Polytech'Montpellier
Réseau Polytech'

ECOLE POLYTECHNIQUE UNIVERSITAIRE DE MONTPELLIER

Représentation et Synthèse des Systèmes Logiques

Serge PRAVOSSOUDOVITCH

ECOLE POLYTECHNIQUE UNIVERSITAIRE DE MONTPELLIER
UNIVERSITE MONTPELLIER II SCIENCES ET TECHNIQUES DU LANGUEDOC
Place Eugène Bataillon 34095 MONTPELLIER CEDEX 5
Tél. : 04 67 14 31 60 – Fax : 04 67 14 45 14
E-mail : scola@polytech.univ-montp2.fr

um2 UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

Table des Matières

Chapitre 1

Algèbre de commutation

Les éléments permettant de réaliser un circuit logique peuvent mettre en jeu des phénomènes physiques très divers : mécaniques, électriques, magnétiques, pneumatiques, électroniques etc ... Ils ont en commun la propriété de pouvoir occuper deux états distincts par action d'une commande externe. La valeur numérique exacte de ces états ou signaux n'a aucune importance. Il est en effet possible de concevoir des circuits réalisant la même fonction avec des valeurs de signaux complètement différentes. Des symboles arbitraires sont donc utilisés pour représenter les deux valeurs possibles des signaux. Une algèbre formelle utilisant ces symboles a été développée par George Boole (1815-1864). Le but de ce premier chapitre est de présenter les lois et opérateurs régissant cette algèbre (algèbre de Boole) en se focalisant sur le cas particulier de l'algèbre de commutation.

1.1. Algèbre de Boole

Autodidacte, Boole est à l'origine de la notion d'ensemble et du "calcul" sur ces ensembles (classes d'objets : *classes of things*) liant la logique mathématique au calcul algébrique (*The Mathematical Analysis of Logic*, 1847). Boole résolut ainsi un des problèmes fondamentaux de formalisation du langage et du raisonnement, que se posaient les mathématiciens depuis Leibniz. La pertinence de ses travaux lui permirent l'obtention (1849) d'une chaire de mathématiques au Queen's Collège de Cork (Irlande du sud).

On peut le considérer comme le créateur de la logique moderne. *L'algèbre de Boole* qu'il exprime en 1854 dans son traité *An investigation of the laws of thought* (*sur les lois de la pensée*), est aussi utilisée de nos jours dans la mise au point des machines automatiques (algèbre des circuits).

Remarquons que cette algèbre logique peut être considérée comme un cas particulier de l'algèbre des parties d'un ensemble (muni de la réunion et de l'intersection) de Cantor.

Un ensemble E possède une structure d'algèbre de Boole s'il est muni de deux lois de composition interne notées + et * telles que:

- Les lois + et * sont commutatives et associatives.
- Les lois + et * sont distributives l'une par rapport à l'autre.
- Les lois + et * admettent un élément neutre (0 et 1 respectivement).
- Tout élément de E est idempotent pour chaque loi : $x + x = x$ et $x * x = x$
- Tout élément x de E possède un unique élément, dit *complémenté* de x, généralement noté généralement x' (ou $\bar{x}$), vérifiant la loi du *tiers exclu* : $x + x' = 1$ et le principe de *contradiction* $x * x' = 0$.

1.2. Postulats de l'algèbre de commutation

L'algèbre de commutation est une algèbre de Boole définie sur un ensemble comprenant deux éléments (B_2) que nous noterons 0 et 1 (B_2={0,1}). Toute variable de commutation (ou variable logique) peut prendre l'une de ces deux valeurs.

L'algèbre de commutation est le système algébrique constitué de l'ensemble (0,1) et des opérateurs ET, OU, INV définis par les postulats suivants :

L'opération OU (ou disjonction), notée + est définie par :

(P1) 1+1 = 1+0 = 0+1 = 1

(P2) 0+0 = 0

L'opération ET (ou intersection), notée . est définie par :

(P1*) 0.0 = 0.1 = 1.0 = 0

(P2*) 1.1 = 1

L'opération INV (ou complément), notée ' est définie par :

(P3) 0' = 1

(P3*) 1' = 0

Remarque : Ces postulats marchent par paire. Chaque élément de la paire peut être obtenu à partir de l'autre élément en échangeant les symboles 0 et 1 et les opérateurs. et +. Cette propriété est un exemple du principe général de dualité. Au même titre que pour les postulats, cette propriété sera vraie pour tous les théorèmes de l'algèbre de commutation.

1.3. Théorèmes de l'algèbre de commutation

Tous les théorèmes que nous allons énoncer peuvent être démontrés par parfaite induction, c'est à dire en considérant l'ensemble des cas possibles, car le nombre de variables intervenant dans ces propriétés est faible.

1.3.1. Théorèmes monovariables

(T1) x+0=x	(T1*) x.1=x	(Identité)
(T2) x+1=1	(T2*) x.0=0	(Elément nul)
(T3) x+x=x	(T3*) x.x=x	(Idempotence)
(T4) x+x'=1	(T4*) x.x'=0	(Complémentation)
(T5) (x')'=x		(Involution)

Remarque : Trois de ces théorèmes (T2,T3,T3*) méritent l'attention car ils ne sont pas vrais dans l'algèbre ordinaire.

