Algèbre booléenne Chapitre 2

Les circuits logiques sont la base fonctionnelle des systèmes informatiques modernes. Pour avoir une idée précise du fonctionnement d'un ordinateur, il faut comprendre la logique numérique et l'algèbre booléenne.

Ce chapitre fournit seulement une introduction de base à l'algèbre de Boole. Ce sujet seul est souvent l'objet d'un livre. Ici, nous n'allons traiter que les concepts qui feront l'objet des chapitres à suivre.

2.0 Vue d'Ensemble du Chapitre

La logique booléenne est à la base de l'informatique et des ordinateurs binaires modernes. Vous pouvez représenter tout algorithme ou tout circuit d'ordinateur à l'aide d'un système d'équations booléennes. Ce chapitre offre une brève introduction à l'algèbre booléenne, aux tables de vérité, à la représentation canonique, aux fonctions booléennes et à leur simplification ; il fait l'object aussi de la conception logique, des circuits combinatoires et séquentiels, ainsi que les équivalences entre le matériel et le logiciel.

Le matériel est particulièrement important pour ceux qui veulent concevoir des circuits électroniques ou qui veulent écrire des programmes pour les manipuler. Mais, même si vous avez une autre orientation, cette introduction est encore importante, car vous pouvez tirer parti de cette information pour optimiser certaines expressions conditionnelles complexes à l'intérieur d'instructions comme IF, WHILE, etc.

La section de ce chapitre sur la réduction (optimisation) des fonctions logiques, se sert des *diagrammes Veitch*, plus connus comme *tables de Karnaugh*. Les techniques d'optimisation utilisées dans ce chapitre consistent à réduire le nombre de *termes*[1] dans une fonction booléenne. Il faut comprendre que certaines personnes considèrent les techniques d'optimisation obsolètes, car, selon eux, réduire le nombre de termes dans une opération n'est pas aussi important qu'autrefois. Ce chapitre utilise la méthode des tables de Karnaugh comme un exemple d'optimisation booléenne, non comme une technique à employer régulièrement. Si vous avez un intérêt réel pour la conception et l'optimisation des circuits, vous devriez consulter un ouvrage dédié à ce sujet, afin de disposer de meilleures techniques.

Bien que ce chapitre soit principalement orienté sur le matériel, gardez à l'esprit que beaucoup de concepts utilisés dans ce livre utilisent des équations booléennes (fonctions logiques). Egalement, certains exercices de programmation qu'on verra plus loin demandent ces connaissances. Par conséquent, vous devez être capable de travailler avec des fonctions boléennes avant de procéder à la lecture des chapitres suivants.

2.1 Algèbre booléenne

L'algèbre booléenne est un système mathématique déductif, basé sur les valeurs 0 (faux) et 1 (vrai). Un opérateur binaire "$\circ$" défini sur cet ensemble de valeurs accepte deux entrées booléennes et produit un seul résultat booléen. Par exemple, l'opérateur AND accepte deux valeurs d'entrée et produit un résultat de sortie (le AND logique des deux valeurs).

Pour tout système d'algèbre donné, il y a certains postulats initiaux (ou axiomes) que le système doit suivre. On peut déduire des règles additionnelles, des théorèmes et d'autres propriétés à partir de l'ensemble des postulats de base. Les systèmes d'algèbre booléenne emploient souvent les postulats suivants :

- *Clôture.* Le système est dit *fermé* par rapport à un opérateur binaire si, pour chaque paire de valeurs booléennes, il produit un résultat booléen. Par exemple, le AND est fermé parce qu'il accepte seulement deux opérandes booléens et produit seulement un résultat booléen.
- *Commutativité*. Un opérateur binaire "$\circ$" est dit *commutatif* si $A \circ B = B \circ A$ pour toutes les valeurs booléennes possibles de A et B.
- *Associativité*. Un opérateur binaire "$\circ$" est *associatif* si
$$(A \circ B) \circ C = A \circ (B \circ C)$$

[1] Anglicisation d'usage commun pour se référer aux éléments des équations booléennes et qui a strictement le même sens que le mot français « terme », n.d.t.

Pour toutes les valeurs booléennes de A, B et C.

- *Distributivité*. Deux opérateurs binaires "$\circ$" et % sont *distributifs* si
  $$A \circ (B \,\%\, C) = (A \circ B) \,\%\, (A \circ C)$$
  pour toutes les valeurs booléennes de A, B et C.
- *Indentité*. Une valeur booléenne I est dite identité d'un opérateur "$\circ$" si
  $$A \circ I = A$$
- *Inverse*. Une valeur booléenne I est dite l'inverse d'un opérateur "$\circ$" si $A \circ I = B$ et $B \neq A$ (c.à.d. que B est la valeur opposée de A).

Pour nos buts, l'algèbre booléenne sera basée sur le jeu suivant d'opérateurs et de valeurs :

Les deux valeurs possibles d'un système booléen sont 0 et 1. Souvent, on appellera ces valeurs *faux* et *vrai* (respectivement).

Le symbole "•" représente l'opérateur AND. Par exemple, A • B est le résultat d'un ET logique entre les valeurs booléennes de A et B. Ce livre omettra ce symbole toutes les fois qu'il utilisera des variables d'une seule lettre ; par exemple AB représente une opération AND entre les variables A et B (ceci pourra également s'appeler le *produit* de A et de B).

Le symbole "+" représente l'opérateur OR ; par exemple, A + B est le résultat d'un OU logique entre les valeurs booléennes A et B (ceci sera appelé également la *somme* de A et de B).

Le *complément* logique qu'on appelle aussi *négation* ou tout simplement NOT, est un opérateur unaire. Ce livre utilisera l'apostrophe pour indiquer cette négation. Par exemple, A' représente le NOT de A.

Si dans une expression booléenne apparaissent plusieurs opérateurs différents, le résultat de cette expression dépend de la *priorité* de ces derniers. On utilisera l'ordre suivant de précédence (du plus haut au plus bas) pour les opérateurs : ( ), NOT, AND, OR. Les opérateurs AND et OR ont une associativité de gauche à droite. Si deux opérandes avec les mêmes priorités sont adjacentes, il faut les évaluer de gauche à droite. Alors que l'opération NOT a une associativité de droite, même si cela n'a pratiquement aucun effet, étant donné que NOT est un opérateur unaire.

On utilise également l'ensemble suivant de postulats :

P1 L'algèbre booléenne est fermée pour les opérations de type AND, OR et NOT.

P2 L'élément identité par rapport à • est 1 et par rapport à + est 0. Il n'y a pas d'élément identitaire par rapport au NOT.

P3 Les opérateurs • et + sont commutatifs.

P4 + et • sont distributifs l'un par rapport à l'autre. Donc, A • (B + C) = (A •B) + (A • C) et A + (B • C) = (A + B) • (A + C).

P5 Pour toute valeur A, il existe une valeur A' telle que A • A' = 0 et A + A' = 1. Cette valeur est le complément logique ou NOT de A.

P6 Les opérateurs • et + sont associatifs. C'est-à-dire, (A • B) • C = A • (B •C) et (A + B) + C = A + (B + C).

Vous pouvez prouver tous les autres théorèmes en utilisant ces postulats. Ce livre ne s'attardera pas sur des preuves formelles, cependant, c'est une bonne idée de vous familiariser avec certains théorèmes importants. En voici quelques uns :

Th1 : A + A = A
Th2 : A • A = A
Th3 : A + 0 = A
Th4 : A •1 = A
Th5 : A • 0 = 0
Th6 : A + 1 = 1
Th7 : (A + B)' = A' • B'
Th8 : (A •B)' = A' + B'
Th9 : A + A •B = A

Th10 : A • (A + B) = A
Th11 : A + A'B = A + B
Th12 : A' • (A + B') = A'B'
Th13 : AB + AB' = A
Th14 : (A' + B') • (A' + B) = A'
Th15 : A + A' = 1
Th16 : A • A' = 0

Les théorèmes sept et huit sont connus comme les *lois de DeMorgan*, d'après le mathématicien qui les a découverts.

Ces théorèmes apparaissent « en paires ». Chaque paire (par exemple, Th1 et Th2, Th3 et Th4) constitue un exemple de l'important *principe de dualité* dans l'algèbre booléenne. Toute expression valide qu'on crée à l'aide des postulats et des théorèmes, reste valide si l'on interchange les opérateurs et les constantes qui paraissent dans l'expression. Spécialement, si dans une expression, on interchange • et + et on permute les 0 et les 1, on obtient une expression obéissant à toutes les règles de l'algèbre booléenne. Ceci ne veut pas dire que deux expressions obéissant au principe de dualité comportent les mêmes valeurs, mais simplement que les deux expressions sont valides. Par conséquent, il s'agit d'une manière facile de générer un deuxième théorème pour toute chose qu'on démontre.

Bien qu'ici, nous n'allons pas prouver les théorèmes booléens, on les utilisera pour montrer comment deux équations peuvent être identiques. Ce procédé est important à l'heure de produire des *représentations canoniques* d'une expression, ou quand on doit tout simplement simplifier quelque chose.

2.2 Fonctions booléennes et tables de vérité

Une expression booléenne est une séquence de 0, de 1 et de littéraux séparés par des opérateurs. Un littéral est un nom de variable *primé*[2] (nié) ou *non primé*. Pour ce qui nous concerne, tout nom de variable est un caractère alphabétique isolé. Une fonction booléenne est une expression booléenne spécifique ; nous donnerons généralement aux fonctions booléennes le nom "F" avec un éventuel indice. Par exemple, considérons le booléen suivant :

$$F_0 = AB + C$$

Cette fonction calcule le AND entre A et B et effectue un OR entre le résultat de AB et C. Si A = 1, B = 0 et C = 1, alors F_0 retourne la valeur un (1 • 0 + 1 = 1).

Un autre moyen de représenter une fonction booléenne, c'est à travers d'une *table de vérité*. Le chapitre précédent a utilisé ces tables pour représenter les fonctions AND et OR. Ces tables prennent les formes :

Table 6 : Table de vérité pour AND

AND	0	1
0	0	0
1	0	1

et

Table 7 : Table de vérité pour OR

OR	0	1
0	0	1
1	1	1

[2] *Primé* veut dire simpelemnt qu'il est suivi d'un apostrophe, donc, il est soumis à une négation, n.d.t.

Pour des opérateurs binaires et deux valeurs d'entrée, cette forme de table de vérité est très naturelle et pratique. Cependant, reconsidérez la fonction booléenne F_0 ci-dessus. Cette fonction comporte *trois* valeurs d'entrée et non deux. Par conséquent, on ne peut pas utiliser le format de la table de vérité qu'on vient de voir. Heureusement, c'est également facile de construire des tables pour trois ou plus variables. L'exemple suivant montre un moyen de le faire pour des fonctions de trois ou quatre variables :

Table 8 : Table de vérité pour une fonction de trois variables

F = AB + C		BA			
		00	01	10	11
C	0	0	0	0	1
	1	1	1	1	1

Table 9 : Table de vérité pour une fonction de quatre variables

F = AB + CD		BA			
		00	01	10	11
DC	00	0	0	0	1
	01	0	0	0	1
	10	0	0	0	1
	11	1	1	1	1

Dans les tables 8 et 9, les quatre colonnes représentent les quatre possibles combinaisons de zéros et de uns pour A et B (B est le bit fort ou le bit le plus à gauche, A est le bit faible ou le bit le plus à droite). De même, les quatre lignes dans la seconde table représentent toutes les combinaisons pour les variables C et D. Ici aussi, D représente le bit fort et C le bit faible.

La table 10 illustre une autre manière de représenter des tables de vérité. Cette forme présente deux avantages sur les deux formes précédentes - la table est plus facile à remplir et fournit une représentation compacte pour deux fonctions ou plus.
Notez également que les tables ci-dessus fournissent trois fonctions de trois variables.

Bien qu'on peut créer une variété infinie de fonctions booléennes, elles ne sont pas toutes uniques. Par exemple, F = A et F = AA sont deux fonctions différentes. En vertu du deuxième théorème, cependant, il est facile de montrer que deux fonctions sont équivalentes quand elles produisent exactement la même sortie pour toutes les combinaisons d'entrée. Si vous établissez le nombre des variables d'entrée, il y a un nombre fini de fonctions uniques possibles. Par exemple, avec deux entrées, il y a 16 fonctions uniques et avec trois, il y en a 256. Etant donné *n* variables d'entrée, il y a pour ces *n* valeurs $2^{(2^n)}$ (deux à la puissance deux à la puissance *n*) fonctions booléennes uniques. Pour deux valeurs on a 2 ^ (2^2) = 2^4 = 16 différentes fonctions uniques. Avec trois variables d'entrée, on a 2 ^ (2^3) = 2^8 ou 256 fonctions possibles. Avec quatre variables, on a 2 ^ (2^4) = 65536 fonctions uniques différentes.

Table 10 : Un autre format pour les tables de vérité

C	B	A	F = ABC	F = AB + C	F = A + BC
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	0

1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

Quand on a affaire seulement à 16 fonctions booléennes, il est assez facile de nommer chaque fonction. La table suivante liste les 16 fonctions booléennes possibles de deux variables d'entrée accompagnéees par certains noms communs :

Table 11 : les 16 possibles fonctions de deux variables

No. de fonction	Description
0	Zéro ou Désactivé. Retourne toujours 0, peu importe les entrées A et B.
1	NOR (NOT(A OR B) = (A + B)'
2	Inhibition = BA' (B NOT A). Equivaut aussi à B > A ou A < B.
3	NOT A. Ignore B et retourne A'.
4	Inhibition. AB' (A NOT B). Equivaut aussi à A > B ou B < A.
5	NOT B. Retourne B' et ignore A.
6	OR exclusif (XOR) = A $\oplus$ B. Equivalent aussi à A $\neq$ B.
7	NAND (NOT(A AND B)) = (A $\bullet$ B)'
8	AND = A $\bullet$ B. Retourne A AND B.
9	Equivalence=(A=B). Connu aussi comme NOR exclusif (NOT OR exclusif)
10	Copier B. Retourne la valeur de B et ignore A.
11	Implication, A implique B = B + A' (si A, alors B). Equivalent à A $\geq$ B.
12	Copier A. Retourne la valeur de A et ignore B.
13	Implication, B implique A = A + B' (si B, alors A). Equivalent à B $\geq$ A.
14	OR = A + B. Retourne A OR B.
15	Un ou Activé. Retourne toujours 1, peu importe les entrées A et B.

Avec plus de deux valeurs d'entrée, il y a trop de fonctions pour que chacune ait un nom ou une description. Par conséquent, on fait référence à ces fonctions par leur numéro, au lieu que par un nom. Par exemple, F_8 constitue le AND entre A et B pour des fonctions à deux entrées, et F_{14} correspond à l'opération OR. Sans doute, le seul problème est de déterminer le numéro de fonction. Par exemple, étant donné la fonction de trois variables F = AB + C, quel est le numéro de fonction correspondant ? Ce nombre est facile à calculer si l'on regarde la table de vérité de cette fonction (voir la table 14). Si on considère les valeurs de A, B et C comme des bits dans un nombre binaire, avec C représentant le bit fort et A le bit faible, ils produisent les nombres binaires de la plage zéro à sept. Il y a un résultat de zéro ou de un associé à chacune de ces chaînes de bits. Si on construit une valeur binaire en plaçant le résultat de la fonction dans la position des bits spécifiée par A, B et C, le nombre binaire obtenu est le numéro de fonction. Considérez la table de vérité pour F = AB + C :

CBA :	7	6	5	4	3	2	1	0
F = AB + C :	1	1	1	1	1	0	0	0

Si on traite la valeur de la fonction F comme un nombre binaire, il produit la valeur $F8_{16}$ ou 248_{10}. On note d'habitude les numéros de fonction en décimal.

