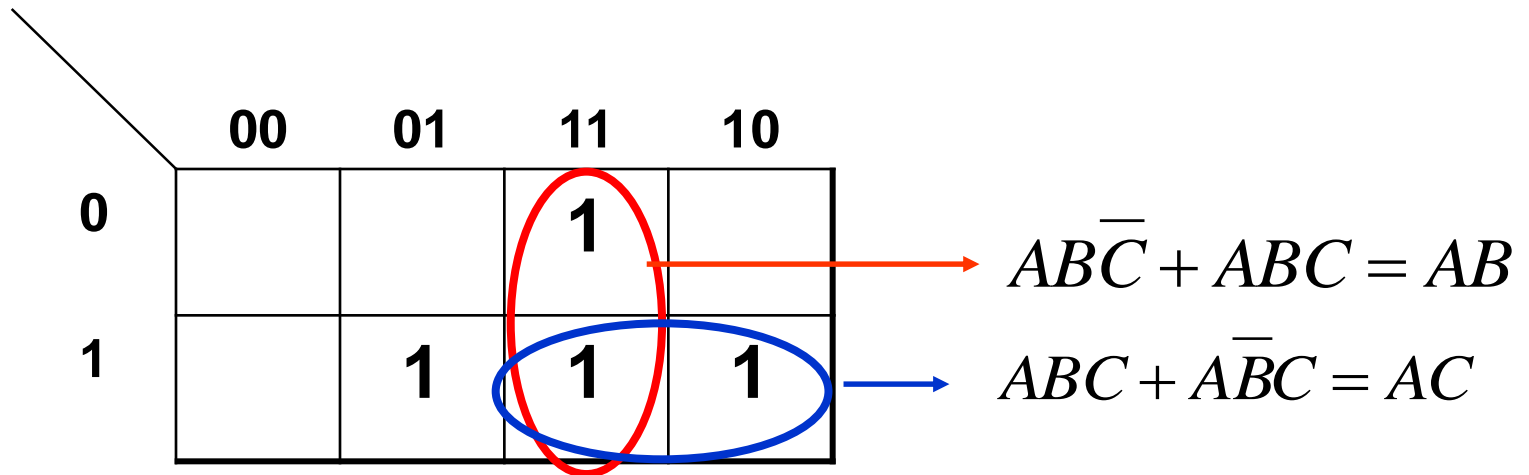
74

- L'idée de base est d'essayer de regrouper (faire des regroupements ) les cases adjacentes qui comportent des 1 ( rassembler les termes adjacents ).
- Essayer de faire des regroupements avec le maximum de cases ( 16,8,4 ou 2 )

# Méthode de simplification (Exemple : 3 variables )

75

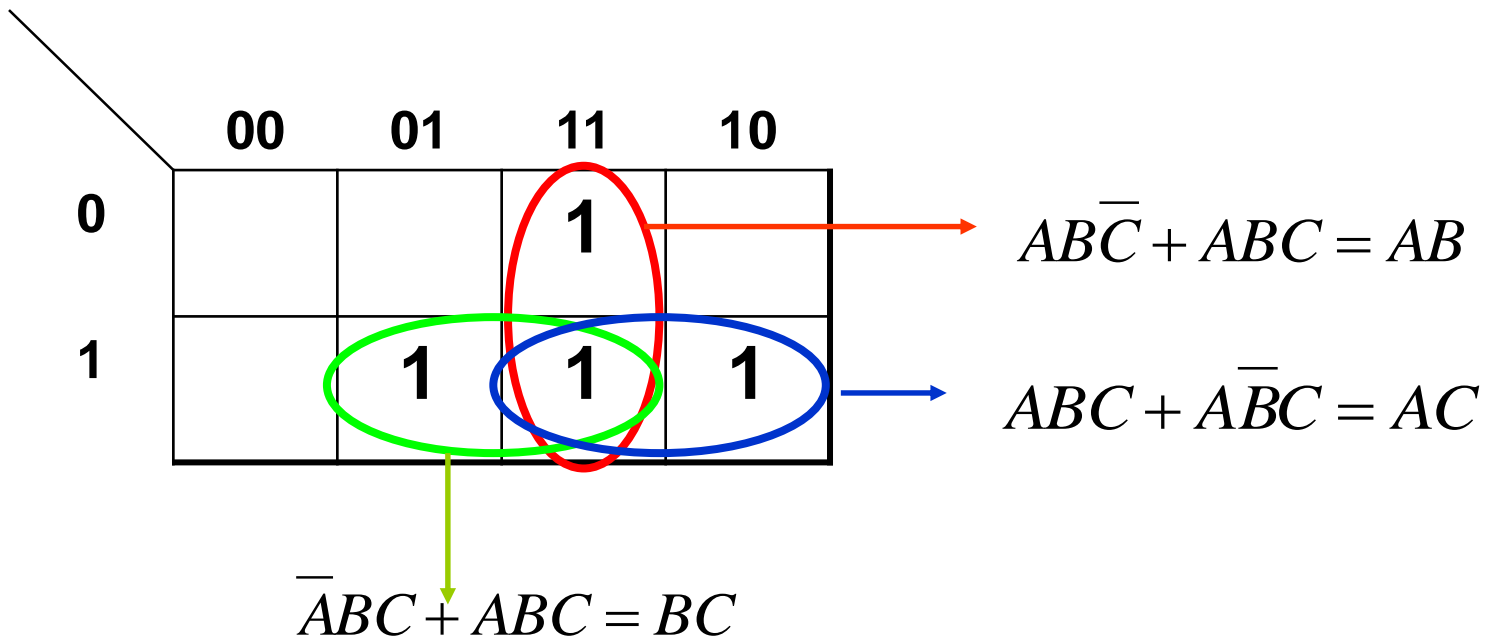
- Puisque il existent encore des cases qui sont en dehors d'un regroupement on refait la même procédure : former des regroupements.
- Une case peut appartenir à plusieurs regroupements



# Méthode de simplification (Exemple : 3 variables )

76

- On s'arrête lorsque il y a plus de 1 en dehors des regroupements
- La fonction final est égale à la réunion ( somme ) des termes après simplification.



$$F(A, B, C) = AB + AC + BC$$

# Méthode de simplification (Exemple : 3 variables )

77

Donc, en résumé pour simplifier une fonction par la table de karnaugh il faut suivre les étapes suivantes :

1. Remplir le tableau à partir de la table de vérité ou à partir de la forme canonique.
2. Faire des regroupements : des regroupements de 16,8,4,2,1 cases ( Les même termes peuvent participer à plusieurs regroupements ) .
3. Dans un regroupement :
  - Qui contient un seule terme on peut pas éliminer de variables.
  - Qui contient deux termes on peut éliminer une variable ( celle qui change d'état ).
  - Qui contient 4 termes on peut éliminer 2 variables.
  - Qui contient 8 termes on peut éliminer 3 variables.
  - Qui contient 16 termes on peut éliminer 4 variables.
5. L'expression logique finale est la réunion ( la somme ) des groupements après simplification et élimination des variables qui changent d'état.

# Méthode de simplification (Exemple : 3 variables )

78

## □ Exemple 1 : 3 variables

	00	01	11	10
0			1	
1	1	1	1	1

$$F(A, B, C) = C + AB$$

# Méthode de simplification (Exemple : 3 variables )

79

## Exemple 2 : 4 variables

	00	01	11	10
00				1
01	1	1	1	1
11				
10		1		

$$F(A, B, C, D) = \overline{C}.D + A.\overline{B}.\overline{C} + \overline{A}.B.C.\overline{D}$$

# Méthode de simplification (Exemple : 3 variables )

80

## □ Exemple 3 : 4 variables

	00	01	11	10
00	1			1
01		1	1	1
11				1
10	1			1

$$F(A, B, C, D) = \overline{A}\overline{B} + \overline{B}\overline{D} + \overline{B}CD$$



# Méthode de simplification (Exemple : 3 variables )

81

## Exemple 4 : 5 variables

	00	01	11	10
00	1			
01	1		1	
11	1		1	
10	1			

U = 0

	00	01	11	10
00	1			
01	1			1
11	1			1
10	1	1		

U = 1

$$F(A, B, C, D, U) = \bar{A} \bar{B} + A.B.D.\bar{U} + \bar{A}.C.\bar{D}.U + A.\bar{B}.D.U$$

# Exercices

82

- Trouver la forme simplifiée des fonctions à partir des deux tableaux ?

	00	01	11	10
0		1	1	1
1	1		1	1

	00	01	11	10
00	1		1	1
01				
11				
10	1	1	1	1

# Cas d'une fonction non totalement définie

83

Examinons l'exemple suivant :

- Une serrure de sécurité s'ouvre en fonction de quatre clés A, B, C D. Le fonctionnement de la serrure est définie comme suite :
  - $S(A,B,C,D) = 1$  si au moins deux clés sont utilisées
  - $S(A,B,C,D) = 0$  sinon
- Les clés A et D ne peuvent pas être utilisées en même temps.
- On remarque que si la clé A et D sont utilisées en même temps l'état du système n'est pas déterminé.
- Ces cas sont appelés cas impossibles ou interdites → comment représenter ces cas dans la table de vérité ?.

# Cas d'une fonction non totalement définie

84

Pour les cas impossibles ou interdites

- il faut mettre un **X** dans la T.V .

Les cas impossibles sont représentées

- aussi par des **X** dans la table de karnaugh

	00	01	11	10
00			1	
01		1	X	X
11	1	1	X	X
10		1	1	1

A	B	C	D	S
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	X
1	0	1	0	1
1	0	1	1	X
1	1	0	0	1
1	1	0	1	X
1	1	1	0	1
1	1	1	1	X

# Cas d'une fonction non totalement définie

85

- Il est possible d'utiliser les **X** dans des regroupements :
  - ▣ Soit les prendre comme étant des **1**
  - ▣ Ou les prendre comme étant des **0**
- Il ne faut pas former des regroupement qui contient uniquement des **X**

	00	01	11	10
00			1	
01		1	X	X
11	1	1	X	X
10		1	1	1
	AB			

# Cas d'une fonction non totalement définie

86

	00	01	11	10
00			1	
01		1	X	X
11	1	1	X	X
10		1	1	1

$$AB + CD$$

	00	01	11	10
00			1	
01		1	X	X
11	1	1	X	X
10		1	1	1

$$AB + CD + BD$$

	00	01	11	10
00			1	
01		1	X	X
11	1	1	X	X
10		1	1	1

$$AB + CD + BD + AC$$



	00	01	11	10
00			1	
01		1	X	X
11	1	1	X	X
10		1	1	1

$$AB + CD + BD + AC + BC$$

# Exercices

90

## □ Exercice 1

Trouver la fonction logique simplifiée à partir de la table suivante ?

	00	01	11	10
00		1	X	
01	1	X		1
11	1		X	1
10	X		1	X

# Exercices

91

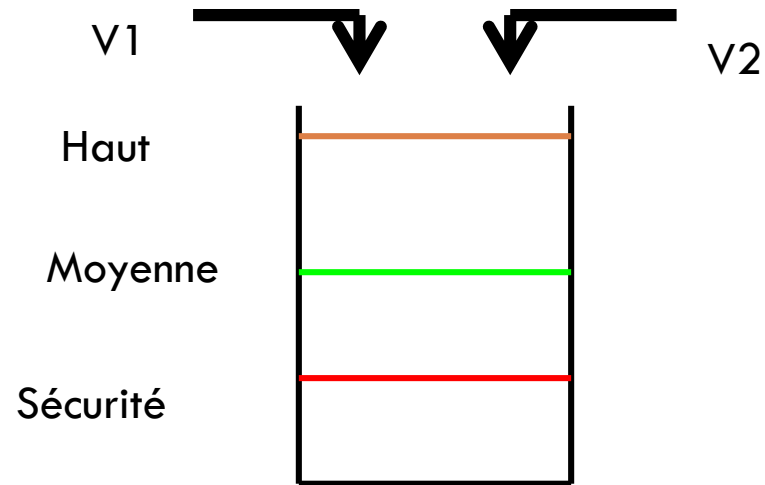
La figure 1 représente un réservoir alimenté par deux vannes V1 et V2. On distingue trois niveaux : Sécurité, Moyen, Haut:

- lorsque le niveau de liquide est inférieur ou égale à Sécurité, V1 et V2 sont ouvertes.
- lorsque le niveau du liquide est inférieur ou égal à Moyen mais supérieur à Sécurité, seule V1 est ouverte.
- lorsque le niveau du liquide est supérieur à Moyen mais inférieur à Haut, seule V2 est ouverte.
- lorsque le niveau de liquide a atteint le niveau Haut, les deux vannes sont fermées.

Question: Donner les équations logiques de l'ouverture de V1 et V2 en fonction du niveau de liquide.

# Exercices

92





# Troisième Partie

94

## Plan

### Conversion de Bases

Définition des différentes bases

Conversion vers la base 10

Conversions de la base 10 vers d'autres bases

# Codage des informations

95

La numération arabe utilise une représentation positionnelle contrairement à la numération romaine : le rang de chaque chiffre indique son poids.

Numération romaine : MCMXCIX où M vaut toujours 1000, C vaut toujours 100, etc.

Numération positionnelle : 1999 où le « 9 » le plus à droite vaut « 9 », celui immédiatement à sa gauche vaut « 90 », etc. La valeur d'un chiffre dépend de sa position.

$XXX_b$  indique que le nombre XXX est écrit en base  $b$ .

Exemples :

- $10^3 10^2 10^1 10^0$
- $5\ 9\ 3\ 1_{10} = 5*1000+9*100+3*10+1*1$
- $2^3 2^2 2^1 2^0$
- $1\ 0\ 0\ 1_2 = 1*8+0*4+0*2+1*1 = 9_{10}$
- $5^2 5^1 5^0$
- $1\ 4\ 3_5 = 1*25+4*5+3 = 48_{10}$

# Codage des informations

96

- Bases usuelles : base 10, 8.
- Représentation des nombres dans une base  $b$  :
  1. Si  $b \leq 10$ , on utilise simplement les chiffres de 0 à  $b-1$  (exemple : base 2).
  2. Si  $b > 10$ , on utilise les chiffres de 0 à 9 et des lettres.

Ainsi, en base 16 (numération hexadécimale) on utilise les chiffres de 0 à 9 et les lettres de A (=10) à F (=15).

$$E2A1_{16} = 14*16^3 + 2*16^2 + 10*16 + 1 = 58017_{10}$$

Exemple : l'adresse mémoire.



# Opérations arithmétiques

97

Les principes sont les mêmes que ceux utilisés en bc

Exemples de calculs en base 2 ( $172 + 101 = 273$ )

– Tables d'addition et de multiplication en base 2.

$$\begin{array}{r} 10101100 \\ + 1100101 \\ \hline = 100010001 \end{array}$$

		101
x		101
<hr/>		
		101
	000	
	<hr/>	
	101	
	<hr/>	
	11001	

Dans le système de calcul binaire la soustraction ressemble en quelque sorte à l'addition, sauf que lorsque l'on soustrait un bit égal à un d'un bit égal à zéro, on obtient une retenue pour le bit de poids plus élevé.

Principe de base :

$$\begin{array}{ll} 0 - 0 = 0 & 0 - 1 = 1 \text{ (avec retenue)} \\ 1 - 0 = 1 & 1 - 1 = 0 \end{array}$$

Exemple de soustraction :

$$\begin{array}{r} \phantom{1}r \phantom{1}r \phantom{1}r \phantom{1}r \\ 1101110 \\ - 10111 \\ \hline 1010111 \end{array}$$

# Les conversions

98

**Conversion** Binaire décimal:

**Passage d'une base quelconque à la base 10**

Il suffit d'écrire le nombre comme ci-dessus et d'effectuer les opérations en décimal. (x 10)

Exemple en hexadécimal :

$$(AB)_{16} = 10 * 16^1 + 11 * 16^0 = 160 + 11 = (171)_{10}$$

(En base 16, A représente 10, B  $\rightarrow$  11, et F  $\rightarrow$  15).

# Les conversions

99

Passage de la base 10 vers une base quelconque

Nombres entiers : On procède par divisions successives. On divise le nombre par la base, puis le quotient obtenu par la base, et ainsi de suite jusqu'à l'obtention d'un quotient nul.

La suite des restes obtenus correspond aux chiffres dans la base visée,  $a_0 a_1 \dots a_n$ .

Exemple : soit à convertir  $(44)_{10}$  vers la base 2.

$$44 = 22 \times 2 + 0 \Rightarrow a_0 = 0$$

$$22 = 11 \times 2 + 0 \Rightarrow a_1 = 0$$

$$11 = 2 \times 5 + 1 \Rightarrow a_2 = 1$$

$$5 = 2 \times 2 + 1 \Rightarrow a_3 = 1$$

$$2 = 1 \times 2 + 0 \Rightarrow a_4 = 0$$

$$1 = 0 \times 2 + 1 \Rightarrow a_5 = 1$$

$$\text{Donc } (44)_{10} = (101100)_2.$$

# Les conversions

100

**Nombres fractionnaires :** On multiplie la partie fractionnaire par la base en répétant l'opération sur la partie fractionnaire du produit jusqu'à ce qu'elle soit nulle (ou que la précision voulue soit atteinte).

