

ARCHITECTURE DES ORDINATEURS ET PROGRAMMATION

2009/2010

MEZAACHE SALAH EDDINE

DÉPARTEMENT D'ÉLECTRONIQUE

CENTRE UNIVERSITAIRE DE BORDJ BOU ARRÉRIDJ

Mez_salah@hotmail.com

PARTIE 1

INTRODUCTION À L'ÉLECTRONIQUE NUMÉRIQUE

Sommaire

1 REPRÉSENTATION DES DONNÉES.....	5
1.1 INTRODUCTION.....	5
1.2 ÉLÉMENTS DE LOGIQUE COMBINATOIRE.....	5
1.2.1 CIRCUITS COMBINATOIRES : DÉFINITION.....	5
1.2.2 TABLES DE VÉRITÉ.....	5
1.2.3 ALGÈBRE ET OPÉRATEURS DE BASE.....	6
1.2.4 DE LA TABLE DE VÉRITÉ À LA FORMULE ALGÈBRIQUE.....	7
1.2.5 THÉORÈMES DE L'ALGÈBRE DE BOOLE.....	7
1.2.6 AUTRES PORTES LOGIQUES.....	8
1.2.7 LE MULTIPLEXEUR.....	9
1.3 MÉTHODES DE SIMPLIFICATION DES FONCTIONS COMBINATOIRES.....	9
1.3.1 TABLES DE KARNAUGH.....	9
1.4 MÉTHODE DE QUINE-MC CLUSKEY.....	14
1.5 ÉLÉMENTS DE LOGIQUE SÉQUENTIELLE.....	16
1.5.1 NOTION D'ÉTAT.....	17
1.5.2 LATCH (BASCULE) RS.....	17
1.5.3 GRAPHE D'ÉTATS.....	18
1.5.4 FRONTS ET NIVEAUX ; SIGNAUX D'HORLOGE.....	18
1.5.5 CIRCUITS SÉQUENTIELS SYNCHRONES ET ASYNCHRONES : DÉFINITIONS.....	18
1.5.6 BASCULES SYNCHRONES.....	19
1.5.7 LATCH RS ACTIF SUR UN NIVEAU D'HORLOGE.....	19
1.5.8 LES BASCULES D.....	19
1.5.9 LOGIQUE SUR FRONT.....	20
1.5.10 ÉQUATION D'ÉVOLUTION.....	20
1.5.11 BASCULE T.....	20
1.5.12 BASCULE JK.....	21
1.6 SYNTHÈSE D'UN CIRCUIT SÉQUENTIEL SYNCHRONE.....	22
1.6.1 GRAPHE D'ÉTATS.....	22
1.6.2 TRANSFORMATION D'UN GRAPHE DE MOORE EN GRAPHE DE MEALY.....	24
1.6.3 TABLES DE TRANSITIONS.....	24
1.6.4 SIMPLIFICATION DES TABLES DE TRANSITION.....	25
1.6.5 ÉTAPES DE LA SYNTHÈSE D'UN CIRCUIT SÉQUENTIEL SYNCHRONE.....	26
1.6.6 SYNTHÈSE DU DÉTECTEUR DE SÉQUENCE, VERSION MOORE.....	27
2 INTRODUCTION À L'ARCHITECTURE.....	32
2.1 INTRODUCTION.....	32
2.2 ARCHITECTURE DE BASE D'UN ORDINATEUR.....	32
2.3 PRINCIPES DE FONCTIONNEMENT.....	32
2.4 ORGANISATION DE LA MÉMOIRE CENTRALE.....	32
2.5 CIRCULATION DE L'INFORMATION DANS UN CALCULATEUR.....	33
2.6 LE PROCESSEUR CENTRAL.....	34
2.6.1 LES REGISTRES ET L'ACCUMULATEUR.....	34
2.6.2 ARCHITECTURE D'UN PROCESSEUR À ACCUMULATEUR.....	35
2.7 LES MÉMOIRES.....	36
2.7.1 SCHÉMA FONCTIONNEL D'UNE MÉMOIRE.....	37
2.7.2 INTERFAÇAGE MICROPROCESSEUR/MÉMOIRE.....	38
2.8 MICROPROCESSEUR INTEL 8086.....	38
2.8.1 JEU D'INSTRUCTION.....	40
2.8.1.1 TYPES D'INSTRUCTIONS.....	40

2.8.1.2 LES INSTRUCTIONS ARITHMÉTIQUES.....	41
2.8.1.3 LES INSTRUCTIONS LOGIQUES.....	42
2.8.1.4 LES INSTRUCTIONS DE BRANCHEMENT.....	43
2.9 LES INTERFACES D'ENTRÉES/SORTIES.....	44
2.9.1 GESTION DES PORTS D'E/S PAR LE 8086.....	45
3 ALGORITHMIQUE ET LANGAGE ÉVOLUÉS.....	47
3.1 ALGORITHMIQUE.....	47
3.2 ALGORITHMIQUE.....	47
3.3 PROGRAMMATION ET LANGAGE.....	47
3.4 DÉFINITIONS.....	48
3.4.1 ALGORITHME.....	48
3.4.2 ORGANIGRAMME.....	48
3.5 QUALITÉ D'UN ALGORITHME.....	48

1 Représentation des données

1.1 Introduction

Les informations traitées par un ordinateur peuvent être de différents types (texte, nombres, etc.) mais elles sont toujours représentées et manipulées par l'ordinateur sous forme binaire. Toute information sera traitée comme une suite de 0 et de 1. L'unité d'information est le chiffre binaire (0 ou 1), que l'on appelle bit (pour binary digit, chiffre binaire).

Le codage d'une information consiste à établir une correspondance entre la représentation externe (habituelle) de l'information (le caractère A ou le nombre 36 par exemple), et sa représentation interne dans la machine, qui est une suite de bits.

On utilise la représentation binaire car elle est simple, facile à réaliser techniquement à l'aide de bistables (système à deux états réalisés à l'aide de transistors). Enfin, les opérations arithmétiques de base (addition, multiplication etc.) sont faciles à exprimer en base 2 (noter que la table de multiplication se résume à $0 \times 0 = 0$, $1 \times 0 = 0$ et $1 \times 1 = 1$).

1.2 Éléments de logique combinatoire

En 1854, **Georges Boole** publia son ouvrage séminal sur une algèbre manipulant des informations factuelles vraies ou fausses. Son travail a été redécouvert et développé sous la forme que nous connaissons maintenant par **Shannon**. Le nom de Boole est entré dans le vocabulaire courant, avec l'adjectif booléen qui désigne ce qui ne peut prendre que deux valeurs distinctes 'vrai' ou 'faux'.

1.2.1 Circuits combinatoires : définition

Un circuit combinatoire est un module tel que l'état des sorties ne dépend que de l'état des entrées. L'état des sorties doit être évalué une fois que les entrées sont stables, et après avoir attendu un temps suffisant à la propagation des signaux. On comprend intuitivement que les circuits combinatoires correspondent à des circuits sans état interne : face à la même situation (les mêmes entrées), ils produisent toujours les mêmes résultats (les mêmes sorties).

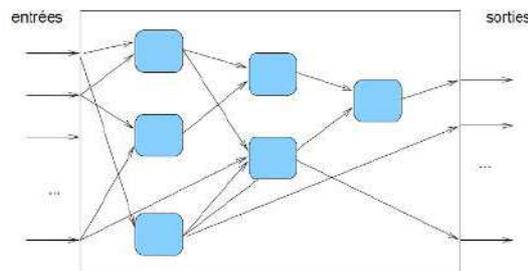


Figure 1.1 Circuits combinatoires

1.2.2 Tables de vérité

Une des contributions essentielles de Boole est la table de vérité, qui permet de capturer les relations logiques entre les entrées et les sorties d'un circuit combinatoire sous une forme tabulaire.

Considérons par exemple un circuit inconnu possédant 2 entrées A et B et une sortie S. Nous pouvons analyser exhaustivement ce circuit en lui présentant les $2^2 = 4$ jeux d'entrées différents, et en mesurant à chaque fois l'état de la sortie. Le tout est consigné dans un tableau qu'on appelle table de vérité du circuit (Figure 1.2)

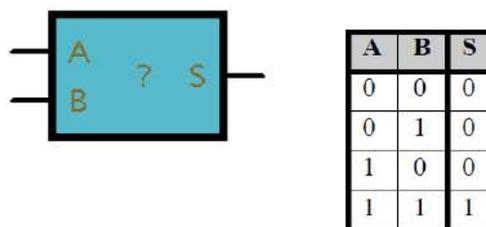


Figure 1.2 Un circuit à deux entrées et une sortie, et sa table de vérité

Ici, on reconnaît l'opération logique ET : S vaut 1 si et seulement si A et B valent 1.

La construction d'une table de vérité est bien sûr généralisable à un nombre quelconque n d'entrées et un nombre m de sorties. Elle possédera alors 2^n lignes, ce qui n'est praticable que pour une valeur de n assez petite.

1.2.3 Algèbre et opérateurs de base

Plutôt que de décrire en extension la relation entre les entrées et une sortie, on peut la décrire en intention à l'aide d'une formule de calcul utilisant seulement trois opérateurs booléens : **NON**, **ET**, **OU**.

Opérateur NON

NON (NOT en anglais) est un opérateur unaire, qui inverse son argument. On notera généralement $\text{NOT}(A) = \bar{A}$; La table de vérité et le schéma représentatif sont donnés par la Figure 1. 3.

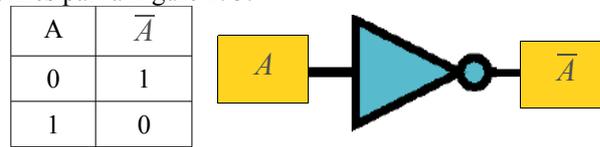


Figure 3 Table de vérité du NON et dessin de la porte correspondante

On a bien sur $\overline{\bar{A}} = A$.

Opérateur ET

ET (AND en anglais) est un opérateur à 2 entrées ou plus, dont la sortie vaut 1 si et seulement si toutes ses entrées valent 1. On le note algébriquement comme un produit, c'est à dire $S = A \times B$ ou $S = AB$. La table de vérité d'un ET à deux entrées, et le dessin usuel de la porte correspondante sont représentés par la Figure 1. 4.

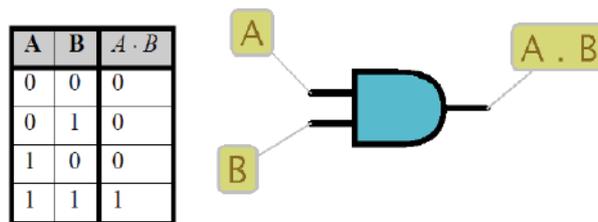


Figure 1. 4 Table de vérité du ET, et symbole de la porte correspondante

De par sa définition, le ET est associatif : $A \times B \times C = (A \times B) \times C = A \times (B \times C)$. Il est également idempotent : $A \times A = A$.

Opérateur OU

OU (OR en anglais) est un opérateur à 2 entrées ou plus, dont la sortie vaut 1 si et seulement si une de ses entrées vaut 1. On le note algébriquement comme une somme, c'est à dire $S = A + B$. La table de vérité d'un OU à deux entrées, et le dessin usuel de la porte correspondante sont représentés Figure 1. 5.

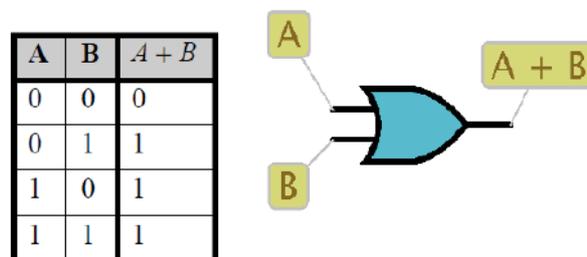


Figure 1. 5 Table de vérité du OU, et symbole de la porte correspondante

De par sa définition, le OU est associatif : $A + B + C = (A + B) + C = A + (B + C)$. Il est également idempotent : $A + A = A$.

1.2.4 De la table de vérité à la formule algébrique

On peut automatiquement écrire la formule algébrique d'une fonction booléenne combinatoire dès lors qu'on a sa table de vérité. Il suffit de considérer seulement les lignes qui donnent une sortie égale à 1, d'écrire le minterm correspondant, qui code sous forme d'un ET la combinaison de ces entrées. Par exemple, le minterm associé au vecteur $(A, B, C) = (1, 0, 1)$ est : $A\bar{B}C$. La formule algébrique qui décrit toute la fonction est le OU de tous ces minterms.

Considérons par exemple la table de vérité de la fonction majorité à 3 entrées, qui vaut 1 lorsqu'une majorité de ses entrées (2 ou 3) vaut 1 (Figure 1. 6).

A	B	C	MAJ(A,B,C)
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

La table de vérité contient 4 lignes avec une sortie S à 1, ce qui donne : $S = \bar{A}BC + A\bar{B}C + \bar{A}BC + ABC$

Cette formule est simplifiable, comme on le verra dans une section ultérieure.

Est-on sûr que la formule obtenue par cette méthode est correcte ? Pour une combinaison des entrées qui doit donner une sortie à 1, la formule possède le minterm correspondant, et donne aussi une valeur de 1. Pour toutes les autres combinaisons d'entrées, la table donne une sortie à 0, et la formule aussi, puisqu'aucun minterm n'est présent qui corresponde à ces combinaisons. La formule donne donc toujours le même résultat que la table.

Corollairement, cela démontre le résultat fondamental suivant :

N'importe quelle fonction combinatoire s'exprime sous forme d'une somme de minterms.

1.2.5 Théorèmes de l'algèbre de Boole

Le tableau de la Figure 1. 6 résume les principaux théorèmes et propriétés de l'algèbre de Boole. On peut les démontrer de façon algébrique, ou en utilisant des tables de vérité.

Les théorèmes de De Morgan permettent de transformer les ET en OU et vice-versa.

Le couple (ET, NON) ou le couple (OU, NON) suffisent donc à exprimer n'importe quelle formule algébrique combinatoire.

Le théorème d'absorption $A + AB = A$ montre qu'un terme plus spécifique disparaît au profit d'un terme moins spécifique. Par exemple dans l'équation $\dots + \bar{A}B + \bar{A}B\bar{C} \dots$, le terme $\bar{A}B\bar{C}$ s'efface au profit de $\bar{A}B$, moins spécifique et qui donc l'inclut.

relation	relation duale	Propriété
$AB = BA$	$A + B = B + A$	Commutativité
$A(B + C) = AB + AC$	$A + BC = (A + B)(A + C)$	Distributivité
$1 \cdot A = A$	$0 + A = A$	Identité
$A\bar{A} = 0$	$A + \bar{A} = 1$	Complément
$0 \cdot A = 0$	$1 + A = 1$	Zéro et un
$A \cdot A = A$	$A + A = A$	Idempotence
$A(BC) = (AB)C$	$A + (B + C) = (A + B) + C$	Associativité
$\overline{\bar{A}} = A$		Involution
$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$	Théorèmes de De Morgan
$A(A + B) = A$	$A + AB = A$	Théorèmes d'absorption
$A + \bar{A}B = A + B$	$A(\bar{A} + B) = AB$	Théorèmes d'absorption

Figure 1. 6 Propriétés et théorèmes de l'algèbre de Boole

1.2.6 Autres portes logiques

Opérateur NAND

NAND (= NOT AND) est un opérateur à 2 entrées ou plus, dont la sortie vaut 0 si et seulement si toutes ses entrées valent 1. On a donc à deux entrées : $\overline{A \cdot B}$. La table de vérité d'un NAND à deux entrées, et le dessin usuel de la porte correspondante sont représentés Figure 1. 7.