1.3.2. Théorèmes multivariables

(T6) x+y =y+x	(Commutativité)
(T6*) x.y =y.x	(Commutativité)
(T7) (x+y)+z = x+(y+z) = x+y+z	(Associativité)

(T7*) $(x.y).z = x.(y.z) = x.y.z$ (Associativité)
(T8) $x+(x.y) = x$ (Absorbtion)
(T8*) $x.(x+y) = x$ (Absorbtion)
(T9) $(x+y').y = xy$
(T9*) $(x.y')+y = x+y$
(T10) $x.(y+z) = (x.y)+(x.z)$ (Distributivité)
(T10*) $x+(y.z) = (x+y).(x+z)$ (Distributivité)
(T11) $(x+y).(x'+z).(y+z) = (x+y).(x'+z)$ (Consensus)
(T11*) $x.y + x'.z + y.z = x.y + x'.z$ (Consensus)
(T12) $(x+y).(x'+z) = x.z + x'.y$
(T12*) <=> (T12)
(T13) $(x_1 + x_2)' = x_1'. x_2'$ (De Morgan)
(T13*) $(x_1.x_2)' = x_1' + x_2'$ (De Morgan)

Remarque : Le théorème T10* mérite l'attention car il n'est pas vrai dans l'algèbre ordinaire alors que sont dual (T10) l'est.

Ces théorèmes peuvent être démontrés par parfaite induction mais un théorème peut également être démontré en utilisant d'autres théorèmes déjà démontrés.

Exemple : Démonstration du théorème T12 en utilisant les autres théorèmes de l'algèbre de commutation.

T12 => $(x+y).(x'+z) = x.z + x'.y$

Soit : Expression1 = $(a+b).(a'+c)$ et Expression2 : $x.z + x'.y$
Expression1 =>$(x+y).(x'+z)$
T10 => $x.x' + x.z + x'.y + y.z$
T4* => $x.z + x'.y + y.z$ (démo. directe par le théorème des consensus T11*)
T4 => $x.z + x'.y + y.z.(x+x')$
T10 => $x.z + x'.y + x.y.z + x'.y.z$
T7 => $(x.z + x.y.z) + (x'.y + x'.y.z)$
T8 => $x.z + x'.y$

Les théorèmes T13 et T13* peuvent se généraliser à n variables (Théorème de De Morgan généralisés). Dans ce cas ces théorèmes s'expriment de la manière suivante :

(T13) $(x_1 + x_2 + ... + x_n)' = x_1'. x_2' ... x_n'$ (De Morgan généralisé)
(T13*) $(x_1.x_2...x_n)' = x_1' + x_2' + ... + x_n'$ (De Morgan généralisé)

Ces théorèmes de De Morgan généralisés ne peuvent plus être prouvés par parfaite induction. Pour ces théorèmes, la preuve peut être faite par récurrence.

Exemple : Démonstration du théorème T13 généralisé par récurrence

T13 => $(x_1 + x_2 + ... + x_n)' = x_1'. x_2' ... x_n'$
On a $(x_1 + x_2)' = x_1'. x_2'$ (Théorème de De Morgan prouvé par parfaite induction)
Supposons : $(x_1 + x_2 + ... + x_{n-1})' = x_1'. x_2' ... x_{n-1}'$
$(x_1 + x_2 + ... + x_{n-1} + x_n)' = (x_1 + x_2 + ... + x_{n-1})' . x_n'$
$= x_1'. x_2' ... x_{n-1}' . x_n'$ (CQFD)

1.4. Dualité

1.4.1. Principe de dualité

Le principe de dualité a été évoqué brièvement dans la partie précédente. Dans cette partie le principe de dualité sera étudié de manière plus complète.

Une expression formelle du principe de dualité est donnée par le méta-théorème suivant :

Méta-théorème : Tous les théorèmes de l'algèbre de commutation où interviennent les trois opérations (INV,ET,OU) restent vrais si les opérateurs (ET,OU) et les valeurs logiques (0, 1) sont interchangées.

Ceci est appelé méta-théorème car il s'agit d'un théorème portant sur plusieurs théorèmes. Il peut être démontré uniquement en remarquant qu'il est vrai pour tous les postulats de base de l'algèbre de commutation. Les différents théorèmes découlant de ces postulats, il sera donc vrai pour l'ensemble des théorèmes.

Définition : Soit une expression logique $E(x_1,x_2,...,x_n,.,+,')$, l'expression ***duale*** de E est définie comme :

$$\text{dual de } E = D[E] = E(x_1,x_2,...,x_n,+,.,')$$

Exemple : $E = ab + c \Rightarrow D[E] = (a+b)c$

$E = a + (b+cd)[e(f+bg) + h] \Rightarrow D[E] = a[b(c+d) + (e + f(b+g))h]$

Propriété : $D[D(E)] = E$

Théorème : Le théorème généralisé de De Morgan (T13) peut être repris en utilisant la notion de dualité. Cette forme du théorème montre les relations entre dual et complément.

$$D[E(x_1,x_2,...,x_n)] = E'(x_1',x_2',...,x_n')$$
$$E'(x_1,x_2,...,x_n) = D[E(x_1',x_2',...,x_n')]$$

1.4.2. Applications du principe de dualité

Une des raisons pour laquelle le principe de dualité est utile est qu'il permet d'obtenir des résultats sur des situations duales sans avoir à détailler les résultats. Par exemple, l'opérateur NOR étant l'opérateur dual du NAND, les résultats obtenus sur les réseaux de NAND peuvent être utilisés sur les réseaux NOR sans avoir à détailler l'étude. On peut par exemple démontrer qu'une fonction peut être réalisée avec deux couches de NAND (Application du théorème de De Morgan sur une fonction somme de produit) et en déduire par dualité qu'une fonction peut être réalisée avec deux couches NOR.

Une autre application importante du principe de dualité est la conversion d'expressions « somme de produits » en expressions « produit de sommes ». En effet, il est relativement facile de convertir une expression « produit de sommes » (ΠΣ) en une expression « somme de produits » (ΣΠ), l'inverse l'est moins.

Une expression « produit de sommes » (ΠΣ) peut être convertie en une expression « somme de produits » (ΣΠ) en utilisant la propriété de distributivité (Théorème T10).

Exemple : $(a+b)(c+d) = (a+b)c + (a+b)d$

$= ac + bc + ad + bd$

Une expression « somme de produits » (ΣΠ) peut être convertie en une expression « produit de sommes » (ΠΣ) par un processus dual utilisant la propriété de distributivité duale (Théorème T10').