Pour la partie entière, on procède par divisions comme pour un entier.

□ Exemple : conversion de  $(54, 25)_{10}$  en base 2

Partie entière :  $(54)_{10} = (110110)_2$  par divisions.

Partie fractionnaire :

- $0,25 \times 2 = 0,50 = 0 + 0,50 \Rightarrow a_{-1} = 0$
- $0,50 \times 2 = 1,00 = 1 + 0,00 \Rightarrow a_{-2} = 1$
- $0,00 \times 2 = 0,00 = 0 + 0,00 \Rightarrow a_{-3} = 0$

# Les conversions

101

## Cas des bases 2, 8 et 16

Ces bases correspondent à des puissances de 2 ( $2^1$ ,  $2^3$  et  $2^4$ ), d'où des passages de l'une à l'autre très simples. Les bases 8 et 16 sont pour cela très utilisées en informatique, elles permettent de représenter rapidement et de manière compacte des configurations binaires.

La base 8 est appelée notation octale, et la base 16 notation hexadécimale.

Chaque chiffre en base 16 ( $2^4$ ) représente un paquet de 4 bits consécutifs. Par exemple :

$$(10011011)_2 = (1001\ 1011)_2 = (9B)_{16}$$

De même, chaque chiffre octal représente 3 bits.

On manipule souvent des nombres formés de 8 bits, nommés octets, qui sont donc notés sur 2 chiffres hexadécimaux.

# Les conversions

102

Exemples :

Décimal  $\rightarrow$  binaire : on procède par division entière successive par 2. Exemple :  $29_{10}$  :

$$29 : 2 = 14 \text{ reste } 1$$

$$14 : 2 = 7 \text{ reste } 0$$

$$7 : 2 = 3 \text{ reste } 1$$

$$3 : 2 = 1 \text{ reste } 1$$

$$1 : 2 = 0 \text{ reste } 1$$

$$\text{donc } 29_{10} = 11101_2$$

# Les conversions

103

Exemple :  $14_5 = 9_{10} = 12_7$  :

– Conversion de la base 5 vers la base 7: division entière par 7 =  $12_7$  en base 7.

$$14_5 : 12_5 = 1 \text{ reste } 2$$

$$1_5 : 12_5 = 0 \text{ reste } 1$$

Le codage de  $14_5$  est donc  $12_7$ .

– Conversion de la base 7 vers la base 5 : division entière par 5 =  $5_7$  en base 7.

x	$0_7$	$1_7$	$2_7$	$3_7$	$4_7$	$5_7$	$6_7$	$10_7$
$5_7$	$0_7$	$5_7$	$13_7$	$21_7$	$26_7$	$34_7$	$42_7$	$50_7$

- $12_7 : 5_7 = 1 \text{ reste } 4$
- $1_7 : 5_7 = 0 \text{ reste } 1$
- Le codage de  $12_7$  est donc  $14_5$ .

# Les conversions

104

## REPRÉSENTATION DES ENTIERS

Exemple plus compliqué : conversion de  $1452_7$  en base 5

$$\begin{array}{r} 1452_7 \\ -13 \\ \hline 15 \\ -13 \\ \hline 22 \\ -21 \\ \hline 1 \end{array} \quad \begin{array}{r} 5_7 \\ 223 \end{array}$$

$$\begin{array}{r} 223_7 \\ -21 \\ \hline 13 \\ -13 \\ \hline 0 \end{array} \quad \begin{array}{r} 5_7 \\ 32 \end{array}$$

$$\begin{array}{r} 32_7 \\ -26 \\ \hline 3 \end{array} \quad \begin{array}{r} 5_7 \\ 4 \end{array}$$

D'où  $1452_7 = 4301_5$ . Les sceptiques peuvent vérifier que  $1452_7 = 1*7^3 + 4*7^2 + 5*7 + 2 = 576$  et  $4301_5 = 4*5^3 + 3*5^2 + 0*5 + 1 = 576...$

En pratique, si  $b_1 \neq 10$ , on évite d'effectuer la division en base  $b_1$  en convertissant  $X$  en base 10 puis en procédant par division sur le nombre obtenu pour effectuer la conversion de la base 10 vers la base  $b_2$ .

Certaines conversions sont très faciles à réaliser comme la conversion binaire vers octal (base 8) ou binaire vers hexadécimal (base 16).



# Les conversions

105

Exemple :  $1010011101_2$

– base 8 découpages par blocs de 3 chiffres :

001	010	011	101	
1	2	3	5	$= 1235_8$

– base 16 découpages par blocs de 4 chiffres :

0010	1001	1101	
2	9	D	$= 29D_{16}$

De manière générale, les conversions sont faciles lorsque  $b_2$  est une puissance de  $b_1$ . Les conversions en base 8 ou 16 sont très fréquentes pour l'affichage des nombres binaires qui, par leurs longueurs, sont rapidement illisibles.

Exemple : octal pour les codes de caractères, hexadécimal pour les adresses mémoires.

# Autres Opérations

106

## Opérations arithmétiques

Les opérations arithmétiques s'effectuent en base quelconque  $b$  avec les mêmes méthodes qu'en base 10. Une retenue ou un report apparaît lorsque l'on atteint ou dépasse la valeur  $b$  de la base.

## Représentation des entiers

### Codage machine

La représentation (ou codification) des nombres est nécessaire afin de les stocker et manipuler par un ordinateur. Le principal problème est la limitation de la taille du codage : un nombre mathématique peut prendre des valeurs arbitrairement grandes, tandis que le codage dans l'ordinateur doit s'effectuer sur un nombre de bits fixé.

# Codage

107

## Entiers naturels

Les entiers naturels (positifs ou nuls) sont codés sur un nombre d'octets fixé (un octet est un groupe de 8 bits). On rencontre habituellement des codages sur 1, 2 ou 4 octets, plus rarement sur 64 bits (8 octets, par exemple sur les processeurs DEC Alpha).

Un codage sur  $n$  bits permet de représenter tous les nombres naturels compris entre 0 et  $2^n - 1$ . Par exemple sur 1 octet, on pourra coder les nombres de 0 à  $2^8 - 1 = 255$ .

On représente le nombre en base 2 et on range les bits dans les cellules binaires correspondant à leur poids binaire, de la droite vers la gauche. Si nécessaire, on complète à gauche par des zéros (bits de poids fort).

## Entiers relatifs

Il faut ici coder le signe du nombre. On utilise le codage en complément à deux, qui permet d'effectuer ensuite les opérations arithmétiques entre nombres relatifs de la même façon qu'entre nombres naturels.

1. Entiers positifs ou nuls : On représente le nombre en base 2 et on range les bits comme pour les entiers naturels. Cependant, la cellule de poids fort est toujours à 0 : on utilise donc  $n - 1$  bits.

Le plus grand entier positif représentable sur  $n$  bits en relatif est donc  $2^{n-1} - 1$ .

# Codage des entiers négatif

109

Pour obtenir le codage d'un nombre  $x$  négatif, on code en binaire sa valeur absolue sur  $n - 1$  bits, puis on complémente (ou inverse) tous les bits et on ajoute 1.

**Exemple** : soit à coder la valeur -2 sur 8 bits. On exprime 2 en binaire, soit

00000010. Le complément à 1 est 11111101. On ajoute 1 et on obtient le résultat : 1111 1110.

Remarque :

- ✓ le bit de poids fort d'un nombre négatif est toujours 1 ;
- ✓ sur  $n$  bits, le plus grand entier positif est  $2^{n-1} - 1 = 011 \dots 1$  ;
- ✓ sur  $n$  bits, le plus petit entier négatif est  $-2^{n-1}$ .

# Représentation des caractères:

## Code ASCII

110

Les caractères sont des données non numériques : il n'y a pas de sens à additionner ou multiplier deux caractères. Par contre, il est souvent utile de comparer deux caractères, par exemple pour les trier dans l'ordre alphabétique.

Les caractères, appelés symboles alphanumériques, incluent les lettres majuscules et minuscules, les symboles de ponctuation (& ~ , . ; # " - etc...), et les chiffres.

Un texte, ou *chaîne de caractères*, sera représenté comme une suite de caractères.

Le codage des caractères est fait par une table de correspondance indiquant la configuration binaire représentant chaque caractère. Les deux codes les plus connus sont l'EBCDIC (en voie de disparition) et le code ASCII (American Standard Code for Information Interchange).

Le code ASCII représente chaque caractère sur 7 bits (on parle parfois de code ASCII étendu, utilisant 8 bits pour coder des caractères supplémentaires).

# Code ASCII

111

Notons que le code ASCII original, défini pour les besoins de l'informatique en langue anglaise) ne permet la représentation des caractères accentués (é, è, à, ù, ...), et encore moins des caractères chinois ou arabes. Pour ces langues, d'autres codages existent, utilisant 16 bits par caractères. A chaque caractère est associée une configuration de 8 chiffres binaires (1 octet), le chiffre de poids fort (le plus à gauche) étant toujours égal à zéro. La table indique aussi les valeurs en base 10 (décimal) et 16 (hexadécimal) du nombre correspondant.

Plusieurs points importants à propos du code ASCII :

- Les codes compris entre 0 et 31 ne représentent pas des caractères, ils ne sont pas affichables. Ces codes, souvent nommés *caractères de contrôles* sont utilisés pour indiquer des actions comme passer à la ligne (CR, LF), émettre un bip sonore, etc.
- Les lettres se suivent dans l'ordre alphabétique (codes 65 à 90 pour les majuscules, 97 à 122 pour les minuscules), ce qui simplifie les comparaisons.
- On passe des majuscules aux minuscules en modifiant le 5ième bit, ce qui revient à ajouter 32 au code ASCII décimal.
- Les chiffres sont rangés dans l'ordre croissant (codes 48 à 57), et les 4 bits de poids faibles définissent la valeur en binaire du chiffre.

# Récapitulatif

112

- Codage en binaire, ici un chiffre est appelé un bit (binary digit : chiffre binaire).
- Les nombres sont codés sur  $n$  octets, généralement 2 (*short* en Langage C) ou 4 (*int* ou *long* en langage C).
- $m$  bits  $\rightarrow 2^m$  nombres différents, si  $n = 2 \rightarrow 2^{16} = 65\,536$  nombres, si  $n = 4 \rightarrow 2^{32} = 4\,294\,967\,296$  nombres.
- le problème c'est que la capacité de la mémoire est limitée. Si le résultat de l'opération est supérieur au nombre maximum représentable  $\rightarrow$  *overflow* (dépassement de capacité).

Exemple sur 4 bits :  $9 + 7 = 16$  qui ne peut être stocké sur 4 bits...

$$\begin{array}{r} 1\ 0\ 0\ 1 \\ +\ 1\ 1\ 1 \\ \hline 0\ 0\ 0\ 0 \end{array}$$



# Récapitulatif

113

## **Complément binaire et codage des entiers négatifs**

- En binaire, le principe du complément est le même, le complément à 1 revient simplement à inverser les bits d'un nombre et le complément à 2 ajoute 1 au complément à 1.

# Conversion en binaire

114

- Exemple : 28,862510 en binaire
  - ▣ Conversion de 28 :  $11100_2$
  - ▣ Conversion de 0,8625 :
    - $0,8625 \times 2 = 1,725 = 1 + 0,725$
    - $0,725 \times 2 = 1,45 = 1 + 0,45$
    - $0,45 \times 2 = 0,9 = 0 + 0,9$
    - $0,9 \times 2 = 1,8 = 1 + 0,8$
    - $0,8 \times 2 = 1,6 = 1 + 0,6 \dots$
  - ▣  $28,862510 = (11100,11011\dots)_2$

# Conversion en hexadécimal

115

- Exemple :  $3,14159_{10}$  en hexadécimal
  - ▣ Conversion de 3:  $3_{16}$
  - ▣ Conversion de 0,14159:
    - $0,14159 \times 16 = 2,26544 = 2 + 0,26544$
    - $0,26544 \times 16 = 4,24704 = 4 + 0,24704$
    - $0,24704 \times 16 = 3,95264 = 3 + 0,95264...$
  - ▣  $3,14159_{10} = (3,243...)_{16}$

# Codage IEEE

116

- Représentation normalisée
- Un nombre représenté en virgule flottante est normalisé s'il est sous la forme:
  - ▣  $\pm 0, M * X^{\pm c}$
  - ▣  $M$  – un nombre dont le premier chiffre est non nul
  - ▣ Exemple:
    - $+ 59,4151 * 10^{-5} \Rightarrow$  Normalisé:  $+0,594151 * 10^{-3}$

# De nombreux défauts pour la représentation en virgule fixe

117

- Pour un nombre très grand comme le nombre d'Avogadro  $N_A$  (environ  $6,022 \times 10^{23}$ ), en écriture décimale cela nécessite au moins 24 chiffres pour une approximation à l'entier près.
- Pour un nombre très petit comme la charge élémentaire d'un électron (environ  $-1,602 \times 10^{-19}$  Coulombs), en écriture décimale cela nécessite au moins 20 chiffres pour une approximation.

# Virgule flottante

118

- Inspiré de l'écriture scientifique
- Exemple:
  - ▣  $173,95 = + 1,7395 \times 10^2$
- Généralisation: soit  $x$  un réel
$$x = \text{signe mantisse} \times 10^n$$
- Avantage: permet de représenter des nombres très grands et très petits sans s'encombrer de zéros

# Application à la base 2

119

- L'écriture devient alors:
  - signe mantisse  $\times 2^n$
  - Avec la mantisse et l'exposant en binaire
- A la fin des années 70, chaque ordinateur avait sa propre représentation pour les nombres à virgule flottante. Il y a donc eu la nécessité de normaliser le codage des nombres flottants

# La norme IEEE754

120

- signe mantisse  $\times 2^n$
- Le signe  $+$  est représenté par 0 et le signe  $-$  par 1
- La mantisse appartient à l'intervalle  $[1; 2[$
- L'exposant est un entier relatif et il est établi de manière à ce que la mantisse soit de la forme «1,...»



# Normalisation dans le format IEEE754

121

- La mantisse est normalisé sous la forme
  - $\pm 1, M * 2^{\pm c}$
  - Pseudo mantisse
  - Le 1 précédant la virgule n'est pas codé en machine et est appelé bit caché
- Exemple:
  - Mantisse: 101000000000000000000000
  - Représentation:  $1.101_2 = 1.625_{10}$

# La norme IEEE 754

122

- Plusieurs formats:
  - ▣ Simple précision : 32 bits (soit 4 octets) 1 bit de signe, 8 bits d'exposant, 23 bits de mantisse
  - ▣ Double précision : 64 bits (soit 8 octets) 1 bit de signe, 11 bits d'exposant, 52 bits de mantisse
  - ▣ Quadruple précision : 128 bits (soit 16 octets) 1 bit de signe, 15 bits d'exposant, 112 bits de mantisse

# La norme IEEE 754

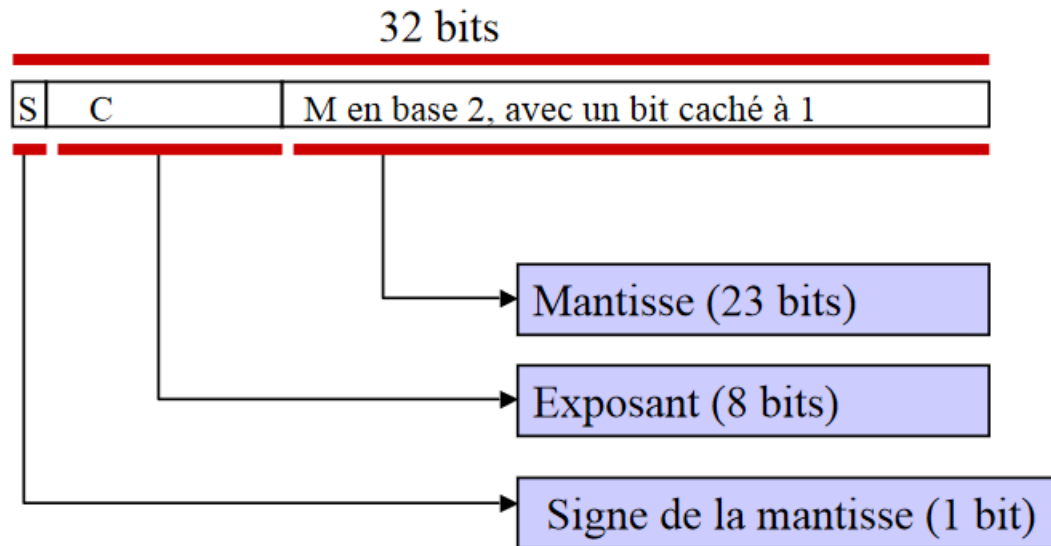
123

- Un format standardisé
- Format simple précision: 32 bits
  - Bit du signe (1 bit)
  - Exposant (8 bits)
  - Mantisse (23 bits)
- Format double précision: 64 bits
  - Bit du signe (1 bit)
  - Exposant (11 bits)
  - Mantisse (52 bits)