Ceci permet également de déduire pourquoi il y a 2 ^ (2^n) fonctions pour *n* variables : si vous avez *n* variables, il y a 2^n bits dans le numéro de fonction. Si vous avez *m* bits, il y a 2^m valeurs différentes. Par conséquent, pour *n* valeurs d'entrée il y a m = 2^n bits possibles et 2^m ou 2 ^(2^n) fonctions possibles.

2.3 Manipulation algébrique d'expressions booléennes

On peut transformer une expression booléenne en une autre expression équivalente en appliquant les postulats et les théorèmes de l'algèbre booléenne. Ceci est important si vous avez besoin de mettre une expression donnée en *forme canonique* (une forme standardisée),ou bien si vous avez besoin de minimiser le

nombre de littéraux (variables primées et non primées) ou de termes dans une expression. Minimiser des termes et des expressions peut être important parce que les circuits électriques consistent souvent en de composantes individuelles qui réalisent chaque terme ou littéral d'une expression donnée. Minimiser l'expression équivaut à utiliser moins de composantes électriques et, par conséquent, à réduire le coût du système.

Malheureusement, il n'y a pas de règles fixes d'optimisation. Tout comme les démonstrations mathématiques, ces transformations demandent de l'expérience pour être réalisées avec facilité. Néanmoins, quelques exemples peuvent montrer le principe :

```
ab + ab' + a'b          = a(b+b') + a'b                 Postulat 4
                        = a•1 + a'b                     P5
                        = a + a'b                       Théorème 4
                        = a + a'b + 0                   Th3
                        = a + a'b + aa'                 P5
                        = a + b(a+a')                   P4
                        = a + b•1                       P5
                        = a + b                         Th4


(a'b + a'b' + b')'      = (a'(b+b') + b')'              P4
                        = (a'+b')'                      P5
                        = ( (ab)')'                     Th8
                        = ab                            Définition du NOT


b(a+c) + ab' + bc' + c = ba + bc + ab' + bc' + c        P4
                        = a(b+b') + b(c+c') + c         P4
                        = a•1 + b•1 + c                 P5
                        = a + b + c                     Th4
```

Bien que cet exemple fait usage des transformations algébriques pour simplifier une expression booléenne, on peut se servir des mêmes transformations pour d'autres buts. Par exemple, la prochaine section décrit la forme canonique des expressions booléennes. Les formes canoniques sont rarement optimales.

2.4 Les Formes canoniques

Etant donné qu'il existe un nombre fini de fonctions booléennes uniques pour *n* variables d'entrée, on peut cependant obtenir, avec ce même nombre de variables, un nombre infini d'expressions logiques possibles ; il y aura certes une infinité d'expressions équivalentes (c'est-à-dire, qui produisent le même résultat à partir des mêmes entrées). Afin d'éliminer plus facilement les confusions éventuelles, les concepteurs logiques se servent de la forme canonique (ou standardisée) pour spécifier une fonction booléenne. Pour toute fonction possible, il existe une forme canonique unique. Ceci rend le traitement de ces fonctions beaucoup plus clair.

Actuellement, il existe plusieurs types de formes canoniques. Ici, on en décrira seulement deux et on emploiera seulement le premier, qu'on nomme *somme des minterms* ; le second est nommé *produit des maxterms*. A l'aide du principe de dualité, la conversion entre ces deux formes canoniques est très facile.

Un *term* est une variable, ou bien un produit (AND) de divers littéraux. Par exemple, si on a deux variables, A et B, il y a huit *terms* possibles : A, B, A', B', A'B', A'B, AB' et AB. Pour trois variables on a 26 *terms* différents : A, B, C, A', B', C', A'B', A'B, AB', AB, A'C', A'C, AC', AC, B'C', B'C, BC', BC, A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC et ABC. Comme vous voyez, si le nombre de variables augmente, le nombre de *terms* augmente exponentiellement. Un *minterm* est un produit contenant exactement *n* littéraux. Par exemple, les *minterms* de deux variables sont A'B', AB', A'B et AB. De même, les *minterms* de trois variables sont A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC et ABC. En général, il y a 2^n *minterms* pour *n* variables. L'ensemble des possibles *minterms* est très facile à générer, puisqu'il correspond à la séquence des nombres binaires :

Table 12 : Minterms pour trois variables d'entrée

Equivaleut binaire	Minterm

(CBA)	
000	A'B'C'
001	AB'C'
010	A'BC'
011	ABC'
100	A'B'C
101	AB'C
110	A'BC
111	ABC

On peut spécifier *toute* fonction booléenne avec une somme (OR) de minterms. Etant donné la fonction F_{248} = AB + C, la forme canonique équivalente est ABC + A'BC + AB'C + A'B'C + ABC'. Algébriquement, on peut montrer que c'est équivalent à ce qui suit :

```
ABC + A'BC + AB'C + A'B'C + ABC'          = BC(A+A') + B'C(A+A') + ABC'
                                          = BC•1 + B'C•1 + ABC'
                                          = C(B+B') + ABC'
                                          = C + ABC'
                                          = C + AB
```

Certainement, la forme canonique n'est pas la forme optimale. Néanmoins, une fonction représentée selon la somme des *minterms* présente un grand avantage : sa table de vérité est très facile à générer. Et c'est également très simple d'en générer l'équation logique depuis la table de vérité.

Pour construire une table de vérité à partir d'une forme canonique, convertir simplement chaque *minterms* en sa valeur binaire, en substituant 1 aux variables non primées et 0 aux variables primées. Puis, placer un "1" à la position correspondante (spécifiée par la valeur binaire du *minterm*) de la table de vérité :

1) Convertir les *minterms* en son équivalent binaire :

F_{248} = CBA + CBA' + CB'A + CB'A' +C'BA

= 111 + 110 + 101 + 100 + 011

2) Mettre un "1" dans la table de vérité pour toutes les entrées qui correspondent à l'expression ci-dessus :

Table 13 : Créer une table de vérité à partir d'une somme de *minterms*, étape 1

C	B	A	F = AB + C
0	0	0	
0	0	1	
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Finalement, placez un zéro dans toutes les entrées de la table qui n'ont pas été remplies par des "1" :

Table 14 : Créer une table de vérité à partir d'une somme de minterms, étape 2

C	B	A	F = AB + C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Inversement, générer une fonction logique à partir d'une table de vérité est également simple. D'abord, localisez les entrées dans la table qui ont été remplies avec des "1". Dans la table ci-dessus, il s'agit des dernières cinq entrées. Le nombre des entrées contenant "1" détermine le nombre de *minterms* présent dans l'équation canonique. Pour générer les *minterms* individuels, remplacez les "1" par des A, B ou C et les "0" par des A', B' ou C'. Puis, calculez la somme de ces éléments. Dans l'exemple en question, F_{248} contient "1" pour CBA = 111, 110, 101, 100 et 011. Par conséquent, F_{248} = CBA + CBA' + CB'A + CB'A' +C'BA. Le premier terme, CBA, vient de la dernière entrée de la table. C, B et A contiennent tous "1", on génère donc le *minterm* CBA (ou ABC, si vous préférez). Le terme de l'avant-dernière entrée contient 110 pour CBA, on obtient donc le *minterm* CBA'. De même, 101 produit CB'A, 100 produit CB'A' et 011 produit C'BA. Sans doute, OR et AND sont tous les deux commutatifs, donc on peut arranger les terms obtenus à notre loisir et de la façon la plus convéniente. Le principe fonctionne également pour tout nombre de variables. Considérez la fonction F_{53504} = ABCD + A'BCD + A'B'CD + A'B'C'D. En mettant 1 aux positions de la table de vérité qui correspondent à chaque term, on obtient :

Table 15 : Créer une table de vérité avec quatre variables à partir des *minterms*

D	C	B	A	F = ABCD + A'BCD + A'B'CD + A'B'C'D
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	1
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	
1	1	1	0	1
1	1	1	1	1

Tous les éléments restants de la table de vérité contiennent zéro.

Probablement, la manière la plus simple pour générer la forme canonique d'une fonction booléenne est de construire d'abord la table de vérité pour cette fonction et ensuite établir la forme canonique depuis la table de

vérité. On utilisera cette technique, par exemple, en convertissant entre les deux formes canoniques que ce chapitre présente. Cependant, générer une somme de minterms algébriquement est aussi simple. L'usage de la loi de distributivité et le théorème 15 (A + A' = 1) rend la tâche facile. Considérez F_{248} = AB + C. Cette fonction contient deux termes, AB et C, mais ils ne sont pas des *minterms*. Les *minterms* contiennent toutes les combinaisons des variables dans leur forme primée et non primée. On peut convertir le premier terme en une somme de *minterms* de la façon suivante :

```
AB      =       AB•1                    Par le théorème 4
        =       AB • (C+C')             Par le théorème 15
        =       ABC + ABC'              Par la loi de distributivité
        =       CBA + C'BA              Par la loi de l'associativité
```

De façon similaire, on peut convertir le second terme de F_{248} en une somme de *minterms* comme suit :

```
C       = C•1                           Par le théorème 4
        = C • (A+A')                    Par le théorème 15
        = CA + CA'                      Par la loi de distributivité
        = CA•1 + CA'•1                  Par le théorème 4
        = CA•(B+B') + CA'•(B+B')        Par le théorème 15
        = CAB + CAB' + CA'B + CA'B'     Par la loi de distributivité
        = CBA + CBA' + CB'A + CB'A'     Par la loi de l'associativité
```

La dernière étape (arranger les termes) dans ces deux conversions est optionnelle. Pour obtenir la forme canonique finale pour l'équation F_{248} on a juste besoin de faire la somme des résultats :

```
F248    =       (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A')
        =       CBA + CBA' + CB'A + CB'A' + C'BA
```

Une autre technique de canonisation est le *produit des maxterms*. Un *maxterm* est la somme (ergo OR) de toutes les variables d'entrée, primées ou non primées. Par exemple, considérez la fonction logique G suivante de trois variables :

```
G = (A+B+C) • (A'+B+C) • (A+B'+C)
```

Comme pour la somme des *minterms*, il y a exactement un produit de *maxterms* pour chaque possible fonction. Sans doute, pour chaque produit de *maxterms*, il y a une forme équivalente de somme de *minterms*. En fait, la fonction G ci-dessus est l'équivalent de :

```
F248 = CBA + CBA' + CB'A + CB'A' + C'BA = AB + C
```

Générer une table de vérité pour ce produit de *maxterms* n'est pas plus difficile que le faire pour une somme de *minterms*. Pour faire ceci, on utilise le principe de dualité. Souvenez-vous que ce principe consiste à passer de AND à OR et à remplacer les 0 pour les 1 (et vice-versa). Par conséquent, pour construire une table de vérité vous devriez d'abord permuter les littéraux primés et non primés. En G ci-dessus, ceci produit :

```
G = (A + B + C) • (A' + B + C) • (A + B' + C)
```

La prochaine étape consiste à appliquer le principe de DeMorgan :

```
G = A'B'C' + AB'C' + A'BC'
```

Finalement, vous aurez besoin de permuter entre les 0 et les 1. Ce qui veut dire que vous placerez des 0 aux endroits de la table de vérité qui correspondent aux termes ci-dessus et des 1 à tous les autres. Ceci remplira de zéros les premières trois entrées ; remplir les entrées restantes avec des 1 produit F_{248}.

Vous pouvez facilement convertir entre ces deux formes canoniques en générant la table de vérité d'une forme et travailler à rebours sur cette table pour produire la seconde forme. Par exemple, considérez la fonction de deux variables F_7 = A + B. La somme des *minterms* est A'B + AB' + AB. La table de vérité prend la forme :

Table 16 : F_7, table de vérité OR pour deux variables

F_7	A	B

0	0	0
0	1	0
1	0	1
1	1	1

Le travail à rebours pour obtenir le produit des maxterms consiste à localiser toutes les entrées qui ont un résultat de zéro. C'est l'entrée où le résultat entre A et B vaut zéro, ce qui donne G = A'B'. Cependant, on a encore besoin de convertir toutes les variables pour obtenir G = AB. Par le principe de dualité, il ne nous reste que à invertir les opérateurs AND et OR, ce qui nous donne finalement G = A + B. C'est la forme canonique correspondant au produit des *maxterms*.

Etant donné que travailler avec les produits de *maxterms* est un peu plus désordonné que travailler avec des sommes de *minterms*, ce livre utilisera généralement les sommes des *minterms*. De plus, c'est plus commun dans le travail en algèbre booléenne. Cependant, si vous étudiez conception logique, vous trouverez les deux formes.

2.5 Simplification de fonctions booléennes

Puisqu'il y a une variété infinie de fonctions booléennes pour *n* variables, mais seulement un nombre fini de fonctions uniques pour ces *n* variables, vous pourriez vous demander s'il existe une méthode permettant de simplifier une fonction donnée pour produire une forme optimale. Pour ce faire, on peut certainement se servir de transformations algébriques, mais ce procédé ne garantit pas de résultat optimal. Il y a pourtant deux méthodes qui *réduisent* une fonction booléenne donnée en sa forme optimale : la table de Karnaugh et la méthode des implicants premiers. Ce livre traitera seulement la méthode de Karnaugh ; pour l'autre, voir tout ouvrage sur la conception logique.

Puisqu'il existe une forme optimale pour toute fonction logique, il vous surprendra peut-être le fait que nous n'allons pas utiliser les formes optimales. Il y a deux raisons à cela. D'abord, il peut y avoir diverses formes optimales et rien ne garantit qu'une forme optimale donnée soit unique. Ensuite, il est facile de convertir entre forme canonique et table de vérité.

La méthode de la table de Karnaugh est pratique seulement pour des fonctions avec deux, trois ou quatre variables. Avec soin, vous pouvez vous en servir avec cinq ou six variables, mais la méthode devient trop encombrante à ce point là. Pour plus de six variables, ce système de simplification n'a plus d'utilité[3].

La première étape avec la méthode de Karnaugh consiste à construire une table de vérité bidimensionnelle pour la fonction (voir figure 2.1)

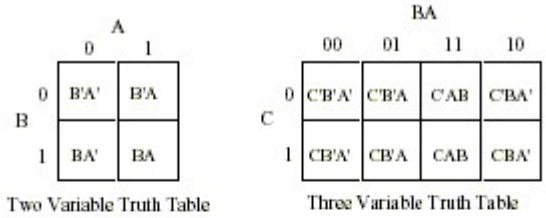
Attention : observez soigneusement ces tables de vérité. Elles ne se servent pas de la même disposition qu'on a vu précédemment. En effet, la progression des valeurs est 00, 01, 11, 10 et non 00, 01, 10, 11. Ceci est très important ! Si vous organisez la table de vérité en séquence binaire, la méthode ne fonctionnera pas. Nous allons appeler cette table une *carte de vérité* pour la distinguer d'une table de vérité standard.