Exemple : $ab + cd = (ab + c)(ab + d)$

$= (a+c)(b+c)(a+d)(b+d)$

La propriété de distributivité duale n'étant pas vraie dans les algèbres ordinaires, on peut avoir des difficultés à l'utiliser pour ces conversions. Ces difficultés peuvent être évitées en convertissant l'expression « somme de produits » en son dual qui est une expression « produit de sommes », en transformant cette expression en « somme de produits » (opération facile à réaliser), et enfin en reprenant le dual de cette expression, ce qui donne une expression « produit de sommes ».

Exemple : E = ab + cd => D[E] = F = (a + b)(c + d)
= (a+b)c +(a+b)d (T10)
= ac + bc + ad + bd

D[F] = E = (a+c)(b+c)(a+d)(b+d)

1.5. Opérateurs et ensembles d'opérateurs

1.5.1. Opérateurs élémentaires

Les opérateurs entrant dans la constitution de l'algèbre de commutation sont les 3 opérateurs INV, ET, OU définis par les postulats de l'algèbre de commutation. Les opérations réalisées par ces opérateurs peuvent être représentées sous forme tabulaire appelée table de vérité (Figure 1.1).

x	x'
0	1
1	0

xy	x.y
00	0
01	0
10	0
11	1

xy	x+y
00	0
01	1
10	1
11	1

Figure 1.1. *Table de vérité des opérations INV, ET, OU*

Ces opérations logiques élémentaires peuvent également être représentées de manière graphique par des diagrammes appelés diagramme de Venn comme illustré sur la figure 1.2.

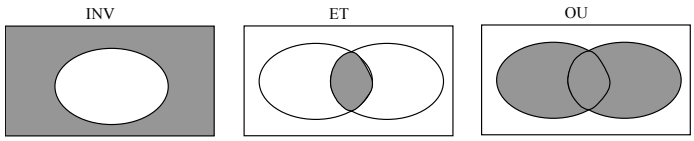


Figure 1.2. *Diagramme de Venn des opérations INV, ET, OU*

Les opérateurs logiques peuvent quant à eux être représentés par des symboles graphiques. Les symboles correspondant aux opérateurs INV, ET, OU sont donnés sur la figure 1.3.

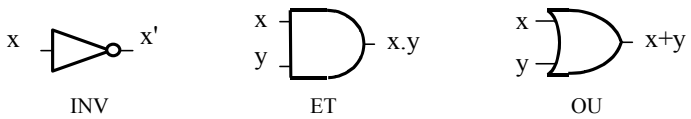


Figure 1.3. *Représentation graphique des opérateurs INV, ET, OU*

Les trois opérateurs INV, ET, OU permettent de réaliser n'importe quelle fonction logique. En ce sens, ils constituent un groupe complet d'opérateurs.

Aucun des opérateurs INV, ET, OU ne forme à lui seul un groupe complet.

(ET, OU) n'est pas un groupe complet. Il n'est en effet pas possible de réaliser un inverseur avec des opérateurs ET et OU.

(ET, INV) est un groupe complet

x => x'

x.y => x.y

x+y => (x'.y')'

(OU, INV) est un groupe complet

x => x'

x+y => x+y

x.y => (x'+y')'

D'autres opérations que les 3 opérations élémentaires de l 'algèbre de commutation peuvent être définies. Le nombre d'opération possible de n variables est $2^{**}2^n$. Ainsi, il existe 16 opérations ou opérateurs différents de 2 variables. La définition de ces opérateurs est donnée sur la figure 1.4.

xy	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}
00	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
01	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
10	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
11	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Figure 1.4. *Opérateurs à 2 entrées*

O0 : Nul
O1 : Nor (Non Ou) => x NOR y = (x + y)'
O2 : Inh (Inhibition) => y INH x = x'.y
O3 : x'
O4 : Inh (Inhibition) => x INIB y = x . y'
O5 : y'
O6 : Xor (Ou_Exlusif) => x XOR y = x.y' + x'.y
O7 : Nand (Non_Et) => x NAND y = (x . y)'
O8 : And (Et)
O9 : Nxor (Non_Ou_Exclusif) => x NXOR y = x.y + x'.y'
O10 : y
O11 : Impl (Implication) => x IMPL y = x' + y
O12 : x
O13 : Impl (Implication) => y IMPL x = x + y'
O14 : Or (Ou)
O15 : Identité

1.5.2. Opérateur NAND

La table de vérité définissant l'opérateur NAND ainsi que les représentations graphiques associées à cet opérateur sont présentées sur la figure 1.5.

xy	x∧y
00	1
01	1
10	1
11	0

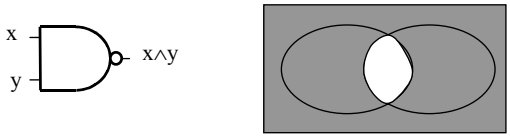


Figure 1.5. *Opérateur NAND*

L'opérateur NAND est commutatif : $x \wedge y = y \wedge x$

L'opérateur NAND n'est pas associatif : $(x \wedge y) \wedge z \neq x \wedge (y \wedge z)$ $(xyz = 110)$

L'opérateur NAND est un opérateur complet. Toute expression logique peut en effet être exprimée uniquement à l'aide d'opérateurs NAND en utilisant les transformations suivantes :

x'	=> (x∧x)	(T3')
x.y	=> (x∧y) ∧ (x∧y)	(T3*, T5)
x+y	=> (x∧x) ∧ (y∧y)	(T3*, T13*, T5)

L'opérateur NAND est particulièrement intéressant car facilement réalisable notamment en technologie MOS.