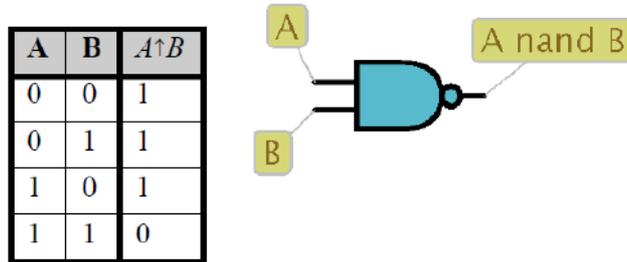


Figure 1. 7 Table de vérité du NAND, et dessin de la porte correspondante

Le NAND est un opérateur dit complet, c'est à dire qu'il permet à lui seul d'exprimer n'importe quelle fonction combinatoire. Il permet en effet de former un NOT, en reliant ses deux entrées : $\overline{A} = \overline{A \cdot A}$. Les théorèmes de De Morgan assurent donc qu'il peut exprimer un ET et un OU, et donc n'importe quelle expression combinatoire.

Opérateur NOR

NOR (= NOT OR) est un opérateur à 2 entrées ou plus, dont la sortie vaut 0 si et seulement au moins une de ses entrées 1. On a donc à deux entrées : $\overline{A + B}$. La table de vérité d'un NOR à deux entrées, et le dessin usuel de la porte correspondante sont représentés Figure 1. 8.

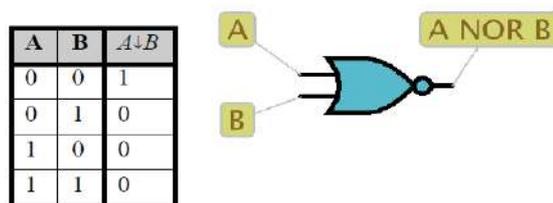


Figure 1. 8 Table de vérité du NOR, et dessin de la porte correspondante

Opérateur XOR

XOR (= EXCLUSIVE OR) est un opérateur à 2 entrées ou plus. On peut le définir de plusieurs façons ; la définition qui permet le plus directement de démontrer ses propriétés est celle qui consiste à en faire un détecteur d'imparité : sa sortie vaut 1 si et seulement si un nombre impair de ses entrées est à 1. On le note ' \oplus '; la table de vérité d'un XOR à deux entrées, et le dessin usuel de la porte correspondante sont représentés Figure 1. 9.

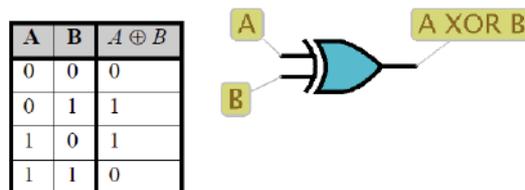


Figure 1. 9 Table de vérité du XOR, et dessin de la porte correspondante

On voit à partir de la table que $A \oplus B = A \overline{B} + \overline{A} B$. Lorsqu'il a deux entrées, il mérite son nom de OU exclusif

puisque sa sortie ne vaut 1 que si l'une de ses entrées vaut 1, mais seulement une. Il possède plusieurs autres propriétés intéressantes :

- lorsqu'on inverse une entrée quelconque d'un XOR, sa sortie s'inverse. C'est évident puisque cela incrémente ou décrémente le nombre d'entrées à 1, et donc inverse la parité.
- le XOR à deux entrées est un inverseur commandé. En effet, en examinant la table de vérité du XOR (Figure 1. 9)), on constate que lorsque A vaut 0, la sortie S reproduit l'entrée B, et lorsque A vaut 1, la sortie S reproduit l'inverse de l'entrée B. L'entrée A peut être alors vue comme une commande d'inversion dans le passage de B vers S.

1.2.7 Le multiplexeur

Le **multiplexeur** est une porte à trois entrées, qui joue un rôle d'aiguillage. Son dessin indique bien cette fonction (Figure 1. 10).

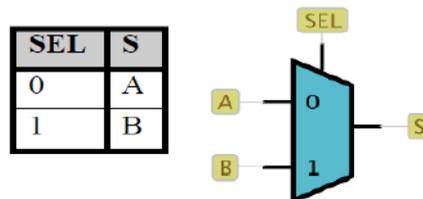


Figure 1. 10 Table de vérité condensée du multiplexeur; et dessin de la porte correspondante

Lorsque la commande de sélection vaut 0, l'aiguillage est du côté de A, et S prend la valeur de A ; lorsqu'elle vaut 1, S prend la valeur de B. Algébriquement, on a donc :

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

À partir de la compréhension du rôle du multiplexeur, il est clair que :

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

1.3 Méthodes de simplification des fonctions combinatoires

1.3.1 Tables de Karnaugh

Une table de Karnaugh est utilisée pour trouver visuellement les simplifications à faire sur une somme de termes. Elle est très facile à utiliser, pour peu qu'il n'y ait pas plus de 4 variables en jeu. Au delà, il vaut mieux employer la méthode de Quine-Mc Cluskey décrite plus bas.

Considérons par exemple la fonction ayant la table de vérité de la Figure 1. 11.

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

Figure 1. 11 Table de vérité d'une fonction à simplifier

On va représenter les valeurs de S dans un tableau 4X4, chaque case correspondant à un des 16 mintermes possibles avec 4 variables A,B,C et D (Figure 1. 12).

		A,B			
		0,0	0,1	1,1	1,0
C,D	0,0			1	
	0,1	1	1	1	1
	1,1	1	1	1	1
	1,0	1		1	

Figure 1. 12 Table de Karnaugh représentant la table de vérité

Avant de poursuivre les explications de la table de Karnaugh, il faut introduire quelques concepts sur lesquels Karnaugh s'est basé pour développer ses tables. Le premier concept est l'utilisation d'une représentation particulière de l'information, il s'agit des codes. Les codes et systèmes représentatifs seront vus en détail aux chapitre 5. Lorsque plusieurs variables logiques sont exprimées ensemble, ce groupe peut être appelé nombre binaire. Par analogie aux nombres décimaux où chaque chiffre est un multiplicateur d'une puissance de dix, chaque chiffre qui le compose est le multiplicateur d'une puissance de deux.

Nombre décimal

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

$$123 = 100 + 20 + 3$$

Nombre binaire

$$123 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$123 = 64 + 32 + 8 + 4 + 2 + 1$$

Cette notation, appelée code binaire naturel, permet de représenter 2^n nombres en utilisant n bits (variables binaires). Avec quatre bits, il est donc possible de représenter 16 nombres allant de 0 à 15.

Le code binaire naturel est utilisé pour énumérer toutes les combinaisons possibles d'entrées lors de la création de la table de vérité d'une fonction.

Le code de Gray, quant à lui, a été élaboré à partir des deux caractéristiques suivantes:

- La transition d'un mot au mot suivant implique qu'un, et seulement un, bit change d'état.
- Le code est cyclique.

Le deuxième concept nécessaire à la compréhension des tables de Karnaugh est l'adjacence logique. Deux mots binaires sont dits adjacents s'ils ne diffèrent que par la complémentarité d'une, et seulement une, variable. Si deux mots sont adjacents sont sommés, ils peuvent être fusionnés et la variable qui diffère est éliminée. Par exemple, les mots ABC et $AB\bar{C}$ sont adjacents puisqu'ils ne diffèrent que par la complémentarité de la variable C. Le théorème stipule donc que $ABC + AB\bar{C} = AB$. La preuve est simple:

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

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

$$AB = AB$$

La première caractéristique du code de Gray spécifie que deux mots consécutifs ne diffèrent que par l'état d'un bit, ils sont donc adjacents.

La méthode de Karnaugh consiste à indiquer dans la table les cases correspondantes aux états de variable d'entrées produisant une sortie vraie. Cela peut être déterminé à partir de l'équation de la fonction ou sa table de vérité. Il faut toutefois porter une attention particulière lors du transfert d'information de la table de vérité puisque celle-ci utilise un code binaire naturel, qui est sensiblement différent du code de Gray de la table de Karnaugh. Lorsque toute la fonction est représentée dans la table, on procède à des regroupements de "1" qui se situent les uns à côté des autres. Puisque la table de Karnaugh utilise un code de Gray, ces groupements identifient des termes adjacents. La Figure 1. suivante identifie certains groupements typiques. Il est important de noter que les groupements sont toujours des rectangles (les carrés sont aussi des rectangles) contenant un nombre de "1" qui est une puissance de deux.

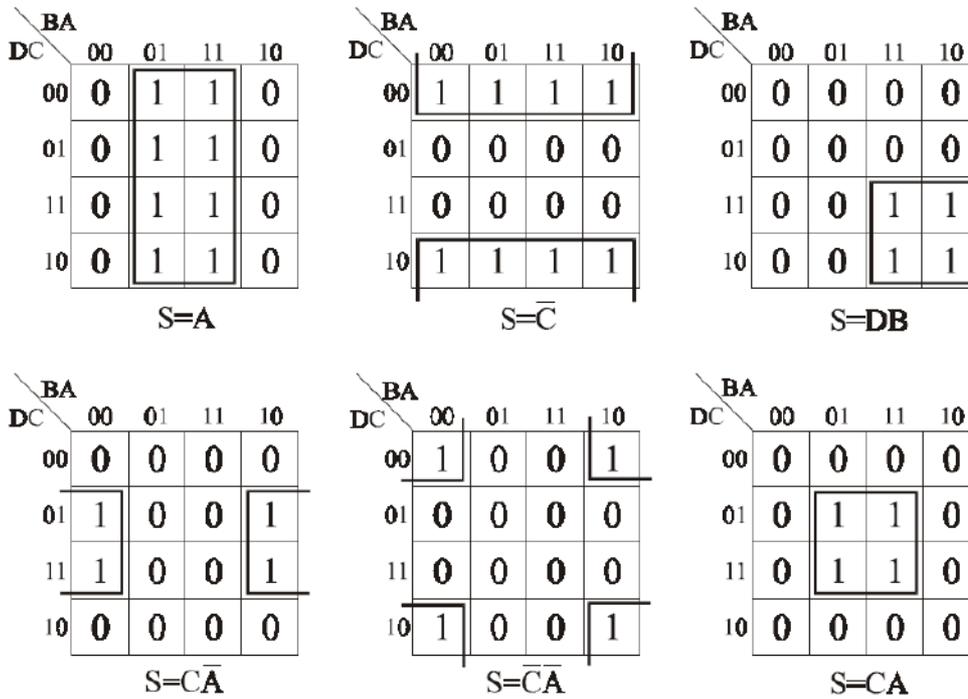


Figure 1.13 Formation d'impliquants dans les tables de Karnaugh

Exemple

On veut simplifier $S = A\bar{C} + \bar{A}B + BC$.

Il faut tout d'abord remplir la table de Karnaugh à l'aide de l'équation de la fonction. Cette étape est très importante puisqu'il faut remplir toutes les cases qui correspondent à une combinaison d'entrées produisant une sortie vraie.

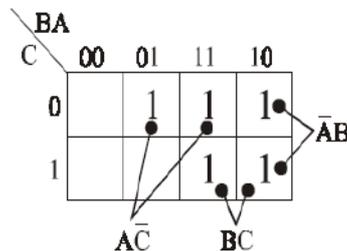


Figure 1.14 Remplissage de la table de Karnaugh

Une fois que la table est remplie, il faut procéder aux groupements. Il est très important d'utiliser tous les "1" de la table sans exceptions.

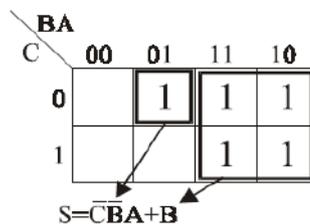


Figure 1.15 Formation des impliquants

La forme obtenue n'est cependant pas la plus compacte. Pour obtenir la forme la plus compacte possible, il faut créer les groupements les plus grands possibles. Notez qu'il est possible d'utiliser les "1" aussi souvent que désiré. Les groupements précédents peuvent alors être exprimés ainsi. La forme présentée à la Figure 1.16 est la plus compacte possible.

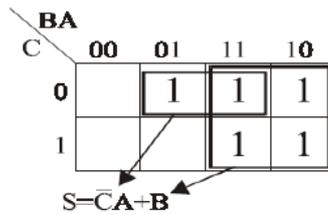


Figure 1. 16 Formation optimale des impliquants

1.4 Méthode de Quine-Mc Cluskey

On va montrer son usage en simplifiant la fonction à 5 variables suivante :

$$f(A, B, C, D, E) = \overline{A}\overline{B}\overline{C}\overline{D}\overline{E} + \overline{A}\overline{B}\overline{C}D\overline{E} + \overline{A}\overline{B}C\overline{D}\overline{E} + \overline{A}\overline{B}CD\overline{E} + \overline{A}B\overline{C}\overline{D}\overline{E} + \overline{A}B\overline{C}D\overline{E} + \overline{A}BC\overline{D}\overline{E} + \overline{A}BCD\overline{E} + ABC\overline{D}\overline{E} + ABCD\overline{E} + ABCDE$$

$$f(A, B, C, D, E) = \overline{A}\overline{B}\overline{C}\overline{D}\overline{E} + \overline{A}\overline{B}\overline{C}D\overline{E} + \overline{A}\overline{B}C\overline{D}\overline{E}$$

Une table de Karnaugh serait délicate à utiliser dans ce cas, car elle aurait une taille de 4 x 8, et certains regroupements possibles pourraient ne pas être connexes.

Par économie d'écriture, on commence par représenter chaque *minterm* par le nombre associé à sa représentation binaire :

$$f(A, B, C, D, E) = \sum (0, 2, 8, 10, 12, 13, 26, 29, 30)$$

On place ensuite ces *minterms* dans un tableau, en les groupant selon le nombre de 1 qu'ils possèdent (Figure 1. 17).

Nombre de 1	minterm	Valeur binaire
0	0	00000
1	2 8	00010 01000
2	10 12	01010 01100
3	13 26	01101 11010
4	29 30	11101 11110

Figure 1. 17 Classement des minterms selon le nombre de 1 de leur valeur binaire

Les simplifications ne peuvent se produire qu'entre groupes adjacents. La simplification entre les minterms 1 et 2 se notera par exemple : 000-0, en mettant un tiret à l'endroit où la variable a disparu. Lorsqu'on a effectué toutes les simplifications possibles, on recommence à partir des nouveaux groupes formés, jusqu'à ce qu'aucune simplification ne soit possible (Figure 1. 18).

étape 1			étape 2			étape 3	
0	00000	✓	0-2	000-0	✓	0-2-8-10	0-0-0
2	00010	✓	0-8	0-000	✓		
8	01000	✓	2-10	0-010	✓		
10	01010	✓	8-10	010-0	✓		
12	01100	✓	8-12	01-00			
13	01101	✓	10-26	-1010			
26	11010	✓	12-13	0110-			
29	11101	✓	13-29	-1101			
30	11110	✓	26-30	11-10			

Figure 1. 18 On simplifie entre groupes adjacents, étape par étape. Les termes qui ont été utilisés dans une simplification sont cochés

Maintenant, il suffit de choisir dans ce tableau les termes qui vont inclure tous les minterms de départ, en essayant de trouver ceux qui correspondent à l'expression la plus simple. On peut le faire de façon purement intuitive, en commençant par les termes les plus à droite : par exemple les groupes 0-2-8-10, 8-12, 13-29, 26-30 vont inclure tous les minterms.