En supposant votre fonction booléenne en forme canonique (somme de *minterms*) insérez des "1" dans toutes les entrées de la table qui correspondent à un *minterm*. Placez des "0" dans ce qui reste. Par exemple, considérez la fonction de trois variables F = C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA. La figure 2.2 montre la table de Karnaugh pour cette fonction.

L'étape suivante consiste à dessiner des rectangles autour de groupes rectangulaires de "1". Les rectangles que vous encerclez doivent avoir des côtés dont les longueurs sont des puissances de deux. Pour des fonctions de trois variables, les rectangles peuvent avoir des côtés de longueur 1, 2 et 4. L'ensemble des rectangles que vous dessinez doit couvrir la totalité des "1" dans la table. L'astuce consiste à dessiner tous les rectangles possibles, sans qu'un rectangle soit totalement encerclé par d'autres. Notez qu'un rectangle peut en chevaucher

[3]Cependant, on peut troujours écrire un programme qui se sert de cette méthode pour simplifier des fonctions de sept ou plus variables.

un autre s'il ne peut pas l'entourer complètement. Dans la table de la figure 2.2, il y a trois de ces rectangles (voir figure 2.3).



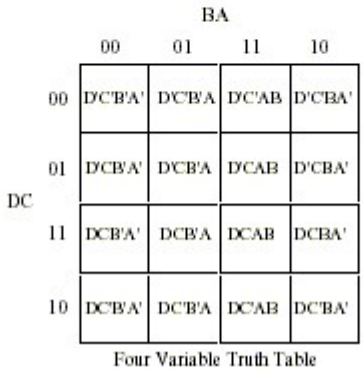


Figure 2.1 Des tables de Karnaugh de deux, trois et quatre dimensions

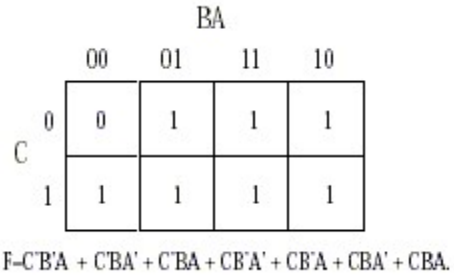


Figure 2.2 Une table de Karnaugh

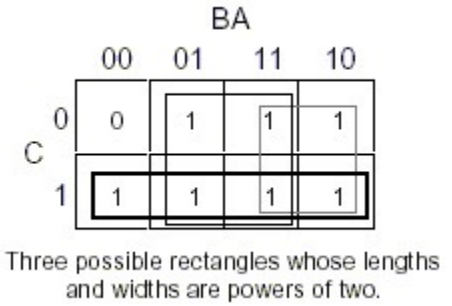


Figure 2.3 Formation de rectangles par groupes de 1

Chaque rectangle représente un terme dans la fonction booléenne simplifiée. Par conséquent, celle-ci contiendra seulement trois terms. Vous construisez chaque terme en utilisant un procédé d'élimination. Vous éliminerez toute variable apparaissant à la fois dans sa forme primée et non primée à l'intérieur de chaque rectangle. Considérez le rectanble long et maigre situé dans la ligne C = 1. Ce rectangle contient soit A soit B

dans leurs formes primées et non primées. Donc, on peut éliminer A et B du terme. Puisque ce rectangle est situé dans la région C = 1, il représente le littéral C isolé.

Maintenant, considérez le carré du centre. Celui-ci comprend C, C', B, B' et A. Par conséquent, il représente un seul terme A. De même, l'autre carré moins foncé contient C, C', A, A' et B. Il représente un seul terme B.

La fonction, optimale, finale est la somme (OR) des termes représentés par les trois quadrilatères, donc F=A+B+C. Les rectangles ne contenant que des zéros ne seront pas pris en compte.

Quand vous encerclez des groupes de "1", vous devez considérer le fait qu'une table de Karnaugh constitue un tore (c'est-à-dire, c'est comme si elle était enroulée autour d'un cylindre). Si un encerclement touche le bord droit, il peut continuer dans le bord gauche et s'il atteint le bord haut, il peut continuer dans le bord bas (et vice-versa). Ceci fournit des possibilités additionnelles d'encerclement. Considérez la fonction F = C'B'A' + C'BA' + CB'A' + CBA'. La figure 2.4, montre la table de cette fonction.

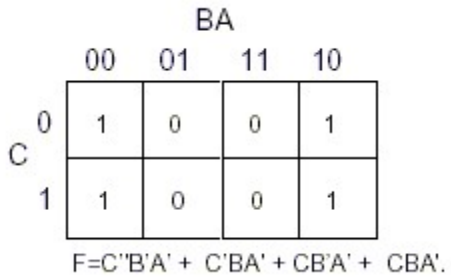


Figure 2.4 Table de Karnaugh pour la fonction F = C'B'A' + C'BA' + CB'A' + CBA'

A première vue, vous allez penser qu'il y a deux rectangles possibles, comme le montre la figure 2.5. Cependant, pusiqu'une table de Karnaugh est un object continu ayant les côtés connectés, on peut construire un seul rectangle, comme montre la figure 2.6.

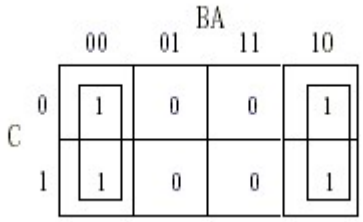


Figure 2.5 Première tentative d'encercler des rectangles

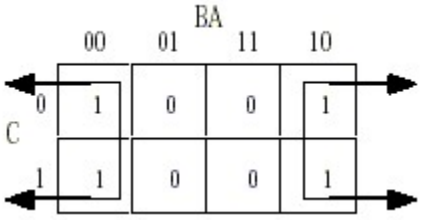


Figure 2.6 Encerclement correct pour la fonction F = C'B'A' + C'BA' + CB'A' + CBA'

Et alors ? Pourquoi se préoccuper d'avoir un rectangle au lieu de deux dans la table de Karnaugh ? La réponse est que, plus grand est le carré obtenu, plus grand sera le nombre de termes qui seront éliminés. Moins de rectangles on a, moins de termes apparaissent dans la fonction finale. Par exemple, les deux rectangles de la figure 2.5 génèrent une fonction avec deux termes. Le premier rectangle (à la gauche) élimine la variable C en laissant A'B' dans le terme. Le second rectangle (à la droite) élimine également la variable C, en laissant comme terme BA'. Par conséquent, la table de vérité produit l'équation F = A'B' + A'B et on sait que ceci n'est pas optimal (voir théorème 13). Maintenant, considérez la table de la figure 2.6. Ici, on a un seul rectangle, donc, la fonction résultante aura seulement un terme. Sans doute, ceci est plus optimal qu'une équation de deux termes. Puisque le rectangle comprend C, C' et B, B', le seul terme qui reste est A'. La fonction booléenne est donc réduite à F = A'.

Il y a seulement deux cas où la méthode de Karnaugh ne peut fonctionner correctement : une table qui ne contient que des "0" et une table qui ne contient que des "1". Ces deux cas correspondent aux fonctions booléennes F = 0 et F = 1 respectivement. Elles sont pourtant faciles à générer, il suffit de regarder la table.

Une chose importante à considérer en vous servant de la méthode de Karnaugh est que vous *tenterez* toujours d'obtenir les plus gros rectangles dont les longueurs des côtés sont des puissances de deux. Vous devez faire cela même dans les cas où les rectangles se chevauchent (sauf quand un rectangle en contient un autre). Considérez la fonction F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA. Elle produit la table de la figure 2.7.

La tentation initiale est de créer des ensembles comme dans la figure 2.8. Cependant, la solution correcte apparaît à la figure 2.9.

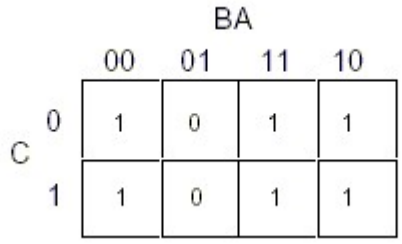


Figure 2.7 Table pour la fonction F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

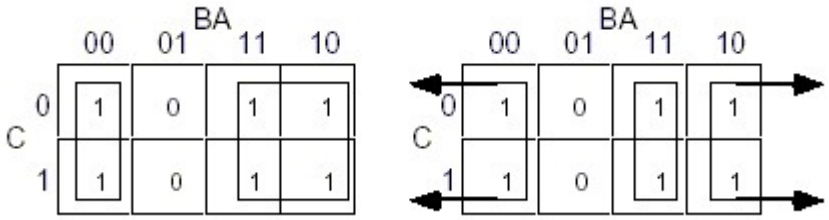


Figure 2.8 Choix tendanciel des rectangles (incorrect)

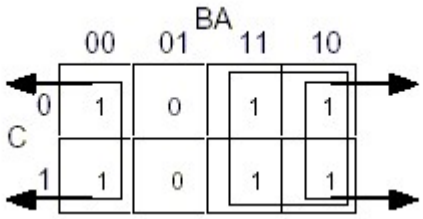


Figure 2.9 Ensemble correct de rectangles pour F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

Toutes les solutions produiront une fonction de deux termes. Cependant, les premières deux (figure 2.8) produisent F = B + A'B' et F = AB + A'. Et la troisième (figure 2.9) produit F = B + A'. Certainement, la dernière forme est plus optimale que les autres deux (voir théorèmes 11 et 12).

Pour des fonctions de trois variables, la taille du rectangle détermine le nombre de termes qu'il représente :

- Un rectangle encerclant une case représente un *minterm*. Celui-ci aura trois littéraux.
- Un rectangle encerclant deux cases représente un terme constitué de deux littéraux.
- Un rectangle encerclant quatre cases représente un terme constitué d'un seul littéral.
- Un rectangle encerclant huit cases représente la fonction F = 1.

Les tables créées à partir de fonctions de quatre variables peuvent être bien plus délicates. Ceci parce qu'il y aura plus de chances de tomber sur des rectangles cachés le long des bords ou dans les coins. La figure 2.10, montre certains de ces rectangles.

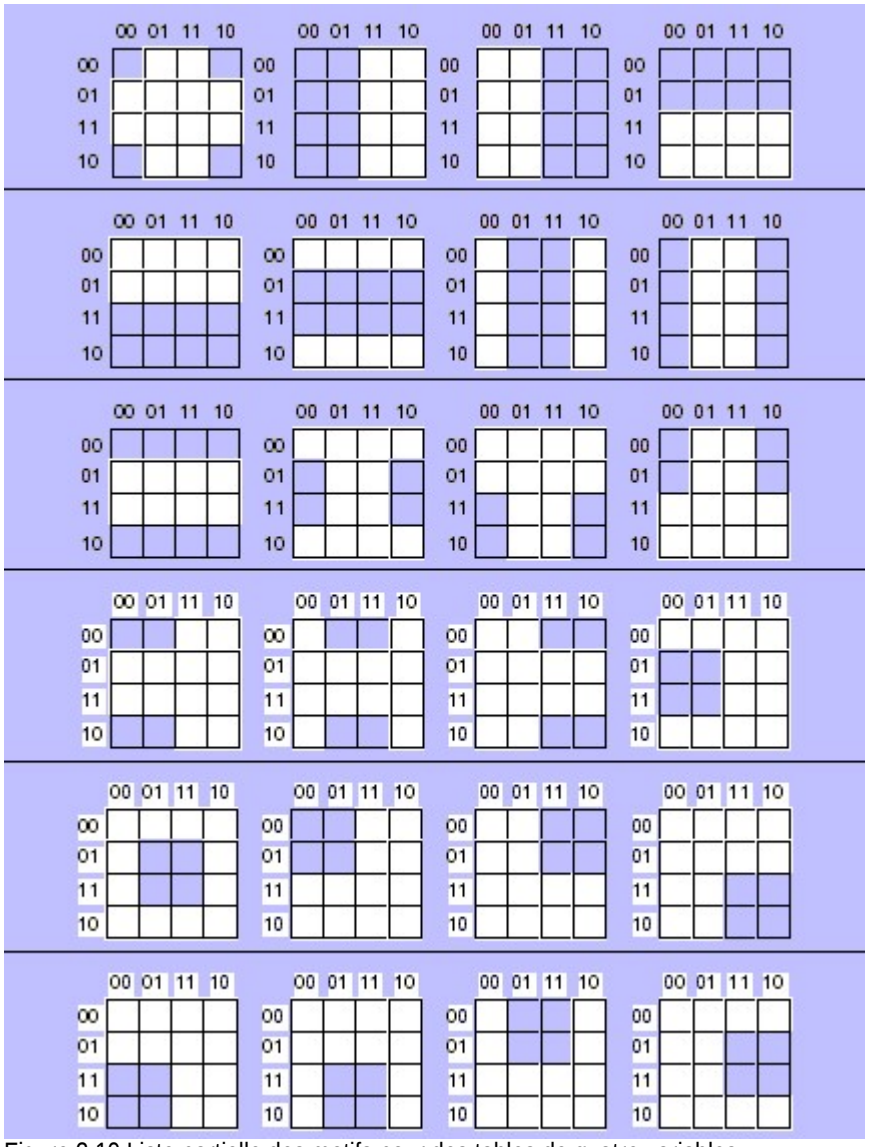


Figure 2.10 Liste partielle des motifs pour des tables de quatre variables

Cette liste des modèles est loin de décrire toutes les combinaisons possibles ! Par exemple, ce diagramme ne montre aucun rectangle 1x2. Vous devez être très soigneux en travaillant avec quatre variables, pour vous assurer d'encercler les plus gros rectangles possibles, spécialement lors des chevauchements. Ceci s'applique surtout quand des rectangles sont à proximité d'un des quatre bords.

Comme pour les fonctions de trois variables, c'est la taille des rectangles qui détermine, dans la table d'une fonction de quatre variables, le nombre de termrs qu'elle représente :

- Un rectangle encerclant une case représente un *minterm*. Celui-ci aura quatre littéraux.
- Un rectangle encerclant deux cases représente un terme constitué de trois littéraux.
- Un rectangle encerclant quatre cases représente un terme constitué de deux littéraux.
- Un rectangle encerclant huit cases représente un terme constitué d'un seul littéral.
- Un rectangle encerclant seize cases représente la fonction F = 1.

Le dernier exemple qui suit montre l'optimisation d'une fonction contenant quatre variables. Cette fonction est F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA'. La table de Karnaugh apparaît à la figure 2.11.

Il y a, pour cette fonction, deux ensembles possibles de rectangles maximaux, chacun produisant trois terms (voir figure 2.12). Les deux fonctions résultantes sont équivalentes ; les deux sont optimales, comme vous pouvez le voir[4]. Les deux remplissent parfaitement nos besoins.