1.5.3. Opérateur NOR

La table de vérité définissant l'opérateur NOR ainsi que les représentations graphiques associées à cet opérateur sont présentées sur la figure 1.6.

xy	x∨y
00	1
01	0
10	0
11	0

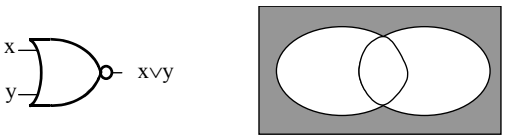


Figure 1.6. *Opérateur NOR*

L'opérateur NOR est commutatif : $x \vee y = y \vee x$

L'opérateur NOR n'est pas associatif $(x \vee y) \vee z \neq x \vee (y \vee z)$ $(xyz = 001)$

L'opérateur NOR est un opérateur complet. Toute expression logique peut en effet être exprimée uniquement à l'aide d'opérateurs NOR en utilisant les transformations suivantes :

x'	=> (x∨x)	(T3)
x.y	=> (x∨x) ∨ (y∨y)	(T3,T13,T5)
x+y	=> (x∨y) ∨ (x∨y)	(T3, T5)

L'opérateur NOR est particulièrement intéressant car facilement réalisable notamment en technologie MOS.

1.5.4. Opérateur INH (Inhibition)

La table de vérité définissant l'opérateur INIB ainsi que les représentations graphiques associées à cet opérateur sont présentées sur la figure 1.7.

xy	x/y
00	0
01	0
10	1
11	0

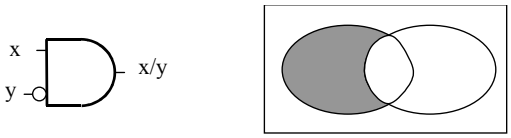


Figure 1.7. *Opérateur INH*

L'opérateur INH n'est pas commutatif : $x / y \neq y / x$ $(xy = 01)$

L'opérateur INH n'est pas associatif : $(x / y) / z \neq x / (y / z)$ $(xyz = 101)$

L'opérateur INH est un opérateur complet. Toute expression logique peut en effet être exprimée uniquement à l'aide d'opérateurs INH en utilisant les transformations suivantes :

x' => (1/x)

x.y => x/(1/y)

x+y => 1/[(1/x)/y]

1.5.5. Opérateur IMPL (Implication)

La table de vérité définissant l'opérateur IMPL ainsi que les représentations graphiques associées à cet opérateur sont présentées sur la figure 1.8.

xy	$x \Rightarrow y$
00	1
01	1
10	0
11	1

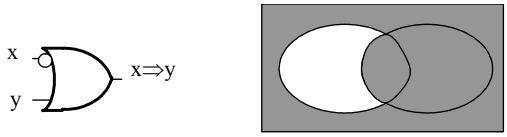


Figure 1.8. *Opérateur IMPL*

L'opérateur IMPL n'est pas commutatif : $x \Rightarrow y \neq y \Rightarrow x$ $(xy = 01)$

L'opérateur IMPL n'est pas associatif : $(x \Rightarrow y) \Rightarrow z \neq x \Rightarrow (y \Rightarrow z)$ $(xyz = 010)$

L'opérateur IMPL (qui est en fait le complément de l'opérateur INH) est un opérateur complet. Toute expression logique peut en effet être exprimée uniquement à l'aide d'opérateurs IMPL en utilisant les transformations suivantes :

x' => $(x \Rightarrow 0)$

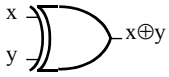
x.y => $[y \Rightarrow (x \Rightarrow 0)] \Rightarrow 0$

x+y => $(x \Rightarrow 0) \Rightarrow y$

1.5.6. Opérateur XOR (OU exclusif)

La table de vérité définissant l'opérateur XOR ainsi que les représentations graphiques associées à cet opérateur sont présentées sur la figure 1.9.

xy	x⊕y
00	0
01	1
10	1
11	0



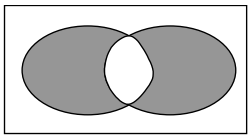


Figure 1.9. *Opérateur XOR*

L'opérateur XOR est commutatif : $x \oplus y = y \oplus x$

L'opérateur XOR est associatif $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

Du fait de ces propriétés de commutativité et d'associativité, l'opérateur XOR à 2 entrées peut être généralisé à un opérateur XOR multi-entrées.

Contrairement aux opérateurs NAND et NOR, INIB et IMPL, l'opérateur XOR ne forme pas un groupe complet à lui seul. Par contre, puisqu'il est possible de réaliser un inverseur avec un XOR et que les groupes d'opérateurs suivants sont des groupes complets (OU, INV), (ET, INV), les groupes (XOR, OU) et (XOR, ET) sont des groupes complets. Le groupe complet (XOR, ET) est un anneau booléen appelé champs de galois.

Toute expression logique peut en effet être exprimée dans le champs de Galois en utilisant les transformations suivantes :

$x' = x \oplus 1$

$x.y = x.y$

$x+y = x \oplus y \oplus x.y$

1.5.6.a. Identités remarquables

Les identités suivantes permettent de manipuler des expressions algébriques faisant intervenir des opérateurs XOR :

I1 : $x \oplus y = xy' + x'y = (x+y)(x'+y')$

I2 : $(x \oplus y)' = x \oplus y' = x' \oplus y = xy + x'y' = (x'+y)(x+y')$

I3 : $x \oplus x = 0$

I4 : $x \oplus x' = 1$

I5 : $x \oplus 1 = x'$

I6 : $x \oplus 0 = x$

I7 : $x(y \oplus z) = xy \oplus xz$

I8 : $x+y = x \oplus y \oplus xy = x \oplus x'y \Rightarrow x+y = x \oplus y$ si $xy=0$

I9 : $x \oplus (x+y) = x'y$

I10 : $x \oplus xy = xy'$

Exemple : Exprimer l'expression logique suivante dans le champs de Galois.