On trouve donc :

$$f(A, B, C, D, E) = \overline{A}\overline{C}\overline{E} + \overline{A}B\overline{D}\overline{E} + BC\overline{D}E + ABDE$$

Dans les cas complexes, ou si on veut programmer cet algorithme, on énumère la liste des implicants premiers, qui sont les termes du tableau précédent qui n'ont été utilisés dans aucune simplification. Dans le tableau précédent, on a coché avec une '3' les termes qui ont été utilisés dans une simplification, donc la liste des implicants premiers est : 8-12, 10-26, 12-13, 13-29, 26-30, 0-2-8-10. Il est clair que le résultat simplifié ne comportera que des termes de cette liste, et le but est de déterminer parmi eux ceux qu'il est indispensable de placer dans le résultat, appelés implicants premiers essentiels. On construit pour cela la table des implicants premiers (Figure 1. 19).

		0	2	8	10	12	13	26	29	30
8-12	01-00			✓		✓				
10-26	-1010				✓			✓		
12-13	0110-					✓	✓			
13-29	-1101	*					✓		✓	
26-30	11-10	*						✓		✓
0-2-8-10	0-0-0	*	✓	✓	✓	✓				

Figure 1. 19 Table des implicants premiers ; les implicants premiers sont en lignes et les minterms de départ sont en colonnes. Un implicant premier est dit essentiel s'il est le seul à couvrir un des minterms en colonne ; on le repère avec une marque '*'. *

Parmi les 5 minterms placés en ligne dans le tableau, 3 sont dits essentiels, parce qu'ils sont les seuls à couvrir un des minterms placés en colonnes : l'implicant premier 13-29 est le seul à couvrir le minterm 29, 26-30 est le seul à couvrir 26 et 30, et 0-2-8-10 est le seul à couvrir les minterms 0 et 2. 8-12 et 12-13 ne sont pas essentiels, car 8, 12 et 13 sont couverts par d'autres implicants premiers.

Les implicants premiers essentiels sont nécessaires, mais pas forcément suffisants. Ici par exemple, si on ne prenait qu'eux, le minterm 12 ne serait pas couvert. Il reste donc une phase heuristique de choix parmi les implicants premiers non essentiels pour couvrir tous les minterms non encore couverts. Dans notre exemple, on peut ajouter, soit 8-12, soit 12-13 pour couvrir le 12 manquant, ce qui donne les deux possibilités suivantes :

version avec 8-12 : $f(A, B, C, D, E) = \overline{A}\overline{C}\overline{E} + \overline{A}B\overline{D}\overline{E} + BC\overline{D}E + ABD\overline{E}$

version avec 12-13 : $f(A, B, C, D, E) = \overline{A}\overline{C}\overline{E} + \overline{A}BC\overline{D} + BC\overline{D}E + ABD\overline{E}$

1.5 Éléments de logique séquentielle

Un circuit est dit séquentiel, si les valeurs de ses sorties ne dépendent pas que des valeurs de ses entrées.

C'est donc le contraire d'un circuit combinatoire, dont les valeurs des sorties ne dépendaient que de celles de ses entrées, avec une propagation sans retour arrière des signaux des entrées vers les sorties.

À l'inverse, les sorties d'un circuit séquentiel dépendent non seulement des valeurs des entrées au moment présent, mais aussi d'un état interne qui dépend de l'historique des valeurs d'entrées précédentes. En vertu de ce que nous avons démontré pour les circuits combinatoires, cela implique un rebouclage vers l'arrière de certains signaux internes au circuit (Figure 1. 20).

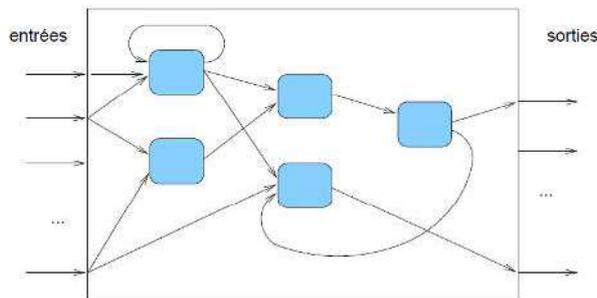


Figure 1. 21 Circuit séquentiel : certains signaux internes ou sorties rebouclent en arrière

Ce type de circuit peut donner lieu à des fonctionnements très complexes ; il peut également être instable dans certaines configurations. Plus encore que pour les circuits combinatoires, des méthodes pour maîtriser leur complexité sont nécessaires.

1.5.1 Notion d'état

Prenons l'exemple suivant : on considère un système à 1 entrée e et une sortie S . La sortie S du système doit changer de valeur à chaque front montant de l'entrée e . Ce cahier des charges peut être représenté par le chronogramme suivant :

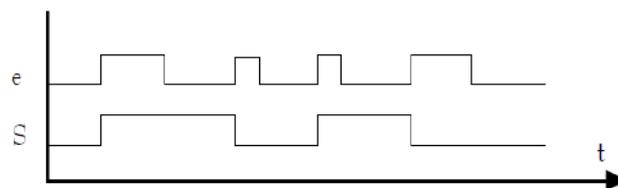


Figure 1. 22 Chronogramme du système

Pour une même valeur de e , S peut prendre deux valeurs 0 ou 1. Ce système n'est pas combinatoire: on ne peut pas définir $S = f(e)$.

Par contre la valeur de S peut être déterminée en utilisant ce qui s'est passé auparavant. Le système a en mémoire la valeur de S avant changement. La réalisation de ce système nécessiterait des bascules.

Un *système séquentiel* est un système dont les sorties à l'instant t dépendent à la fois des entrées à cet instant, mais aussi de ce qui s'est passé auparavant : l'histoire du système. Cette histoire sera représentée par une succession d'*états* que prend le système au cours du temps. Le changement d'état sera provoqué par une variation des entrées. Les sorties sont fonction de l'état du système.

Un système séquentiel pourra être représenté par le schéma de la Figure 1. 23.

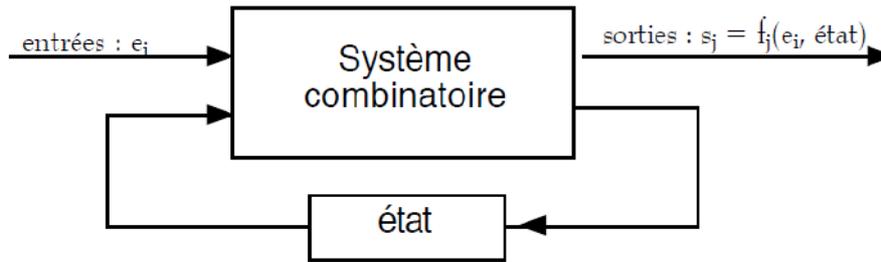


Figure 1. 23 Représentation d'un système séquentiel

1.5.2 Latch (Bascule) RS

La notion de circuit séquentiel avec ses états internes incorpore la notion de mémoire (retour sur son passé). La cellule mémoire la plus simple est le latch RS (RS étant les initiales de Reset-Set), appelé aussi bistable ou flip-flop (Figure 1. 24).

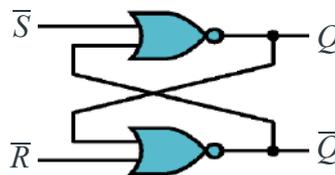


Figure 1. 24 Schéma latch RS

En entrée, les signaux Set et Reset, actifs à l'état bas, commandent le système.

- Lorsque Set devient actif, le système change d'état et Q prend pour valeur 1.
- Lorsque Set devient inactif, le système conserve son état et Q reste à 1.
- Reset à l'effet inverse sur le système. Les deux signaux de contrôle ne peuvent être actifs en même temps sans quoi les sorties ne seraient plus complémentaires.

La table de vérité de ce circuit est la suivante : (on note 't' l'instant avant modification des entrées).

\bar{S}	\bar{R}	Q_{t+1}
0	0	NA
0	1	1
1	0	0
1	1	Q_t

Ce circuit est bien séquentiel : ses sorties ne dépendent pas uniquement de ses entrées. La notion d'état interne est ici clairement présente sous la forme d'un bit mémorisé.

1.5.3 Graphe d'états

Les modes de fonctionnement décrits précédemment peuvent être résumés dans le graphe de la Figure 1. 25.

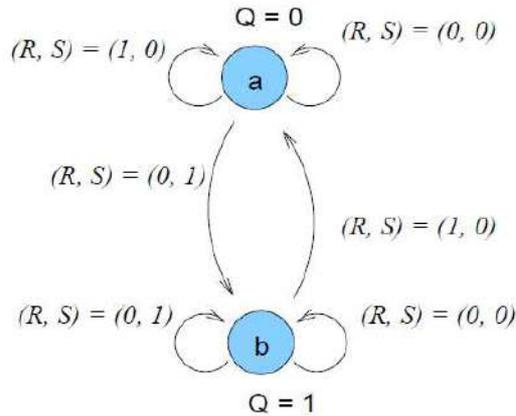


Figure 1.25 Graphe d'états d'un latch RS. Les nœuds correspondent aux états stables du circuit, et les arcs représentent les transitions entre les états

1.5.4 Fronts et niveaux ; signaux d'horloge

Le niveau d'un signal, c'est sa valeur 0 ou 1. On parle de front lors d'un changement de niveau : front montant lorsque le signal passe de 0 à 1, front descendant lorsqu'il passe de 1 à 0. En pratique, ces transitions ne sont pas tout à fait instantanées, mais leur durée est très inférieure aux temps de propagation des portes (Figure 1.26).

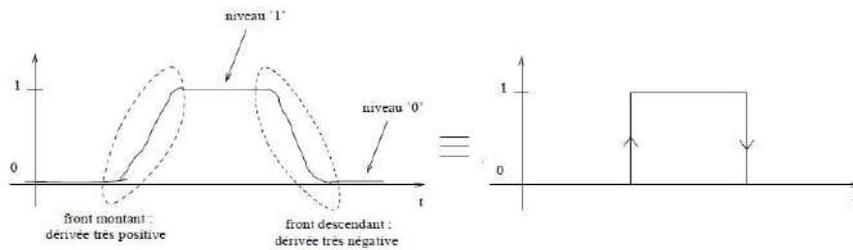


Figure 1.26 Fronts et niveaux d'un signal réel, et leur équivalent logique

Une horloge *simplifié* est un signal périodique Il produit donc une succession périodique de fronts montants et/ou descendants.

1.5.5 Circuits séquentiels synchrones et asynchrones : définitions

Circuits asynchrones purs

On appelle circuits séquentiels asynchrones purs des circuits séquentiels sans signaux d'horloge, et dans lesquels ce sont les changements de valeur des entrées qui sont à la source des changements d'états. Le latch RS est un exemple de circuit asynchrone pur : il n'est pas gouverné par une horloge, et ce sont les changements des entrées qui provoquent les changements d'état.

Circuits synchrones purs

On appelle circuit séquentiels synchrones purs des circuits séquentiels qui possèdent une horloge unique, et dont l'état interne se modifie précisément après chaque front (montant ou descendant) de l'horloge.

1.5.6 Bascules synchrones

Notre but est maintenant de réaliser à l'aide de circuits logiques les machines séquentielles telles qu'elles peuvent être décrites par les graphes d'états. Pour faciliter cette synthèse, il a été imaginé de créer des composants appelés bascules.

Une bascule mémorise un seul bit et permet de représenter seulement deux états distincts, mais l'utilisation d'un vecteur de n bascules permet de coder jusqu'à 2^n états distincts.

Par ailleurs, pour que tout un vecteur de bascules évolue de façon fiable d'état en état, il est souhaitable que chacune d'elles ait une entrée d'horloge reliée à une horloge commune (circuit synchrone), et ne puisse changer d'état qu'au

moment précis d'un front de cette horloge (montant ou descendant), et jamais entre deux fronts, même si les autres entrées du circuit sont modifiées plusieurs fois entre temps.

1.5.7 Latch RS actif sur un niveau d'horloge

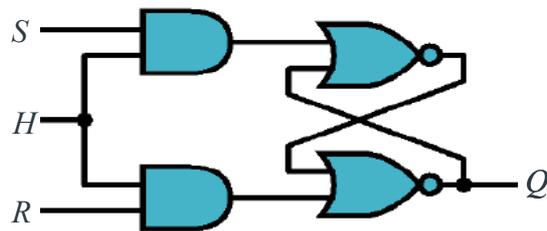


Figure 1. 27 Latch RS

L'état de ce circuit est modifiable lorsque $H = 1$, c'est à dire sur un niveau du signal H. Tant que H est au niveau 1, les modifications de S et de R vont avoir un effet sur l'état du bistable. Un tel latch peut avoir certaines applications, mais il n'est pas adapté à la réalisation de circuits séquentiels synchrones, où les changements d'états de leurs différentes parties doivent se produire de façon parfaitement synchronisée, au moment précis défini par un front d'une horloge commune.

1.5.8 Les bascules D

Les bistables RS sont tout à fait utilisables pour la mémorisation, toutefois, ils nécessitent la connaissance de l'état à mémoriser de sorte à passer la bonne commande Set ou Reset. Plus généralement, le besoin est la mémorisation d'une valeur inconnue à partir d'un instant donné.

Il a donc été créé à partir d'un bistable RS, un système ne nécessitant qu'une seule entrée pour fixer le niveau à mémoriser. Ce système a en outre un second signal permettant la commande de mémorisation.

On a en fait ajouté un inverseur entre Set et Reset pour n'avoir plus qu'un seul signal d'entrée. Le bistable ainsi créé est appelé bascule D transparente. On notera que dans un tel montage, il n'y a plus de combinaisons d'entrées invalides.

D	C	Q_{t+1}
0	0	Q_t
1	0	Q_t
0	1	0
1	1	1

Tableau de vérité de la bascule D

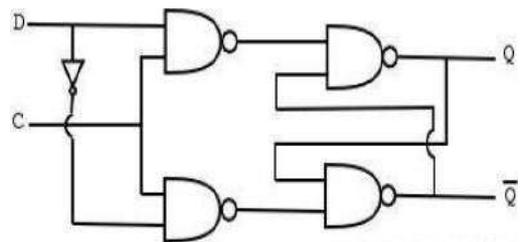


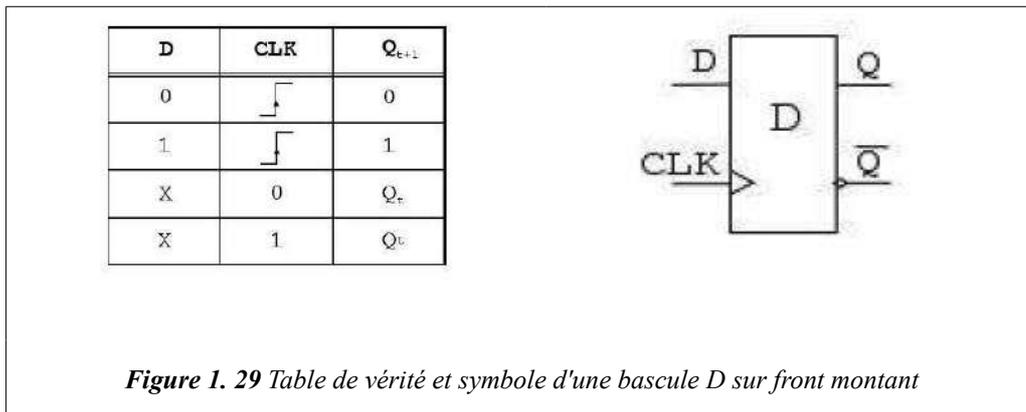
Figure 1. 28 Schéma d'une bascule D

1.5.9 Logique sur front

Dans le cas d'un bistable D, la sortie Q copie le niveau de D durant toute la période d'activité du signal C. Dès que C devient inactif, et aussi longtemps qu'il le reste, le dernier niveau est mémorisé. Ce comportement est celui de la mémoire statique élémentaire, répandue dans tous les systèmes logiques.