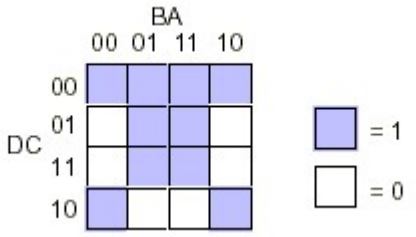


Figure 2.11 Table de Karnaugh pour la fonction F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA'

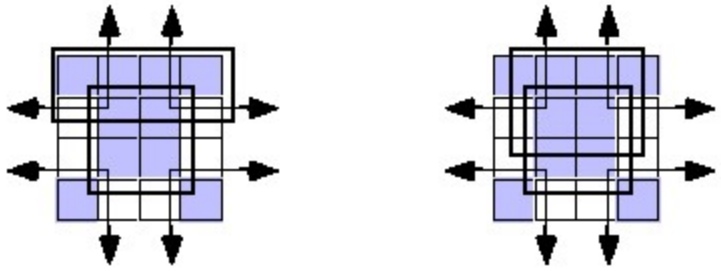


Figure 2.12 Deux combinaisons dans l'encerclement des valeurs portant à un résultat de trois *minterms*.

D'abord, considérez le terme constitué par le rectangle formé des quatre coins. Ce rectangle contient B, B', D et D', on peut donc éliminer ces littéraux ; ce qui reste est C' et A', donc il représente le terme C'A'.

Le second rectangle commun aux deux solutions de la figure, est le rectangle formé par les quatre cases centrales. Celui-ci comprend A, B, B', C, D et D'. Après avoir éliminé B, B', D et D' (car les deux formes primées et non primées de ces littéraux sont présentes), on obtient CA comme résultat.

La table de gauche de la figure 2.12 a un troisième terme représenté par le rectangle en haut. Celui-ci comprend les variables A, A', B, B', C' et D'. Puisqu'il contient A, A', B et B', on peut éliminer ces littéraux. Ce qui laisse le terme C'D'. Par conséquent, la fonction représentée par la table de gauche est F = C'A' + CA + C'D'.

La table de droite de la figure 2.12 a un troisième terme représenté par le rectangle qui chevauche le carré central. Ce rectangle comprend les variables A, B, B', C, C' et D'. On peut éliminer B, B', C et C' car elles apparaissent en version primée et non primée, ce qui laisse le terme AD. Par conséquent, la fonction représentée par la table[5] de droite est F = C'A' + CA + AD'.

Puisque les deux expressions contiennent le même nombre de termes et le même nombre d'opérateurs, elles sont équivalentes. A moins d'avoir une raison spéciale pour préférer une expression à l'autre, on peut utiliser n'importe laquelle des deux.

2.6 En tout cas, quel est le rapport avec les ordinateurs ?

Bien qu'il y a une relation faible entre les fonctions et les expressions boléennes dans des langages comme C ou Pascal, ce n'est pas le cas de l'assembleur, il ne faut donc pas se surprendre si on y consacre autant d'attention ici. La relation entre logique booléenne et systèmes informatiques est, en effet, très forte. Il existe une correspondance directe entre fonctions logiques et circuits électroniques. Les ingénieurs qui conçoivent des CPU et d'autres composants relatifs à l'ordinateur doivent avoir une connaissance intime de cette matière. Même si

[4]Rappelez-vous : l'existence d'une seule solution optimale n'est pas garantie.
[5]L'original ici parle de "fonction de droite", mais le terme correct est "table". Sans doite une erreur de l'auteur, n.d.t.

vous ne pensez pas créer vos propres circuits, comprendre ces concepts est important pour tirer le meilleur parti d'un ordinateur.

2.6.1 Correspondance entre circuits électroniques et fonctions booléennes

Il y a une relation bilatérale (ou réciproque) entre circuits électriques et fonctions booléennes. On peut réaliser un circuit à partir de toute fonction booléenne et vice-versa. Puisque ces fonctions ne demandent que des opérateurs comme AND, OR et NOT, on peut construire tout circuit utilisant exclusivement ces opérateurs. Les fonctions AND, OR et NOT correspondent aux portes AND, OR et inverseur (NOT) (voir figure 2.13).

Un fait intéressant à noter est qu'on pourrait implémenter n'importe quel circuit en n'utilisant que des portes NAND (voir figure 2.14).

Pour démontrer qu'on peut construire toute fonction booléenne en n'utilisant que des NAND, on n'a besoin que de montrer comment construire un inverseur, une porte AND et une porte OR à partir de portes NAND (puisqu'on peut créer toute fonction en n'utilisant que AND, OR et NOT). Construire un inverseur est facile, il suffit de connecter ensemble deux entrées (voir figure 2.15).

Une fois construit un inverseur, implémenter une porte AND est encore plus facile - il suffit d'inverser la sortie d'une porte NAND. D'après tout, NOT(NOT(A AND B)) est équivalent à A AND B (voir figure 2.16). Sans doute, cela prend deux portes NAND pour construire une porte AND, mais personne ne prétend que des circuits construits uniquement avec des NAND soient optimaux ; on veut juste montrer que c'est possible.

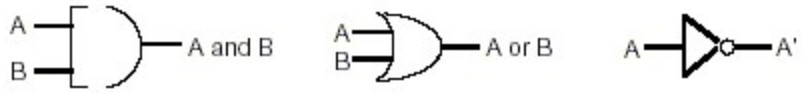


Figure 2.13 Des portes AND, OR et inverseur

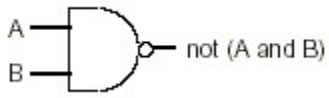


Figure 2.14 Une porte NAND

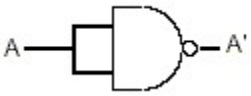


Figure 2.15 Inverseur construit à partir d'une porte NAND

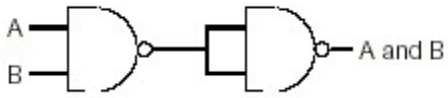


Figure 2.16 Construction d'une porte AND à partir de deux portes NAND

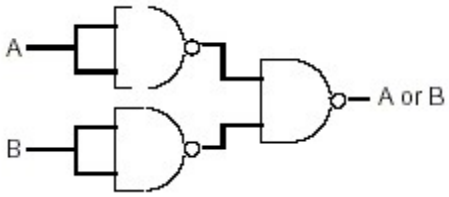


Figure 2.17 Construction d'une porte OR à partir de portes NAND

La porte qui reste à synthétiser est la porte OR. On peut la construire aisément à partir de portes NAND si on applique le théorème de DeMorgan.

```
(A OR B)'      =  A' AND B'          Théorème de DeMorgan
A OR B         = (A' AND B')'        Inversion des deux côtés de l'équation
A OR B         =  A' NAND B'         Définition de l'opération NAND
```

En appliquant ces transformations, on obtient le circuit de la figure 2.17.

Toute cette discussion doit sans doute vous étonner. D'après tout, pourquoi ne pas utiliser des portes AND, OR et NOT directement ? Il y a deux raisons à ceci. D'abord, les portes NAND sont moins onéreuses à construire que les autres portes. Ensuite, il est beaucoup plus facile de construire des circuits intégrés complexes en utilisant les mêmes blocs de construction de base, plutôt que de le faire à partir de portes de base différentes.

Notez que c'est également possible de construire n'importe quel circuit logique en n'utilisant que des portes NOR[6]. La correspondance entre les logiques NAND et NOR est orthogonale à la correspondance entre les deux formes canoniques qu'on a vues dans ce chapitre (somme de *minterms* versus produit de *maxterms*). Bien que la logique du NOR est utile pour certains circuits, la plupart des concepteurs utilise NAND. Voir les exercices pour plus d'exemples.

2.6.2 Circuits combinatoires

Un circuit combinatoire est un système contenant des opérations booléennes de base (AND, OR, NOT), certaines entrées et un ensemble de sorties. Puisque chaque sortie correspond à une fonction logique individuelle, un circuit combinatoire implémente souvent diverses fonctions booléennes. C'est très important de se souvenir de ce fait - chaque sortie représente une fonction booléenne différente.

Un CPU d'ordinateur est construit sur la base de plusieurs circuits combinatoires. Par exemple, on peut implémenter un circuit d'addition en utilisant des fonctions booléennes. Soient deux nombres d'un bit, A et B. Vous pouvez produire la somme et la retenue sur un bit de cette addition en utilisant les fonctions booléennes :

```
S       =       AB' + A'B               Somme entre A et B
C       =       AB                      Retenue de l'addition A + B
```

Ces deux fonctions booléennes constituent un *demi-additionneur* (half-adder). Les ingénieurs électroniciens l'appellent ainsi parce qu'il additionne deux bits mais il ne peut pas ajouter la retenue d'une opération précédente. Un *additionneur complet* (full adder) fait la somme de trois entrées d'un bit (deux bits plus une retenue d'une précédente addition) et produit deux sorties : la somme et la retenue. Les deux équations pour un additionneur complet sont :

$$S = A'B'C_{in} + A'BC_{in}' + AB'C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Bien que ces équations produisent un résultat d'un seul bit (en ignorant la retenue), il est facile de construire une somme de *n* bits en combinant des circuits d'addition (voir figure 2.18). Comme cet exemple l'illustre clairement, on peut utiliser des fonctions logiques pour implémenter des opérations arithmétiques et booléennes.

Un autre circuit commun est le décodeur *sept segments* (seven-segment decoder). C'est un circuit combinatoire qui accepte quatre entrées et détermine quel des sept segments (dans un panneau les illustrant) doit être allumé (un logique) ou éteint (zéro logique). Puisqu'un affichage de ce type contient sept valeurs de sortie (une pour chaque segment), il y aura sept fonctions logiques associées à cet affichage (les segments de

[6]NOR n'est autre chose que NOT(A OR B).

zéro à six). Voir la figure 2.19 pour la numérotation des segments. La figure 2.20 montre la disposition de l'état *allumé* des segments pour les dix valeurs décimales.

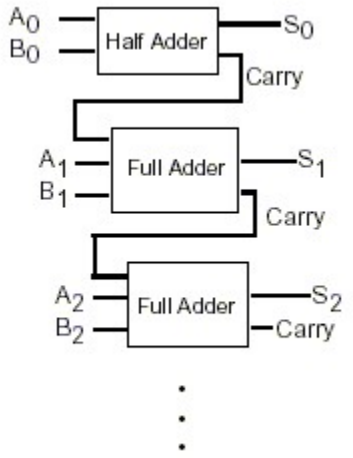


Figure 2.18 Un additionneur de n bits se servant de demi-additionneurs et d'additionneurs complets

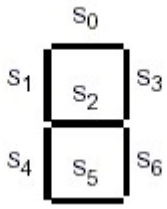


Figure 2.19 Affichage d'un décodeur sept segments

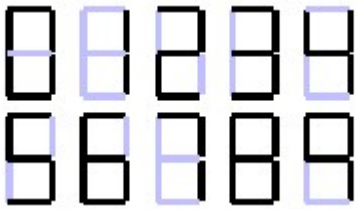


Figure 2.20 Etats du décodeur pour les valeurs de 0 à 9

Les quatre entrées pour chacune de ces sept fonctions booléennes sont les quatre bits d'un nombre binaire sur la plage 0...9, où D est le bit fort et A le bit faible. Chaque fonction logique produit un résultat de 1 (le segment est allumé) pour une entrée donnée, pour tout segment qui doit être allumé. Par exemple, S_4 (le segment quatre) serait allumé pour les valeurs binaires 0000, 0010, 0110 et 1000. Pour chaque valeur qui allume un segment, on a un *minterm* dans l'équation logique :

$$S_4 = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'$$

Comme second exemple, S_0 est allumé pour les valeurs zéro, deux, trois, cinq, six, sept, huit et neuf. Par conséquent, la fonction logique pour S_0 est :

$$S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A$$

Vous pouvez générer les autres cinq fonctions de façon similaire (voir les exercices).

Les circuits combinatoires représentent les bases pour beaucoup de composants d'un système informatique. Vous pouvez construire des circuits pour l'addition, la soustraction, la comparaison, la multiplication, la division et plusieurs autres opérations.

2.6.3 Logique séquentielle et synchrone

Un des problèmes principaux avec la logique combinatoire est qu'elle n'a pas de mémoire. En théorie, toute sortie d'une fonction booléenne dépend seulement des entrées courantes. Tout changement dans ces valeurs se reflète immédiatement dans les sorties[7]. Mais, il faut cependant que les ordinateurs aient la capacité de se souvenir des résultats d'une opération antérieure. C'est le sujet de la logique séquentielle et synchronisée.

Une cellule de mémoire est un circuit électronique qui retient une valeur d'entrée après la suppression de cette valeur d'entrée. L'unité de mémoire la plus basique est pourtant le *set/reset flip-flop* (ou *bistable*). On peut construire un tel engin via deux portes NAND, comme montré à la figure 2.21.

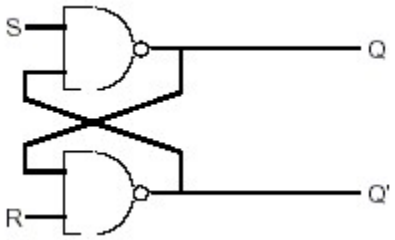


Figure 2.21 Un set-reset flip-flop construit à partir de deux portes NAND

Les entrées S et R sont normalement à l'état haut (valant 1). Si on met temporairement l'entrée S à 0 et on la remet ensuite à 1, ceci force la sortie Q à valoir 1. De même, si l'on fait passer l'entrée R à 0 et ensuite on la remet à 1, ceci produit la valeur 0 à la sortie Q. Q' est généralement l'inverse de Q. Notez que si, S et R valent 1, la sortie Q dépend de Q. Quelle qu'était sa valeur, la porte NAND continuera à produire le même résultat. Si Q valait 1, alors il y a deux 1 aux entrées du NAND inférieur (Q NAND R). Ceci produit 0 à la sortie Q'. Par conséquent, les deux entrées du NAND supérieur sont 0 et 1. Ce qui produit la valeur 1 à la sortie (correspondante à la valeur d'origine de Q).

Si cette valeur d'origine était 0, alors les entrées du NAND inférieur sont Q = 0 et R = 1. Par conséquent, la sortie de cette porte NAND est 1. Et ce qui rentre dans le NAND supérieur est S = 1 et Q' = 1. Ce qui produit une sortie de zéro, et cela correspond à la valeur d'origine de Q.

Supposez que Q = 0, S = 0 et R = 1. Ceci met les deux entrées du NAND supérieur à 1 et 0 et produit 1 à la sortie Q. Faire retourner S à 1 ne change pas du tout cette sortie. Vous pouvez obtenir les mêmes résultats si Q = 1, S = 0 et R = 1. Ceci produit encore un résultat de 1 dans Q. Et cette valeur reste inaltérée, même si S passe de 0 à 1. Par conséquent, changer l'entrée S à 0 et la remettre à 1 produit 1 comme résultat (ce qui active le flip-flop). La même idée s'applique à l'entrée R, mais ceci force la sortie à 0 au lieu de 1.

Il y a cependant une lacune dans le circuit : il ne fonctionnera pas correctement si on met S et R à 0 de façon simultanée. Ceci produit 1 soit dans Q, soit dans Q' (ce qui est logiquement inconsistant). Quelle que soit l'entrée qui reste à zéro, celle qui maintient cet état le plus longtemps détermine l'état final du flip-flop. Un circuit opérant de telle manière est dit *instable*.

Le seul problème avec le flip-flop de type S/R est qu'il faut se servir de différentes entrées pour retenir une valeur de zéro ou de un. Une cellule de mémoire serait plus efficace si l'on pouvait spécifier une valeur à retenir dans une entrée et fournir, pour l'autre, une *entrée d'horloge* pour balancer cette valeur. Un tel type de flip-flop est dit un D flip-flop (pour les données) et utilise le circuit de la figure 2.22.