# Format simple précision

124

## □ IEEE 754 simple precision



# La norme IEEE754

125

- Simple précision:
- les caractéristiques
  - ▣ Exposant (n): de  $-126$  à  $127$
  - ▣ On effectue la somme  $n + 127$  afin de coder l'exposant en binaire
  - ▣ Mantisse: de  $1$  à  $2 \cdot 2^{-23}$
  - ▣ Plus petit nombre normalisé:  $2^{-126}$
  - ▣ Plus grand nombre normalisé: presque  $2^{128}$
  - ▣ Les exposants  $00000000$  et  $11111111$  sont interdits

# La norme IEEE754

126

- Simple précision: Exemple
- Codons le nombre  $-6,625$ 
  - $6,625_{10} = 110,1010_2$
  - $110,1010 = 1,101010 \times 2^2$
  - $101010000000000000000000$
  - $127 + 2 = 129_{10} = 10000001_2$
  - $1\ 10000001\ 101010000000000000000000$
  - En hexadécimal : C0 D4 00 00

# La norme IEEE754

127

- Exemple:  $0.75_{10}$ 
  - ▣  $0.75_{10} = 1.1 \times 2^{-1}$
  - ▣ La mantisse est donc :  $.1000\dots0$  L'exposant est donc :  $-1 + 127 = 126 = 01111110$
  - ▣ L'exposant est donc :  $-1 + 127 = 126 = 01111110_2$
  - ▣ Le codage du nombre sur 32 bits est donc :
    - $0\ 01111110\ 100000000000000000000000$
    - $3F400000_{16}$

# La norme IEEE754

128

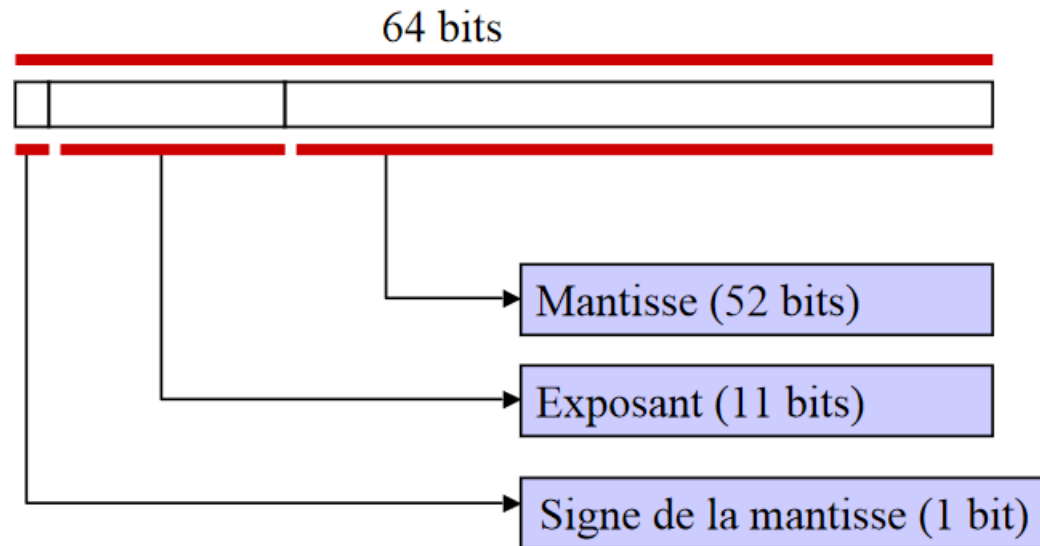
- Double précision: les caractéristiques
  - ▣ Exposant (n): de  $-1022$  à  $1023$
  - ▣ On effectue la somme  $n + 1023$  afin de coder l'exposant en binaire
  - ▣ Mantisse: de  $1$  à  $2 \cdot 2^{-52}$
  - ▣ Plus petit nombre normalisé:  $2^{-1022}$
  - ▣ Plus grand nombre normalisé: presque  $2^{1024}$



# Format double precision

129

## □ IEEE 754 double precision



# Le texte en informatique

130

- la façon dont on stocke du texte dans un ordinateur.
- Un ordinateur ne peut stocker que des nombres, plus précisément des 0 et des 1 « *bits* » qu'on regroupe pour former des nombres en binaire. Comment fait-on alors pour écrire du texte ? on associe à chaque **caractère** (une lettre, un signe de ponctuation, un espace...) un nombre. Un texte est alors une suite de ces nombres, on parle de **chaîne de caractères**.
- Il y a des caractères particuliers, dits « de contrôle », qui ne servent pas pour un symbole « imprimable » mais donnent des indications aux programmes qui manipulent les chaînes de caractères. Comme le caractère « fin de chaîne » qui sert à indiquer où s'arrête la chaîne.
- De plus, il y a aussi les lettres minuscules. Il faudrait aussi pouvoir gérer les accents, les symboles de monnaie... voire, les langues non latines (Arabe et Chinois).là c'est un vrai problème.
- Il faut aussi considérer les contraintes matérielles. En effet, un ordinateur ne connaît que le binaire. Les bits sont regroupés par groupes de 8 appelés « octets ». Un octet ne peut stocker que les nombres entiers de 0 à 255 (soit  $256 = 2^8$  possibilités). Si cela ne suffit pas, on peut rassembler les octets par 2, 4 ou plus pour avoir de plus grands nombres.

# Le code ASCII

131

- Le premier encodage historique est l'**ASCII**, soit l'*American Standard Code for Information Interchange* (en français, le *code américain normalisé pour l'échange d'informations*). C'est une norme américaine, inventée en 1961, qui avait pour but d'organiser le bazar informatique à l'échelle nationale. Ce n'est pas le premier encodage utilisé mais on peut oublier les précédents.
- Le jeu de caractères ASCII utilise **7 bits** (et non 8 !) et dispose donc de **128 (27) caractères uniquement, numérotés de 0 à 127**. En effet, il est paru à une époque où des regroupements par 7 au lieu de 8 étaient encore assez fréquents. Sans plus attendre, voici la table de l'ASCII en attaché.

# ISO 8859 : 8 bits pour les langues latines

132

- Plus tard, une norme mieux pensée a fait son apparition : la norme **ISO 8859**. Cette fois, elle utilise **8 bits donc 256 caractères au maximum**. Le standard ISO 8859 comporte en fait plusieurs « parties », c'est-à-dire des pages de code indépendantes, nommées ISO 8859-*n* où *n* est le numéro de la partie. ISO 8859 a été pensée afin que les parties soient le plus largement compatibles entre elles. Ainsi, elle englobe l'ASCII (codes 0 à 127) comme base commune, et les codes 128 à 255 devaient accueillir les caractères propres à chaque page de code, en s'arrangeant pour que des caractères identiques ou proches d'une page à l'autre occupent le même code.
- Ce standard a principalement servi aux langues latines d'Europe pour mettre au point une page de code commune. À elles seules, elles utilisent finalement 10 parties d'ISO 8859, parfois appelées *latin-1*, *latin-2*, etc. ; ces parties correspondent à des évolutions dans la page de code latine de base (*latin-1*, ou officiellement ISO 8859-1) afin de rajouter certains caractères pour compléter des langues. En voici deux que vous devriez connaître :
  - **ISO 8859-1 (*latin-1* ou « Occidental »)** est un encodage très courant dans les pays latins et sur la toile. C'est en effet celui qu'utilisent Linux et de nombreux documents et pages web. Les systèmes Windows utilisent également un jeu proche. Il a l'avantage de permettre d'écrire *grosso modo* toutes les langues latines, et ceci avec des caractères d'un octet seulement.
  - **ISO 8859-15 (*latin-9* ou « Occidental (euro) »)**, datant de 1998, introduit le signe de l'euro (€) et complète le support de quelques langues dont le français (avec Œ) en abandonnant des symboles peu utilisés (dont le mystérieux ¤ signifiant « monnaie »). Il est néanmoins peu utilisé par rapport à son grand frère ci-dessus.

# ISO 2022 : du multi-octet pour les langues asiatiques

133

Les langues latines s'en sont plutôt bien sorties finalement. Elles ont réussi à ne pas dépasser la limite fatidique de l'octet, ce qui restait quand même le plus pratique pour les traitements (et la consommation mémoire). Mais les langues asiatiques comme le japonais, le coréen ou le chinois disposent de bien trop de caractères pour que tout tienne sur 8 bits. Les encodages mis au point en Asie de l'Est ont donc franchi le saut du **multi-octet**. Certains utilisaient 2 octets, ce qui permet 65 536 (216) codes différents.

Comme pour les langues latines, un standard a été mis au point pour les organiser, on l'appelle **ISO 2022**. C'est un concept un peu spécial. Il permet de jongler entre plusieurs pages de code à l'aide de « séquences d'échappement » codées sur 3 octets (parfois 4) **et commençant par le caractère ASCII ESC (0x1B)** ; celles-ci indiquent aux programmes quelle page de code il faut utiliser pour interpréter ce qui suit. Les différentes pages de codes sont totalement indépendantes et peuvent utiliser un octet ou deux par caractère.

ISO 2022 a été utilisé pour le chinois (ISO 2022-CN), le coréen (ISO 2022-KR) et le japonais (ISO 2022-JP). Ces encodages, surtout le japonais, restent encore très répandus même si l'Unicode se développe

# Unicode

134

- Avec des normes comme ISO 8859 ou ISO 2022, on commençait à s'en tirer pas trop mal. Les problèmes sont atténués, mais subsistent (et si vous rédigez un document en français mais voulez y insérer de l'arabe ?). Finalement, des illuminés se sont dit : « Et si on créait un jeu de caractères unique pour tout le monde ? » De cette idée toute simple sont nés deux monuments : le standard ISO 10646 et Unicode.

# Quatrième Partie

135

## Plan

### Assembleur

Architecture RISC & CISC

Architecture PIPELINE

Registres

segmentation de la mémoire

Programmation en Assembleur 80x86

Directives d'assembleur 80x86

la pile

les procédures

# Rappel: Microprocesseur

136

- le microprocesseur est constituer de:
  - ❖ UAL
  - ❖ UC
  - ❖ Bus internes
  - ❖ Registres



# Historique

137

- le premier microprocesseur était le 4004 inventé par INTEL contient un bus de transfert de longueur 4 bits, mémorise un volume mémoire de 640 octets(bytes).
- Puis le 80x86 avec un bus de transport de 16 bits et un bus d'adresse de volume 20bits, il adresse 1MB de mémoire
- Puis apparu le pentium avec un bus de transfert de 32bits puis les PC courants d'un bus de transport de volume 64bits

# Historique du 8086

138

Depuis 1978, le 8086 fut le premier microprocesseur 16 bits fabriqué par Intel. Parmi ses caractéristiques principales:

Il est de la forme d'un boîtier de 40 broches alimenté de 5V.

Il possède un bus multiplexé adresse/donnée de 20bits

Le bus de donnée occupe 16 bits ce qui permet d'échanger des mots de 2 octets

Le bus d'adresse occupe 20 bits ce qui permet d'adresser  $2^{20}$  bits soit 1MO

Il est compatible avec la majorité des processeur du marché

Tous les registres sont de 16 bits de taille, mais pour garder la compatibilité avec les autres processeurs comme 8088, certains registres sont découpés en deux et on peut accéder séparément à la partie haute et à la partie basse.

# Caractéristique du 8086

139

- Le microprocesseur 8086 est un processeur 16 bits , il peut adresser 1MO par l'intermédiaire de son bus d'adresse
- Les registres du 8086:
  - 4 registres de travail de 16 bits (AX, BX, CX, DX) qu'on peut les utiliser sous forme de 8 registres de 8 bits
  - 2 registres d'index SI et DI de 16 bits
  - 2 registres de pointeurs BP et SP de 16 bits
  - 4 registres de segment 16 bits, CS, DS, SS, ES permettant la pagination de la mémoire,
  - 1 registre d'état

# Architecture Interne du 8086

140

Il existe deux unités internes distinctes: l'UE (**Unité** d'Exécution) et l'UIB (**Unité** d'Interfaçage avec le Bus). Le rôle de l'UIB est de récupérer et stocker les informations à traiter, et d'établir les transmissions avec les bus du système. L'UE exécute les instructions qui lui sont transmises par l'UIB. Les processeurs actuels de la famille x86 (compatible 8086) traitent les informations de la même façon.

Le traitement des instructions se passait comme suit:

- Extraction des instructions par l'UIB
- Exécution des instructions
- Extraction des nouvelles instructions

# Architecture Interne du 8086

141

Lorsque l'exécution d'une instruction est terminée, l'UE reste inactif un court instant, pendant que l'UIB extrait l'instruction suivante. Pour remédier à ce temps d'attente, le prétraitement ou traitement pipeline a été introduit dans cette famille de microprocesseur.

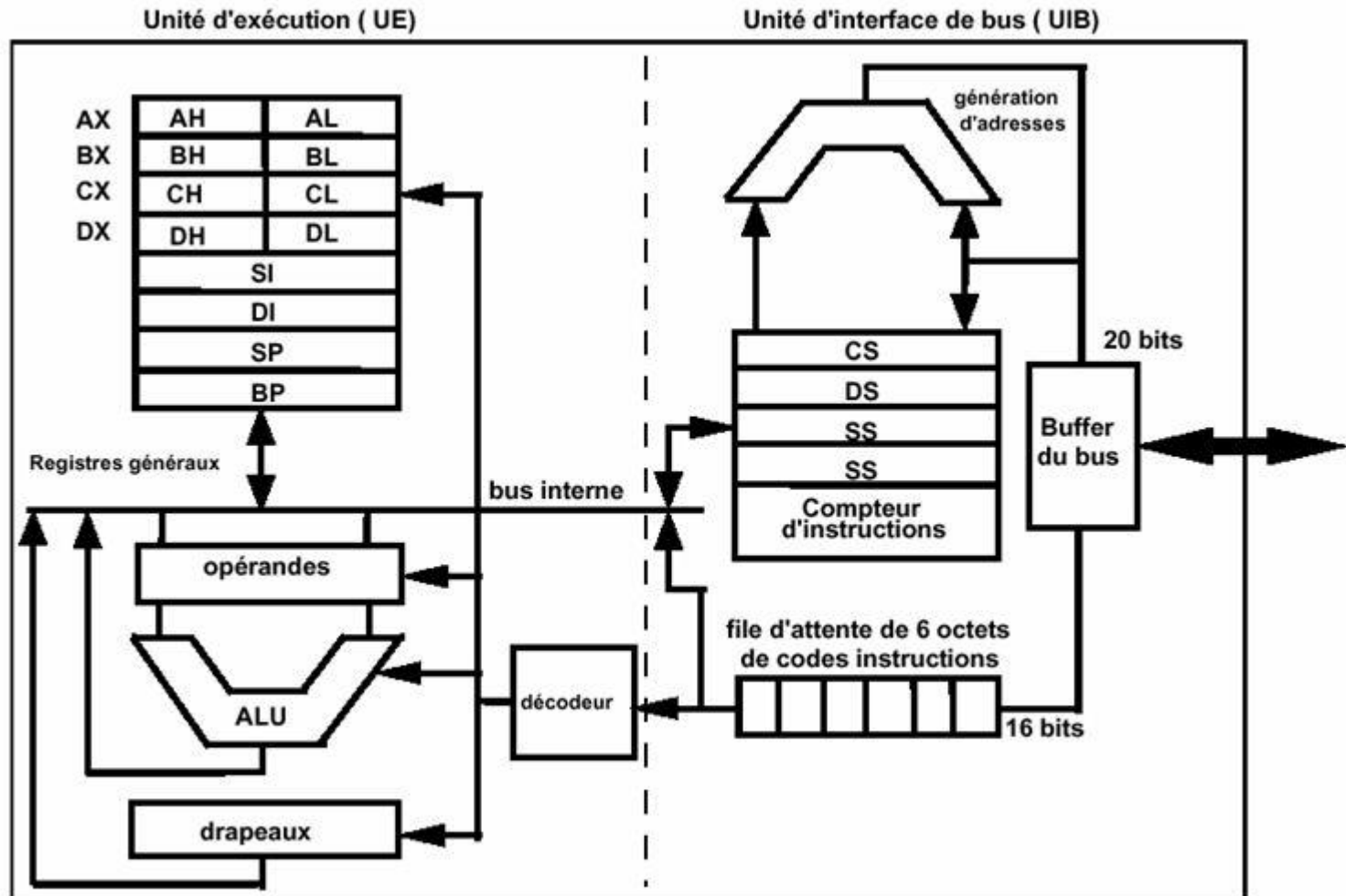
Pendant que l'UE exécute les informations qui lui sont transmises, l'instruction suivante est chargée dans l'UIB.