Exp: a.b' + b'.c' + a'.c'

$\Rightarrow (a.b' \oplus a'.c') + b'.c'$

$\Rightarrow (a.b' \oplus a'.c') \oplus b'.c' \oplus b'.c'.(a.b' \oplus a'.c')$

$\Rightarrow a.b' \oplus a'.c' \oplus b'.c' \oplus a.b'.c' \oplus a'.b'.c'$

$\Rightarrow a.b'.c \oplus a.b'.c' \oplus a'.c'$

$\Rightarrow a.b'.(c \oplus c') \oplus a'.c'$

$\Rightarrow a.b' \oplus a'.c'$

=> a.(b⊕1) ⊕ (a⊕1).(c⊕1)

=> a.b ⊕ a ⊕ a.c ⊕ a ⊕ c ⊕ 1

=> a.b ⊕ a.c ⊕ c ⊕ 1

1.5.6.b. Exemples d'utilisation de l'opérateur XOR

Application 1 : Si la constante 1 est appliquée sur une entrée de l'opérateur XOR, cet opérateur se comporte comme un inverseur (1 ⊕ x=x'). De ce fait, l'opérateur XOR peut être utilisé comme un inverseur contrôlé (figure 1.10).

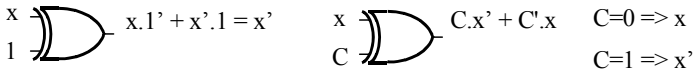


Figure 1.10. *Inverseur contrôlé*

Lorsque le signal de commande C est à 0, la sortie est égale à x et lorsque ce signal de commande est à 1, la sortie est égale à x. Ce type de structure est souvent utilisé dans les circuits pour accroître leur utilité.

Application 2 : L'opérateur XOR est également très utilisé dans les circuits de détection et de correction d'erreurs. La fonction parité ou somme modulo 2 du nombre de bits est effectivement la fonction la plus souvent employée dans ce type de circuits.

b0 ⊕ b1 ⊕ ... ⊕ bn = 0 si le nombre de bits à 1 est pair
b0 ⊕ b1 ⊕ ... ⊕ bn = 1 si le nombre de bits à 1 est impair

Application 3 : L'opérateur XOR réalise la fonction SOMME_MODULO_2. Ainsi, la sortie somme d'un additionneur binaire dont les entrées sont x, y est : s=x⊕y. La sortie somme d'un additionneur binaire dont les entrées sont x, y et r (bits de retenue de l'étage précédent) est : s=x⊕y⊕r.

1.5.7. Opérateur MUX21 (Multipleur 2 vers 1)

L'opérateur MUX21 (Multiplexeur) est un opérateur portant sur 3 variables défini de la manière suivante (Figure 1.11):

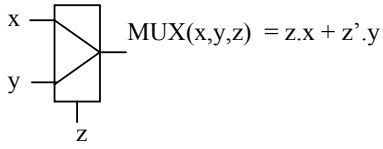


Figure 1.11. Opérateur MUX

L'opérateur MUX21 est un opérateur complet.

x' => MUX (0,1,x)
x.y => MUX (x,0,y)
x+y => MUX (1,x,y)

Cet opérateur est particulièrement intéressant dans le cadre de certaines applications (structures itératives). Il est notamment très utilisé pour réaliser les circuits programmables de type FPGA.

1.6. Implantation technologique des opérateurs logiques

Le but n'est pas ici de rentrer dans les détails électroniques de l'implantation technologique des opérateurs logiques, mais simplement de donner quelques notions d'implantation qui peuvent être utiles pour orienter les traitement à réaliser sur les expressions logiques

1.6.1. Implantation à relais

Le composant historique permettant la réalisation de fonctions logiques est le relais ou interrupteur. La représentation et le fonctionnement de relais à commande directe (a) et à commande inversée (b) sont donnés sur la figure 1.12.

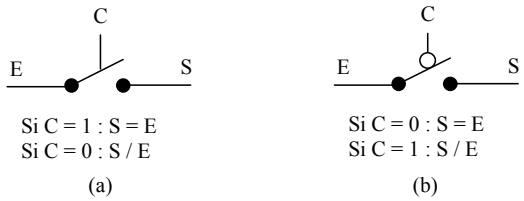


Figure 1.12. *Relais à commande directe (a) et inversée (b)*

Les opérations logiques élémentaires (INV, ET, OU) peuvent être réalisées par la mise en série ou en parallèle de relais comme indiqué sue la figure 1.13.

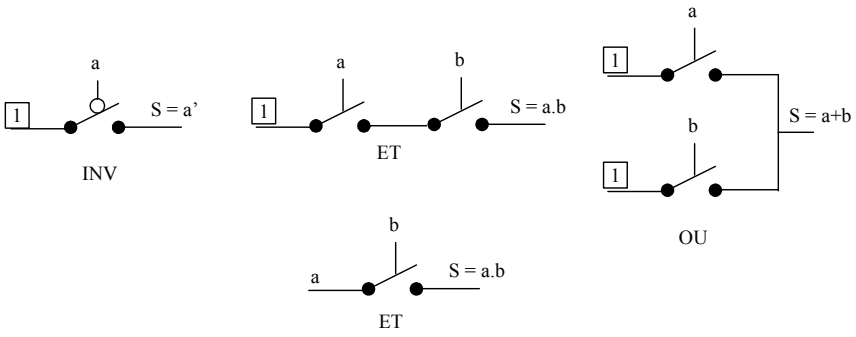


Figure 1.13. *Implantation à relais des opérateurs logiques élémentaires*

Dans la mesure ou le 0 logique sur les entrées est une absence d'alimentation (et non une connexion à la masse), l'opérateur OU peuvent être optimisés (figure 1.14). Cette structure n'est effectivement valides que si le 0 logique n'est qu'une absence d'alimentation et non une connexion à la masse (sinon cela entraînerai des possibilités de court-circuit entre masse et alimentation.