En présumant fixer les sorties Q et Q' à 0-1 ou 1-0, envoyer un impulsion d'horloge passant de 0 à 1 et revenant à 0 a pour effet de copier l'entrée D dans la sortie Q. Et de copier également D' dans Q'. Les exercices à la fin du chapitre s'attendent à une explication détaillée de votre part, donc étudiez ce diagramme avec soin.

Bien que retenir un simple bit est souvent important, se souvenir de groupes de bits l'est encore plus. On peut retenir une séquence de bits en combinant en parallèle différents flips-flops de type D. Concaténer une série de flips-flops pour stocker une valeur de *n* bits constitue un *registre*. Le schéma électronique de la figure 2.23 montre comment construire un registre de 8 bits à partir d'un ensemble de flips-flopss.

[7]En pratique, dans toute implémentation électronique des fonction booléennes, il y a un court délai de propagation entre le changement des entrées et les sorties correspondantes.

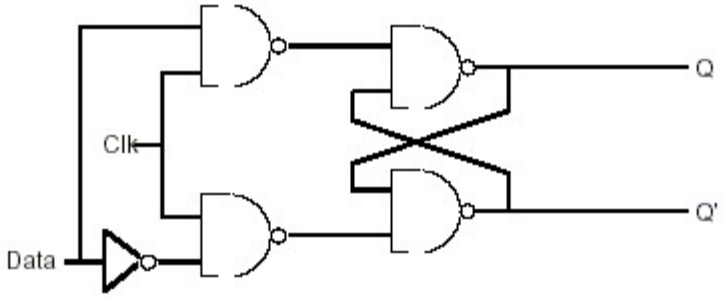


Figure 2.22 Implémentation d'un flip-flop de type D avec des portes NAND

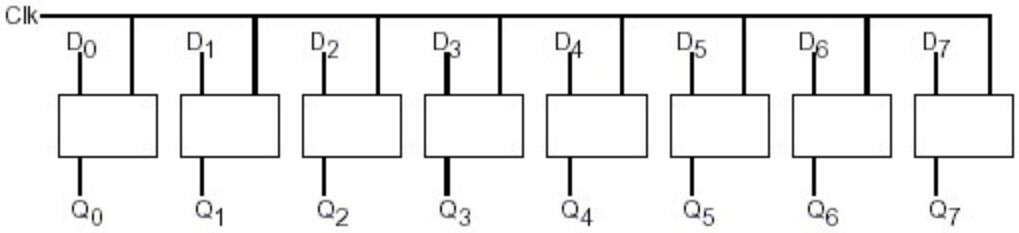


Figure 2.23 Un registre de huit bits implémenté avec huit flips-flopss de type D

Notez que les huit flips-flops utilisent une ligne d'horloge commune. Ce diagramme n'illustre pas les sorties Q', car elles sont rarement requises dans un registre.

Les flip-flops D sont utiles pour construire beaucoup d'autres circuits, en plus des registres, mais au-delà aussi. Par exemple, on peut construire un registre de décalage qui décale les bits d'une position vers la gauche à chaque impulsion d'horloge. Un registre de décalage de quatre bits est montré à la figure 2.24.

On peut également construire, à l'aide des flips-flops, un compteur comptant le nombre des fois qu'une horloge passe de 1 à 0 et revient à 1. Le circuit de la figure 2.25 implémente un compteur de quatre bits qui utilise des flips-flops D.

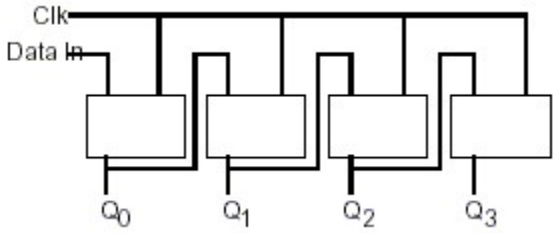


Figure 2.24 Un registre de décalage de quatre bits construit à partir de flip-flops D

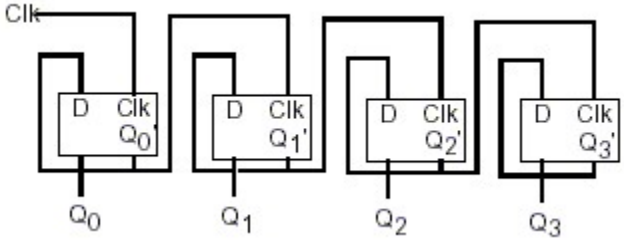


Figure 2.25 Un compteur de quatre bits construit à partir de flips-flops de type D

Cela vous paraîtra surprenant, mais on peut construire tout un CPU avec des circuits combinatoires et quelques circuits séquentiels additionnels.

2.7 D'accord, mais quel est le rapport avec la programmation, encore ?

Une fois qu'on a des registres, des compteurs et des registres de décalage, on peut construire des machines à état (state machines). L'implémentation d'un algorithme matériel se servant de ces machines est bien hors de la portée de cet ouvrage. Cependant on peut tirer une importante conclusion à partir de cette discussion : *tout algorithme qu'on peut implémenter par le logiciel, peut être aussi implémenté par le matériel.* Ce qui suggère que la logique booléenne est la base du fonctionnement de tous les systèmes informatiques modernes. Tout programme qu'on peut écrire, peut être spécifié comme une séquence d'équations booléennes.

Il va sans dire que résoudre un problème de programmation à l'aide d'un langage comme Pascal, C ou même assembleur, est beaucoup plus facile que le résoudre en utilisant des équations booléennes. Donc, c'est fort improbable que vous allez jamais écrire un programme en utilisant un ensemble de machines d'états et de circuits logiques. Néanmoins, il y a des fois où une implémentation matérielle est plus adaptée. Une solution hardware peut être d'un, de deux, de trois ou plus ordres de grandeur plus rapide qu'une solution logicielle équivalente. Par conséquent, certaines opérations critiques de performance peuvent demander une programmation matérielle.

Un fait encore plus intéressant est que l'inverse de ce qui a été dit est également vrai. Non seulement on peut implémenter toute solution logicielle dans le matériel, mais on peut aussi implémenter toute solution matérielle dans le logiciel. Et cette révélation est importante parce que beaucoup d'opérations qui normalement sont réalisées du côté matériel, sont beaucoup moins dispendieuses si on les développe de façon logicielle sur un microprocesseur. En vérité, l'application primaire de l'assembleur dans beaucoup de systèmes modernes est de remplacer des circuits électroniques complexes par un support logiciel vraiment bon marché. C'est souvent possible de remplacer des composantes électroniques coûtant de dizaines de centaines de dollars par des puces de micro ordinateur à 25$. Tout le domaine des systèmes intégrés se penche sur cette question problématique. Les systèmes intégrés (embedded systems) sont des systèmes informatiques incorporés dans d'autres produits. Par exemple, certains fours à micro-ondes, les TV, les systèmes de jeux vidéo, les lecteurs de CD et d'autres appareils domestiques contiennent un ou plus systèmes informatiques dont le seul but est de remplacer toute une conception matérielle complexe. Les ingénieurs utilisent des ordinateurs pour atteindre ce but, parce que cela comporte moins de dépenses ; de plus, réaliser des circuits logiciels est techniquement plus simple que travailler avec les circuits électroniques traditionnels.

On peut parfaitement concevoir des logiciels qui lisent des interrupteurs (variables d'entrées) et peuvent mettre en marche des moteurs, allumer ou éteindre des lumières, fermer ou ouvrir des portes, etc. (sorties des fonctions). Pour écrire de tels logiciels, vous avez besoin de comprendre les fonctions booléennes et de comprendre la manière de les implémenter selon une perspective logicielle.

Sans doute, même si vous n'écrirez jamais des logiciels destinés aux systèmes intégrés ou des logiciels manipulant des objets du monde réel, il y a une autre raison qui justifie l'étude des fonctions booléennes. Les langages de haut niveau traitent des expressions booléennes (par exemple, ces expressions qui contrôlent des instructions *if* ou *while*). En appliquant des transformations telles que le théorème de DeMorgan ou la table de Karnaugh, c'est souvent possible d'améliorer les performances des programmes. Par conséquent, étudier les fonctions booléennes est important, quel que soit l'usage que vous en ferez. Même en travaillant uniquement avec des langages de haut niveau, les fonction bolléennes permettent d'être un meilleur programmeur.

Par exemple, supposez avoir l'instruction Pascal suivante :

```
if((x=y) and (a<>b)) or ((x=y) and (c<=d)) then SomeStmt;
```

Vous pouvez utiliser la loi distributive pour simplifier ceci à :

```
if((x=y) and ((a<>b) or (c<=d)) then SomeStmt;
```

De même, on peut utiliser le théorème de DeMorgan pour réduire

```
while(not((a=b) and (c=d)) do Something;
```

à

```
while(a<>b) or (c<>d) do Something;
```

2.8 Fonctions booléennes génériques

Pour une application spécifique, on peut créer une fonction logique qui obtient un résultat donné. Supposez cependant qu'on voulait écrire un programme qui *simule* toute possible fonction. Par exemple, un des programmes accompagnant ce livre permet d'entrer une fonction booléenne arbitraire d'une à quatre variables différentes. Ce programme lit les entrées et produit les résultats nécessaires. Puisque le nombre de fonctions uniques de quatre variables est grand (65536 pour être exacts), ce n'est pas pratique d'inclure une solution spécifique pour chaque fonction. Ce qui est nécessaire est une *fonction logique générique*, une fonction qui peut calculer les résultats de toute fonction arbitraire. Cette section décrit la manière de l'écrire.

Une fonction booléenne générique de quatre variables, requiert cinq paramètres - les quatre paramètres d'entrée et un cinquième argument spécifiant la fonction spécifique à calculer. Alors qu'il y a beaucoup de façons de spécifier cette fonction, on utilisera comme cinquième argument le numéro de fonction.

Vous pourriez vous demander comment on peut effectuer le calcul d'une fonction à partir de son numéro. Cependant, gardez à l'esprit que les bits qui constituent un numéro de fonction proviennent directement de la table de vérité de cette fonction. Par conséquent, si on extrait ces bits, on peut construire la table de vérité de la fonction. En effet, si on choisit le $i_{ème}$ bit du numéro de fonction, où i = D*8 + C*4 + B*2 + A, on obtient le résultat de la fonction pour les valeurs particulières de A, B, C et D[8]. Les exemples suivants, en C et en Pascal, montrent comment écrire ces fonctions :

```
/************************************************************************/
/*                                                                      */
/* Ce programme C démontre comment écrire une fonction logique générique qui*/
/* calcule toute fonction logique de quatre variables. Etant donné que le   */
/* langage C dispose d'opérateurs de manipulation de bits, cette tâche est  */
/* simple à accomplir.                                                  */
/*                                                                      */
/************************************************************************/

#include <stdlib.h>
#include <stdio.h>

/*Fonction logique générique. Le paramètre "func" contient un nombre de     */
/*fonction de 16 bits. Ce nombre est en réalité une table de vérité encodée */
/*pour cette fonction. Les paramètres a, b, c et d sont les entrées de      */
/*cette fonction. Si on considère "func" comme un tableau de bits           */
/*2x2x2x2, cette fonction particulière sélectionne le bit "func[d,c,b,a]"   */
/*dans le tableau func.                                                  */

int generic (int func, int a, int b, int c, int d)
{
       /*retourne le bit spécifié par a, b, c et d */
       return(func >> (a + b * 2 + c * 4 + d * 8)) & 1;
}

/*Programme principal pour piloter la fonction logique générique écrite en C*/
main()
{
       int func, a, b, c, d;

       /*Répéter ce qui suit tant que l'utilisateur n'entre pas zéro*/

       do
       {
              /*Obtenir le numéro de fonction (table de vérité*/
              printf("Entrez le numéro de fonction (en hex) : ");
              scanf("%x", &func);

              /*Si l'utilisateur a entré le nombre zéro, alors arrêter le
                programme*/
              if(func != 0)
              {
```

[8]Le chapitre 5 expliquera pourquoi cette multiplication fonctionne.

```
                        printf("Entrer les valeurs pour d, c, b et a: ");
                        scanf("%d%d%d%d", &d, &c, &b, &a);
                        printf("Le résultat est : %d\n", generic(func,a,b,c,d));
                        printf("Func=%x, A=%d, B=%d, C=%d, D=%d\n",
                                func,a,b,c,d);
                }
        }while(func != 0);
}
```

Le programme Pascal qui suit est écrit pour le Pascal standard, qui ne fournit pas d'opérateurs de manipulation de bits ; donc, ce programme est plus long parce qu'il doit simuler les bits à l'aide d'un tableau d'entiers. Beaucoup de versions modernes de Pascal (spécialement Turbo Pascal) fournissent des librairies de routines intégrées pour travailler avec les bits. Ce programme aurait été beaucoup plus facile à écrire en utilisant ces caractéristiques non standard.

```
program GenericFunc(input, output);

(*Puisque le Pascal standard n'offre pas de moyen facile de manipuler        *)
(*directement les bits dans un entier, on simulera le numéro de fonction en  *)
(*utilisant un tableau de 16 entiers. "GFTYPE" est le type de ce tableau.    *)

type
        gftype = array[0..15] of integer;

var
        a, b, c, d:integer;
        fresult:integer;
        func:gftype;

(*Le Pascal standard ne fournit pas de moyen pour décaler une valeur entière *)
(*à gauche ou à droite. Par conséquent, on simulera une valeur de 16         *)
(*bits en utilisant un tableau de 16 entiers. On peut simuler des décalages  *)
(*en déplaçant les données à l'intérieur du tableau.                         *)
(*                                                                           *)
(*Noter que Turbo Pascal *est* en mesure de fournir les opérations shl et    *)
(*shr. Cependant, ce code est écrit pour le Pascal Standard et ne se limite  *)
(*pas seulement à Turbo Pascal.                                             *)
(*                                                                           *)
(*ShiftLeft décale les valeurs dans func d'une position vers la gauche et    *)
(*insère les valeurs qui sortent à droite à la position zéro.                *)

procedure ShiftLeft(shiftin:integer);
var i:integer;
begin
        for i := 15 downto 1 do func[i] := func[i-1];
        func[0] := shiftin;
end;

(*ShiftNibble décale les données de func vers la gauche de quatre positions *)
(*et insère les quatre bits a (bit faible), b, c et d (bit fort) dans les   *)
(*positions vacantes.                                                       *)

procedure ShiftNibble(d, c, b, a:integer);
begin
        ShiftLeft(d);
        ShiftLeft(c);
        ShiftLeft(b);
        ShiftLeft(a);
end;
```

```
(*ShiftRight décale les données dans func d'une position vers la droite.    *)
(*Il remplit tous les bits forts par des zéros.                             *)

procedure ShiftRight;
var i:integer;
begin
        for i := 0 to 14 do func[i] := func[i+1];
        func[15] := 0;
end;

(*ToUpper convertit un caractère minuscule en majuscule.                    *)

procedure toupper(var ch:char)
begin
        if(ch in ['a'..'z']) then ch := chr(ord(ch) - 32);
end;

(*ReadFunc lit un numéro de fonction hexadécimal entré par l'utilisateur    *)
et place cette valeur dans le tableau de la fonction bit par bit.           *)

function ReadFunc:integer;
var ch:char;
        i, val:integer;
begin
        write('Entrez le numéro de fonction (en hexadécimal):');
        for i := 0 to 15 do func[i] := 0;
        repeat
                read(ch);
                if not eoln then begin

                        toupper(ch);
                        case ch of
                                '0' : ShiftNibble(0, 0, 0, 0);
                                '1' : ShiftNibble(0, 0, 0, 1);
                                '2' : ShiftNibble(0, 0, 1, 0);
                                '3' : ShiftNibble(0, 0, 1, 1);
                                '4' : ShiftNibble(0, 1, 0, 0);
                                '5' : ShiftNibble(0, 1, 0, 1);
                                '6' : ShiftNibble(0, 1, 1, 0);
                                '7' : ShiftNibble(0, 1, 1, 1);
                                '8' : ShiftNibble(1, 0, 0, 0);
                                '9' : ShiftNibble(1, 0, 0, 1);
                                'A' : ShiftNibble(1, 0, 1, 0);
                                'B' : ShiftNibble(1, 0, 1, 1);
                                'C' : ShiftNibble(1, 1, 0, 0);
                                'D' : ShiftNibble(1, 1, 0, 1);
                                'E' : ShiftNibble(1, 1, 1, 0);
                                'F' : ShiftNibble(1, 1, 1, 1);
                               else write(chr(7), chr(8));
                        end;
                end;
        until eoln;
        val := 0;
        for i:= 0 to 15 do val := val + func[i];
        ReadFunc := val;
end;

(*Generic - Calcule la fonction logique générique spécifiée par le numéro *)
(*de fonction "func" sur les quatre valeurs d'entrée a, b, c et d. Elle   *)
(*fait ceci en retournant le bit d*8 + c*4 + b*3 + a*2 à partir de        *)
(*func.                                                                    *)

function Generic(var func:gftype; a, b, c, d:integer):integer;
```

```
begin
        Generic := func[a + b*2 + c*4 + d*8];
end;

begin   (*main*)
        repeat
                fresult := ReadFunc;
                if(fresult <> 0) then begin
                        write('Entrez les valeurs pour D, C, B et A (0/1) : ');
                        readln(d, c, b, a);
                        writeln('Le résultat est : ',Generic(func, a, b, c, d));
                end;
        until fresult = 0;
end.
```