Les instructions qui suivront sont placées dans une file d'attente. Lorsque l'UE a fini de traiter une instruction l'UIB lui transmet instantanément l'instruction suivante, et charge la troisième instruction en vue de la transmettre à l'UE. De cette façon, l'UE est continuellement en activité. Dans la figure suivante nous pouvons observer un schéma plus détaillé de l'UE et l'UIB. Nous y retrouvons les éléments dont il a été question précédemment

En conclusion on peut dire que le 8086 se compose essentiellement de deux unités : la BIU qui fournit l'interface physique entre le microprocesseur et le monde extérieur et l'EU qui comporte essentiellement l'UAL de 16 bits qui manipule les registre généraux de 16 bits aussi.

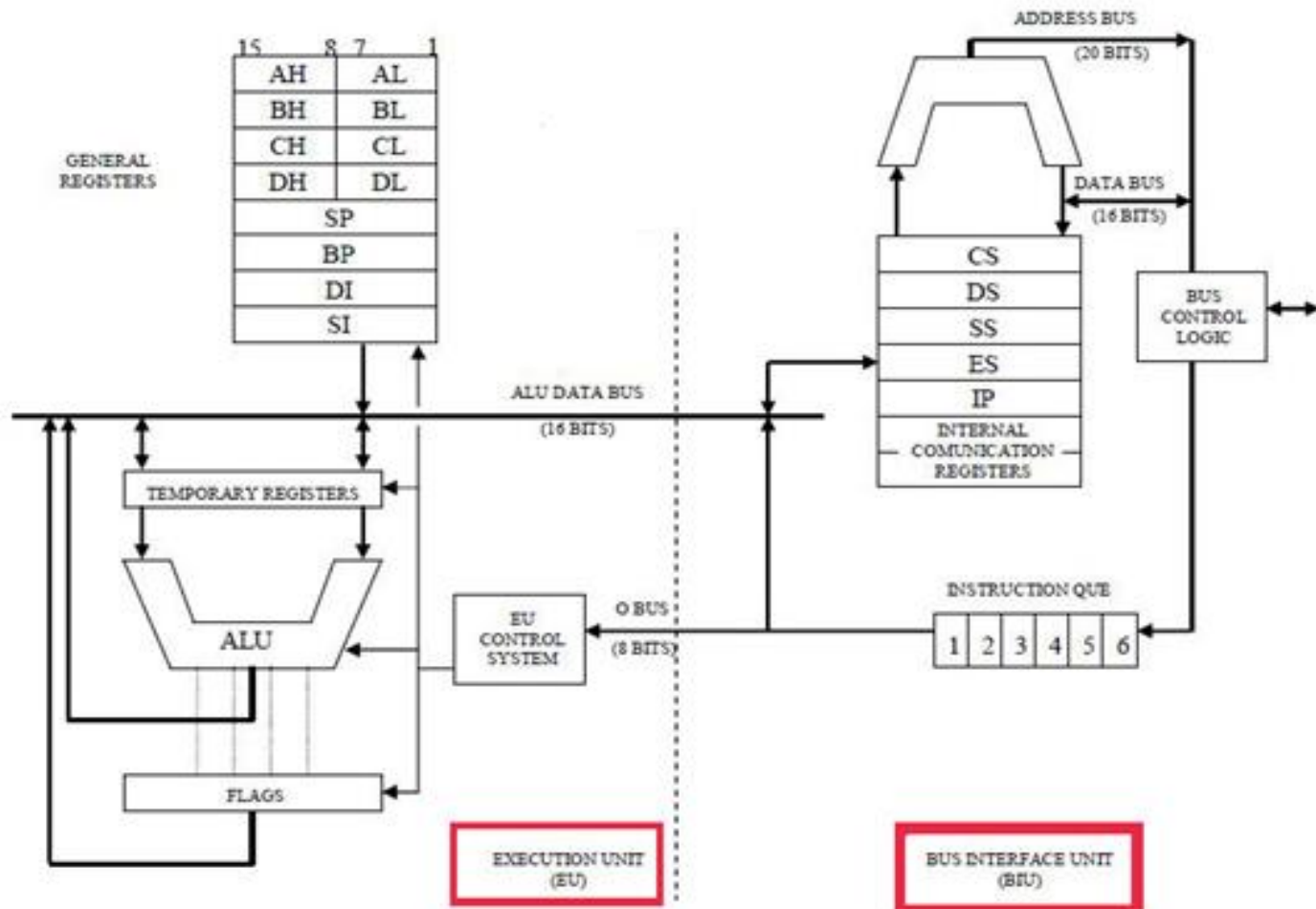
# Architecture Interne du 8086

142



# Architecture Interne du 8086

143



# Architecture Interne du 8086

144

- Dans un ordinateur il y'a 3 bus:

Bus de Données: il relie entre le microprocesseur et la mémoire, sa principale fonction est le transfert des données de la mémoire et vers la mémoire

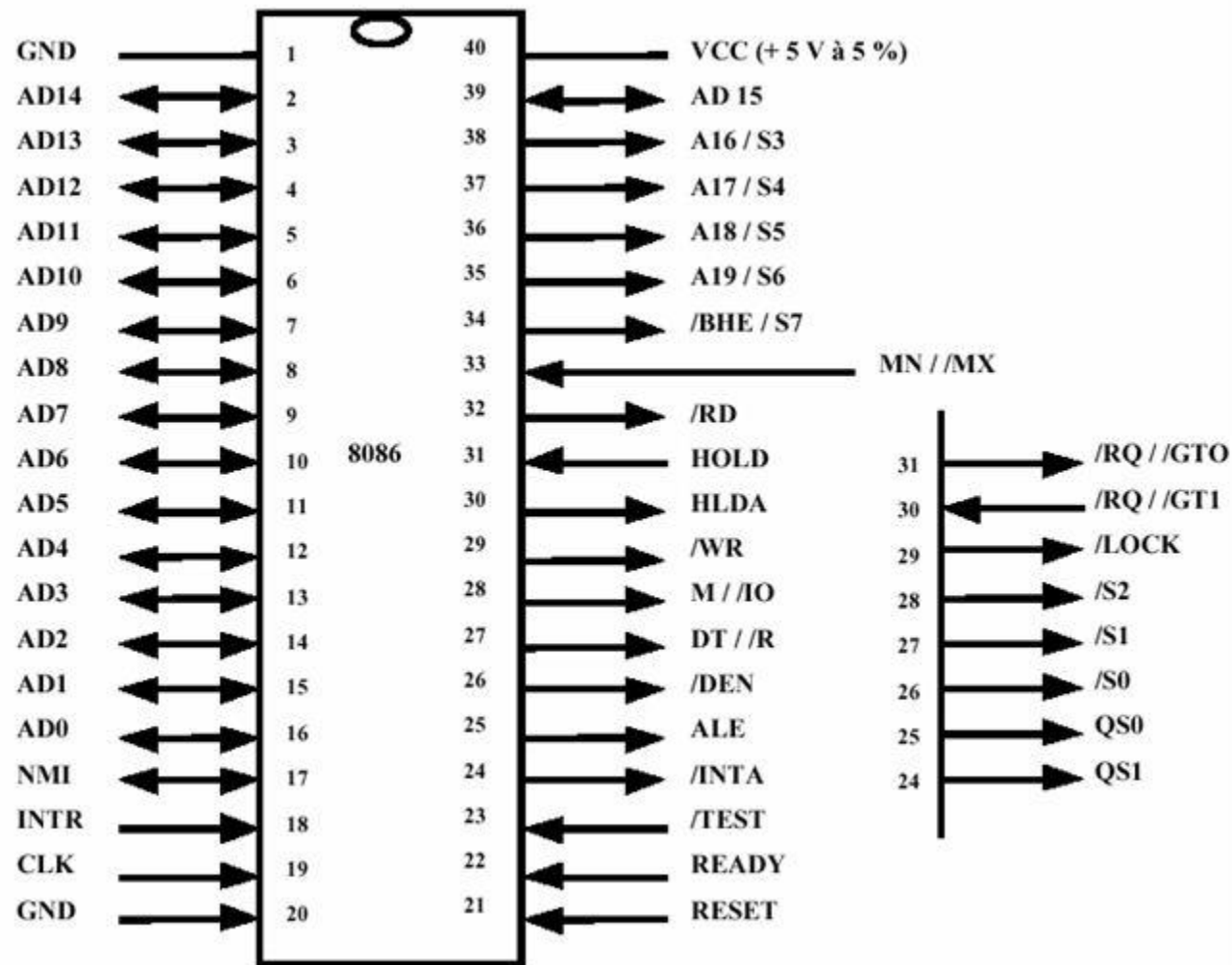
Bus d'Adresse: il relie entre le microprocesseur et la mémoire sa principale fonction est le transport des adresses du processeur vers la mémoire

Bus de Contrôle: il coordonne entre le bus de donnée et le bus d'adresse



# Architecture Externe du 8086

145



# Les registres

146

- Le processeur utilise toujours des registres, qui sont des petites mémoires internes intégré au microprocesseur, très rapides d'accès utilisées pour stocker temporairement une donnée, une instruction ou une adresse(durant l'exécution d'un programme). Certains d'entre eux sont affectés à des opérations d'ordre général et sont accessibles au programmeur à tout moment. Nous disons alors qu'il s'agit de registres généraux. D'autres registres ont des rôles bien plus spécifiques et ne peuvent pas servir à un usage non spécialisé

Le nombre exact de registres dépend du type de processeur et varie typiquement entre une dizaine et une centaine.

Parmi les registres, le plus important est le registre *accumulateur*, qui est *utilisé pour* stocker les résultats des opérations arithmétiques et logiques. L'accumulateur intervient dans une proportion importante des instructions.

# Les registres généraux

147

Les registres généraux peuvent être utilisés dans toutes les opérations arithmétiques et logiques que le programmeur insère dans le code assembleur. Un registre complet présente une grandeur de 16 bits. Comme le montre la figure suivante, chaque registre est en réalité divisé en deux registres distincts de 8 bits. De cette façon, nous pouvons utiliser une partie du registre si nous désirons y stocker une valeur n'excédant pas 8 bits. Si, au contraire, la valeur que nous désirons y ranger excède 8 bits, nous utiliserons le registre complet, c'est à dire 16 bits. Nous verrons plus loin qu'il est possible de manipuler facilement les registres généraux.

Le programmeur dispose de 8 registres internes de 16 bits qu'on peut diviser en deux groupes:

- groupe de données : formé par 4 registres de 16 bits (AX, BX, CX, et DX), chaque registre peut être divisé en deux registres de 8 bits (AH, AL, BH, BL, CH, CL, DH et DL )
- groupe de pointeur et indexe : formé de 4 registres de 16 bits (SI, DI, SP, BP) et font généralement référence à un emplacement en mémoire.

# Les registres

148

- Les registres généraux: Groupe de données:

	15	8	7	0
AX	AH		AL	
BX	BH		BL	
CX	CH		CL	
DX	DH		DL	

- Les Registres d'Adresse: Groupe de pointeur et index:

	15	0
Stack pointer	SP	
Base pointer	BP	
Source index	SI	
Destination index	DI	

# Groupe de données

149

## Registre AX : (Accumulateur)

Toutes les opérations de transferts de données avec les entrées-sorties ainsi que le traitement des chaînes de caractères se font dans ce registre, de même les opérations arithmétiques et logiques.

Les conversions en BCD du résultat d'une opération arithmétique (addition, soustraction, multiplication et la division) se font dans ce registre.

## Registre BX : (registre de base)

Il est utilisé pour l'adressage de données dans une zone mémoire différente de la zone code : en général il contient une adresse de décalage par rapport à une adresse de référence. ). (Par exemple, l'adresse de début d'un tableau). De plus il peut servir pour la conversion d'un code à un autre.

# Groupe de données

150

## Registre CX : (Le Compteur)

Lors de l'exécution d'une boucle on a souvent recours à un compteur de boucles pour compter le nombre d'itérations, le registre CX a été fait pour servir comme compteur lors des instructions de boucle.

### Remarque :

Le registre CL sert en tant que compteur pour les opérations de décalage et de rotation, dans ce cas il va compter le nombre de décalages (rotation) de bits à droite ou à gauche.

## Registre DX : (données)

On utilise le registre DX pour les opérations de multiplication et de division mais surtout pour contenir le numéro d'un port d'entrée/sortie pour adresser les interfaces d'E/S.

# Groupe de Pointeur d'Index

151

Ces registres sont plus spécialement adaptés au traitement des éléments dans la mémoire. Ils sont en général munis de propriétés d'incrémentement et de décrémentation.

Un cas particulier de pointeur est le pointeur de pile (Stack Pointer : SP). Ce registre permet de pointer la pile pour stocker des données ou des adresses selon le principe du "Dernier Entré Premier Sorti" ou "LIFO " (Last In First Out).

# Groupe de Pointeur d'Index

152

## **L'index SI : (source index) :**

Il permet de pointer la mémoire il forme en général un décalage (un offset) par rapport à une base fixe (le registre DS), il sert aussi pour les instructions de chaîne de caractères, en effet il pointe sur le caractère source

## **L'index DI : (Destination index) :**

Il permet aussi de pointer la mémoire il présente un décalage par rapport à une base fixe (DS ou ES), il sert aussi pour les instructions de chaîne de caractères, il pointe alors sur la destination

## **Les pointeurs SP et BP : ( Stack pointer et base pointer )**

Ils pointent sur la zone pile (une zone mémoire qui stocke l'information avec le principe FILO : voir plus loin), ils présentent un décalage par rapport à la base (le registre SS). Pour le registre BP il a un rôle proche de celui de BX, mais il est généralement utilisé avec le segment de pile.



# Les Registres de segment

153

Le 8086 a 4 registres segments de 16 bits chacun: **CS** (code segment, **DS** (Data segment), **ES** (Extra segment) et **SS** (stack segment), ces registres sont chargés de sélectionner les différents segments de la mémoire en pointant sur le début de chacun d'entre eux. Chaque segment de mémoire ne peut excéder les 65535 octets.

	15	0
Code segment	CS	
Data segment	DS	
Stack segment	SS	
Extra segment	ES	

# Les Registres de segment

154

## Le registre CS (code segment) :

Il pointe sur le segment qui contient les codes des instructions du programme en cours.

### Remarque :

Si la taille du programme dépasse les 65535 octets alors on peut diviser le code sur plusieurs segments (chacun ne dépasse pas les 65535 octets) et pour basculer d'une partie à une autre du programme il suffit de changer la valeur du registre CS et de cette manière on résout le problème des programmes qui ont une taille supérieure à 65535 octets.

## Le registre DS (Data segment) :

Le registre segment de données pointe sur le segment des variables globales du programme, bien évidemment la taille ne peut excéder 65535 octets (si on a des données qui dépassent cette limite, on utilise la même astuce citée dans la remarque précédente mais dans ce cas on change la valeur de DS).

# Les Registres de segment

155

## Le registre ES (Extra segment) :

Le registre de données supplémentaires ES est utilisé par le microprocesseur lorsque l'accès aux autres registres est devenu difficile ou impossible pour modifier des données, de même ce segment est utilisé pour le stockage des chaînes de caractères.

## Le segment SS (Stack segment) :

Le registre SS pointe sur la pile : la pile est une zone mémoire où on peut sauvegarder les registres ou les adresses ou les données pour les récupérer après l'exécution d'un sous programme ou l'exécution d'un programme d'interruption, en général il est conseillé de ne pas changer le contenu de ce registre car on risque de perdre des informations très importantes (par exemple les passages d'arguments entre le programme principal et le sous programme).

Remarque: Les valeurs des registres CS, DS et SS sont automatiquement initialisées par le système d'exploitation au lancement du programme. Dès lors, ces segments sont implicites, c'est-à-dire que si l'on désire accéder à une donnée en mémoire, il suffit de spécifier son offset sans avoir à se soucier du segment.

## Autres Registres

156

## Le registre IP : (Le compteur de programme) :

Instruction Pointer ou Compteur de Programme, contient l'adresse de l'emplacement mémoire où se situe la prochaine instruction à exécuter. Autrement dit, il doit indiquer au processeur la prochaine instruction à exécuter. Le registre IP est constamment modifié après l'exécution de chaque instruction afin qu'il pointe sur l'instruction suivante.

## Le registre d'état (Flags)

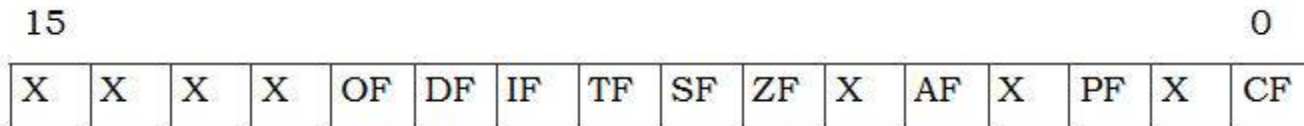
Le registre d'état FLAG sert à contenir l'état de certaines opérations effectuées par le processeur. Par exemple, quand le résultat d'une opération est trop grand pour être contenu dans le registre cible (celui qui doit contenir le résultat de l'opération), un bit spécifique du registre d'état (le bit OF) est mis à 1 pour indiquer le débordement.

Remarque : Drapeaux (flags)



Les drapeaux sont des indicateurs qui annoncent une condition particulière suite à une opération arithmétique ou logique.