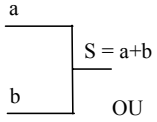


Figure 1.14. *Cas particulier d'implantation de l'opérateurs OU*

Tous les opérateurs ou expressions logiques peuvent être implantés en utilisant les opérateurs élémentaires ou en construisant directement, sur le même principe l'expression logique équivalente. Les opérateurs XOR et MUX peuvent, par exemple s'implanter de la manière suivante (figure 1.15):

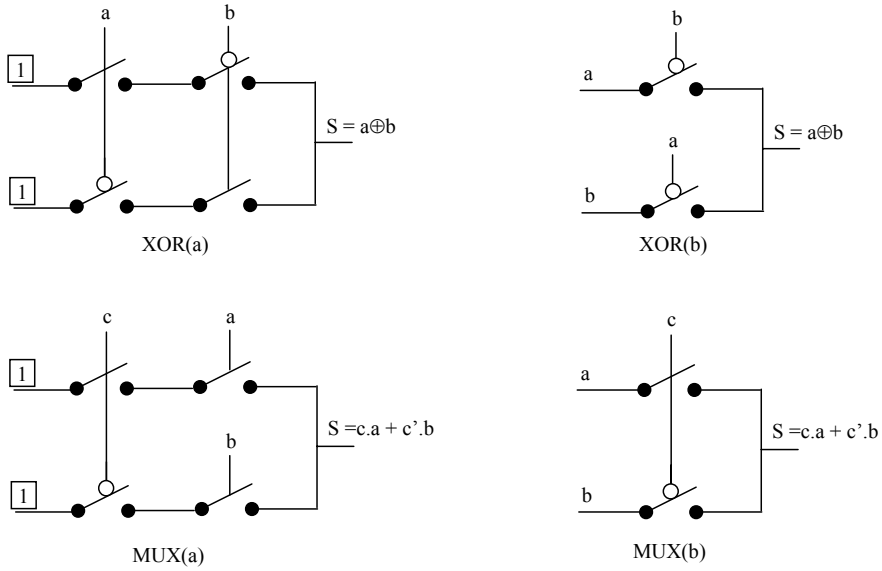


Figure 1.15. *Implantation à relais des opérateurs XOR et MUX*

Remarque : La fonctionnalité du XOR et du MUX est telle que les implantations XOR(b) et MUX(b) restent valide même si le 0 logique d'une entrée est une connexion à la masse. Dans les deux cas, il n'y a effectivement pas de possibilité de court-circuit entre masse et alimentation.

Sur ce principe, nous pouvons imaginer des structures à base de multiplexeurs permettant de réaliser une fonction logique quelconque. Ainsi, la structure représentée sur la figure 1.16 réalise l'expression logique S = ab + a'b'. Ce type de structure est à la base des circuits programmables de type FPGA.

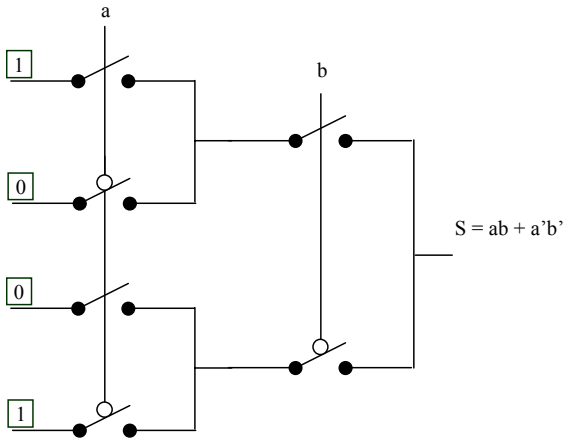


Figure 1.16. *Implantation multiplexeur d'une expression logique*

1.6.2. Implantation en technologie NMOS

D'un point de vue fonctionnel, un transistor NMOS peut être assimilé à un relais à commande directe (Figure 1.17). Cependant pour assurer le bon fonctionnement de ce dispositif, il faut rajouter une contrainte électrique imposant que le 0 logique soit une connexion à la masse.

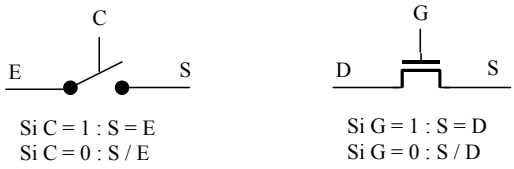


Figure 1.17. *Relais et transistor NMOS*

Si le comportement fonctionnel du transistor est identique à celui du relais, la contrainte électrique imposant que le 0 logique soit une connexion à la masse fait que les structures présentées dans le cadre d'une technologie à relais doivent être adaptées pour être transposables à la technologie MOS. Les structures présentées sur la figure 1.18 sont telles que le 0 logique est effectivement obtenu par une connexion à la masse. La résistance présente dans ces structures permet d'éviter la mise en court-circuit des alimentations.