Le code suivant démontre le pouvoir des opérateurs de manipulation de bits. Cette version du programme utilise les caractéristiques additionnelles de Turbo Pascal, permettant aux programmeurs d'effectuer facilement des opérations de décalage et de comparaisons logiques :

```
 program GenericFunc(input, output);
 const
       hex = ['a'..'f', 'A'..'F'];
       decimal = ['0'..'9'];
 var
      a, b, c, d:integer;
      fresult:integer;
      func:integer;

(*Voici une seconde version de la fonction générique écrite en Pascal,       *)
(*version utilisant les nouvelles possibilités de manipulation de bits       *)
(*de Turbo Pascal pour simplifier le programme.                              *)

function ReadFunc:integer;
var     ch:char;
        i, val:integer;
begin

        write('Entrez le numéro de fonction (en hexadécimal) : ');
        repeat

                read(ch);
                func := 0;
                if not eoln then begin

                        if(ch in Hex) then
                                func := (func shl 4) + (ord(ch) and 15) + 9
                        else if (ch in Decimal) then
                                func := (func shl 4) + (ord(ch) and 15)
                        else write(ch(7));

                end;
        until eoln;
        ReadFunc := func;
end;

(*Generic - Calcule la fonction logique générique spécifiée par le numéro   *)
(*de fonction "func" sur les quatre valeurs d'entrée a, b, c et d. Elle fait*)
(*ceci en retournant le bit d*8 + c*4 + b*3 + a*2 à partir de func. Cette   *)
(*version se base sur la possibilité d'utiliser l'opérateur de décalage à   *)
(*droite et les opérateurs logiques sur les bits offerte par Turbo Pascal.  *)
```

```
function Generic(func, a, b, c, d:integer):integer;
begin
        Generic := (func shr(a + b*2 + c*4 + d*8)) and 1;
end;

begin   (*main*)

        repeat

                fresult := ReadFunc;
                if(fresult <> 0) then begin

                        write('Entrez des valeurs pour D, C, B et A (0/1) :');
                        readln(d, c, b, a);
                        writeln('Le résultat est : ',Generic(func, a, b, c, d));

                end;
        until fresult = 0;
end.
```

2.9 Exercices de Laboratoire

Ce laboratoire utilise divers programmes Windows pour manipuler des tables de vérité, travailler avec des expressions logiques et optimiser et simuler des équations. Ces programmes vous aideront à comprendre les relations entre les équations logiques et les tables de vérité ; et aussi à comprendre pleinement les systèmes logiques.

Le programme WLOGIC.EXE simule le fonctionnement des circuits logiques. Il maintient des équations décrivant un circuit électronique et il simule des interrupteurs et des indicateurs lumineux (LEDs) pour les sorties. Pour ceux qui aiment des laboratoires réalistes, il y a un programme optionnel que vous pouvez faire tourner sous DOS, LOGICEV.EXE qui contrôle un ensemble réel d'indicateurs et d'interrupteurs qu'on construit et on attache sur un port parallèle du PC. Les directives de construction matérielle se trouvent dans les appendices. L'utilisation des deux programmes permet d'observer facilement le comportement d'un ensemble de fonctions logiques.

Si vous ne l'avez déjà fait, veuillez installer les logiciels supportant ce livre. Voir les exercices de laboratoire du chapitre 1 pour plus de détails.

2.9.1 Exercices sur les tables de vérité et les équations logiques

Dans cet exercice, vous allez créer diverses tables de vérité de deux, trois et quatre variables. Le programme TRUTHTBL.EXE (qui se trouve dans le sous-répertoire CH2), convertit automatiquement en forme canonique (somme de *minterms*) les tables de vérité que vous entrez.

Le fichier TRUTHTBL.EXE est un programme Windows et requiert certaines versions de Windows pour fonctionner correctement. En particulier, il ne fonctionnera pas sous DOS. Il devrait tout de même fonctionner convenablement sur des versions de Windows à partir de 3.1.

Le programme TRUTHTBL.EXE fournit trois boutons vous permettant de choisir des tables de trois ou quatre variables. Cliquer sur un de ces boutons vous permet d'arranger la table de façon appropriée. Par défaut, TRUTHTBL suppose que vous voulez travailler avec quatre variables. Essayez de presser les boutons permettant de choisir le nombre de variables et observez les résultats. Décrivez ce qui arrive dans votre rapport.

Pour changer les entrées de la table, vous aurez juste besoin de cliquer sur la case associée à la valeur de la table de vérité que vous voulez changer. Cliquer sur une de ces cases inverse la valeur qui y est inscrite. Par exemple, essayez de cliquer sur la case DCBA plusieurs fois et observez les résultats.

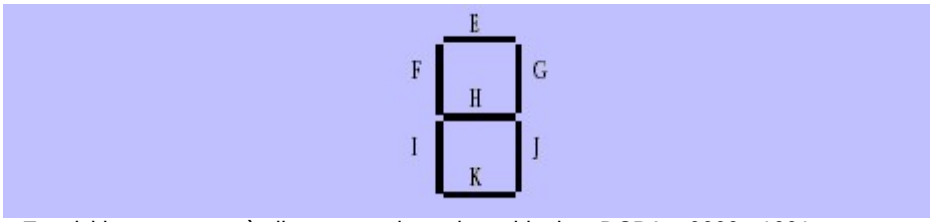
Notez que, dès que vous cliquez sur différentes entrées, le programme recalcule automatiquement la somme des *minterms* et l'affiche au bas de la fenêtre. Quelle équation sera affichée si vous mettez toutes les entrées à zéro[9] ?

Réglez le programme pour travailler avec quatre variables. Mettez à un la position DCBA. Maintenant, pressez le bouton pour mettre en mode deux variables. Mettez de nouveau le programme en mode quatre variables et remplissez toute la table avec des 1. Maintenant, revenez encore en mode deux variables. Finalement, pressez le bouton "quatre variables" et examinez les résultats. Qu'est-ce que le programme fait quand vous changez la taille des tables de vérité ? Expérimentez plusieurs modes, puis notez vos résultats.

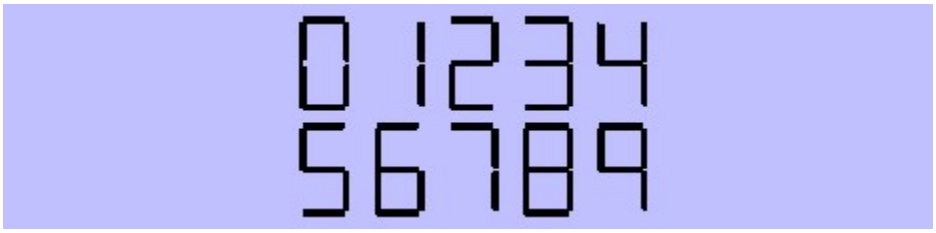
Passez en mode deux variables. Donnez les entrées pour les fonctions AND, OR, XOR et NAND. Vérifiez la justesse des équations logiques produites et annotez vos résultats dans votre rapport de laboratoire. Note : si vous avez une imprimante compatible avec Windows, vous pouvez imprimer toutes les tables que vous créez en pressant le bouton Print dans la fenêtre. Ceci rend très facile d'inclure dans votre rapport les tables et les équations correspondantes. **Pour aller plus loin :** faites les tables de vérités pour toutes les 16 fonctions de deux variables. Dans votre rapport, notez tous les résultats.

Concevez à la main plusieurs tables de vérités de deux, trois et quatre variables. Déterminez toujours à la main les équations logiques sous forme de somme de minterms. Puis passez ces tables au programme et vérifiez l'exactitude de votre travail. Incluez dans votre rapport vos tables et vos équations avec les résultats du programme.

Considérez la disposition suivante de l'affichage d'un circuit sept segments :



Et voici les segments à allumer pour les valeurs binaires DCBA = 0000 - 1001 :

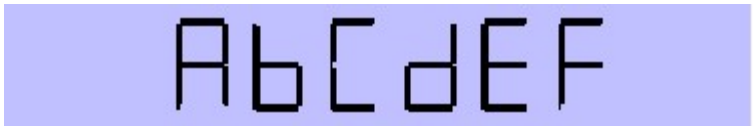


E = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A
F = D'C'B'A' + D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A
G = D'C'B'A' + D'C'B'A + D'C'BA' +D'C'BA + D'CB'A' + D'CBA + DC'B'A' + DC'B'A
H = D'C'B'A' + D'C'BA + D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A
I = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'
J = D'C'B'A' + D'C'B'A + D'C'BA + D'CB'A' + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A
K = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + DC'B'A'

Convertissez chacune de ces équations en table de vérité en mettant toutes les entrées de la table à zéro et en cliquant ensuite sur chaque case qui correspond à chaque minterm de l'équation. En observant l'équation que TRUTHTBL produit, vérifiez chacune de vos conversions. Placez les tables et la description de vos résultats dans votre rapport de laboratoire.

Pour aller plus loin : modifiez les équations ci-dessus pour produire les caractères hexadécimaux suivants. Déterminez les nouvelles tables et utilisez le programme pour vérifier que vos tables et vos équations sont correctes.

[9]Note : à l'entrée du programme, aucune équation ne sera affichée. Par conséquent, vous devrez changer au moins une case et ensuite la remettre à zéro pour voir cette équation.



2.9.2 Exercices sur les fonctions logiques en forme canonique

Dans ce laboratoire, vous allez entrer diverses équations logiques, calculer leur forme canonique et générer leur table de vérité. Dans un certain sens, cet exercice est l'opposé du précédent où vous génériez une forme canonique à partir d'une table de vérité.

Cet exercice utilise le programme CANON.EXE qui se trouve dans le sous-répertoire CH2. Exécutez ce programme à partir de Windows en double-cliquant sur son icône. Il affiche une boîte de texte, une table de vérité et divers boutons. Contrairement à TRUTHTBL.EXE de l'exercice précédent vous ne pourrez pas modifier la table de vérité, car c'est une table d'affichage uniquement. Pour calculer les résultats qui seront affichés dans cette table, il faudra entrer des équations logiques dans le champ de texte et presser le bouton "Compute". Ce programme produit aussi la somme des minterms pour l'équation logique entrée (d'où le nom du programme).

Des équations logiques valides prennent les formes suivantes :

- Un *terme* est soit une variable (A, B, C ou D), soit une expression logique entourée par des parenthèses.
- Un *facteur* est soit un terme soit un facteur suivi par le symbole ' Ce caractère effectue la négation logique du facteur qui le précède immédiatement.
- Un *produit* est soit un facteur, soit un facteur concaténé avec un produit. La concaténation illustre l'opération AND.
- Une expression est un produit, ou un produit suivi d'une somme (laquelle illustre l'opération OR) avec une autre expression.

Notez que OR a la précédence la plus basse, AND a une précédence intermédiaire et NOT la plus élevée. Vous pouvez utiliser des parenthèses pour imposer des priorités de votre choix. L'opérateur NOT, puisque sa priorité est aussi élevée, s'applique seulement à une variable isolée ou bien à une expression entre parenthèses. Les exemples qui suivent illustrent des expressions valides :

```
AB'C + D(B'+C')
AB(C+D)' + ABCD + A(B+C)
A'B'C'D' + ABCD + A(B+C)
(A+B)' + A'B'
```

Pour cet ensemble d'exercices, vous devrez créer diverses expressions logiques et les insérer dans CANON.EXE. Incluez les tables de vérité et les formes canoniques dans votre rapport de laboratoire. Vérifiez aussi la validité des théorèmes illustrés dans ce chapitre (regardez le paragraphe 2.1). Entrez les deux versions des dualités et constatez qu'elles produisent la même table de vérité (par exemple (AB)' = A' + B'). Pour des expériences additionnelles, créez différentes équations logiques complexes et générez leurs tables de vérité et leurs formes canoniques à la main. Puis entrez les équations dans CANON.EXE et vérifiez la justesse de vos calculs.

2.9.3 Exercices d'optimisation

Dans cet ensemble d'exercices de laboratoire, le programme OPTIMIZP.EXE (se trouvant toujours dans le sous-répertoire CH2), vous guidera à travers les étapes de l'optimisation des fonctions logiques. OPTIMIZP.EXE utilise les tables de Karnaugh pour produire une équation avec le nombre minimal de terms.

Exécutez le programme en cliquant sur son icône ou à partir de la commande *Run* (*Exécuter*, si Windows est en français) du menu de démarrage. Ce programme permet d'entrer une fonction logique arbitraire, tout comme on pouvait faire avec CANON.EXE de l'exercice précédent.

Après avoir entré l'équation et pressé le bouton Optimize, le programme construit la table de vérité, l'équation canonique et la forme optimale. Une fois l'équation optimisée, OPTIMIZP.EXE active le bouton "Step", qui vous permet de parcouris tout le processus de l'optimisation étape par étape.

Pour cet exercice, entrez les sept équation pour l'affichage à sept segments. Générez et annotez la forme optimisée de ces équations dans votre rapport de laboratoire. Afin de vous assurer de comprendre de quelle façon le programme effectue ses opérations, utilisez le bouton Step pour analyser l'optimisation une étape à la fois.