Un comportement plus évolué est parfois souhaitable. Le fait que Q puisse changer lorsque C est actif peut être gênant dans certains cas. Pour modifier ce comportement, la bascule synchrone a été définie. Dans une telle bascule la sortie Q correspond à l'état de D au moment précis où C est passé de l'état inactif à l'état actif. Q ne change pas tant que C ne change pas, que le niveau soit 0 ou 1. On parle donc de logique sur front car seul un front d'horloge, et non pas un niveau, a une influence sur l'état du système.

Pour des raisons de testabilité des circuits, le signal C est généralement issu d'une horloge. Ce signal sera donc plus généralement appelé H ou encore CLK. Alors, le symbole utilisé est légèrement différent : un triangle au niveau de l'entrée CLK vient en indiquer un fonctionnement sur front.



1.5.10 Équation d'évolution

L'équation d'évolution d'une bascule, c'est l'équation de la valeur que prendra l'état après le front d'horloge. Dans le cas de la bascule D c'est tout simplement la valeur présente à son entrée : $Q \leftarrow D$

1.5.11 Bascule T

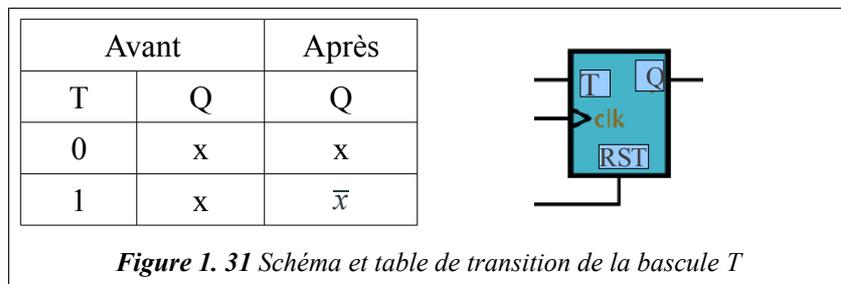
Tout comme la bascule D, la bascule T (**Trigger**) mémorise également un bit de façon synchrone, mais elle a des modalités différentes d'utilisation. Son schéma et sa table de transitions synthétique associée sont donnés par la Figure 1.30.

L'état mémorisé de la bascule T s'inverse (après le front d'horloge) si et seulement si son entrée T vaut 1. Lorsque son entrée T vaut 0, cet état reste inchangé. Elle est donc adaptée aux problématiques de changements d'une valeur et non à un simple stockage comme la bascule D.

ÉQUATION D'ÉVOLUTION

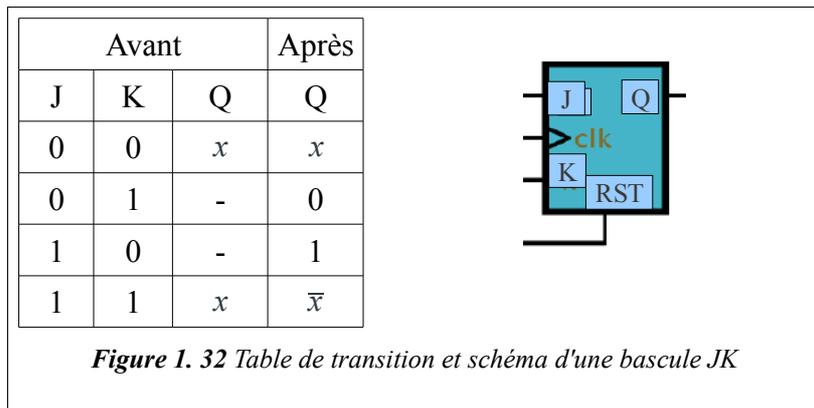
$$Q \leftarrow T Q + T \bar{Q}$$

Le premier terme exprime bien que, si T est à 0, Q ne change pas, et que si T est à 1, Q s'inverse.



1.5.12 Bascule JK

Comme les bascules D et T, la bascule JK (Jack-Kilby; le nom d'un des inventeurs des circuits intégrés) mémorise un bit de façon synchrone. Son schéma, sa table de transitions simplifiée et l'écriture SHDL associée sont donnés Figure 1.32.



Équation d'évolution

$$Q = \bar{K}Q + J\bar{Q}$$

On vérifie que dans tous les cas, on a le résultat attendu :

$$- \text{si } J=0 \text{ et } K=0, \bar{K}Q + J\bar{Q} = 0 + Q = Q$$

$$- \text{si } J=0 \text{ et } K=1, \bar{K}Q + J\bar{Q} = 0 + 0 = 0$$

$$- \text{si } J=1 \text{ et } K=0, \bar{K}Q + J\bar{Q} = Q + \bar{Q} = 1$$

$$- \text{si } J=1 \text{ et } K=1, \bar{K}Q + J\bar{Q} = 0 + \bar{Q} = \bar{Q}$$

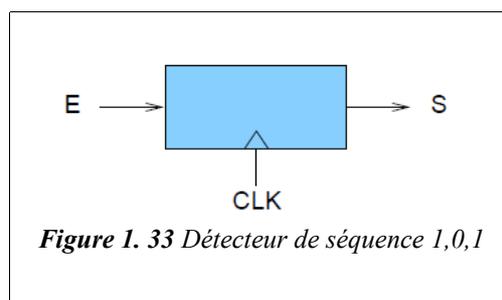
La bascule D est clairement adaptée aux situations où on doit stocker un bit présent. La bascule T est adaptée aux situations qui se posent en termes de changement ou d'inversion. La bascule JK semble plus versatile, et elle est d'ailleurs souvent appelée *bascule universelle*. Par analogie avec les portes combinatoires, on pourrait se demander si cette bascule JK est 'plus puissante' que les bascules D ou T, c'est à dire s'il est possible de fabriquer avec elle des circuits que les autres ne pourraient pas faire. La réponse est non.

Ces trois types de bascules ont une puissance d'expression équivalente : elle mémorisent un bit, et elles ne diffèrent que par les modalités de stockage de ce bit. Par exemple, on vérifie facilement qu'on peut fabriquer une bascule JK avec une bascule T et réciproquement.

1.6 Synthèse d'un circuit séquentiel synchrone

1.6.1 Graphes d'états

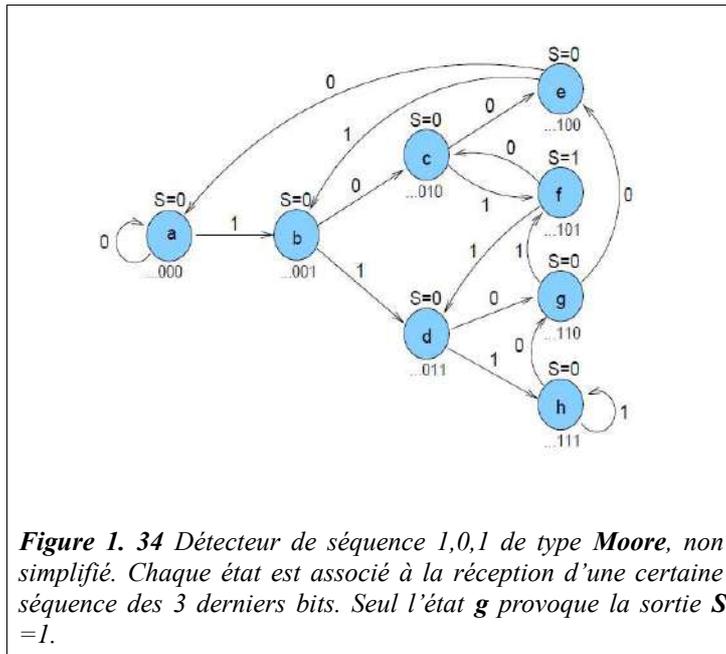
Considérons le fonctionnement du circuit séquentiel synchrone de la Figure 1. 33, détecteur de la séquence '1,0,1' :



Une suite de chiffres binaires arrive sur E, et les instants d'échantillonnage (moments où la valeur de E est prise en compte) sont les fronts montants de CLK. On souhaite que la sortie S vaille 1 si et seulement si les deux dernières valeurs de E sont : 1,0,1.

Cette spécification est en fait imprécise, car elle ne dit pas exactement à quel moment par rapport à l'horloge CLK la sortie doit prendre sa valeur. Il y a deux possibilités :

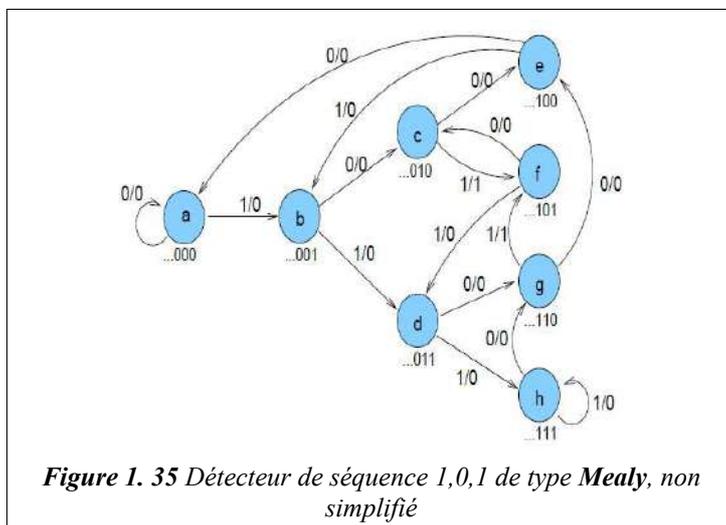
1. la sortie ne dépend que de l'état interne du circuit, et ne peut changer que juste après chaque front d'horloge. Elle ne pourra changer ensuite qu'au prochain front d'horloge, même si les entrées changent entre temps. On appelle schéma de MOORE une telle conception, et elle conduit au graphe de MOORE de la Figure 1. 34.



La valeur de la sortie S est clairement associée aux états, et ne peut donc changer qu'avec eux. Les changements d'état se produisent à chaque front de l'horloge CLK, qui n'est pas représentée sur la graphe, mais dont la présence est implicite. Un arc libellé '0' par exemple indique un changement d'état lorsque E = 0.

- la sortie dépend de l'état interne du circuit et de ses entrées, et on dit alors qu'il s'agit d'un schéma de MEALY. Pour le détecteur de séquence par exemple, la sortie S pourrait valoir 1 dès que E passe à 1 pour la deuxième fois, avant même que sa valeur ne soit 'officiellement' échantillonnée au prochain front d'horloge. Cela conduit au graphe de MEALY de la Figure 1.35.

L'arc libellé '1/0' de a vers b par exemple indique un changement d'état de a vers b lorsque E = 1, avec une valeur de la sortie S=0. Mais attention : le changement d'état ne se produira qu'au prochain front d'horloge, alors que le changement de sortie se produit immédiatement après le changement des entrées.



1.6.2 Transformation d'un graphe de Moore en graphe de Mealy

On voit sur la Figure 1.36 comment un arc dans un graphe de MOORE se transforme en un arc dans un graphe de MEALY équivalent.

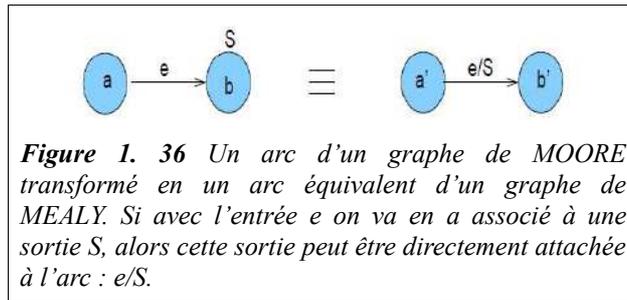


Figure 1. 36 Un arc d'un graphe de MOORE transformé en un arc équivalent d'un graphe de MEALY. Si avec l'entrée e on va en a associé à une sortie S, alors cette sortie peut être directement attachée à l'arc : e/S.

On a donc une méthode automatique pour transformer les graphes de MOORE en graphes de MEALY, très utile car les graphes de MOORE sont souvent plus faciles à concevoir. C'est elle qu'on a utilisée par exemple pour transformer le graphe de MOORE de la Figure 1. 34 en graphe de MEALY de la Figure 1. 35.

1.6.3 Tables de transitions

Les tables de transitions, encore appelées tables de Huffman ne sont rien d'autre qu'une version tabulaire des graphes de transitions. Par exemple, les tables associées aux graphes de MOORE et de MEALY du détecteur de séquence sont représentées Figure 1. 37.

avant		après
état	E	état
a	0	a
a	1	b
b	0	c
b	1	d
c	0	e
c	1	f
d	0	g
d	1	h
e	0	a
e	1	b
f	0	c
f	1	d
g	0	e
g	1	f
h	0	g
h	1	h

état	S
a	0
b	0
c	0
d	0
e	0
f	1
g	0
h	0

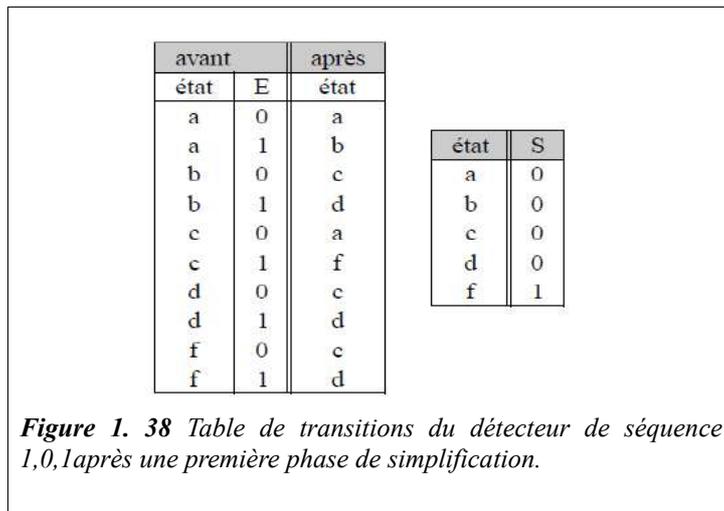
Figure 1. 37 Table de transitions du graphe de MOORE pour le détecteur de séquence 1,0,1. À chaque arc du graphe correspond une ligne dans la table. On notera la table complémentaire, qui donne les sorties associées à chaque état.