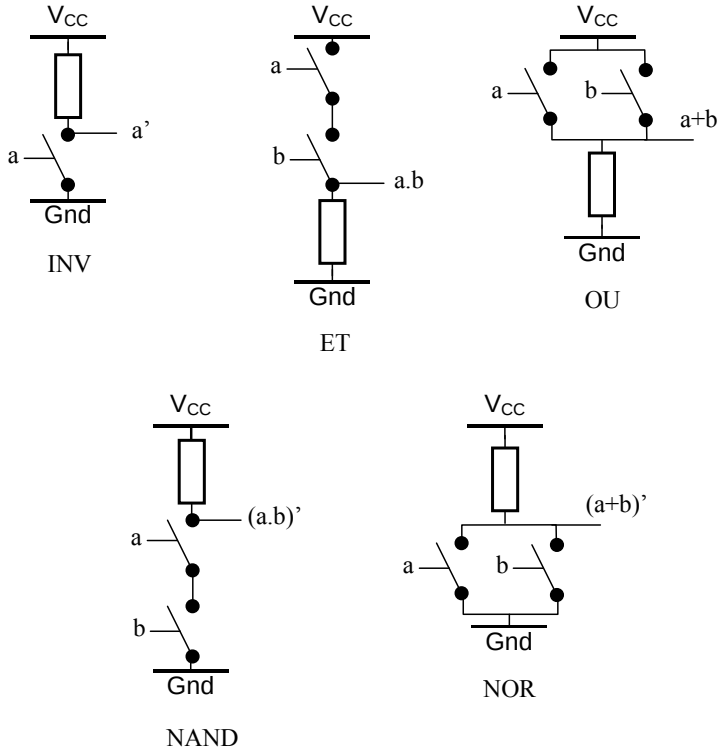


Figure 1.18. *Structure des opérateurs logiques principaux avec 0 logique à la masse*

La réalisation de ces structures en technologie NMOS passe maintenant par la transposition des relais en transistors NMOS et par la réalisation de la résistance. Les contraintes technologiques liées à la réalisation de cette résistance (transistor déplété avec grille et source reliées et drain à Vcc) font que parmi les 5 structures présentées sur la figure 1.18, les seules réalisables en technologie NMOS sont celles ou la résistance est

connectée à Vcc, c'est à dire les structures inverseuses (INV, NAND, NOR). Le schéma de ces structures en technologie NMOS est présenté sur la figure 1.19.

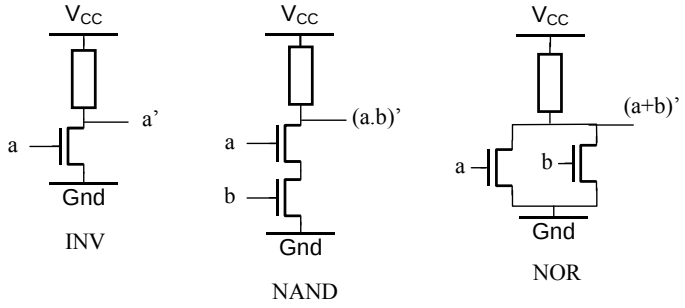


Figure 1.19. *Structure NMOS des opérateurs logique INV, NAND, NOR*

En généralisant ce principe, la structure générale d'un composant logique (porte) NMOS est illustrée sur la figure 1.20.

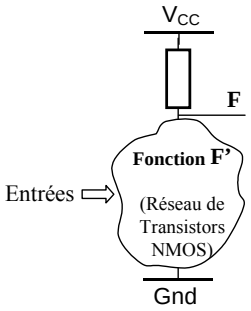


Figure 1.20. *Structure générale d'une porte NMOS*

Tous les opérateurs ou expressions logiques peuvent ainsi être implantés en utilisant les opérateurs élémentaires ou en construisant directement, sur le principe précédent, l'expression logique équivalente. A titre d'exemple, les opérateurs XOR et MUX peuvent s'implanter de la manière suivante (figure 1.21):

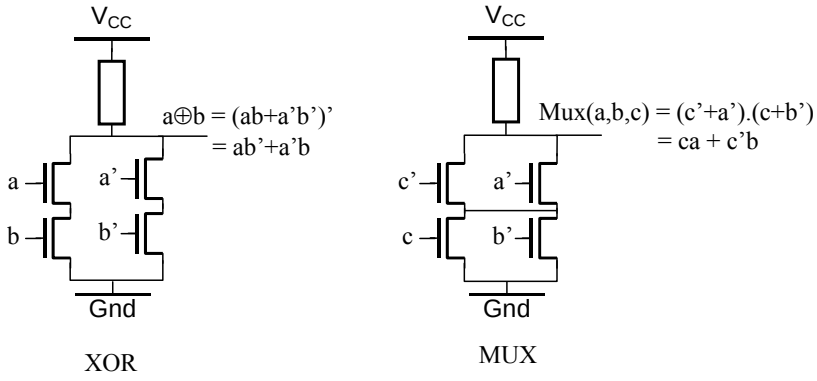


Figure 1.21. *Implantation en technologie NMOS des opérateurs XOR et MUX*

Remarque : Les contraintes technologiques liées notamment aux tensions de seuil des transistors conduisent à des limitations sur le nombre de transistors pouvant être mis en série ou en parallèle entre les alimentations. Ces contraintes, ne relevant pas de l'aspect logique du dispositif mais plutôt de l'aspect électronique ou technologique, elles ne seront pas développées ici.

Par contre, nous pouvons souligner le gros inconvénient de la technologie NMOS qui est sa consommation d'énergie. En effet, sur le niveau bas de la sortie, un courrant s'établi entre les alimentations Vcc et Gnd à travers la résistance. Ce courrant dans la résistance est à l'origine d'une consommation importante. La technologie CMOS permet de palier à cet inconvénient.

1.6.3. Implantation en technologie CMOS

En technologie CMOS deux types de transistors sont utilisés ; les transistors NMOS et les transistors PMOS. D'un point de vue fonctionnel, ces transistors peuvent être assimilés à des relais à commande directe (NMOS) ou inversée (PMOS) (Figure 1.22).

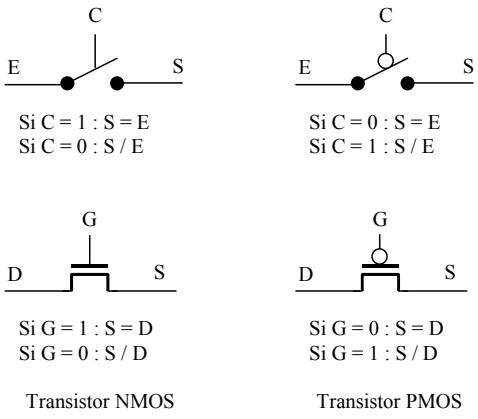


Figure 1.22. *Relais et transistors NMOS et PMOS*

L'objectif principal de cette technologie CMOS est de réduire la consommation d'énergie par rapport à la technologie NMOS. Le principe de base est remplacer la résistance des structures NMOS par une structure à transistors PMOS assurant la fonction complémentaire de celle réalisée par les transistors NMOS. Les deux fonctions ou plans (NMOS et PMOS) étant complémentaires elles interdisent toute connexion (et donc courrant) entre l'alimentation et la masse. Le principe de réalisation d'une cellule CMOS est illustré sur la figure 1.23.