Pour aller plus loin : OPTIMIZP.EXE génère une seule version d'expression optimale pour toute fonction donnée. Mais d'autres versions optimales peuvent exister. A l'aide de la technique de la table de Karnaugh essayez de déterminer s'il existe d'autres expressions équivalentes. Insérez les expressions optimales produites par OPTIMIZP.EXE, ainsi que celles produites par vous, dans CANON.EXE et viréfiez si les formes canoniques produites sont identiques (ce qui voudrait dire que les vôtres aussi le sont).

2.9.4 Exercices d'évaluation logique

Dans cet ensemble d'exercices de laboratoire, vous utiliserez le programme LOGIC.EXE pour entrer, éditer, initialiser et évaluer des expressions logiques. Ce programme vous permet d'entrer jusqu'à 22 équations différentes impliquant 26 variables plus une valeur d'horloge. LOGIC.EXE fournit quatre variables d'entrée et onze variables de sortie (quatre indicateurs simulés et un affichage sept segments simulé). Note : ce programme demande l'installation de deux fichiers dans votre répertoire WINDOWS\SYSTEM. Voir le fichier README.TXT du sous-répertoire CH2 pour plus de détails.

Exécutez LOGIC.EXE en double-cliquant sur son icône ou à partir de la commande Run (Exécuter, si Windows est en français). Ce programme est divisé en trois parties principales : un éditeur d'équations, un écran d'initialisation et un module d'exécution. LOGIC.EXE se sert d'un ensemble d'onglets pour passer d'un module à l'autre. Pour en sélectionner un, il suffit d'y cliquer dessus. Généralement, on crée un ensemble d'équations logiques dans la page de création pour les exécuter ensuite dans le module d'exécution. De façon optionnelle, on peut initialiser toute variable nécessaire (D-Z) dans la page d'initialisation. A tout moment, vous pouvez facilement passer d'un module à l'autre en choisissant l'onglet approprié. Par exemple, vous pourriez créer un ensemble d'équations, les exécuter, plus revenir en arrière - onglet Create - pour en modifier les énoncés (et éventuellement, pour corriger certaines erreurs).

L'onglet *Create* vous permet d'ajouter, éditer et effacer des équations logiques. Celles-ci peuvent être représentées par les variables A-Z, plus le symbole "#", qui indique l'horloge. Les équations sont soumises à une syntaxe très semblable aux expressions logiques utilisées dans les exercices précédents de ce chapitre. En effet, il y a seulement deux différences principales entre les fonctions permises dans LOGIC.EXE et celles permises par les autres programmes. D'abord, LOGIC.EXE permet l'usage des variables A-Z plus # (alors que les autres programmes allouent seulement des fonctions de quatre variables A-D). La seconde différence est que les fonctions de LOGIC.EXE doivent prendre la forme :

variable = *expression*

où *variable* est un caractère alphabétique de la plage E-Z[10] et *expression* est une expression logique utilisant les variables de la plage A-Z et #. Une expression peut utiliser un maximum de quatre variables différentes, plus la valeur d'horloge. Durant l'évaluation, le programme LOGIC.EXE calcule la fonction et stocke le résultat dans la variable de destination spécifiée.

Si vous incluez plus de quatre variables, le programme signalera l'erreur, car il peut seulement évaluer des expressions contenant un maximum de quatre caractères alphabétiques (sans compter la variable à la gauche du signe d'égalité). Notez que la variable de destination peut apparaître à l'intérieur de l'équation ; ce qui suit est parfaitement admis :

```
F = FA+FB
```

[10]A-D sont des variables de lecture seule qu'on lit depuis un ensemble d'interrupteurs. Par conséquent, on n'y peut rien stocker.

Cette expression utilise la valeur courante de F parmi les valeurs courantes de A et de B et calcule la nouvelle expression pour F.

À différence des langages de programmation comme C++, LOGIC.EXE n'évalue pas cette expression seulement une fois avant de la stocker dans F. Il l'évaluera plusieurs fois *jusqu'à ce que la variable F se stabilise*. Autrement dit, l'expression sera évaluée plusieurs fois jusqu'à ce que le même résultat soit produit au moins deux fois de suite. Certaines expressions produiront des *boucles infinies* puisqu'elles ne produiront jamais deux fois le même résultat. Par exemple, la fonction suivante est instable :

```
F = F'
```

Notez que l'instabilité peut persister au-delà des limites d'une fonction. Considérez les deux équations suivantes :

```
F = G
G = F'
```

LOGIC.EXE essaiera d'exécuter cet ensemble jusqu'à ce que les variables du résultat cessent de changer. Mais le résultat sera une boucle infinie.

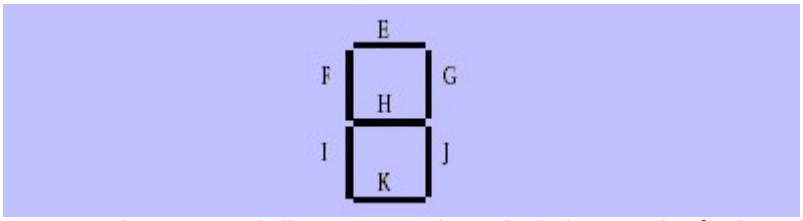
Parfois, un système logique produira de telles boucles seulement pour certaines valeurs données. par exemple, l'équation suivante est susceptible de produire ce résultat :

```
F = GF' + G'F  (F = G xor F)
```

Si la valeur de G est 1, le système sera instable. Si elle est 0, alors l'équation aura un résultat défini. Des équations instables comme celle-ci sont souvent difficiles à découvrir.

LOGIC.EXE détecte et alerte les instabilités quand elles se présentent. Malheureusement, il ne pourra pas indiquer le problème. Il se limitera seulement à vous signaler qu'il existe et s'attendra à une intervention de votre part.

Les variables A-D, E-K et W-Z sont spéciales. A-D sont des valeurs d'entrée de lecture seule. E-K correspondent à l'affichage sept segments de la page d'exécution.



La plage W-Z correspond aux quatre indicateurs LED de sortie de la page d'exécution. Si les variables des plages E-K ou W-Z contiennent 1, le segment correspondant devient rouge (allumé). Sinon, il reste éteint.

L'onglet de création, contient d'importants boutons : *Add, Edit* et *Delete*. Quand vous pressez le bouton Add, LOGIC.EXE ouvre une boîte de dialogue vous permettant d'entrer une équation. Tapez votre équation (ou éditez celle par défaut), puis pressez le bouton *Okay*. S'il y a un problème, le programme signalera l'erreur et vous demandera de la corriger ; sinon il tentera d'insérer quand même cette équation dans le système que vous serez en train d'établir. Si une fonction existe déjà et a la même variable de destination de l'équation précédente, le programme vous demandera si vous avez réellement l'intention de remplacer la fonction, avant de procéder. Chaque fois que LOGIC.EXE ajoute une équation dans sa liste, il en affiche aussi la table de vérité. Vous pouvez ajouter jusqu'à 22 équations (puisqu'il y a 22 variables de destination possibles : de E à Z). LOGIC.EXE affiche ces fonctions dans la zone de liste à la droite de la fenêtre.

Une fois que vous avez entré deux fonctions ou plus, vous pouvez voir la table de vérité de chacune en cliquant simplement sur l'équation désirée dans la liste.

Si vous commettez une erreur en entrant une fonction vous pouvez l'effacer en la sélectionnant avec la souris, puis en cliquant sur le bouton Delete, ou bien vous pouvez tout simplement corriger l'erreur en la sélectionnant, puis en cliquant le bouton Edit. Vous pouvez également éditer une équation déjà existante en double-cliquant sur elle dans la zone de liste.

L'onglet d'initialisation affiche des cases pour chacune des 26 variables possibles en vous permettant aussi de voir la valeur courante pour chacune de ces variables ou de la modifier, si elle appartient à la plage E-Z (n'oubliez pas que les variables A-D sont de lecture seule). Comme règle générale, vous n'aurez pas besoin d'initialiser toutes ces variables, donc vous pouvez sauter cet écran si vous n'en aurez pas un besoin effectif.

La page d'exécution contient cinq boutons importants : *A-D* et *Pulse*. Les interrupteurs A-D vous permettent de fixer les valeurs d'entrée pour les variables A-D. L'interrupteur *Pulse* fait passer la valeur de l'horloge de zéro à un et vice-versa, permettant d'évaluer le comportement des fonctions logiques pour chaque état.

En plus des boutons d'entrée, l'écran d'exécution présente diverses sorties. D'abord, les quatre indicateurs (W, X, Y et Z) et l'affichage du circuit sept segments (variables de sortie E-K, comme indiqué ci-dessus). Et, à part les indicateurs, vous trouverez aussi un voyant d'instabilité qui devient rouge si LOGIC.EXE en détecte une dans le système. Il y a aussi un petit panneau affichant la valeur courante de toutes les variables en usage.

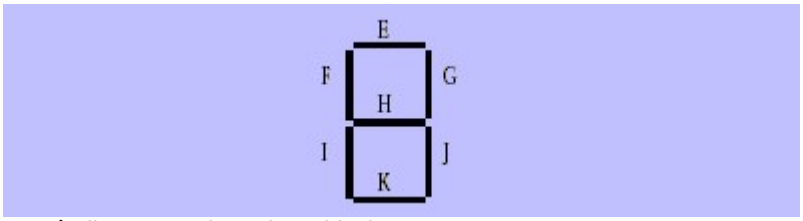
Pour exécuter le système d'équations changez simplement une des valeurs d'entrée (A-D) ou pressez le bouton Pulse. LOGIC.EXE refera automatiquement l'évaluation chaque fois que ces valeurs changeront.

Pour vous familiariser avec LOGIC.EXE, entrez dans l'éditeur les équations suivantes :

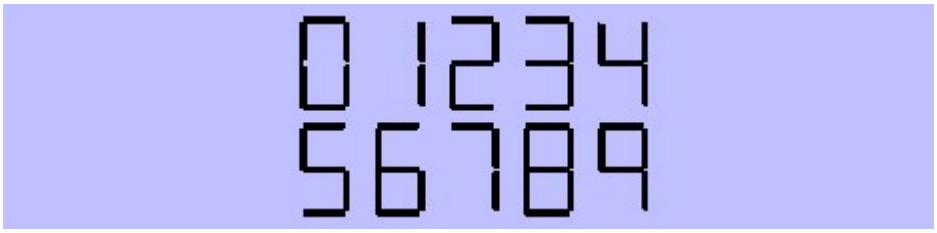
```
W = AB                          A and B
X = A + B                       A or B
Y = A'B + AB'                   not A
Z = A'                          idem
```

Après avoir entré les équations, allez dans l'onglet d'exécution et entrez les quatre valeurs 00, 01, 10 et 11 pour BA. Notez les valeurs de W, X, Y et Z dans votre rapport de laboratoire.

Le programme LOGIC.EXE simule un circuit sept segment. Les variables E-K allument les segments individuels comme suit :



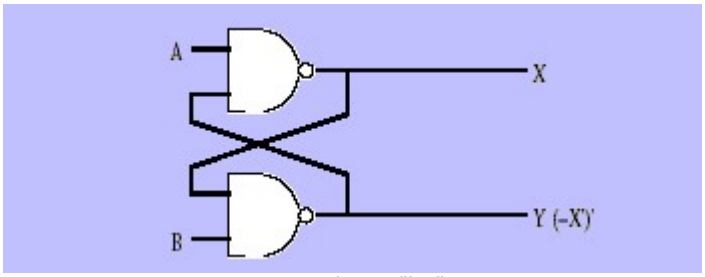
Voici les segments à allumer pour les valeurs binaires DCBA = 0000 - 1001 :



Entrez les sept équations pour ces segments dans LOGIC.EXE et essayez chacun des modes (0000 à 1111). **Truc** : utilisez les équations optimisées que vous avez développées plus tôt.

Expériences additionnelles optionnelles : entrez les équations pour les 16 valeurs hexadécimales et parcourez ces valeurs. Incluez le résultat dans le rapport.

Voici un simple circuit séquentiel. Pour cet exercice, il vous suffira d'entrer les équations logiques pour un simple set/reset flip-flop. Le diagramme du circuit est :



Un set/reset flip-flop

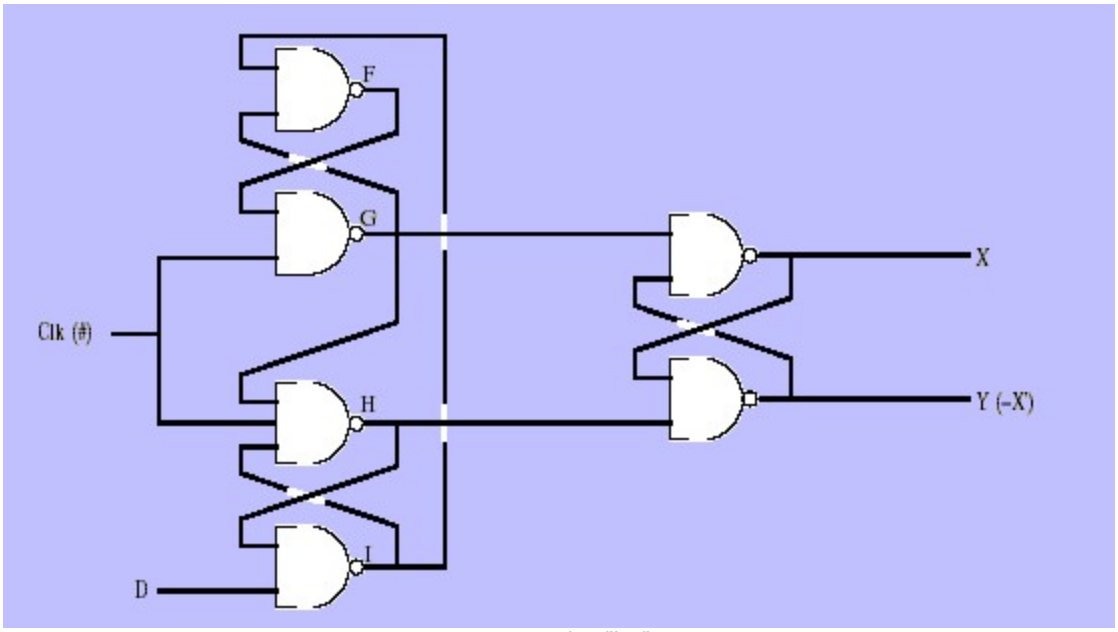
Puisqu'il y a deux sorties, ce circuit a deux équations logiques correspondantes. Elles sont :

```
X = (AY)'
Y = (BX)'
```

Ces deux équations constituent un circuit séquentiel, étant donné que les deux utilisent des variables qui apparaissent dans la sortie. En particulier, en calculant de nouvelles valeurs pour X et Y, Y utilisera la valeur précédente de X et X la valeur précédente de Y.