1.6.4 Simplification des tables de transition

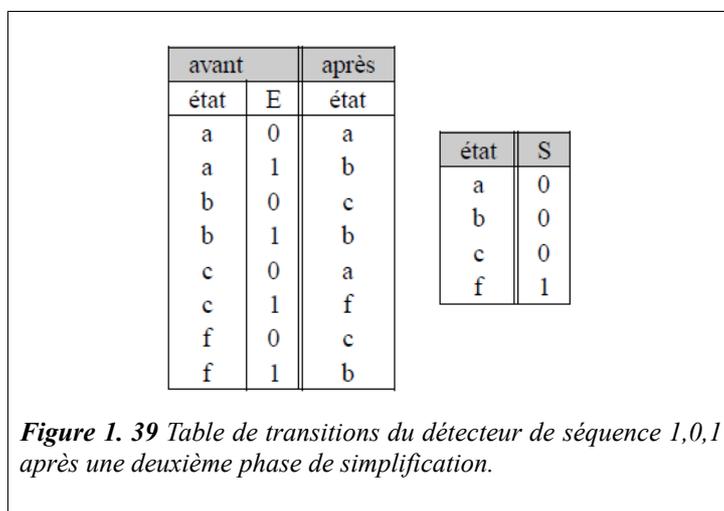
Une table de transitions permet notamment de repérer facilement des états équivalents. Un groupe G d'états sont dits équivalents si, pour chaque combinaison possible des entrées, ils conduisent à des états du groupe G avec les mêmes sorties. On peut alors diminuer le nombre d'états en remplaçant tous les états du groupe G par un seul, puis tenter de rappliquer cette procédure de réduction sur l'ensemble d'états réduit.

Par exemple, sur la table de transitions précédente associée au diagramme de MOORE, on constate que a et e sont équivalents, que c et g sont équivalents, que d et h sont équivalents.

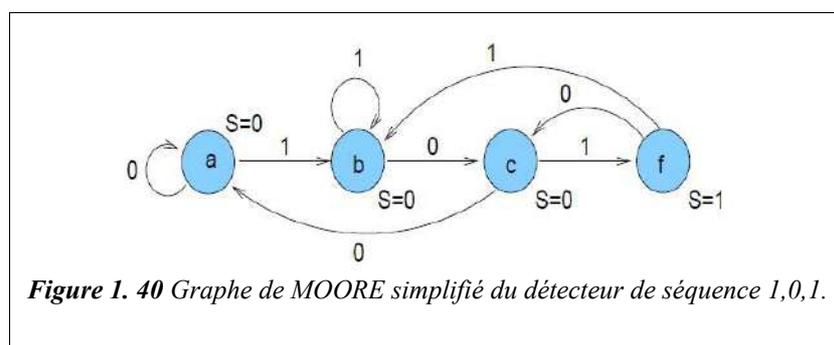
Attention : b et f ne sont pas équivalents car, bien qu'ils aient mêmes états suivants, ils sont associés à des sorties différentes. On peut réécrire la table de transitions en supprimant les lignes associées à e, g et h, et en remplaçant partout e par a, g par c, h par d (Figure 1. 38).



Sur cette table, on constate que b et d sont équivalents, ce qui ne pouvait pas être déterminé à l'étape précédente. La Figure 1. 39 montre la table encore simplifiée.



Le processus de simplification s'arrête ici. Finalement, 4 états suffisent pour ce détecteur de séquence. Ceux-ci ont perdu la signification qu'ils avaient initialement dans le graphe à 8 états. Le graphe simplifié est représenté Figure 1. 40.



1.6.5 Étapes de la synthèse d'un circuit séquentiel synchrone

Lors de la synthèse d'un circuit séquentiel synchrone à l'aide de bascules, qu'il soit de type MOORE ou de type MEALY, on obtiendra le circuit le plus simple en suivant les étapes suivantes :

1. dessin du graphe d'états : il permet une spécification claire du circuit.
2. table de transitions : forme plus lisible du graphe, qui permet de vérifier qu'aucune transition n'est oubliée, et

- prépare la phase de simplification.
3. simplification de la table de transitions : on applique la méthode vue précédemment, parfois en plusieurs étapes.
 4. détermination du nombre de bascules : le nombre d'états du graphe simplifié permet d'établir un nombre minimum n de bascules à employer. On ne choisit pas encore le type des bascules à employer, car la phase d'assignation fournira de nouvelles informations.
 5. assignation des états : pour chaque état, un vecteur unique de n bits est assigné, appelé vecteur d'état. Des règles heuristiques d'assignation permettent de trouver celle qui conduira à des calculs simples.
 6. table de transitions instanciée : on réécrit la table de transitions, en remplaçant chaque symbole d'état par le vecteur associé.
 7. choix des bascules : en observant la table de transitions instanciée, certains types de bascules peuvent être préférés. On emploiera une bascule D lorsqu'un état suivant est la copie d'une valeur de l'état précédent, une bascule T lorsque l'état suivant est l'inverse d'un état précédent ; dans les autres cas, la bascule JK peut être employée.
 8. calcul des entrées des bascules et calcul des sorties du circuit.

1.6.6 Synthèse du détecteur de séquence, version MOORE

DÉTERMINATION DU NOMBRE DE BASCULES

Il y a quatre états à coder, donc 2 bascules seront employées, dont on appellera les sorties X et Y.

ASSIGNATION DES ÉTATS

La phase d'assignation consiste donc à affecter un vecteur d'état à chacun des états du circuit ; ici un couple (X,Y) pour chacun des 4 états. N'importe quelle assignation peut être employée. Néanmoins, on peut toujours affecter à l'état initial le vecteur d'état (0,0,...), qui sera forcé lors d'un RESET asynchrone du circuit. Par ailleurs, certaines assignations donnent lieu à des calculs plus simples que d'autres. On les trouve souvent en appliquant les règles heuristiques suivantes, par ordre de priorité décroissante :

1. rendre adjacents les états de départ qui ont même état d'arrivée dans le graphe.
2. rendre adjacents les états d'arrivée qui ont même état de départ dans le graphe.
3. rendre adjacents les états qui ont mêmes sorties.

L'idée est de minimiser le nombre de bits qui changent (état ou sorties) lors des changements d'états.

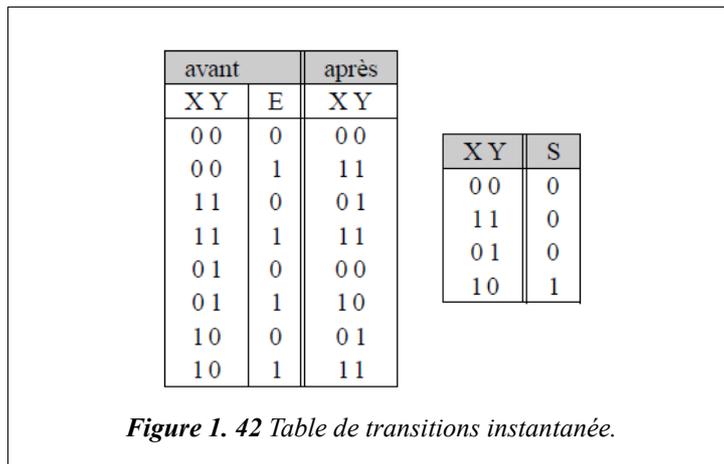
Appliqué à notre circuit, la première règle préconise d'associer a et c, a et f, b et f ; la deuxième règle préconise d'associer a et b, b et c, a et f ; la troisième règle préconise d'associer a, b et c. La Figure 1. 41 propose une assignation qui respecte au mieux ces règles.

		X	
		0	1
Y	0	a	f
	1	c	b

Figure 1. 41 Assignation des états : chaque état est associé à une configuration binaire des bascules.

TABLE DE TRANSITIONS INSTANCIÉE

On recopie la table de transitions précédente, en remplaçant chaque symbole d'état par son vecteur d'état (Figure 1. 42).



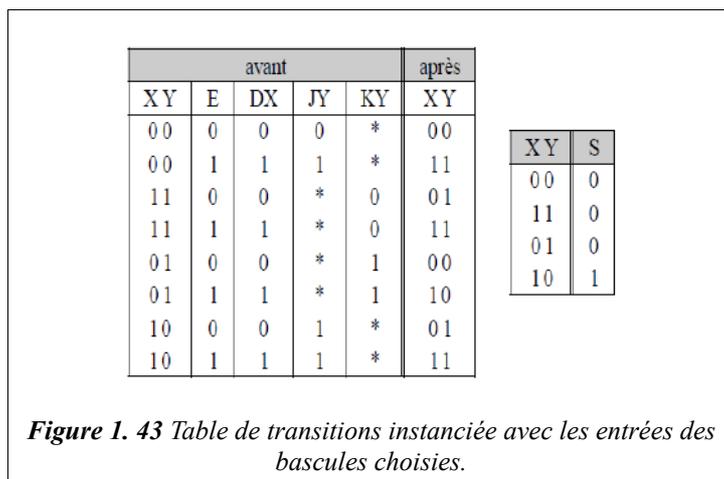
CHOIX DES BASCULES

En observant la table instanciée, on constate que X à l'état suivant reproduit l'entrée E :

une bascule D s'impose pour X. Dans le cas de Y, il reproduit presque la valeur de X de l'état précédent, sauf pour la deuxième ligne. A titre d'exemple, on va choisir une bascule JK.

CALCUL DES ENTRÉES DES BASCULES ET DE LA SORTIE DU CIRCUIT

Plusieurs méthodes sont possibles pour ces calculs. Celle qui sera employée ici est basée sur une réécriture de la table de transitions, à laquelle on rajoute les entrées des bascules, ici DX pour la bascule X et JY, KY pour la bascule Y (Figure 1. 43).



En fait on connaissait déjà le résultat pour X : $DX = E$, qui est la raison pour laquelle on avait choisi une bascule D. Pour Y, on trouve facilement que $JY = E + X$ et $KY = \overline{X}$. Par ailleurs, on a $S = X \overline{Y}$.

La synthèse est terminée, et notre séquenceur se réduit au schéma de la Figure 1. 44.

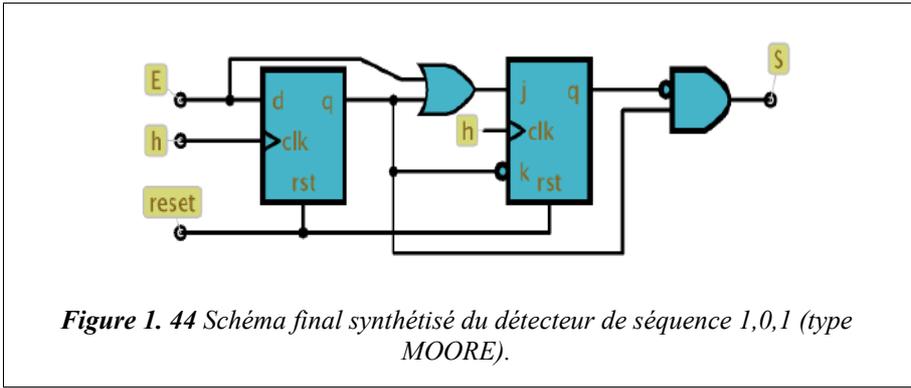


Figure 1. 44 Schéma final synthétisé du détecteur de séquence 1,0,1 (type MOORE).

PARTIE 2

INTRODUCTION À L'ARCHITECTURE DES ORDINATEURS

2 Introduction à l'architecture

2.1 Introduction

Le cours d'Architecture des Ordinateurs expose les principes de fonctionnement des ordinateurs. Il ne s'agit pas ici d'apprendre à programmer, mais de comprendre, à bas niveau, l'organisation de ces machines.

2.2 Architecture de base d'un ordinateur

Un ordinateur est une machine de traitement de l'information. Il est capable d'acquérir de l'information, de la stocker, de la transformer en effectuant des traitements quelconques, puis de la restituer sous une autre forme. Le mot informatique vient de la contraction des mots information et automatique.

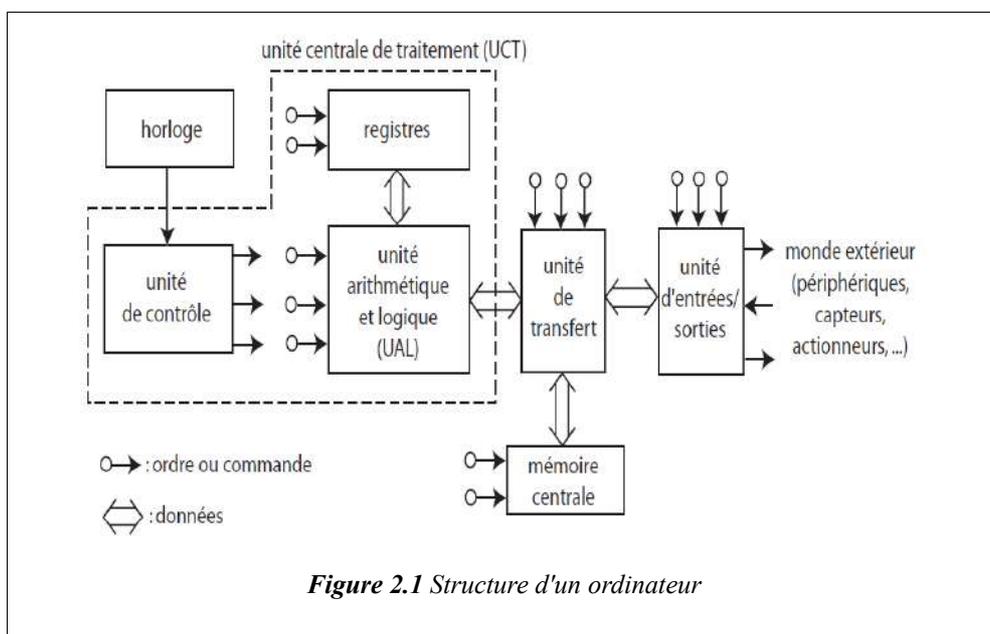
Nous appelons information tout ensemble de données. On distingue généralement différents types d'informations : textes, nombres, sons, images, etc., mais aussi les instructions composant un programme. Comme on l'a vu dans la première partie, toute information est manipulée sous forme binaire (ou numérique) par l'ordinateur.

2.3 Principes de fonctionnement

Un ordinateur est constitué de :

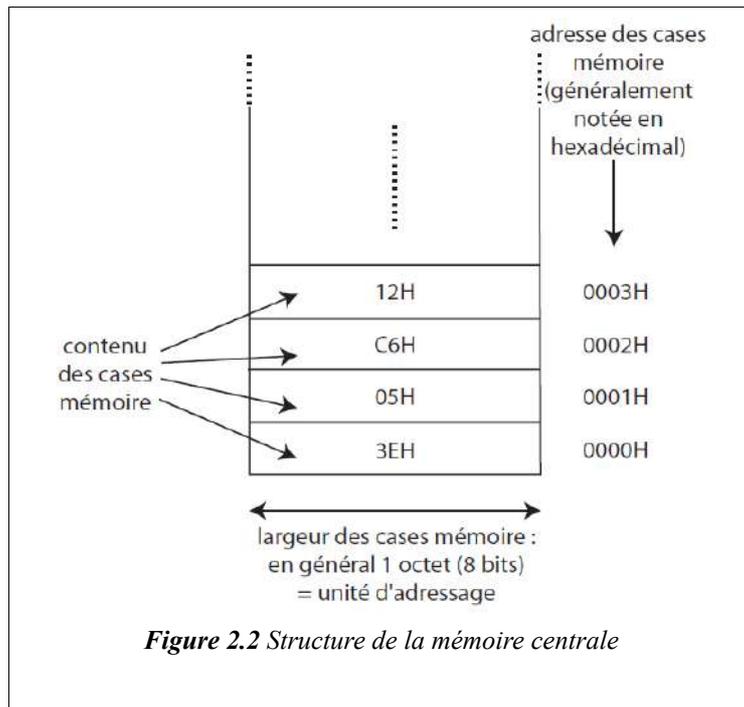
- Unité centrale de traitement (CPU : Central Processing Unit). L'UCT est constituée :
 - d'une unité arithmétique et logique (UAL, ALU : Arithmetic and Logic Unit) : c'est l'organe de calcul du calculateur ;
 - de registres : zones de stockage des données de travail de l'UAL (opérandes, résultats intermédiaires) ;
 - d'une unité de contrôle (UC, CU : Control Unit) : elle envoie les ordres (ou commandes) à tous les autres éléments du calculateur afin d'exécuter un programme.
- Mémoire centrale contient :
 - le programme à exécuter : suite d'instructions élémentaires ;
 - les données à traiter.
- Unité d'entrées/sorties (E/S) est un intermédiaire entre le calculateur et le monde extérieur.
- Unité de transfert est le support matériel de la circulation des données.