Le registre d'état du 8086 est formé par les bits suivants :



# Autres Registres généraux

157

## CF (Carry Flag) :

Retenue: cet indicateur est mis à 1 lorsque il y a une retenue du résultat à 8 ou 16 bits. il intervient dans les opérations d'additions (retenue) et de soustractions (borrow) sur des entiers naturels. Il est positionné en particulier par les instructions ADD, SUB et CMP (comparaison entre deux valeurs).

CF = 1 s'il y a une retenue après l'addition ou la soustraction du bit de poids fort des opérandes. Exemples (sur 8 bits pour simplifier) :

$$\begin{array}{r} 10010110 \\ + 01010100 \\ \hline \end{array}$$

CF=0 11101010

$$\begin{array}{r} 11011001 \\ + 01010010 \\ \hline \end{array}$$

CF=1 00101011

## PF (Parity Flag) :

Parité : si le résultat de l'opération contient un nombre pair de 1 cet indicateur est mis à 1, sinon zéro.

# Autres Registres généraux

158

## AF (Auxiliary Carry) :

Demie retenue : Ce bit est égal à 1 si on a une retenue du quarter de poids faible dans le quarter de poids plus fort.

## ZF (Zero Flag) :

Zéro : Cet indicateur est mis à 1 quand le résultat d'une opération est égal à zéro. Lorsque l'on vient d'effectuer une soustraction (ou une comparaison), ZF=1 indique que les deux opérandes étaient égaux. Sinon, ZF est positionné à 0.

## SF (Sign Flag) :

SF est positionné à 1 si le bit de poids fort du résultat d'une addition ou soustraction est 1 ; sinon SF=0. SF est utile lorsque l'on manipule des entiers signés, car le bit de poids fort donne alors le signe du résultat. Exemples (sur 8 bits) :

$$\begin{array}{r} 10010110 \\ + 01010100 \\ \hline \end{array}$$

SF=1 11101010

$$\begin{array}{r} 11011001 \\ + 01010010 \\ \hline \end{array}$$

SF=0 00101011

# Autres Registres généraux

159

## OF (Overflow Flag) :

Débordement : si on a un débordement arithmétique ce bit est mis à 1. c.à.d. le résultat d'une opération excède la capacité de l'opérande (registre ou case mémoire), sinon il est à 0.

## DF (Direction Flag) :

Auto Incrémentation/Décrémentation : utilisée pendant les instructions de chaîne de caractères pour auto incrémenter ou auto décrémenter le SI et le DI.

## IF (Interrupt Flag) :

Masque d'interruption : pour masquer les interruptions venant de l'extérieur ce bit est mis à 0, dans le cas contraire le microprocesseur reconnaît l'interruption de l'extérieur.

## TF (Trap Flag) :

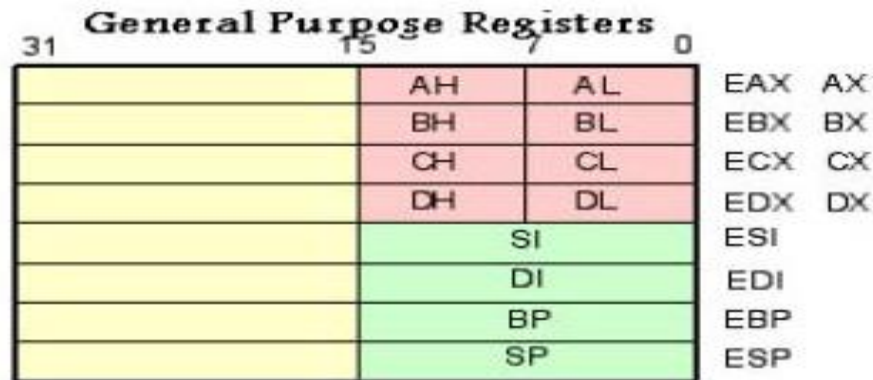
Piège : pour que le microprocesseur exécute le programme pas à pas du.

## Remarque :

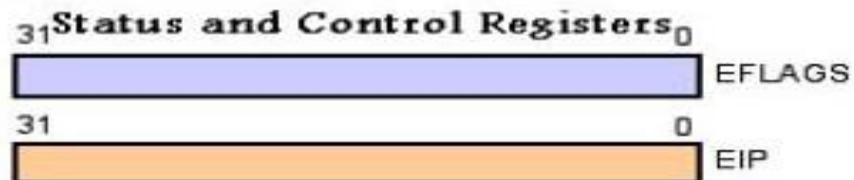
Les instructions de branchements conditionnels utilisent les *indicateurs* (drapeaux), qui sont des bits spéciaux positionnés par l'UAL après certaines opérations. Chaque indicateur est manipulé individuellement par des instructions spécifiques.

# Schéma récapitulatif

160



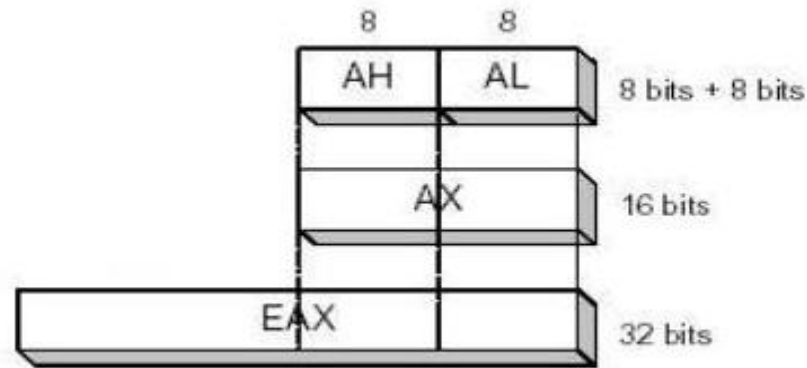
## Segment registers





# Schéma récapitulatif

161



Quand on utilise ces registres on arrive jusqu'à 32 bit grâce à EAX, et jusqu'à 16 bits grâce à AX  
Aussi jusqu'au premier ou deuxième Byte grâce à AL et AH

# Gestion de la mémoire

162

La mémoire se constitue de cases bien structurées de capacité 8bits soit un 1 octet, ces cases se numérotent de 0 jusqu'à la fin de la mémoire, et généralement le système de calcul hexadécimal est utilisé pour numéroté, et on appelle ce numéro Adresse mémoire de la case.

Entre le microprocesseur et la mémoire existe deux bus: un bus de données de largeur 16 bits, et un bus d'adresse de largeur 20 bits

Exemple: quand le UP a besoin de la valeur stocké dans la case 100 : le UP convertit 100 en binaire et la pose dans le bus d'adresse puis l'envoie à la mémoire, quand la mémoire reçoit ce nombre, elle renvoie au UP le contenu de cette case à travers le bus de données

Le fait que le bus d'adresse est de largeur de 20 bits(il peut transporter un nombre binaire de 20 digits), donc la plus grande valeur qu'il peut poser dans le bus d'adresse est  $2^{20}=1048576$  bits soit 1 MO donc le 8086 peut adresser 1MO de mémoire au maximum. On peut aussi enlever quatre segments que le UP utilise directement: CS,DS, SS, ES

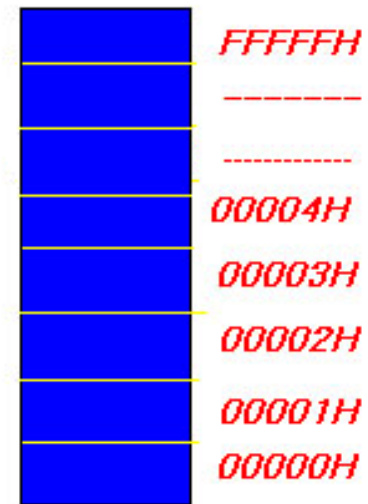
# Gestion de la mémoire

163

0000  
0001  
0010  
0011  
.....  
.....  
1111

**Et jusqu'à la dernière adresse en décimal: 1048575**

**Mais avec le système hexadécimal c'est très simple:**



# Gestion de la mémoire

164

Donc des adresses de largeur 20 bits dans des registres de taille 16 bits? Ce qui n'est pas logique

Méthodes d'adressage:

Pour accéder à un octet, le processeur propose 3 mécanismes d'adressages :

- l'adressage direct (un registre 32 bits contient une adresse) ;
- l'adressage semi-direct;
- l'adressage logique. Pour ce faire on utilise une *adresse logique*. Cette adresse est composée du numéro du segment contenu dans un registre 16 bits et d'un offset contenu dans un registre 32 bits.

# adressage logique

165

Au lieu d'élargir les tailles des registres, on a mis en place une méthode facile qui permet d'arriver vite à l'adresse voulue, on rassemblant deux registres, et diviser la mémoire en segments de 64 KO, à l'intérieur on utilise une autre adresse qui s'appelle offset. Donc pour arriver à n'importe quelle adresse dans la mémoire on utilise son adresse premièrement puis pour arriver à l'emplacement exact on utilise offset.

Donc l'adresse est segment:offset s'appelle adresse logique qu'à son travers on arrive à l'adresse affective dans la mémoire: adresse physique( pour convertir une adresse logique en adresse physique en multiplie par 16 **ajout d'un 0** au début puis on l'additionne avec offset, et on trouve l'adresse physique.

Bref, pour arriver à n'importe quelle adresse dans la mémoire on utilise l'adresse: segment:offset, la première adresse segment existe dans les registres de segment et la deuxième adresse offset existe dans l'un des registres SP ou IP

Donc:  $\text{adresse physique} = \text{segment} \times 16 + \text{offset}$

▣  $\text{adresse physique} = \text{segment} \times 16 + \text{offset}$

# Segmentation de la mémoire

166

Exemple 1:

L'adresse logique F000:FFFD === ?Adresse physique

$$F0000 + FFFD = FFFFD$$

Donc l'adresse physique recherchée est FFFFD

Exemple 2:

923F:E2FF ===== ? Adresse physique

$$923F0 + E2FF = A06EF$$

Donc l'adresse physique de cet exemple est: A06EF

Exemple 3:

0007:7B90 combien d'adresse physique lui correspond

0008:7B80 combien d'adresse physique lui correspond

Une seule adresse physique pour les deux adresses logiques différentes!

CHEVAUCHEMENT(OVERLAPPING)!!!!

# Segmentation de la mémoire

167

La mémoire est découpée à plusieurs segments de même taille 64KO soit 56536 octets(bytes)

Après tout 16 octets du début du segment commence le deuxième segment et après 16 octets commence le troisième et ainsi de suite. Qui veut dire que les segments sont interférés l'un dans l'autre, donc je peux arriver à une adresse dans le troisième segment à travers le premier segment ou le deuxième

Exemple: une adresse dans le segment 100, je peux y arriver à travers le segment 99 ou 98 et les segment avant ==> quand le nombre de segment est grand le nombre d'adresse logiques qu'on peut utilisé pour arriver cette adresse physique est grand

Dans le schéma suivant: le premier segment:00000 ==> 0FFFFh

le deuxième segment commence de 00010h soit après 16 octets du début du premier ...

La région du début du premier au début du suivant s'appelle: PARAGRAPH

# Segmentation de la mémoire

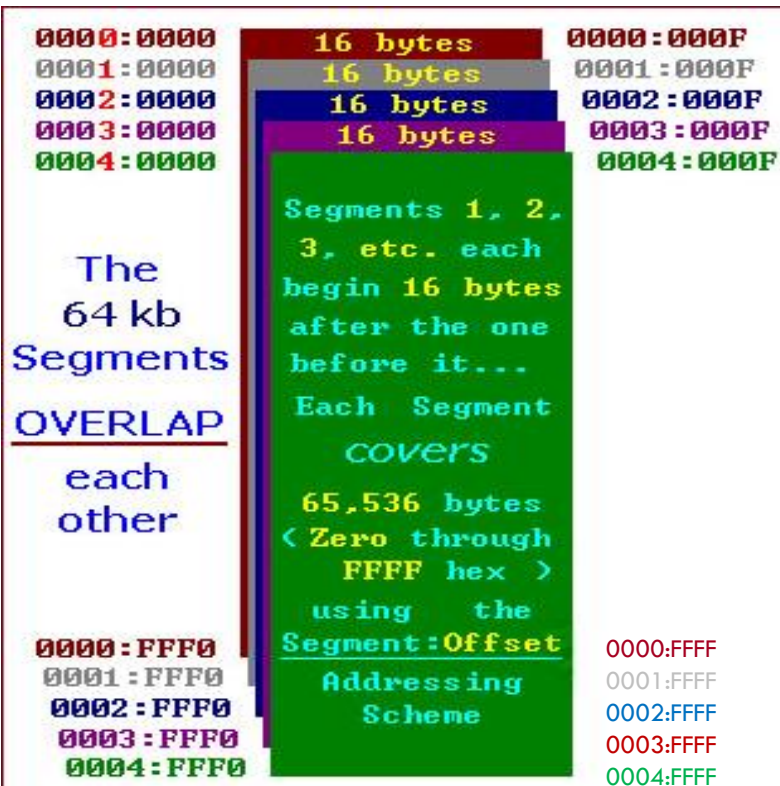
168

	Début du segment		Fin du segment	
1	0000:0000	00000h	0000:FFFF	0FFFFh
2	0001:0000	00010h	0001:FFFF	1000Fh
3	0002:0000	00020h	0002:FFFF	2000Fh
4	0003:0000	00030h	0003:FFFF	3000Fh
.....				
L	FFFF:0000	FFFF0h	FFFF:FFFF	10FFEFh



# Segmentation de la mémoire

169



Segment	Memory Location
0: ----- >	00h à 0FFFFh(0 to 65535)
1: ----- >	10h à 1000Fh(16 to 65551)
2: ----- >	20h à 1001Fh(32 to 65567)
3: ----- >	30h à 1002Fh(48 to 65583)
4: ----- >	40h à 1003Fh(64 to 65599)

# Segmentation de la mémoire

170

Dans le schéma précédent nous avons 5 premiers segments de la mémoire, on a ignoré le reste parce que c'est le même principe. Sur le schéma on trouve bien l'interférence entre les segments, après 16 octets du début d'un segment commence le suivant.

Si je veux arriver au segment vert (4<sup>ième</sup>), je peux utiliser 0, 1, 2, 3, 4 parce que ce point existe dans les 64KO de ces segments ==> ce qui permet le fait que pour une même adresse physique est associé plusieurs adresses logiques

# Langage de programmation: bas niveau

171

Le langage **machine** est le langage compris par le microprocesseur. Ce langage est difficile à maîtriser puisque chaque instruction est codée par une séquence propre de bits. Afin de faciliter la tâche du programmeur, on a créé différents langages plus ou moins évolués.

Le langage **assembleur** est le langage le plus « proche » du langage machine. Il est composé par des instructions en général assez rudimentaires que l'on appelle des **mnémoniques**. Ce sont essentiellement des opérations de transfert de données entre les registres et l'extérieur du microprocesseur (mémoire ou périphérique), ou des opérations arithmétiques ou logiques. Chaque instruction représente un code machine différent. Chaque microprocesseur peut posséder un assembleur différent.

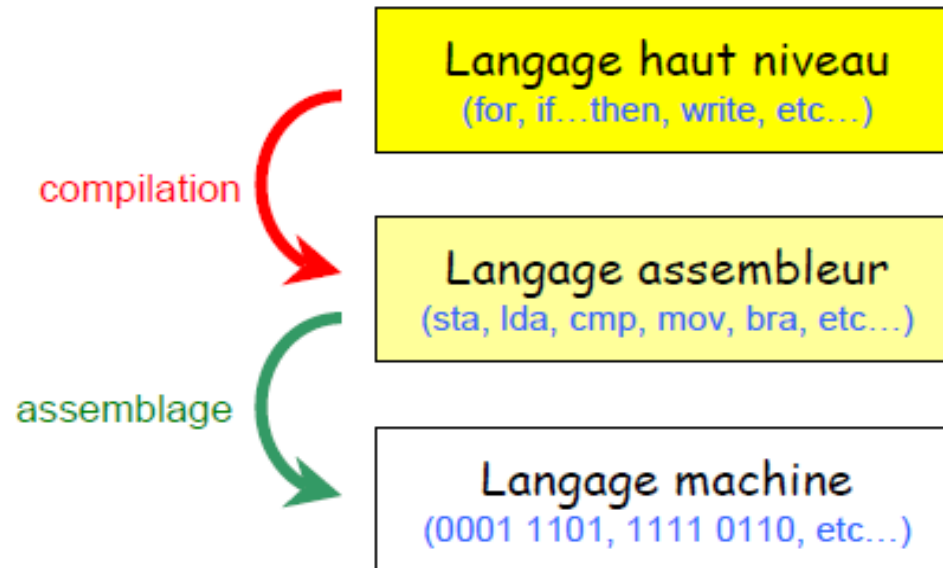
# Langage de programmation: assembleur

172

La difficulté de mise en œuvre de ce type de langage, et leur forte dépendance avec la machine, a nécessité la conception de langages de **haut niveau**, plus adaptés à l'homme, et aux applications qu'il cherchait à développer. Faisant abstraction de toute architecture de machine, ces langages permettent l'expression d'algorithmes sous une forme plus facile à apprendre, et à dominer (C, Pascal, Java, etc...). Chaque instruction en langage de haut niveau correspondra à une succession d'instructions en langage assembleur. Une fois développé, le programme en langage de haut niveau n'est donc pas compréhensible par le microprocesseur. Il faut le **compiler** pour le traduire en assembleur puis l'**assembler** pour le convertir en code machine compréhensible par le microprocesseur. Ces opérations sont réalisées à partir de logiciels spécialisés appelés *compilateur* et *assembleur*.