Entrez ces deux équations dans LOGIC.EXE. Mettez les entrées A et B à 1 (ce qui correspond à l'état normal), puis exécutez la simulation logique. Essayez de faire passer A à 0 et remarquez ce qu'il arrive. Pressez le bouton Pulse plusieurs fois avec A encore à zéro et observez. Puis remettez A à 1 et répétez le procédé. Maintenant essayez encore l'expérience, cette fois en mettant B à 0. Puis, remettez A à 1. Essayez de mettre les deux à zéro puis remettez B à 1. **Pour votre rapport de laboratoire :** fournissez des diagrammes pour les interrupteurs d'entrée et pour les indicateurs de sortie chaque fois que vous ferez passez un bouton à 1.

Un vrai flip-flop de type D fait passer les données uniquement par l'entrée D durant une transition d'horloge de bas à haut (0 à 1). Dans cet exercice vous simulerez un D flip-flop. Le diagramme pour ce circuit est :



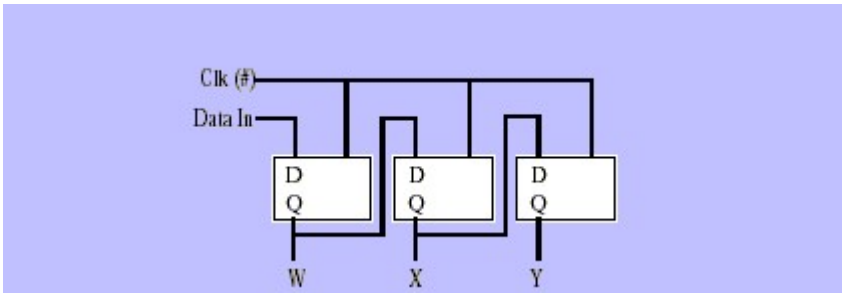
Un vrai D flip-flop

```
F = (IG)'
G = (#F)'
H = (G#I)'
I = (DH)'
X = (GY)'
Y = (HX)'
```

Incluez dans le programme l'ensemble de ces équations et testez votre flip-flop en entrant différentes valeurs dans l'interrupteur d'entrée D et en pressant le bouton d'horloge. Expliquez vos résultats dans votre rapport de laboratoire.

Dans cet exercice vous allez construire un registre de décalage de trois bits en utilisant les équations logiques pour un vrai D flip-flop. Pour construire un tel registre, vous devrez connecter les sorties de chaque flip-flop aux entrées du flip-flop suivant. La ligne d'entrée D fait passer l'information pour le premier flip-flop, la dernière sortie correspond à la retenue sortie du circuit. Utilisez un simple rectangle pour représenter un flip-flop et ignorez les sorties Q (puisqu'on ne les utilise pas) ; le schéma de notre registre ressemble à ceci :

Un registre de décalage de trois bits construit à partir de flip-flops



Dans l'exercice précédent, vous avez utilisé six expressions booléennes pour définir un flip-flop de type D. Par conséquent, on aura besoin d'un total de 18 expressions booléennes pour implémenter ce circuit. Ces expressions sont :

```
Flip-Flop #1 :

       W = (GR)'
       F = (IG)'
```

```
       G = (F#)'
       H = (G#I)'
       I = (DH)'
       R = (HW)'


Flip-Flop #2 :

       X = (KS)'
       J = (MK)'
       K = (J#)'
       L = (K#M)'
       S = (LX)'


Flip-Flop #3 :

       Y = (OT)'
       N = (QO)'
       O = (N#)'
       P = (O#Q)'
       Q = (XP)'
       T = (PY)'
```

Entrez ces équations dans LOGIC.EXE. Initialisez W, X et Y à zéro. Mettez D à 1 et presse le bouton Pulse une fois pour insérer un 1 dans W. Maintenant mettez D à zéro et pressez Pulse plusieurs fois pour décaler ce bit à travers chacun des bits de sortie. **Pour votre rapport de laboratoire :** dans ce registre de décalage, essayez de décaler les bits de diverses façons. Décrivez l'opération étape par étape dans votre rapport.

Pour aller plu loin : décrivez comment créer un *registre de décalage de rotation (ou de recirculation de bits)*. Le bit sortant du bit quatre revient dans le bit zéro. Quelles seraient les équations logiques pour un tel registre de décalage ? Comment pourriez-vous l'initialiser à l'aide de LOGIC.EXE (puisque vous ne pouvez pas utiliser l'entrée D) ?

Expériences additionnelles post-lab : concevez un additionneur complets de deux bits calculant la somme de BA et DC et stockant le résultat binaire dans les indicateurs WXY. Incluez les équations et certains échantillons du résultat dans votre rapport.

2.10 Projets de programmation

Vous pouvez rédiger ces programmes dans tout langage de haut niveau que votre instructeur vous permet d'utiliser (normalement, C, C++ ou certaines versions de Borland Pascal ou Delphi). Si vous le désirez, vous pouvez vous servir des fonctions génériques vues dans ce chapitre.

1) Ecrivez un programme qui lit quatre valeurs de l'utilisateur, I, J, K et L et place ces valeurs dans une table de vérité, avec B'A' = I, B'A = J, BA' = K et BA = L. Assurez-vous que ces valeurs d'entrée soient uniquement 0 ou 1. Puis, invitez l'utilisateur à entrer d'autres valeurs et placez-les dans la table. Affichez le résultat de chaque calcul. Note : pour ce programme, n'utilisez pas la fonction logique générique.

2) Ecrivez un programme qui, à partir d'un numéro de fonction logique de quatre bits, affiche la table de vérité pour cette fonction de deux variables.

3) Ecrivez un programme qui, à partir d'un numéro de fonction logique de huit bits, affiche la table de vérité pour cette fonction de trois variables.

4) Ecrivez un programme qui, à partir d'un numéro de fonction logique de seize bits, affiche la table de vérité pour cette fonction de quatre variables.

5) Ecrivez un programme qui, à partir d'un numéro de fonction logique de seize bits, affiche l'équation canonique de cette fonction (truc : construisez la table de vérité).

2.11 Résumé

L'algèbre booléenne constitue la fondation des deux aspects de l'informatique : le matériel et le logiciel.

L'algèbre booléenne est un système mathématique avec son propre ensemble de règles (postulats), théorèmes et valeurs. Sous beaucoup d'aspects, elle est semblable à l'algèbre traditionnelle, sous d'autres, elle est même plus simple. Ce chapitre a commencé par traiter diverses caractéristiques des systèmes booléens, opérateurs clôture, commutativité, associativité, distributivité, identité et inverse. Puis, il a traité certains postulats et théorèmes importants, dont le *principe de dualité*, permettant de démontrer aisément beaucoup de théorèmes. Pour les détails, voir :

· "Algèbre booléenne", au paragraphe 2.1

La *table de vérité* est un moyen commode de représenter visuellement une fonction ou une expression booléenne. Chaque fonction (ou chaque expression) a une table de vérité correspondante fournissant tous les résultats possibles pour toute combinaison des valeurs d'entrée. Ce chapitre a présenté plusieurs moyens de construire des ces tables.

Bien qu'avec un nombre *n* de valeurs d'entrée il y ait un nombre infini de fonctions booléennes, il est tout de même évident que, pour un nombre d'entrées donné, il y aura un nombre limité de fonctions uniques. Autrement dit, il y a $2\text{^}2^n$ fonctions uniques pour *n* entrées. Par exemple, avec deux variables, il y a 16 fonctions uniques possibles ($2\text{^}2^2 = 16$).

Puisque pour deux entrées le nombre de fonctions est aussi réduit, on peut donner un nom spécifique à chacune d'elles (par exemple, AND, OR, NAND, etc.). Pour les fonctions de trois ou plus variables, le nombre de fonctions uniques est trop grand pour que chacune ait un nom. Par conséquent, pour distinguer une fonction de l'autre, on assignera un numéro à chacune et ce numéro sera basé sur les bits qui apparaissent dans la table de vérité de cette fonction. Pour en savoir plus, voir :

· "Fonctions booléennes et tables de vérité", au paragraphe 2.2

On peut manipuler algébriquement les fonctions et les expressions booléennes. Ceci permet de pouvoir démontrer de nouveaux théorèmes, de simplifier des expressions, de convertir des fonctions en forme canonique ou de montrer l'équivalence de deux fonctions. Pour voir certains exemples de manipulation algébrique, voir :

· "Manipulation algébrique d'expressions booléennes", paragraphe 2.3

Puisque pour un nombre fixe d'entrées, il existe une variété infinie de fonctions booléennes possibles et un nombre fini de fonctions uniques, il est évident que parmi ce nombre infini de fonctions, la plupart engendre le même résultat. Pour éviter les confusions, les ingénieurs font souvent référence à une fonction en utilisant la *forme canonique*. Si deux équations canoniques sont différentes, elles doivent représenter des fonctions booléennes différentes. Ce livre décrit deux formes canoniques : la *somme des minterms* et le *produit des maxterms*. Pour des détails sur les formes canoniques, les conversions en formes canoniques ou les conversions entre formes canoniques, voir :

· "Formes canoniques", au paragraphe 2.4

Quoiqu'une forme canonique est en mesure de fournir une unique représentation d'une fonction booléenne donnée, les expressions qui résultent d'une forme canonique sont rarement *optimales*. Ce qui veut dire que les formes canoniques utilisent souvent plus de littéraux et d'opérateurs que d'autres expressions équivalentes. En concevant un circuit électronique ou une section de programme concernant des expressions booléennes, les ingénieurs préfèrent utiliser des circuits ou des programmes optimisés, puisque les versions optimisées sont moins onéreuses et, probablement, plus rapides. Par conséquent, connaître le moyen de créer une forme optimisée à partir d'une expression booléenne est très important. Ce chapitre traite le sujet dans :

· "Simplification de fonctions booléennes", au paragraphe 2.5

L'algèbre booléenne n'est pas un système mathématique conçu par quelque mathématicien fou et sans importance dans le monde réel. Elle est est la base de la logique numérique, autrement dit, la base de la conception électronique. De plus, il y a une correspondance de un-à-un entre matériel numérique et logiciel. Tout ce que l'on peut concevoir du côté matériel, on peut le concevoir aussi du côté logiciel et vice-versa. Ce texte explique comment implémenter l'addition, les décodeurs, la mémoire, les registres de décalage et les compteurs

en utilisant des fonctions booléennes. D'autre part, il décrit aussi comment améliorer les performances logicielles (par exemple un programme Pascal) en appliquant les règles et les théorèmes de l'algèbre booléenne. Pour plus de détails, voir :

· "En tout cas, qu'est-ce que ceci a à voir avec les ordinateurs ?", paragraphe 2.6
· "Correspondance entre circuits électroniques et fonctions booléennes", paragraphe 2.6.1
· "Circuits combinatoires", paragraphe 2.6.2
· "Logique séquentielle et synchrone", paragraphe 2.6.3
· "D'Accord, mais quel est le rapport avec la programmation, encore ?", paragraphe 2.7

2.12 Questions

1. Quel est l'élément identité par rapport à :

   a) AND b) OR c) XOR d) NOT e) NAND f) NOR

2. Construisez les tables de vérité pour les fonctions suivantes de deux variables :

   a) AND b) OR c) XOR d) NAND e) NOR

   f) Equivalence g) A < B h) A > B i) A implique B

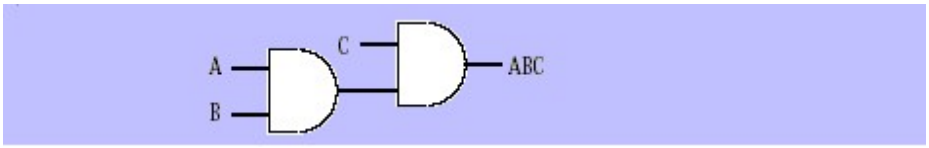
3. Construisez les tables de vérité pour les fonctions suivantes de trois variables :

   a) ABC (and) b) A + B + C (or) c) (ABC)' (nand) d) (A + B + C)' (nor)

   e) Equivalence (ABC) + (A'B'C') f) XOR(ABC + A'B'C')'

4. Dessinez des schémas (des diagrammes des circuits électroniques) pour montrer comment implémenter chacune des fonctions de la question 3, en utilisant seulement deux portes d'entrée et des inverseurs. Par exemple :

   A) ABC =



5. Fournissez des implémentations pour des portes AND, OR et inverseur, en utilisant une ou plus portes NOR.

6. Qu'est-ce que c'est le principe de dualité ? En quoi il peut nous aider ?

7. Dressez une simple table de vérité qui fournit les sorties pour les trois fonctions booléennes suivantes de trois variables :

   Fx = A + BC
   Fy = AB + C'B
   Fz = A'B'C' + ABC + C'B'A

8. Trouvez les numéros des trois fonctions de la question précédente.

9. Combien de fonctions (uniques) possibles peut-on avoir si la fonction a :
   a) une entrée b) deux entrées c) trois entrées d) quatre entrées e) cinq entrées

10. Simplifiez les fonctions booléennes suivantes en utilisant des transformations algébriques. Montrez les étapes de votre travail.

a) F = AB + A'B' b) F = ABC + BC' + AC + ABC'

c) F = A'B'C'D' + A'B'C'D + A'B'CD + A'B'CD'

d) F = A'BC + ABC' + A'BC' + AB'C' + ABC + AB'C

11. Simplifiez les fonctions booléennes de la question 10 en utilisant la méthode de Karnaugh.

12. Fournissez les équations logiques en forme canonique pour les fonctions booléennes S_0...S_6 de l'affichage du circuit sept segments (voir "Circuits combinatoires" au paragraphe 2.6.2).

13. Donnez les tables de vérité pour chacune des fonctions de la question 12.

14. Minimisez chacune des fonctions de la question 12 à l'aide de la méthode de Karnaugh.

15. L'équation logique d'un demi-additionneur (en forme canonique) est

Somme = AB' + A'B Retenue = AB

a) Fournissez le diagramme du circuit électronique pour ce demi-additionneur en utilisant des portes AND, OR et inverseur.
b) Fournissez le diagramme du circuit en utilisant seulement des portes NAND.

16. Les équations canoniques d'un additionneur complet sont :

Somme = A'B'C + A'BC' + AB'C' + ABC Retenue = ABC + ABC' + AB'C + A'BC

a) Fournissez le schéma du circuit électronique en utilisant des portes AND, OR et inverseur.
b) Optimisez ces équations en utilisant la méthode de Karnaugh.
c) Fournissez le circuit électronique de la version optimisée (en utilisant des portes AND, OR et inverseur).

17. Présumez avoir un flip-flop de type D (utilisez la définition présentée ici) dont les sorties sont couramment Q = 1 et Q' = 0. Décrivez en grand détail et exactement qu'est-ce qui arrive si la ligne d'horloge passe :

a) de 0 à 1 avec D = 0.
b) de 1 à 0 avec D = 0.

18. Récrivez les instructions Pascal suivantes pour les rendre plus efficaces :

a) if (x or (not x and y)) then write('1');
b) while (not x and not y) do somefunc(x, y);
c) if not ((x <> y) and (a = b)) then Something;

19. Mettez en forme canonique (somme des *minterms*) chacune des fonctions suivantes :

a) F(A,B,C) = A'BC + AB + BC b) F(A,B,C,D) = A + B + CD' + D

c) F(A,B,C) = A'B + B'A d) F(A,B,C,D) = A + BD'

e) F(A,B,C,D) = A'B'C'D + AB'C'D' + CD + A'BCD'

20. Convertissez les résultats de la question 19 en un produit de *maxterms*.