Les échanges d'ordres et de données dans le calculateur sont synchronisés par une horloge qui délivre des impulsions (signal d'horloge) à des intervalles de temps fixes.



2.4 Organisation de la mémoire centrale

La mémoire peut être vue comme un ensemble de cellules ou cases contenant chacune une information : une instruction ou une donnée. Chaque case mémoire est repérée par un numéro d'ordre unique : son adresse.



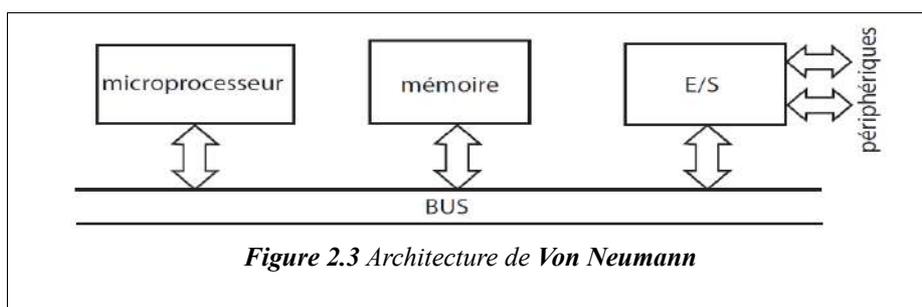
Une case mémoire peut être lue ou écrite par le microprocesseur (cas des mémoires vives) ou bien seulement lue (cas des mémoires mortes).

La capacité (taille) de la mémoire est le nombre d'emplacements, exprimé en général en kilo-octets ou en méga-octets, voire davantage. Rappelons que le kilo informatique vaut 1024 et non 1000 ($2^{10} = 1024 \approx 1000$). Voici les multiples les plus utilisés :

1 K (Kilo)	2^{10}	1024
1 M (Méga)	2^{20}	1048576
1 G (Giga)	2^{30}	1 073 741 824
1 T (Téra)	2^{40}	1 099 511 627 776

2.5 Circulation de l'information dans un ordinateur

La réalisation matérielle des ordinateurs est généralement basée sur l'architecture de **Von Neumann** :



Le microprocesseur échange des informations avec la mémoire et l'unité d'E/S, sous forme de mots binaires, au moyen d'un ensemble de connexions appelé **bus**.

Un bus permet de transférer des données sous forme parallèle, c'est-à-dire en faisant circuler n bits simultanément.

Les microprocesseurs peuvent être classés selon la longueur maximale des mots binaires qu'ils peuvent échanger avec la mémoire et les E/S : microprocesseurs 8 bits, 16 bits, 32 bits, ...

Le bus peut être décomposé en trois bus distincts (Figure 2.4) :

- le bus d'adresses permet au microprocesseur de spécifier l'adresse de la case mémoire à lire ou à écrire ;
- le bus de données permet les transferts entre le microprocesseur et la mémoire ou les E/S ;

- le bus de commande transmet les ordres de lecture et d'écriture de la mémoire et des E/S.

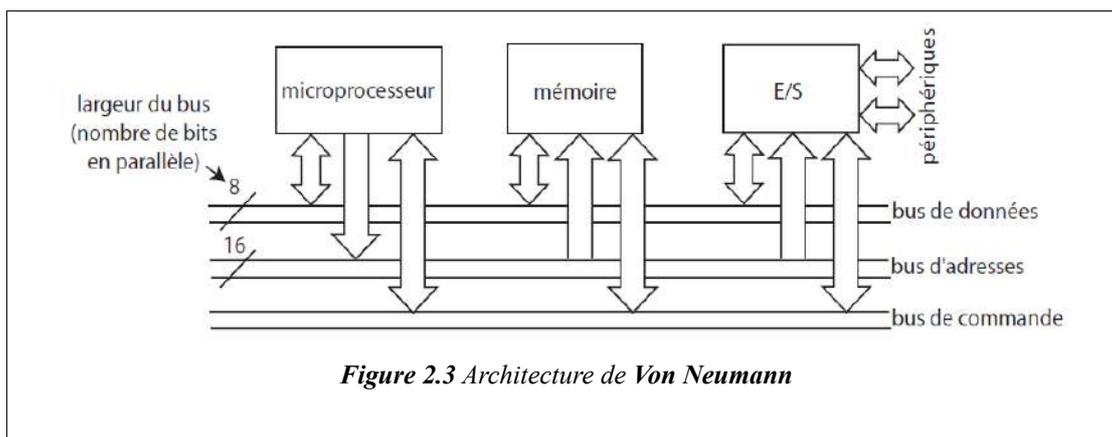


Figure 2.3 Architecture de Von Neumann

REMARQUE : les bus de données et de commande sont bidirectionnels, le bus d'adresse est unidirectionnel : seul le microprocesseur peut délivrer des adresses (il existe une dérogation pour les circuits d'accès direct à la mémoire, DMA).

2.6 Le processeur central

Le processeur est parfois appelé CPU (de l'anglais Central Processing Unit) ou encore MPU (Micro-Processing Unit) pour les microprocesseurs.

Un microprocesseur n'est rien d'autre qu'un processeur dont tous les constituants sont réunis sur la même puce électronique (pastille de silicium), afin de réduire les coûts de fabrication et d'augmenter la vitesse de traitement. Les microordinateurs sont tous équipés de microprocesseurs.

L'architecture de base des processeurs équipant les gros ordinateurs est la même que celle des microprocesseurs.

2.6.1 Les registres et l'accumulateur

Le processeur utilise toujours des registres, qui sont des petites mémoires internes très rapides d'accès utilisées pour stocker temporairement une donnée, une instruction ou une adresse. Chaque registre stocke 8, 16 ou 32 bits.

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.

L'exécution des instructions peut se découper en grandes étapes :

- chargement de l'instruction à exécuter;
- décodage de l'instruction;
- localisation dans la mémoire des données utilisées par l'instruction;
- chargement des données si nécessaire;
- exécution de l'instruction;
- sauvegarde des résultats à leurs destinations respectives;
- passage à l'instruction suivante.

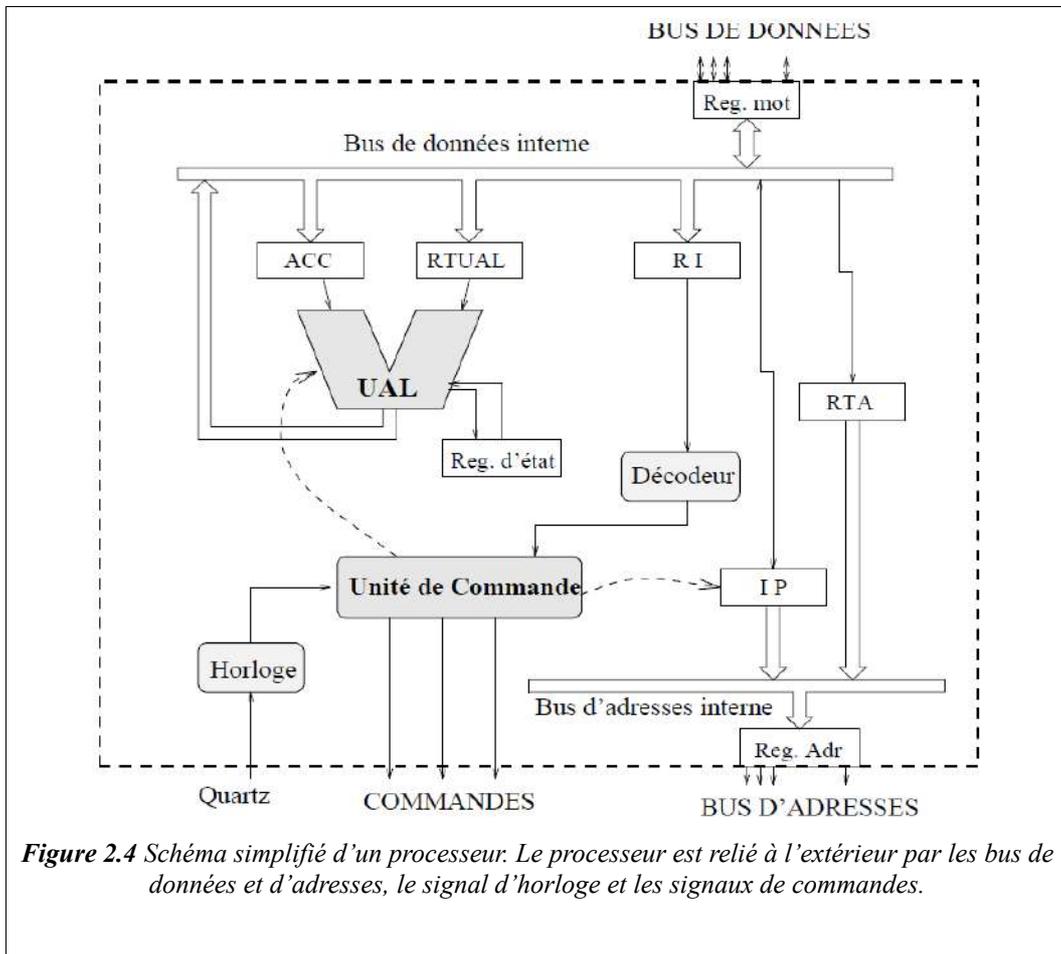
Par exemple, examinons ce qu'il se passe lorsque le processeur exécute une instruction comme "Ajouter 5 au contenu de la case mémoire d'adresse 180" :

1. Le processeur lit et décode l'instruction ;
2. le processeur demande à la mémoire la contenu de l'emplacement 180 ;
3. la valeur lue est rangée dans l'accumulateur ;
4. l'unité de traitement (UAL) ajoute 5 au contenu de l'accumulateur ;
5. le contenu de l'accumulateur est écrit en mémoire à l'adresse 180.

C'est l'unité de commande ou de contrôle qui déclenche chacune de ces actions dans l'ordre. L'addition proprement dite est effectuée par l'UAL.

2.6.2 Architecture d'un processeur à accumulateur

La figure 2.4 représente l'architecture interne simplifiée d'un MPU ou CPU à accumulateur. On y distingue l'unité de commande, l'UAL, et le décodeur d'instructions, qui, à partir du code de l'instruction lu en mémoire actionne la partie de l'unité de commande nécessaire.



Les informations circulent à l'intérieur du processeur sur deux bus internes, l'un pour les données, l'autre pour les instructions.

On distingue les registres suivants :

ACC : Accumulateur ;

RTUAL : Registre Tampon de l'UAL, stocke temporairement l'un des deux opérandes d'une instructions arithmétiques (la valeur 5 dans l'exemple donné plus haut) ;

Reg. d'état : stocke les indicateurs, que nous étudierons plus tard ;

RI : Registre Instruction, contient le code de l'instruction en cours d'exécution (lu en mémoire via le bus de données) ;

IP : Instruction Pointer ou Compteur de Programme, contient l'adresse de l'emplacement mémoire où se situe la prochaine instruction à exécuter ;

RTA : Registre Tampon d'Adresse, utilisé pour accéder à une donnée en mémoire.

Les signaux de commandes permettent au processeur de communiquer avec les autres circuits de l'ordinateur. On trouve en particulier le signal R/W (Read/Write), qui est utilisé pour indiquer à la mémoire principale si l'on effectue un accès en lecture ou en écriture.

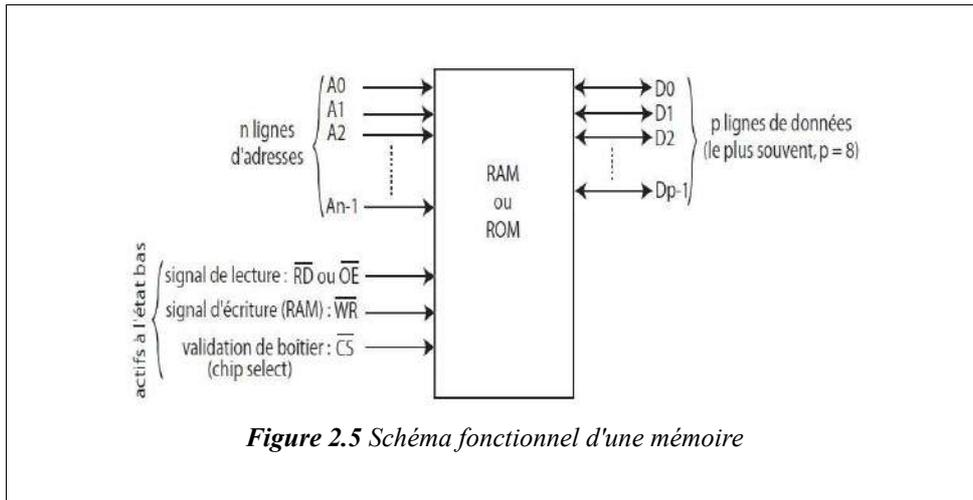
2.7 Les mémoires

On distingue deux types de mémoires :

- les mémoires vives (**RAM** : Random Access Memory) ou mémoires volatiles. Elles perdent leur contenu en cas de coupure d'alimentation. Elles sont utilisées pour stocker temporairement des données et des programmes. Elles peuvent être lues et écrites par le microprocesseur ;
- les mémoires mortes (**ROM** : Read Only Memory) ou mémoires non volatiles. Elles conservent leur contenu en cas de coupure d'alimentation. Elles ne peuvent être que lues par le microprocesseur (pas de possibilité d'écriture). On les utilise pour stocker des données et des programmes de manière définitive.

Les mémoires sont caractérisées par leur capacité : nombre total de cases mémoire contenues dans un même boîtier.