# Assembleur 8086

173



# Assembleur 8086

174

## □ Pourquoi l'assembleur ?

Lorsque l'on doit lire ou écrire un programme en langage machine, il est difficile d'utiliser la notation hexadécimale (un ensemble de 0101ABE...). On écrit les programmes à l'aide de symboles comme MOV, ADD, etc. Les concepteurs de processeur, comme Intel, fournissent toujours une documentation avec les codes des instructions de leur processeur, et les symboles correspondant.

# Écriture et exécuter programme

175

- L'assembleur est un utilitaire qui n'est pas interactif, contrairement à Java, C, ....
- Le programme que l'on désire traduire en langage machine (on dit *assembler*) doit être placé dans un fichier texte (avec l'extension .ASM sous DOS).
- La saisie du programme source au clavier nécessite un programme appelé *éditeur de texte*.
- L'opération d'assemblage traduit chaque instruction du programme source en une instruction machine. Le résultat de l'assemblage est enregistré dans un fichier avec l'extension .OBJ (*fichier objet*).
- Le fichier .OBJ n'est pas directement exécutable. En effet, il arrive fréquemment que l'on construise un programme exécutable à partir de plusieurs fichiers sources. Il faut "relier" les fichiers objets à l'aide d'un utilitaire nommé *éditeur de lien* (*même si l'on en a qu'un seul*). L'éditeur de liens fabrique un fichier exécutable, avec l'extension .EXE.
- Le fichier .EXE est directement exécutable. Un utilitaire spécial du système d'exploitation, le *chargeur* est responsable de la lecture du fichier exécutable, de son implantation en mémoire principale, puis du lancement du programme.

# Déclaration de variables

176

- On déclare les variables à l'aide de directives. L'assembleur attribue à chaque variable une adresse. Dans le programme, on repère les variables grâce à leur nom.
- Les noms des variables (comme les étiquettes) sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre. Le nom peut comporter des majuscules, des minuscules, des chiffres, plus les caractères @, ? et \_.



# Structure d'un Programme

177

```
data          SEGMENT      ; data est le nom du segment de donnees
                        ; directives de declaration de donnees
data          ENDS          ; fin du segment de donnees
                        ASSUME DS:data, CS:code
code          SEGMENT      ; code est le nom du segment d'instructions
debut:                                ; lere instruction, avec l'etiquette debut
                        ; suite d'instructions
code          ENDS
                        END debut ; fin du programme, avec l'etiquette
                                ; de la premiere instruction.
```

Structure d'un programme en assembleur (fichier .ASM).

# Variables de 8 ou 16 bits

178

- Les directives DB (*Define Byte*) et DW (*Define Word*) permettent de déclarer des variables de respectivement 1 ou 2 octets.

Exemple d'utilisation :

```
data        SEGMENT
entree      DW    15 ; 2 octets initialisés à 15
sortie      DW    ? ; 2 octets non initialisés
cle         DB    ? ; 1 octet non initialisé
nega        DB    -1 ; 1 octet initialisé à -1
data        ENDS
```

# Variables de 8 ou 16 bits

179

- Les valeurs initiales peuvent être données en hexadécimal (constante terminée par H) ou en binaire (terminée par b) :

```
data SEGMENT
```

```
truc          DW      0F0AH ; en hexa
```

```
masque        DB      01110000b ; en binaire
```

```
data ENDS
```

Les variables s'utilisent dans le programme en les désignant par leur nom. Après la déclaration précédente, on peut écrire par exemple :

```
MOV           AX, truc
```

```
AND           AL, masque
```

```
MOV           truc, AX
```

L'assembleur se charge de remplacer les noms de variable par les adresses correspondantes.

# Directive MOV

180

MOV op1, op2: pose dans op1 une copie de op2.

Exemple: AL = 20h, BL=50h

Mov AL, BL : après l'exécution: BL = 50h, AL=50h.

Mov BL, 60h: après l'exécution: BL=60h

Mov ne change pas les drapeaux.

*Remarque: l'op1 doit être en même taille que op2*

Directive	Position	Cause
MOV AL, BX	F	Op2<op1
MOV 123h, AL	F	On peut pas stocker un nombre de trois chiffres en hexa, dans un registre de taille 8bits. Il faut 1 ou 2 chiffres
MOV 120, BL	V	120=78h
MOV DX,12345h	F	Dans un registre de taille 16bits il faut y enregistrer une valeur de 4 bit maximum en hexa

# Directive ADD

181

ADD OP1, OP2: Ajoute le contenu de op2 au contenu de op1 et ça peut affecter les drapeaux

Le premier op peut être: variable, adresse, numéro

Op2 peut être référence mémoire

Exemple:

Mov AX, 3d

Add AX, 2d

Résultats: AX=5d

# Directive SUB

182

- ❑ SUB OP1, OP2: cette directive permet de soustrait la valeur de op2 du contenue de op1 le résultat est stocké dans l'op1.
- ❑ Cette directive affecte les drapeaux

# Directive CMP

183

**CMP OP1,OP2:** s'utilise souvent pour comparer les valeurs des op1 et op2.

Exemple: es-ce que le registre AX contient la valeur 5

**CMP AX, 5;** cette opération permet de soustraire 5 de AX, ni l'un ni l'autre ne change de valeur, mais les registre qui s'affectent:

RESULTAT	COMPARAISON	FLAGS
ZERO	EGALITE	SF =0, ZF = 1
POSITIVE	OP1 > OP2	SF=0,ZF=0
NEGATIVE	OP1 < OP2	SF=1,ZF=0

# DIRECTIVE AND

184

Cette opération est logique

Les valeurs possibles de AND:

1 AND 1	1
1 AND 0	0
0 AND 0	0
0 AND 1	0

Exemple: AND AX, BX

AX avant	0	0	0	0	1	0	1	1	1	1	0	0	0	1	1
BX avant	1	0	0	1	1	0	0	0	0	1	0	0	0	0	1
AX après	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1

AND s'utilise souvent pour déterminer la valeur d'un bit dans la mémoire



# DIRECTIVE OR

185

Cette opération est logique

Les valeurs possibles de OR:

1 OR 1	1
1 OR 0	1
0 OR 0	0
0 OR 1	1

Exemple: OR AX, BX

AX avant	0	0	0	0	1	0	1	1	1	1	0	0	0	1	1
BX avant	1	0	0	1	1	0	0	0	0	1	0	0	0	0	1
AX après	1	0	0	1	1	0	1	1	1	1	0	0	0	1	1

ADD s'utilise souvent pour déterminer la valeur d'un bit dans la mémoire

# DIRECTIVE XOR

186

Cette opération est logique

Les valeurs possibles de XOR:

1 XOR 1	0
1 OR 0	1
0 OR 0	0
0 OR 1	1

Exemple: XOR AX, BX

AX avant	0	0	0	0	1	0	1	1	1	1	0	0	0	1	1
BX avant	1	0	0	1	1	0	0	0	0	1	0	0	0	0	1
AX après	1	0	0	1	0	0	1	1	1	0	0	0	0	1	0

ADD s'utilise souvent pour déterminer la valeur d'un bit dans la mémoire

# DIRECTIVE TEST

187

- Même chose que AND, mais ne change que la valeur des drapeaux (FLAGS)

# DIRECTIVE MUL ET IMUL

188

- Même chose pour les deux la seule différence est que IMUL permet de faire une multiplication par signe par contre Mul permet de faire une multiplication sans signe.
- Ces directives utilisent par défaut AX comme OP2

Exemple :  $-2 * -4 = 8 = 1000b$

Mov AL,-2

Mov BL,-4

IMUL BL

AX=8h=1000b

$1700 * 520 = 0D7D20h = 11010111110100100000b$

MOV AX, 1700

MOV BX, 520

MUL BX

DX = 000D h = 00000000000001101 b

AX = 7D20 h = 0111110100100000 b

# IDIV et DIV

189

Taille	Résultat	Reste	/
BYTE	AL	AH	AX
WORD	AX	DX	DX:AX
DWORD	EAX	EDX	EDX:EAX

Exemple:

11CCCEEE44BBBAAA h / 33A33A33 = 583EF4DF h =

1011000001111101111010011011111 b

MOV EDX, 11CCCEEE h

MOV EAX, 44BBBAAA h

MOV ECX, 33A33A33 h

IDIV ECX

EAX = 583EF4DF h = 1011000001111101111010011011111 b

EDX = 15B96C3D h

# XCHG, INC, DEC

190

- ❑ XCHG Permet d'échanger le contenu des deux opérandes

Exemple XCHG AX,BX

SI BX=10 et AX=5

On aurait:

BX=5 et AX=10

- ❑ INC incrémente la donnée avec UN

Exemple: INC AX  $\implies$  AX = AX + 1

INC affecte les drapeaux: ZF,SF,OF,PF,AF

- ❑ DEC décrémente avec 1:

Exemple: DEC AX  $\implies$  AX = AX - 1  $\implies$  SUB AX,1

DEC affecte les drapeaux: ZF,SF,OF,PF,AF

# NOT, NEG

191

□ NOT permet d'inverser la valeur contenue dans la donnée  
Exemple: AL=01110010 on veut lui affecter la valeur  
10001101

Mov AL, 01110010 b

NOT AL

====>AL = 10001101b

NEG permette de changer le signe de la donnée

MOV AX, 100 ====> AX =100

NEG AX ====> AX=FF00 = -100

NEG 1X ====> AX=100

NEG affecte les flags: CF, ZF, SF, OF, PF, AF

# SHL

192

- SHL fais une translation vers la gauche (MSB) d'un octet les bits qui restent à droite (LSB) prennent 0

Nom bre	0	1	0	0	1	1	0	0
SHL	1	0	0	1	1	0	0	0

Si cette opération cause un débordement, le bit qui s'ajoute va être stocker dans le CARRY-FLAG

Exemple:

MOV AL, 01001010B

SHL AL, 1D

Après l'exécution de SHL, AL devient: 10010100B, on pose 1 dans le CARRY-FLAG

RQ: Exécution de SHL une fois permet de multiplier le nombre\*1

**SHL AX,1 ==> AX\*2**

**SHL AX,2 ==> AX\*4**

**SHL AX,3 ==> AX\*8**



# SHR

193

- SHL fais une translation vers la droite (LSB) d'un octet les bit à droite(MSB) prennent 0

Nom bre	0	1	0	0	1	1	1	1
SHL	0	0	1	0	0	1	1	1

Exemple:

MOV AL, 01100111B

SHR AL, 1D

Après l'exécution de SHR, AL devient: 00110011B, on pose 1 dans le CARRY-FLAG

RQ: Exécution de SHR une fois permet de diviser le nombre par 1

**SHL AX,1 ==> AX/2**

# ROR

194

- ROR permet de faire tourner la donnée à droite

Exemple:

1. MOV AL, 01010111B

ROR AL, 1

Après exécution, la valeur de AL devient 10101011B

2. MOV AL, 00111111B

ROR AL, 3

AL:

Premier tours 10011111

Deuxième tours 11001111

Troisième tours 11100111

CF:1

RQ: ROR AX, 16 ==> AX revient à son état initiale

- Pareil pour ROL: MOV AL, 00111111B

ROL AL, 1 ==> 01111110B    CF: 0

# CALL, RET, INT

195

- ❑ CALL Son nom l'indique, c'est un ordre

CALL Adresse

- ❑ RET: permet le retour d'un ou la fin d'une fonction.
- ❑ INT: permet l'appel d'un ordre généralement existant dans le bios.

Exemple: INT 10h

# Segment de code et de données

196

La valeur du segment est stockée dans des registres spéciaux de 16 bits. Le registre DS (*Data Segment*) est utilisé pour le segment de données, et le registre CS (*Code Segment*) pour le segment d'instructions.

## **Registre CS**

Lorsque le processeur lit le code d'une instruction, l'adresse 32 bits est formée à l'aide du registre segment CS et du registre déplacement IP. La paire de ces deux registres est notée CS:IP.

## **Registre DS**

Le registre DS est utilisé pour accéder aux données manipulées par le programme. Ainsi, l'instruction

`MOV AX, [0145]`

donnera lieu à la lecture du mot mémoire d'adresse DS:0145H.

# Initialisation des registres segment

197

Dans ce cours, nous n'écrirons pas de programmes utilisant plus de 64 Ko de code et 64 Ko de données, ce qui nous permettra de n'utiliser qu'un seul segment de chaque type.

Par conséquent, la valeur des registres CS et de DS sera fixée une fois pour toute au début du programme.

Le programmeur en assembleur doit se charger de l'initialisation de DS, c'est-à-dire de lui affecter l'adresse du segment de données à utiliser.

Par contre, le registre CS sera automatiquement initialisé sur le segment contenant la première instruction au moment du chargement en mémoire du programme (par le chargeur du système d'exploitation).

# Déclaration d'un segment en assembleur

198

Comme nous l'avons vu précédemment les directives `SEGMENT` et `ENDS` permettent de définir les segments de code et de données.

La directive `ASSUME` permet d'indiquer à l'assembleur quel est le segment de données et celui de code, afin qu'il génère des adresses correctes. Enfin, le programme doit commencer, avant toute référence au segment de données, par initialiser le registre segment `DS`, de la façon suivante :

```
MOV AX, nom_segment_de_donnees
```

```
MOV CS, AX
```

(Il serait plus simple de faire

`MOV CS, nom_segment_de_donnees` mais il se trouve que cette instruction n'existe pas.)

# Exemple de programme en assembleur

199

```
; Programme calculant la somme de deux entiers de 16 bits
data                SEGMENT
A                   DW          10 ; A = 10
B                   DW          1789 ; B = 1789
Result              DW          ? ; resultat
data                ENDS
code                SEGMENT
ASSUME              DS:data,    CS:code
debut:              MOV         AX, data ; etiquette car 1ere instruction
                   MOV         DS, AX ; initialise DS
                   ; Le programme:
                   MOV         AX, A
                   ADD         AX, B
                   MOV         result, AX ; range resultat
                   ; Retour au DOS:
                   MOV         AH, 4CH
                   INT 21H
code                ENDS
END                 debut ; etiquette de la 1ere inst.
```

# Spécification de la taille des données

200

Dans certains cas, l'adressage indirect est ambigu. Par exemple, si l'on écrit

`MOV [BX], 0 ;` range 0 a l'adresse spécifiée par BX  
l'assembleur ne sait pas si l'instruction concerne 1, 2 ou 4 octets consécutifs.

Afin de lever l'ambiguïté, on doit utiliser une directive spécifiant la taille de la donnée à transférer :

`MOV byte ptr [BX], val ;` concerne 1 octet

`MOV word ptr [BX], val ;` concerne 1 mot de 2 octets



# La pile

201

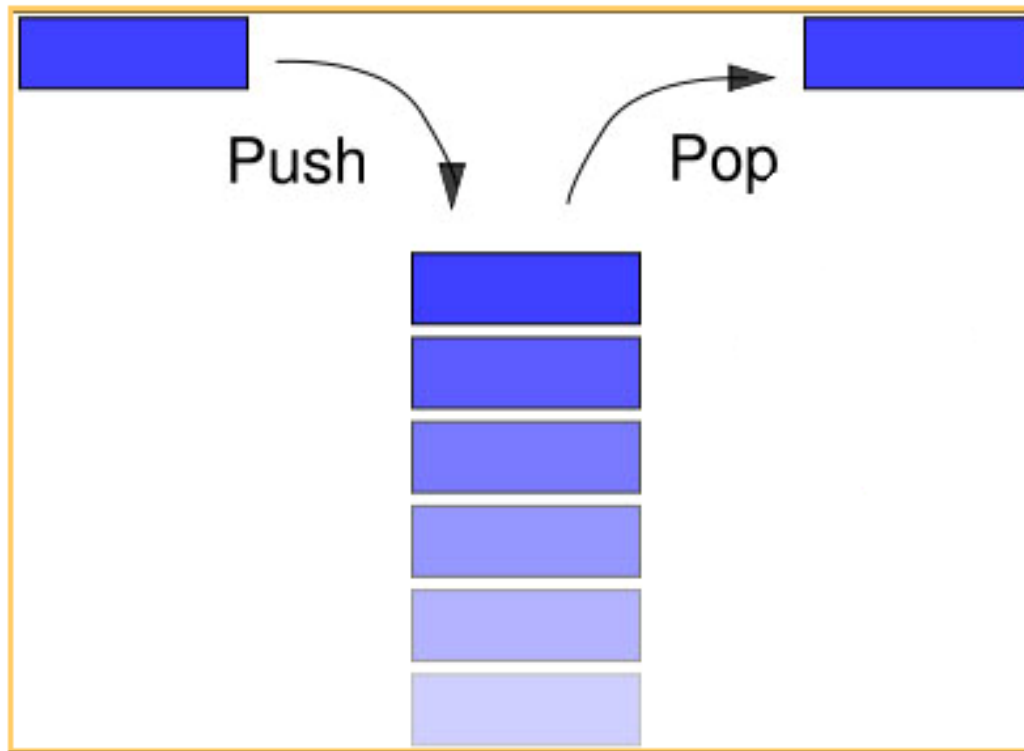
## Notion de pile

- Les *piles* offrent un nouveau moyen d'accéder à des données en mémoire principale, qui est très utilisé pour stocker temporairement des valeurs.
- Une pile est une zone de mémoire et un pointeur qui conserve l'adresse du *sommet* de la pile.