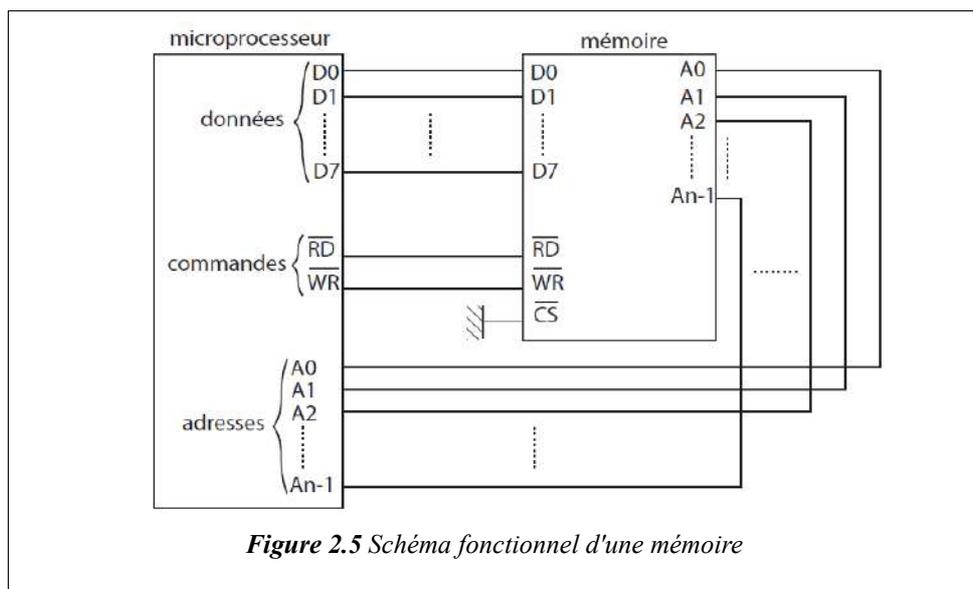
2.7.1 Schéma fonctionnel d'une mémoire



Le nombre de lignes d'adresses dépend de la capacité de la mémoire : n lignes d'adresses permettent d'adresser 2^n cases mémoire : 8 bits d'adresses permettent d'adresser 256 octets, 16 bits d'adresses permettent d'adresser 65536 octets (= 64 Ko), ...

EXEMPLE : mémoire RAM 6264, capacité = 8K x 8bits : 13 broches d'adresses A0 à A12, $2^{13} = 8192 = 8 \text{ Ko}$.

2.7.2 Interfaçage microprocesseur/mémoire



2.8 Microprocesseur Intel 8086

La gamme de microprocesseurs 80x86 équipe les micro-ordinateurs de type PC et compatibles. Les premiers modèles de PC, commercialisés au début des années 1980, utilisaient le 8086, un microprocesseur 16 bits.

Les modèles suivants ont utilisé successivement le 80286, 80386, 80486 et Pentium (ou 80586). Chacun de ces processeurs est plus puissant que les précédents : augmentation de la fréquence d'horloge, de la largeur de bus (32 bits d'adresse et de données), introduction de nouvelles instructions (par exemple calcul sur les réels) et ajout de registres.

Chacun d'entre eux est compatible avec les modèles précédents ; un programme écrit dans le langage machine du 286 peut s'exécuter sans modification sur un 486. L'inverse n'est pas vrai, puisque chaque génération a ajouté des instructions nouvelles. On parle donc de compatibilité ascendante.

Voici les caractéristiques du processeur 8086 :

CPU 16 bits à accumulateur :

- bus de données 16 bits ;
- bus d'adresse 20 bits ;

Le microprocesseur 8086 contient 14 registres répartis en 4 groupes :

- Registres généraux : 4 registres sur 16 bits.
 - AX = (AH,AL) ; AX = accumulateur
 - BX = (BH,BL) ;
 - CX = (CH,CL) ;
 - DX = (DH,DL).
- Registres de pointeurs et d'index : 4 registres sur 16 bits.
 - Pointeurs :
 - SP : Stack Pointer, pointeur de pile (la pile est une zone de sauvegarde de données en cours d'exécution d'un programme) ;
 - BP : Base Pointer, pointeur de base, utilisé pour adresser des données sur la pile.
 - Index :
 - SI : Source Index ;
 - DI : Destination Index.
 Ils sont utilisés pour les transferts de chaînes d'octets entre deux zones mémoire.
- Les pointeurs et les index contiennent des adresses de cases mémoire.
 - Pointeur d'instruction et indicateurs (flags) : 2 registres sur 16 bits.

Flags :

				O	D	I	T	S	Z		A		P		C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- CF : indicateur de retenue (carry) ;
- PF : indicateur de parité ;
- AF : indicateur de retenue auxiliaire ;
- ZF : indicateur de zéro ;
- SF : indicateur de signe ;
- TF : indicateur d'exécution pas à pas (trap) ;
- IF : indicateur d'autorisation d'interruption ;
- DF : indicateur de décrémentation ;
- OF : indicateur de dépassement (overflow).

- Registres de segments : 4 registres sur 16 bits.
 - CS : Code Segment, registre de segment de code ;
 - DS : Data Segment, registre de segment de données ;
 - SS : Stack Segment, registre de segment de pile ;
 - ES : Extra Segment, registre de segment supplémentaire pour les données ;

Les registres de segments, associés aux pointeurs et aux index, permettent au microprocesseur 8086 d'adresser l'ensemble de la mémoire.

Le 8086 est constitué de deux unités fonctionnant en parallèle (figure 2.6) :

- l'unité d'exécution (EU : Execution Unit) : l'unité d'interface de bus (BIU) recherche les instructions en mémoire et les range dans une file d'attente ;
- l'unité d'interface de bus (BIU : Bus Interface Unit) : l'unité d'exécution (EU) exécute les instructions contenues dans la file d'attente.

Les deux unités fonctionnent simultanément, d'où une accélération du processus d'exécution d'un programme (fonctionnement selon le principe du pipe-line).

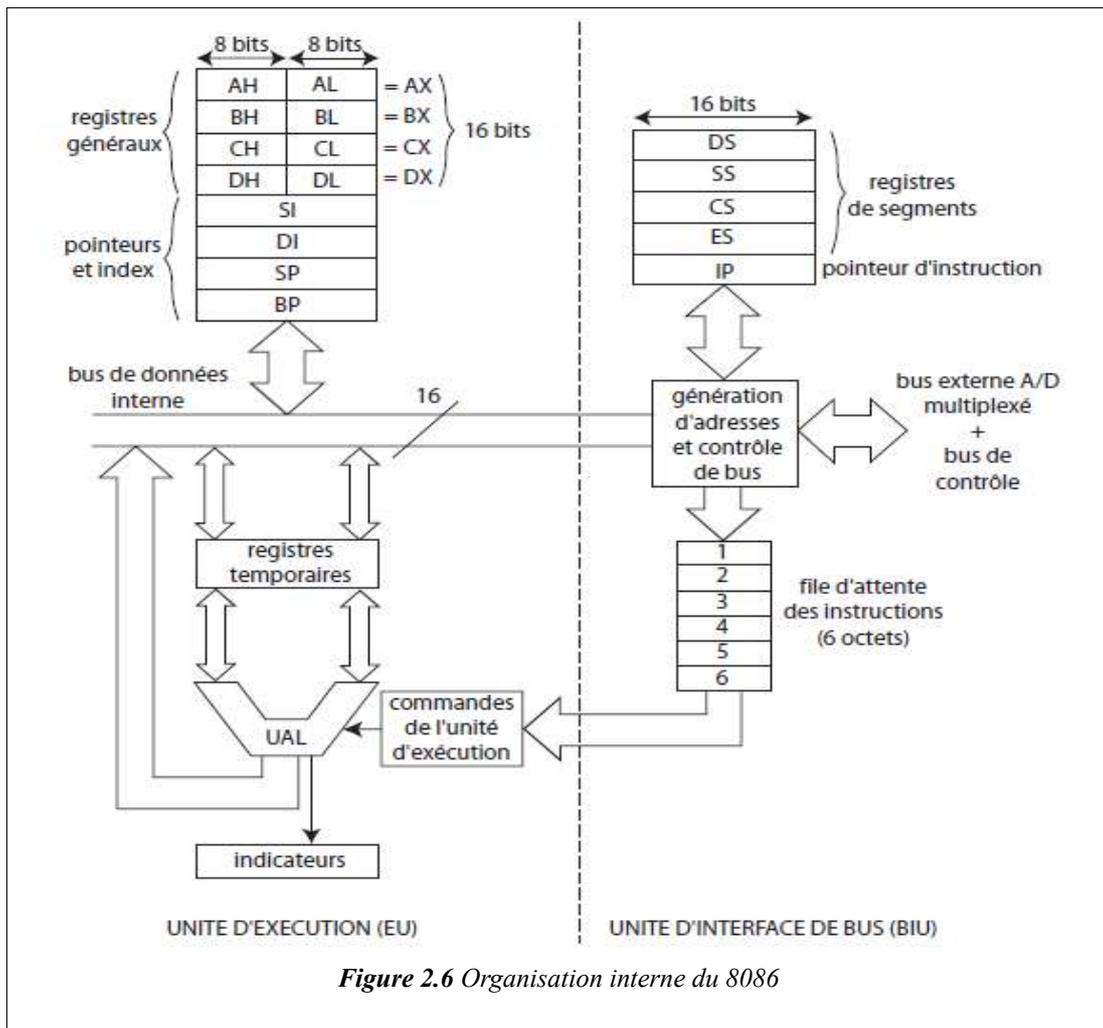


Figure 2.6 Organisation interne du 8086

2.8.1 Jeu d'instruction

Chaque microprocesseur reconnaît un ensemble d'instructions appelé jeu d'instructions (Instruction Set) fixé par le constructeur.

Une instruction est définie par son code opératoire, valeur numérique binaire difficile à manipuler par l'être humain. On utilise donc une notation symbolique pour représenter les instructions : les mnémoniques. Un programme constitué de mnémoniques est appelé programme en assembleur.

Les instructions peuvent être classées en groupes :

- instructions de transfert de données ;
- instructions arithmétiques ;
- instructions logiques ;
- instructions de branchement ...

2.8.1.1 Types d'instructions

LES INSTRUCTIONS DE TRANSFERT

Elles permettent de déplacer des données d'une source vers une destination :

- registre vers mémoire ;
- registre vers registre ;
- mémoire vers registre.

Remarque : le microprocesseur 8086 n'autorise pas les transferts de mémoire vers mémoire (pour ce faire, il faut passer par un registre intermédiaire).

SYNTAXE : MOV destination,source

Il existe différentes façons de spécifier l'adresse d'une case mémoire dans une instruction : ce sont les modes d'adressage.

Exemples de modes d'adressage simples :

- **mov ax,bx** : charge le contenu du registre BX dans le registre AX. Dans ce cas, le transfert se fait de registre à registre : adressage par registre ;
- **mov al,12H** : charge le registre AL avec la valeur 12H. La donnée est fournie immédiatement avec l'instruction : adressage immédiat.
- **mov bl,[1200H]** : transfère le contenu de la case mémoire d'adresse effective (offset) 1200H vers le registre BL. L'instruction comporte l'adresse de la case mémoire où se trouve la donnée : adressage direct. L'adresse effective représente l'offset de la case mémoire dans le segment de données (segment dont l'adresse est contenue dans le registre DS) : segment par défaut.

REMARQUE :

dans le cas de l'adressage immédiat de la mémoire, il faut indiquer le format de la donnée : octet ou mot (2 octets) car le microprocesseur 8086 peut manipuler des données sur 8 bits ou 16 bits. Pour cela, on doit utiliser un spécificateur de format :

- **mov byte ptr [1100H], 65H** : transfère la valeur 65H (sur 1 octet) dans la case mémoire d'offset 1100H ;
- **mov word ptr [1100H], 65H** : transfère la valeur 0065H (sur 2 octets) dans les cases mémoire d'offset 1100H et 1101H.

2.8.1.2 Les instructions arithmétiques

Les instructions arithmétiques de base sont l'**addition**, la **soustraction**, la **multiplication** et la **division** qui incluent diverses variantes. Plusieurs modes d'adressage sont possibles.

Addition : ADD opérande1,opérande2

L'opération effectuée est : opérande1 ← opérande1 + opérande2.

EXEMPLES :

- **add ah,[1100H]** : ajoute le contenu de la case mémoire d'offset 1100H à l'accumulateur AH (adressage direct) ;
- **add ah,[bx]** : ajoute le contenu de la case mémoire pointée par BX à l'accumulateur AH (adressage basé) ;
- **add byte ptr [1200H],05H** : ajoute la valeur 05H au contenu de la case mémoire d'offset 1200H (adressage immédiat).

Soustraction : SUB opérande1,opérande2

L'opération effectuée est : opérande1 ← opérande1 - opérande2.

Multiplication : MUL opérande, où opérande est un registre ou une case mémoire.

Cette instruction effectue la multiplication du contenu de AL par un opérande sur 1 octet ou du contenu de AX par un opérande sur 2 octets. Le résultat est placé dans AX si les données à multiplier sont sur 1 octet (résultat sur 16 bits), dans (DX,AX) si elles sont sur 2 octets (résultat sur 32 bits).

EXEMPLES :

mov al,51 mov bl,32 mul bl	mov ax,4253 mov bx,1689 mul bx	mov al,43 mov byte ptr [1200H],28 mul byte ptr [1200H]
AX = 51 × 32	(DX, AX) = 4253 × 1689	AX = 43 × 28

Division : DIV opérande, où opérande est un registre ou une case mémoire.

Cette instruction effectue la division du contenu de AX par un opérande sur 1 octet ou le contenu de (DX,AX) par un opérande sur 2 octets. Résultat : si l'opérande est sur 1 octet, alors AL = quotient et AH = reste ; si l'opérande est sur

2 octets, alors AX = quotient et DX = reste.

Autres instructions arithmétiques :

- **ADC** : addition avec retenue ;
- **SBB** : soustraction avec retenue ;
- **INC** : incrémentation d'une unité ;
- **DEC** : décrémentation d'une unité ;
- **IMUL** : multiplication signée ;
- **IDIV** : division signée.

2.8.1.3 Les instructions logiques

Ce sont des instructions qui permettent de manipuler des données au niveau des bits. Les opérations logiques de base sont :

- ET → AND opérande1,opérande2
- OU → OR opérande1,opérande2
- OU exclusif → XOR opérande1,opérande2
- complément à 1 → NOT opérande
- complément à 2 →NEG opérande
- décalages et rotations

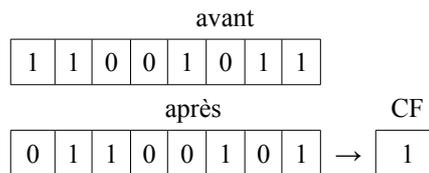
Les différents modes d'adressage sont disponibles.

Les instructions de décalage déplacent d'un certain nombre de positions les bits d'un mot vers la gauche ou vers la droite.

Dans les décalages, les bits qui sont déplacés sont remplacés par des zéros. Il y a les décalages logiques (opérations non signées) et les décalages arithmétiques (opérations signées).

Dans les rotations, les bits déplacés dans un sens sont réinjectés de l'autre côté du mot.

Décalage logique vers la droite (Shift Right) : SHR opérande,n



Remarque : si le nombre de bits à décaler est supérieur à 1, ce nombre doit être placé dans le registre CL ou CX.

EXEMPLE : décalage de AL de trois positions vers la droite :

```
mov cl,3
shr al,cl
```

Décalage logique vers la gauche (Shift Left) : SHL opérande,n

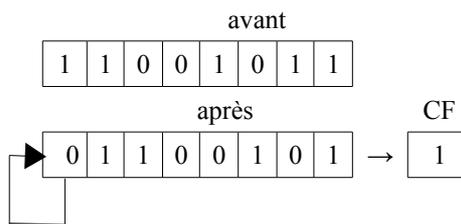
Même mécanisme que le décalage à droite.