# POP & PUSH

202

PUSH , POP



# Instructions PUSH et POP

203

Ces deux nouvelles instructions, PUSH et POP, permettent de manipuler la pile.

**PUSH** *registre empile le contenu du registre sur la pile.*

**POP** *registre retire la valeur en haut de la pile et la place dans le registres spécifié.*

Exemple : transfert de AX vers BX en passant par la pile.

```
PUSH  AX          ;          Pile <- AX
```

```
POP   BX          ;          BX <- Pile
```

(Note : dans cet exemple il vaut mieux employer MOV AX, BX.) La pile est souvent utilisée pour sauvegarder temporairement le contenu des registres. AX et BX contiennent des données à conserver

```
PUSH  AX
```

```
PUSH  BX
```

```
MOV   BX, truc    ;          on utilise AX
```

```
ADD   AX, BX      ;          et BX
```

```
MOV   truc, BX
```

```
POP   BX          ;          récupère l'ancien BX
```

```
POP   AX          ;          et l'ancien AX
```

On voit que la pile peut conserver plusieurs valeurs. La valeur dépilée par POP est la *dernière valeur empilée* ; c'est pourquoi on parle ici de pile LIFO (Last In First Out, Premier Entré Dernier Sorti).

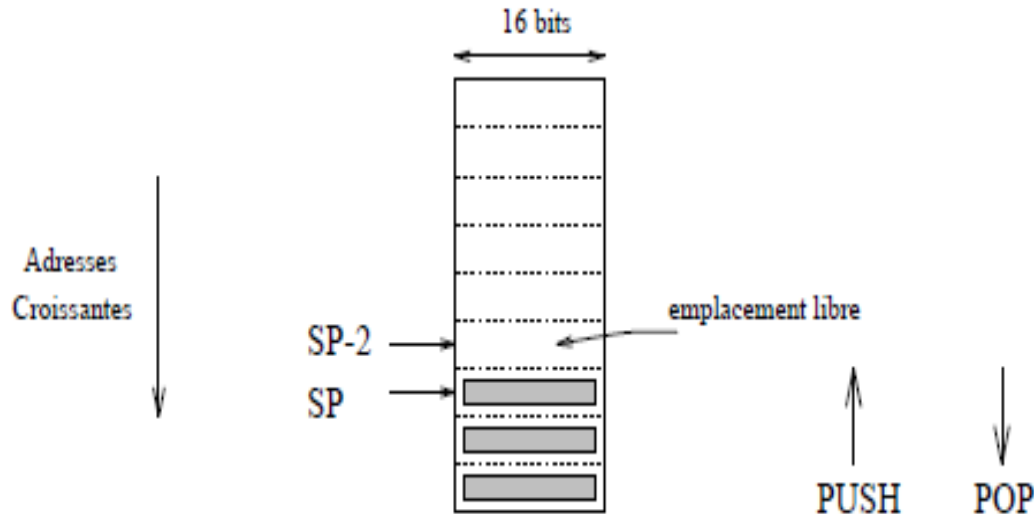
# Registres SS et SP

204

La pile est stockée dans un segment séparé de la mémoire principale. Le processeur possède deux registres dédiés à la gestion de la pile, SS et SP.

Le registre SS (*Stack Segment*) est un registre segment qui contient l'adresse du segment de pile courant (16 bits de poids fort de l'adresse). Il est normalement initialisé au début du programme et reste fixé par la suite.

Le registre SP (*Stack Pointer*) contient le déplacement du sommet de la pile (16 bits de poids faible de son adresse).



L'instruction PUSH effectue les opérations suivantes :

$SP = SP - 2$

[SP] valeur du registre 16 bits.

Notons qu'au début (pile vide), SP pointe "sous" la pile.

L'instruction POP effectue le travail inverse :  
registre destination [SP]

$SP = SP + 2$

Si la pile est vide, POP va lire une valeur en dehors de l'espace pile, donc imprévisible

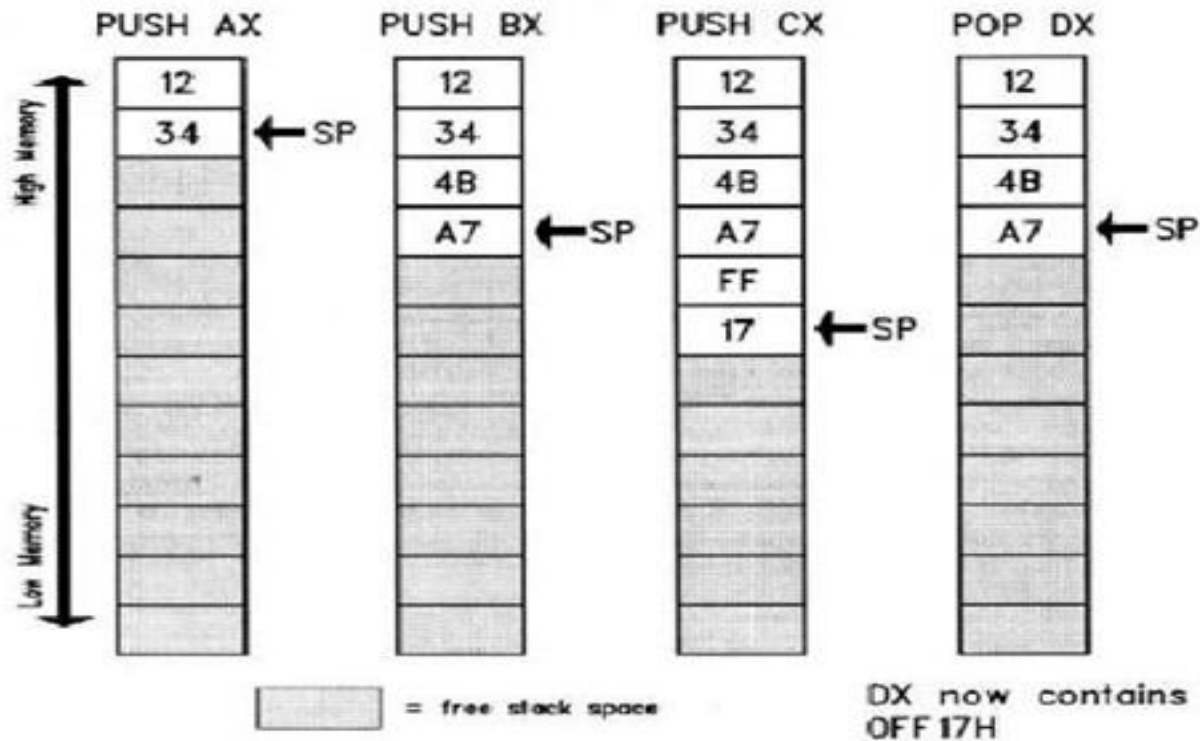
# Exemple

205

Given these  
initial register  
values:

AX = 01234H  
BX = 04BA7H  
CX = 0FF17H  
DX = 034E0H

After...



# Exemple

206

- en remarque que si on conserve les valeurs des registres comme ça:  
AX,BX,DX,CX successivement avec les instructions:

PUSH AX

PUSH BX

PUSH CX

PUSH DX

Donc on doit les récupérer comme ça:

POP DX

POP CX

POP BX

POP AX

On suit le principe LIFO.

# Déclaration d'une pile

207

Pour utiliser une pile en assembleur, il faut déclarer un segment de pile, et y réserver un espace suffisant. Ensuite, il est nécessaire d'initialiser les registres SS et SP pour pointer sous le sommet de la pile.

Voici la déclaration d'une pile de 200 octets :

```
seg_pile SEGMENT stack ; mot clef stack car pile
DW 100 dup (?) ; réserve espace
base_pile EQU this word ; étiquette base de la pile
seg_pile ENDS
```

Noter le mot clef “stack” après la directive SEGMENT, qui indique à l'assembleur qu'il s'agit d'un segment de pile. Afin d'initialiser SP, il faut repérer l'adresse du bas de la pile ; c'est le rôle de la ligne

# Déclaration d'une pile

208

Après les déclarations ci-dessus, on utilisera la séquence d'initialisation :

```
ASSUME  SS:seg_pile
```

```
MOV     AX, seg_pile
```

```
MOV     SS, AX ; init Stack Segment
```

```
MOV     SP, base_pile ; pile vide
```

Noter que le registre SS s'initialise de façon similaire au registre DS ; par contre, on peut accéder directement au registre SP.



# Les Branchements

209

- **JMP**: branchement non conditionnel, permet de transférer le traitement dans un autre point du programme sans condition.

Exemple:

```
MOV EAX, 1
```

```
JMP @1
```

```
ADD EAX, 2
```

```
JMP @2
```

```
:@1
```

```
ADD EAX, 3
```

```
:@2
```

La valeur final de EAX est 4 et non pas 3 parce qu'il saute une partie qu'elle n'est plus utilisée

# Les Branchements conditionnels

210

Les branchement conditionnels sont: JE/JZ, JNE/JNZ, JG, JGE, JA,, JAE, JL, JLE, JBE ...les plus utilisés: JZ, JNZ et JE.

Ces directives sont toujours suivies par une condition particulière par exemple JE est utilisée quand  $ZF=1$  et ce drapeau s'affecte par une comparaison

Exemple:

```
CMP EAX,EBX
```

```
JE SMP
```

```
...
```

```
: FPO
```

```
...
```

```
:SMP
```

Dans ce cas si la valeur de EAX vau la valeur de EBX le programme va faire un jump vers SMP si non il va poursuivre son exécution normale

# Exemple de branchement conditionnel

211

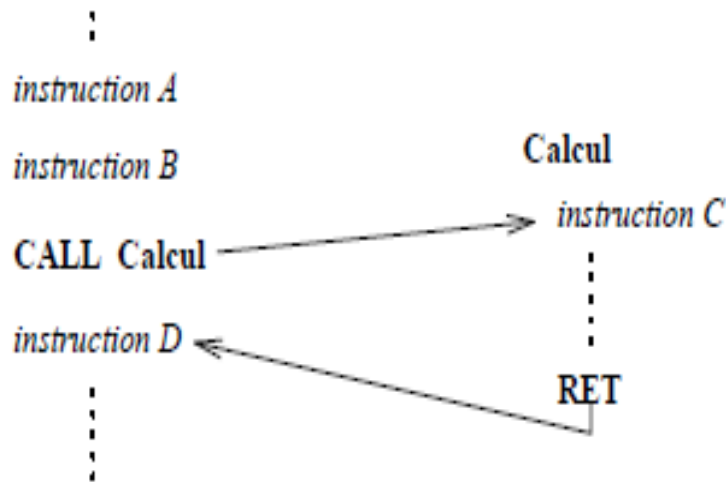
Directive	Sens
JC	Jump if CF =1
JNC	Jump if CF=0
JO	Jump if OF=1
JNO	Jump if OF = 0
JS	Jump if SF = 1
JNS	Jump if SF = 0
JZ/JE	If égalité/résultat égal à 0
JNL/JGE	If >=/si n'est pas inferieur
JNZ/JNE	If différent/ if résultat n'est pas nul

# Procédures

212

## □ Notion de procédure

La notion de procédure en assembleur correspond à celle de fonction en langage C, ou de sous-programme dans d'autres langages.



Appel d'une procédure. La procédure est nommée **calcul**. **Après l'instruction B**, le processeur passe à l'instruction C de la procédure, puis continue jusqu'à rencontrer RET et revient à l'instruction D.

# Procédures

213

Une procédure est une suite d'instructions effectuant une action précise, qui sont regroupées par commodité et pour éviter d'avoir à les écrire à plusieurs reprises dans le programme.

Les procédures sont repérées par l'adresse de leur première instruction, à laquelle on associe une étiquette en assembleur.

L'exécution d'une procédure est déclenchée par un programme *appelant*. Une procédure peut elle-même appeler une autre procédure, et ainsi de suite

# Instructions CALL et RET

214

L'appel d'une procédure est effectué par l'instruction CALL.

CALL                      *adresse\_debut\_procedure*

L'adresse est sur 16 bits, la procédure est donc dans le même segment d'instructions.

CALL est une nouvelle instruction de branchement inconditionnel.

La fin d'une procédure est marquée par l'instruction RET :

RET ne prend pas d'argument ; le processeur passe à l'instruction placée immédiatement après le CALL.

RET est aussi une instruction de branchement : le registre IP est modifié pour revenir à la valeur qu'il avait avant l'appel par CALL. Comment le processeur retrouve-t-il cette

# Déclaration d'une procédure

215

L'assembleur possède quelques directives facilitant la déclaration de procédures.

On déclare une procédure dans le segment d'instruction comme suit :

```
Calcul      PROC  near ;  procedure nommee Calcul
              ... ;      instructions
              RET      ;   dernière instruction
Calcul      ENDP   ;      fin de la procédure
```

Le mot clef PROC commence la définition d'une procédure, near indiquant qu'il s'agit d'une procédure située dans le même segment d'instructions que le programme appelant.

L'appel s'écrit simplement :

```
CALL      Calcul
```

# Passage de paramètres

216

En général, une procédure effectue un traitement sur des données (*paramètres*) qui sont fournies par le programme appelant, et produit un résultat qui est transmis à ce programme.

Plusieurs stratégies peuvent être employées :

- ❖ *Passage par registre : les valeurs des paramètres sont contenues dans des registres du processeur. C'est une méthode simple, mais qui ne convient que si le nombre de paramètres est petit (il y a peu de registres).*
- ❖ *Passage par la pile : les valeurs des paramètres sont empilées. La procédure lit la pile.*



# Exemple avec passage par registre

217

On va écrire une procédure “SOMME” qui calcule la somme de 2 nombres naturels de 16 bits. Convenons que les entiers sont passés par les registres AX et BX, et que le résultat sera placé dans le registre AX.

La procédure s’écrit alors très simplement :

```
SOMME      PROC    near      ;      AX <- AX + BX
ADD        AX, BX
           RET
SOMME      ENDP
```

et son appel, par exemple pour ajouter 6 à la variable Truc :

```
MOV        AX, 6
MOV        BX, Truc
CALL       SOMME
MOV        Truc, AX
```

# Exemple avec passage par la pile

218

Cette technique met en œuvre un nouveau registre, BP (*Base Pointer*), qui permet de lire des valeurs sur la pile sans les dépiler ni modifier SP.

Le registre BP permet un mode d'adressage indirect spécial, de la forme :

```
MOV          AX, [BP+6]
```

cette instruction charge le contenu du mot mémoire d'adresse BP+6 dans AX.

Ainsi, on lira le sommet de la pile avec :

```
MOV          BP, SP ;    BP pointe sur le sommet
```

```
MOV          AX, [BP] ; lit sans dépiler
```

et le mot suivant avec :

```
MOV          AX, [BP+2] ;    2 car 2 octets par mot de pile.
```

L'appel de la procédure "SOMME2" avec passage par la pile est :

```
PUSH 6
```

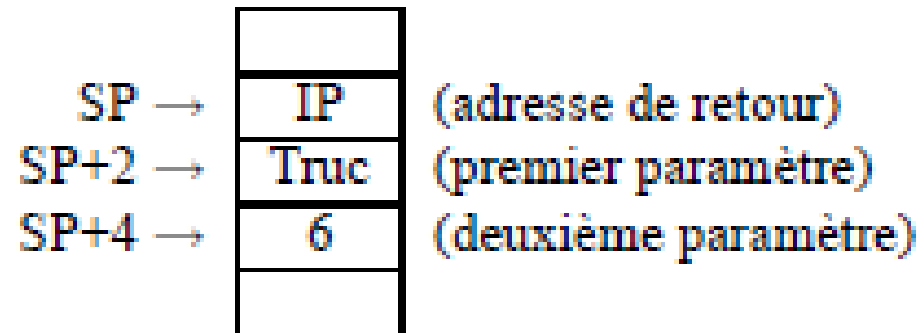
```
PUSH      Truc
```

```
CALL      SOMME2
```

La procédure SOMME2 va lire la pile pour obtenir la valeur des paramètres. Pour cela, il faut bien comprendre quel est le contenu de la pile après le CALL :

# Exemple avec passage par la pile

219



Le sommet de la pile contient l'adresse de retour (ancienne valeur de IP empilée par CALL). Chaque élément de la pile occupe deux octets.