Décalage arithmétique vers la droite : SAR opérande,n

Ce décalage conserve le bit de signe bien que celui-ci soit décalé.

EXEMPLE :

```
mov al,11001011B
sar al,1
```



- le bit de signe est réinjecté.

Décalage arithmétique vers la gauche : *SHL opérande,n*

Identique au décalage logique vers la gauche.

Rotation à droite (Rotate Right) : *ROR opérande,n*

Cette instruction décale l'opérande de n positions vers la droite et réinjecte par la gauche les bits sortant.

Rotation à gauche (Rotate Right) : *ROL opérande,n*

Cette instruction décale l'opérande de n positions vers la gauche et réinjecte par la droite les bits sortant.

Rotation à droite avec passage par l'indicateur de retenue (Rotate Right through Carry) : *RCR opérande,n*

Cette instruction décale l'opérande de n positions vers la droite en passant par l'indicateur de retenue CF.

- le bit sortant par la droite est copié dans l'indicateur de retenue CF et la valeur précédente de CF est réinjectée par la gauche.

Rotation à gauche avec passage par l'indicateur de retenue (Rotate Left through Carry) : *RCL opérande,n*

Cette instruction décale l'opérande de n positions vers la gauche en passant par l'indicateur de retenue CF.

- le bit sortant par la gauche est copié dans l'indicateur de retenue CF et la valeur précédente de CF est réinjectée par la droite.

2.8.1.4 Les instructions de branchement

Les instructions de branchement (ou saut) permettent de modifier l'ordre d'exécution des instructions du programme en fonction de certaines conditions. Il existe 3 types de saut :

- saut inconditionnel ;
- sauts conditionnels ;
- appel de sous-programmes.

Instruction de saut inconditionnel : *JMP label*

Cette instruction effectue un saut (**jump**) vers le label spécifié. Un label (ou étiquette) est une représentation symbolique d'une instruction en mémoire :

EXEMPLE :

```
boucle :  inc ax
          dec bx
          jmp boucle
```

Instructions de sauts conditionnels : *Jcondition label*

Un saut conditionnel n'est exécuté que si une certaine condition est satisfaite, sinon l'exécution se poursuit séquentiellement à l'instruction suivante.

La condition du saut porte sur l'état de l'un (ou plusieurs) des indicateurs d'état (flags) du microprocesseur :

Instruction	Nom	Condition
<i>JZ label</i>	Jump if Zero	saut si ZF = 1
<i>JNZ label</i>	Jump if Not Zero	saut si ZF = 0
<i>JE label</i>	Jump if Equal	saut si ZF = 1
<i>JNE label</i>	Jump if Not Equal	saut si ZF = 0
<i>JC label</i>	Jump if Carry	saut si CF = 1
<i>JNC label</i>	Jump if Not Carry	saut si CF = 0
<i>JS label</i>	Jump if Sign	saut si SF = 1
<i>JNS label</i>	Jump if Not Sign	saut si SF = 0

<i>JO label</i>	Jump if Overflow	saut si OF = 1
<i>JNO label</i>	Jump if Not Overflow	saut si OF = 0
<i>JP label</i>	Jump if Parity	saut si PF = 1
<i>JNP label</i>	Jump if Not Parity	saut si PF = 0

REMARQUE : les indicateurs sont positionnés en fonction du résultat de la dernière opération.

EXEMPLE :

on veut additionner deux nombres signés N1 et N2 se trouvant respectivement aux offsets 1100H et 1101H. Le résultat est rangé à l'offset 1102H s'il est positif, à l'offset 1103H s'il est négatif et à l'offset 1104H s'il est nul :

```

mov     al, [1100H]
add     al,[1101H]
js      negatif
jz      nul
mov     [1102H],al
jmp     fin
negatif : mov     [1103H],al
        jmp     fin
nul :    mov     [1104H],al
fin :    hlt

```

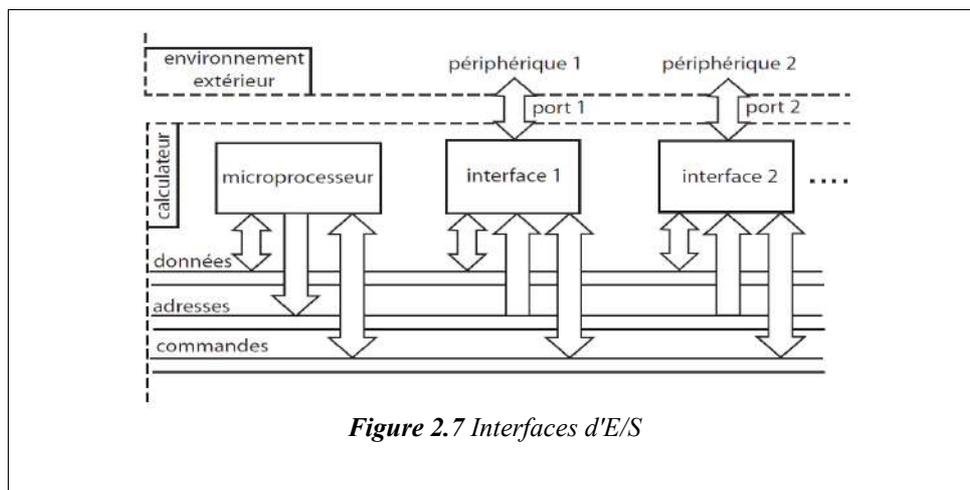
Appel de sous-programmes : pour éviter la répétition d'une même séquence d'instructions plusieurs fois dans un programme, on rédige la séquence une seule fois en lui attribuant un nom (au choix) et on l'appelle lorsqu'on en a besoin.

Le programme appelant est le programme **principal**. La séquence appelée est un **sous-programme** ou **procédure**.

2.9 Les interfaces d'entrées/sorties

Une interface d'entrées/sorties est un circuit intégré permettant au microprocesseur de communiquer avec l'environnement extérieur (périphériques) : clavier, écran, imprimante, modem, disques, processus industriel, ...

Les interfaces d'E/S sont connectées au microprocesseur à travers les bus d'adresses, de données et de commandes (figure 2.7).



Les points d'accès aux interfaces sont appelés ports.

EXEMPLES :

interface	port	exemple de périphérique
interface parallèle	port parallèle	imprimante
interface série	port série	modem

2.9.1 Gestion des ports d'E/S par le 8086

Le 8086 dispose d'un espace mémoire de 1 Mo (adresse d'une case mémoire sur 20 bits) et d'un espace d'E/S de 64 Ko (adresse d'un port d'E/S sur 16 bits).

Le signal permettant de différencier l'adressage de la mémoire de l'adressage des ports d'E/S est la ligne M/\overline{IO} :

- pour un accès à la mémoire, $M/\overline{IO} = 1$;
- pour un accès aux ports d'E/S, $M/\overline{IO} = 0$.

Les instructions de lecture et d'écriture d'un port d'E/S sont respectivement les instructions **IN** et **OUT**. Elles placent la ligne M/\overline{IO} à 0 alors que l'instruction MOV place celle-ci à 1.

Lecture d'un port d'E/S :

- si l'adresse du port d'E/S est sur un octet :
 IN AL,adresse : lecture d'un port sur 8 bits ;
 IN AX,adresse : lecture d'un port sur 16 bits.
- si l'adresse du port d'E/S est sur deux octets :
 IN AL,DX : lecture d'un port sur 8 bits ;
 IN AX,DX : lecture d'un port sur 16 bits.

où le registre DX contient l'adresse du port d'E/S à lire.

Écriture d'un port d'E/S :

- si l'adresse du port d'E/S est sur un octet :
 OUT adresse,AL : écriture d'un port sur 8 bits ;
 OUT adresse,AX : écriture d'un port sur 16 bits.
- si l'adresse du port d'E/S est sur deux octets :
 OUT DX,AL : écriture d'un port sur 8 bits ;
 OUT DX,AX : écriture d'un port sur 16 bits.

où le registre DX contient l'adresse du port d'E/S à écrire.

PARTIE 3

NOTIONS DE PROGRAMMATION EN LANGAGE ÉVOLUÉ (CAS DU C)

3 Algorithmique et Langage évolués

3.1 Algorithmique

Après avoir étudié dans les chapitres précédents le langage machine et l'assembleur, nous abordons ici les langages dites évolués, qui permettent de programmer plus facilement des tâches complexes.

3.2 Algorithmique

Un ordinateur n'est capable de rien si quelqu'un (le programmeur en l'occurrence) ne lui fournit les actions à exécuter. L'ordinateur ne peut faire aucune interprétation des ordres fournis. Ils seront exécutés de manière purement mécanique. De plus, les opérations élémentaires que peut exécuter un ordinateur sont en nombre restreint et doivent être communiquées de façon précise dans un langage qu'il comprendra. Le problème principal de l'utilisateur est donc de lui décrire la suite des actions élémentaires permettant d'obtenir, à partir des données fournies, les résultats escomptés. Cette description doit être précise, envisager le moindre détail et prévoir les diverses possibilités de données. Cette marche à suivre porte le nom d'algorithme dont l'Encyclopédie **UNIVERSALIS** donne la définition suivante:

" Un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données. "

Le mot algorithme provient du nom d'un célèbre mathématicien arabe de la première moitié du 9^{ème} siècle: *Muhammad ibn Musa al Khawarizmi*. Le rôle de l'algorithme est fondamental. En effet, sans algorithme, il n'y aurait pas de programme (qui n'est jamais que sa traduction dans un langage compréhensible par l'ordinateur). De plus, les algorithmes sont fondamentaux en un autre sens: ils sont indépendants à la fois de l'ordinateur qui les exécute, des langages dans lequel ils sont énoncés et traduits.

3.3 Programmation et langage

Parallèlement aux modifications techniques, les moyens de communiquer à l'ordinateur ce qui lui est demandé de faire ont fortement changé. On appelle langages de première génération les langages-machine ou "codes-machine" utilisés initialement (1945).

Un programme en "code-machine" est une suite d'instructions élémentaires, composées uniquement de 0 et de 1, exprimant les opérations de base que la machine peut physiquement exécuter: instructions de calcul (addition, ...) ou de traitement ("et" logique, ...), instructions d'échanges entre la mémoire principale et l'unité de calcul ou entre la mémoire principale et une mémoire externe, des instructions de test qui permettent par exemple de décider de la prochaine instruction à effectuer.

Voici en code machine de l'IBM 370 l'ordre de charger 293 dans le registre "3":

01011000 0011 0000000000100100100

charger 3 293

Ce type de code binaire est le seul que la machine puisse directement comprendre et donc réellement exécuter. Tout programme écrit dans un langage évolué devra par conséquent être d'abord traduit en code-machine avant d'être exécuté.

La deuxième génération est caractérisée par l'apparition de langages d'assemblage (1950) où chaque instruction élémentaire est exprimée de façon symbolique. Un programme dit "assembleur" assure la traduction en code exécutable. L'expression suivante (assembleur INTEL) remplace l'instruction ci-dessus. **MOV R3,293**

La troisième génération débute en 1957 avec le 1^{er} langage dit évolué: FORTRAN (acronyme de mathematical FORmula TRANslating system). Apparaissent ensuite ALGOL (ALGOrithmic Language en 1958), COBOL (COMmon Business Oriented Language en 1959), BASIC (Beginner's All-purpose Symbolic Instruction Code en 1965), Pascal (1968), ... Les concepts employés par ces langages sont beaucoup plus riches et puissants que ceux des générations précédentes et leur formulation se rapproche du langage mathématique. Il existe deux méthodes pour les rendre exécutables:

1. la compilation: l'ensemble du programme est globalement traduit par un autre programme appelé compilateur et le code-machine produit est optimisé. Ce code-machine est ensuite exécuté autant de fois qu'on le veut.
2. l'interprétation: un programme (interpréteur) décode et effectue une à une et au fur et à mesure les instructions du programme-source.

La quatrième génération qui commence au début des années 80 devait mettre l'outil informatique à la portée de tous, en supprimant la nécessité de l'apprentissage d'un langage évolué. Ses concepts fondamentaux sont "convivialité" et

"non-procéduralité" (il suffit de "dire" à la machine ce qu'on veut obtenir sans avoir à préciser comment le faire). Cet aspect a été rencontré avec les langages du type Visual qui prennent en charge l'élaboration de l'interface graphique.

3.4 Définitions

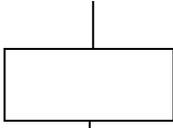
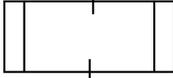
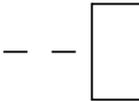
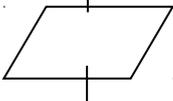
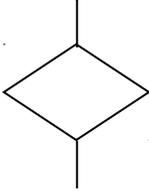
3.4.1 Algorithme

C'est un ensemble de règles opératoires rigoureuses, ordonnant à un processeur d'exécuter dans un ordre déterminé un nombre fini d'opérations élémentaires.

Un algorithme est écrit en utilisant un langage de description d'algorithme (LDA). L'algorithme ne doit pas être confondu avec le programme proprement dit.

3.4.2 Organigramme

C'est une représentation graphique de l'algorithme. Pour le construire, on utilise des symboles normalisés.

SYMBOLE	DESIGNATION	SYMBOLE	DESIGNATION
Symboles de traitement		Symboles auxiliaires	
	Symbole général Opération ou groupe d'opérations sur des données, instructions, pour laquelle il n'existe aucun symbole normalisé.		Début –Fin- Interruption Début, fin ou interruption d'un programme
	Sous programme Portion de programme considéré comme une simple opération.		Commentaire Symbole utilisé pour donner des indications sur les opérations effectuées
	Entrée-Sortie Mise à disposition d'une information (introduction-sortie)		
	Branchement Exploitation de conditions variables impliquant un choix parmi plusieurs		
Sens conventionnel des liaisons:			
Le sens général des liaisons doit être:			
<ul style="list-style-type: none"> ✓ De haut en bas ✓ De gauche à droite 			

3.5 Qualité d'un algorithme

Tout programme fourni à l'ordinateur n'est que la traduction dans un langage de programmation d'un algorithme mis au point pour résoudre un problème donné. Pour obtenir un bon programme, il faut partir d'un bon algorithme. Il doit, entre autres, posséder les qualités suivantes:

1. être clair, facile à comprendre par tous ceux qui le lisent (structure et documentation) présenter la plus grande généralité possible pour répondre au plus grand nombre de cas possibles
2. être d'une utilisation aisée même par ceux qui ne l'ont pas écrit et ce grâce aux messages apparaissant à l'écran qui indiqueront quelles sont les données à fournir et sous quelle forme elles doivent être introduites ainsi que les différentes actions attendues de la part de l'utilisateur.

3. être conçu de manière à limiter le nombre d'opérations à effectuer et la place occupée en mémoire.

Une des meilleures façons de rendre un algorithme clair et compréhensible est d'utiliser une programmation structurée n'utilisant qu'un petit nombre de structures indépendantes du langage de programmation utilisé. Une technique d'élaboration d'un bon algorithme est appelée méthode descendante (top down). Elle consiste à considérer un problème dans son ensemble, à préciser les données fournies et les résultats à obtenir puis à décomposer le problème en plusieurs sous-problèmes plus simples qui seront traités séparément et éventuellement décomposés eux-mêmes de manière plus fine.