La procédure SOMME2 s'écrit donc :

```
SOMME2    PROC    near    ;           AX <- arg1 + arg2
MOV       BP, SP      ;           adresse sommet pile
MOV       AX, [BP+2]   ;           charge argument 1
ADD       AX, [BP+4]   ;           ajoute argument 2
RET
SOMME2    ENDP
```

# Exemple avec passage par la pile

220

La valeur de retour est laissée dans AX.

La solution avec passage par la pile paraît plus lourde sur cet exemple simple. Cependant, elle est beaucoup plus souple dans le cas général que le passage par registre. Il est très facile par exemple d'ajouter deux paramètres supplémentaires sur la pile.

Une procédure bien écrite modifie le moins de registres possible. En général, l'accumulateur est utilisé pour transmettre le résultat et est donc modifié. Les autres registres utilisés par la procédure seront normalement sauvegardés sur la pile. Voici une autre version de SOMME2 qui ne modifie pas la valeur contenue par BP avant l'appel :

```
SOMME2      PROC      near      ;      AX <- arg1 + arg2
PUSH        BP          ;      sauvegarde BP
MOV         BP, SP      ;      adresse sommet pile
MOV         AX, [BP+4]   ;      charge argument 1
ADD         AX, [BP+6]   ;      ajoute argument 2
POP         BP          ;      restaure ancien BP
            RET
SOMME2      ENDP
```

Noter que les index des arguments (BP+4 et BP+6) sont modifiés car on a ajouté une valeur au sommet de la pile.

# Mode d'Adressage

221

Les instructions et leurs opérandes (paramètres) sont stockées en mémoire principale. La taille totale d'une instruction (nombre de bits nécessaires pour la représenter en mémoire) dépend du type d'instruction et aussi du type d'opérande. Chaque instruction est toujours codée sur un nombre entier d'octets, afin de faciliter son décodage par le processeur. Une instruction est composée de deux champs :

- le code opération, qui indique au processeur quelle instruction réaliser.
- le champ opérande qui contient la donnée, ou la référence à une donnée en mémoire (son adresse).



# Mode d'Adressage

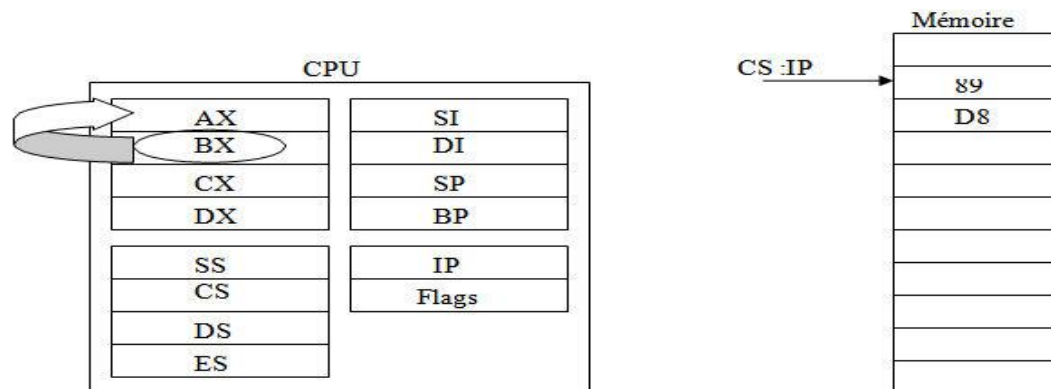
222

Les façons de désigner les opérandes constituent les "modes d'adressage". Selon la manière dont l'opérande (la donnée) est spécifié, c'est à dire selon le mode d'adressage de la donnée, une instruction sera codée par 1, 2, 3 ou 4 octets.

Le microprocesseur 8086 possède 7 modes d'adressage :

- Mode d'adressage registre
- Mode d'adressage immédiat
- Mode d'adressage direct
- Mode d'adressage registre indirect.
- Mode d'adressage relatif à une base.
- Mode d'adressage direct indexé.
- Mode d'adressage indexée.

## 223



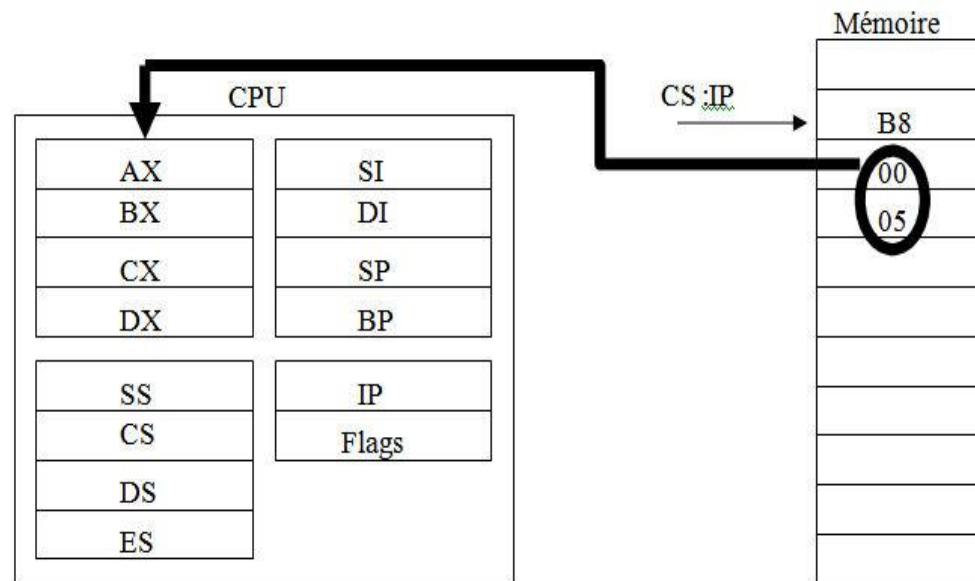
# Mode d'adressage immédiat

224

Dans ce mode d'adressage l'opérande apparaît dans l'instruction elle-même(valeur),

exemple :

`MOV AX,500H` ; cela signifie que la valeur 500H sera stockée immédiatement dans le registre AX





# Mode d'adressage direct

225

Dans ce mode un des opérandes se trouve en mémoire. L'adresse de la case mémoire ou plus précisément son offset est précisé directement dans l'instruction. L'adresse Rseg:Off doit être placée entre [], si le segment n'est pas précisé, DS est pris par défaut

==>On spécifie directement l'adresse de l'opérande dans l'instruction

INST AX, [adr]; INST [adr], R ; INST taille [adr], im ; La valeur adr est une constante (un déplacement) qui doit être ajouté au contenu du registre DS pour former l'adresse physique de 20 bits.

exemples:

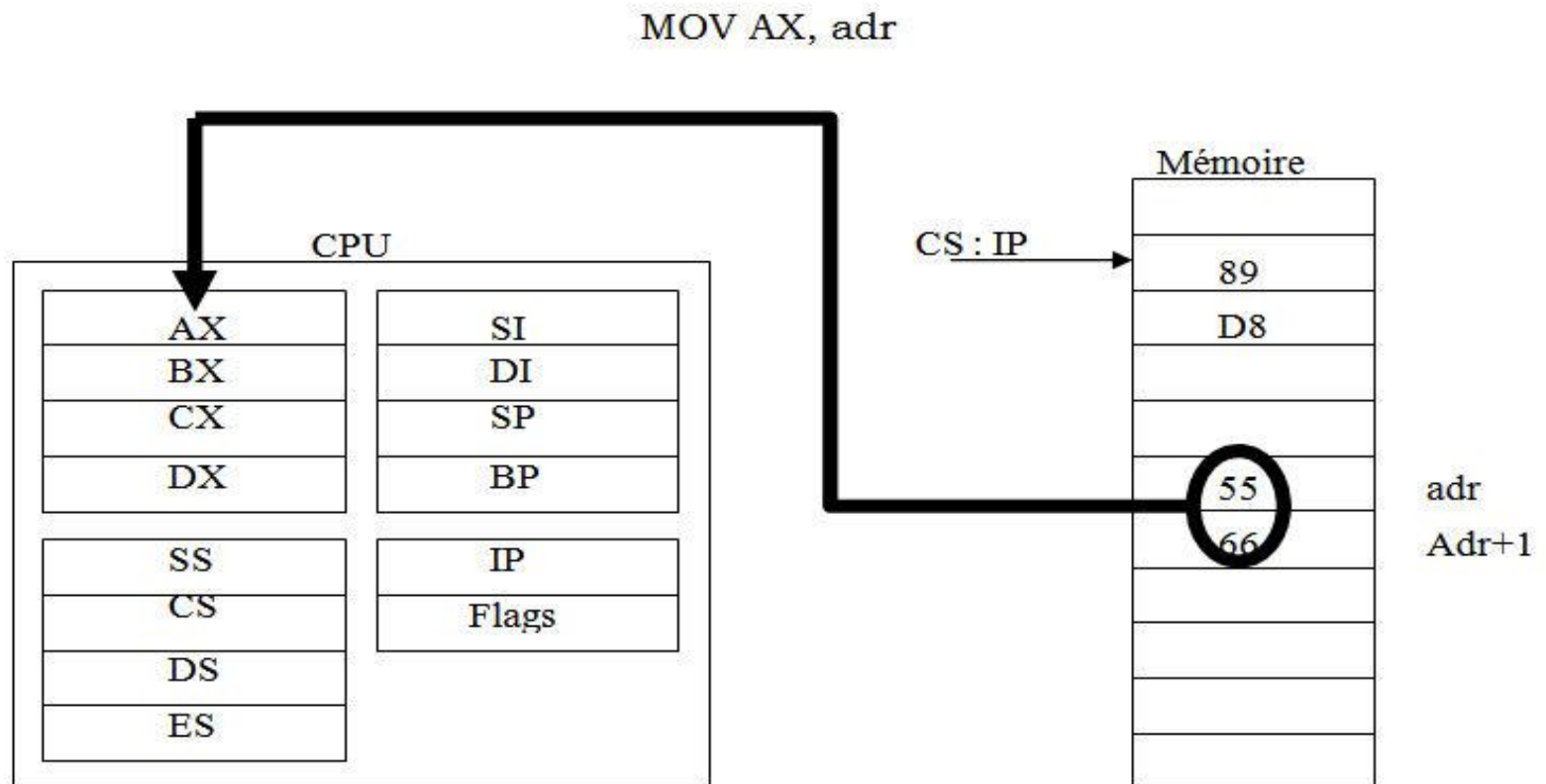
MOV AX, [234] : Copier la case mémoire d'adresse DS:234 dans AX

MOV [123], AX ; Copier AX dans la mémoire d'adresse DS:243

MOV AX, [SS:234] ; Copier dans AX le contenu de la case mémoire SS:234

# Mode d'adressage direct

226



# Mode d'adressage registre indirect

227

UN Des deux opérandes se trouve en mémoire. L'offset de l'adresse n'est pas précisé directement dans l'instruction, il se trouve dans l'un des 4 registres d'offset BX, BP, SI, ou DI et c'est le registre qui sera précisé dans l'instruction: [Rseg:Roff]. Si Rseg n'est pas précisé, le segment par défaut sera utilisé

INST R;[Rseg:Roff]

INST [RSEG:Roff], R

INST taille [Rseg:Roff], im

Exemple :

MOV BX,offset adr

MOV AX, [BX] ; charger AX par la mémoire d'adresse DS:BX

MOV AX, [BP] ; charger AX par la mémoire d'adresse SS:BP

MOV AX, [SI] ; charger AX par la mémoire d'adresse DS:SI

MOV AX, [DI] ; charger AX par la mémoire d'adresse DS:DI

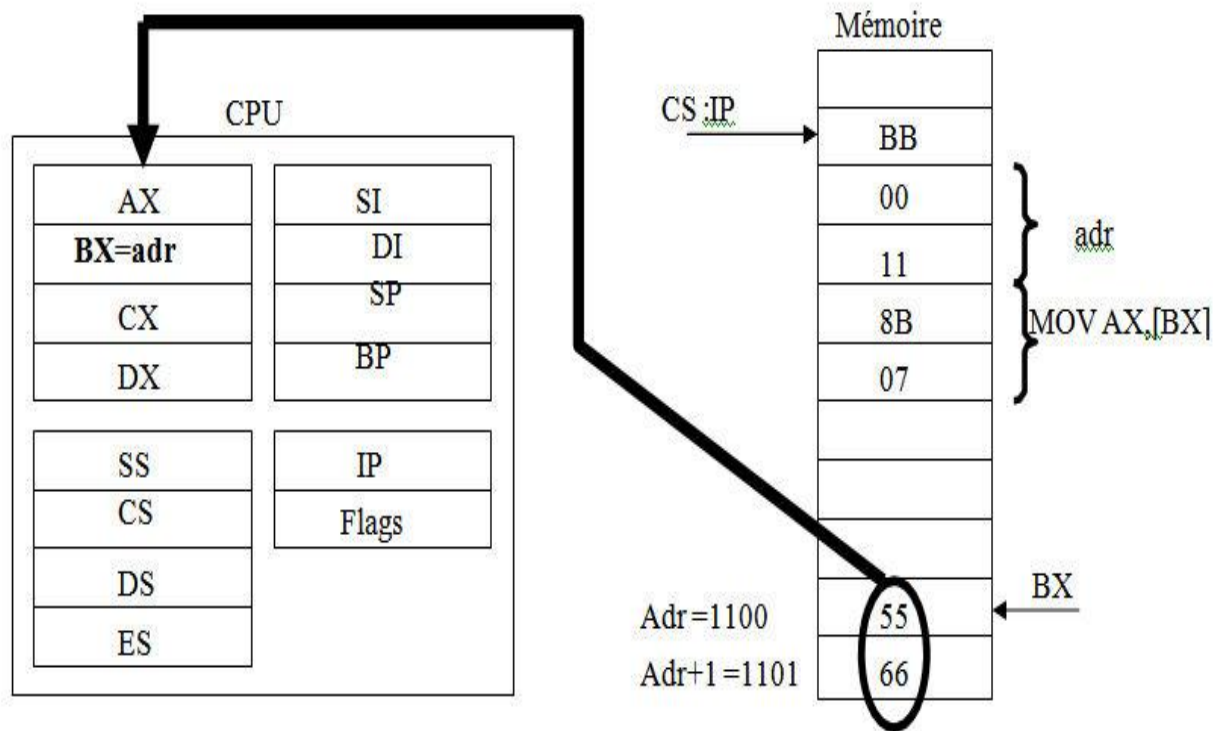
MOV AX, [ES:BP] ; charger AX par la mémoire d'adresse ES:BP

Le contenu de la case mémoire dont l'adresse se trouve dans le registre BX (c.a.d : Adr) est mis dans le registre AX

Remarque: Le symbole [ ] design l'adressage indirect.

# Mode d'adressage registre indirect

228



# Adressage Indirect: Adressage Basé

229

L'offset se trouve dans l'un des deux registres de base BX ou BP. On peut préciser un déplacement qui sera ajouté au contenu de Roff pour déterminer l'offset,

INST R, [Rseg:RB+dep]

INST [Rseg:Rb+dep], R

INST taille[Rseg:Rb + dep], im

Exemples:

MOV AX,[BX] ; Charger AX par la mémoire d'adresse DS:BX

MOV AX,[BI+5] ; Charger AX par la mémoire d'adresse DS:BX+5

MOV AX,[BP-200] ; Charger AX par la mémoire d'adresse SS:BP-200

MOV AX,[ES:BP] ; Charger AX par la mémoire d'adresse ES:BP

# Adressage Indirect: Adressage Indexé(X)

230

L'offset se trouve dans l'un des 2 registres d'index SI ou DI. On peut préciser un déplacement qui sera ajouter au contenu de Ri pour déterminer l'offset

INST R, [Rseg:Ri+dep]

INST [Rseg:Ri+dep], R

INST taille [Rseg:Ri+dep], im

Exemples:

MOV AX,[SI] ; Charger AX par la mémoire d'adresse DS:SI

MOV AX,[SI+500] ; Charger AX par la mémoire d'adresse DS:SI+500

MOV AX,[DI-8] ; Charger AX par la mémoire d'adresse DS:DI-8

MOV AX,[ES:SI+4] ; Charger AX par la mémoire d'adresse ES:SI+4

# Adressage Indirect: Adressage Basé Indexé(X)

231

L'offset de l'adresse de l'opérande est la somme d'un registre de base, d'un registre d'index et d'un registre optionnel.

Si Rseg n'est pas spécifié, le segment par défaut du registre de base est utilisé:

INST R, [Rseg:Bp+Ri+dep]

INST [Rseg:Rb+Ri+dep], R

INST taille [Rseg:Rb+Ri+dep], im

Exemples:

MOV AX, [BX+SI]; AX est chargée par la mémoire d'adresse DS:BX+SI

MOV AX, [BX+DI+5] ; AX est chargé par la mémoire d'adresse DS:BX+DI+5

MOV AX, [BX+SI-8] ; AX est chargé par la mémoire d'adresse SS:BP+SI-8

MOV AX,[BP + DI] ; AX est chargé par la mémoire d'adresse SS:BP+DI