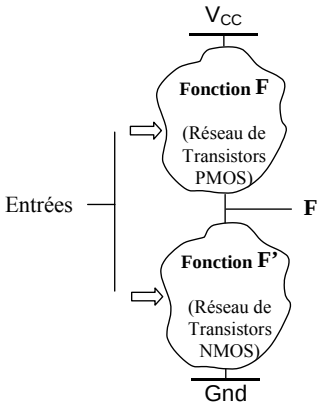


Figure 1.23. *Structure générale d'une porte CMOS*

Le plan PMOS étant construit avec des transistors à commande inversée (transistors PMOS), la fonction implantée dans ce plan (fonction F) est la fonction duale de celle implantée dans la partie NMOS (F(X) = D[F'(X'))]). La structure CMOS ainsi obtenue des opérateurs INV, NAND, NOR est présentée sur la figure 1.24.

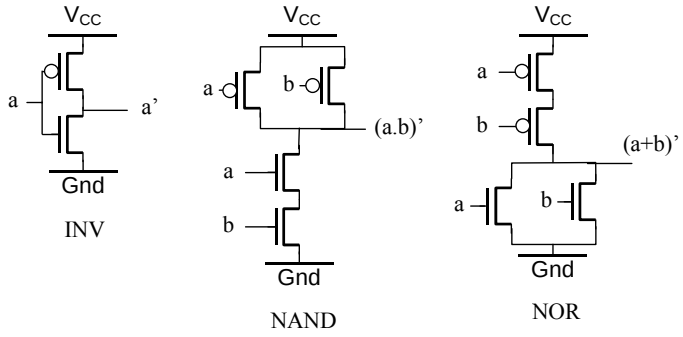


Figure 1.24. *Structure CMOS des opérateurs INV, NAND, NOR*

Chapitre 2

Fonctions et équations logiques

Un circuit logique est un ensemble d'éléments physiques connectés qui réalise une fonction logique. Les notions de fonctions logiques et équations logiques seront développées dans ce chapitre.

2.1. Fonctions logiques

2.1.1. Domaine de définition

B_2 étant l'ensemble des valeurs logiques (B_2={0,1}), B_2^n est l'ensemble formé de quantités X=(x_1,x_2,...x_n) ou les « x_i » sont des variables booléennes. B_2^n est un ensemble comprenant 2^n points (figure 2.1).

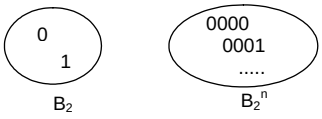


Figure 2.1. *Domaine de définition*

On appelle ***combinaison*** ou encore ***point*** de B_2^n une quantité X=(x_1,x_2,...x_n) ou « xi » est une variable booléenne

On définit une distance sur B_2^n appelée ***distance de hamming*** : soit X_0 et X_1 deux points, la distance de X_0 à X_1 notée d(X_0,X_1) est le nombre de composantes différentes dans X_0 et X_1. C'est le nombre d'arêtes séparant 2 points de l'hypercube.

Exemple : D(00101,01001) = 2.

D a bien les propriétés d'une distance, en particulier D vérifie bien la propriété d'inégalité triangulaire :

$$D(X_0,X_1) \leq D(X_0,X_2) + D(X_2,X_1)$$

Lorsque la distance entre deux combinaisons de B_2^n est égale à 1 les deux points sont dits adjacents. Toute combinaison de n variables est adjacente à n autres combinaisons obtenues en changeant l'une des composantes.

2.1.2. Fonctions logiques simples

Une ***fonction logique simple*** de n variables est une application qui à toute combinaison X appartenant à B_2^n, fait correspondre un élément y appartenant à B_2. Elle sera notée y=f(X).

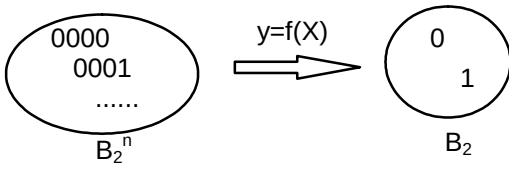


Figure 2.2. *Fonctions logiques simples*

Une fonction simple divise l'ensemble des combinaisons de B_2^n en deux parties : l'ensemble des points "vrais" qui sont les combinaisons X_i telles que $f(X_i)=1$ et l'ensemble des points "faux" qui sont les combinaisons X_j telles que $f(X_j)=0$.

Un point X tel que f(X) = 1 est dit ***couvert*** par la fonction. La ***couverture*** de la fonction est l'ensemble des points couverts.

Le nombre de fonctions logiques de n variables est $2^{**}(2^n)$ puisqu'à chacune des 2^n combinaisons on peut faire correspondre l'une des deux valeurs 0,1. Ce nombre comprend toutes les fonctions de n variables au plus c'est à dire toutes les fonctions de 0,1,2,...,n variables.

La ***fonction complémentaire*** d'une fonction f(X) notée f'(X) est une fonction qui prend la valeur complémentaire de f(X) pour toutes les combinaisons X_i.

Un ordre partiel est défini sur les fonctions. On dit que $f \leq g$ si et seulement si pour tout X tel que f(X) = 1 alors g(X) =1, i.e. l'ensemble des points couverts par g contient l'ensemble des points couverts par f.

2.1.3. Fonctions logiques multiples

Une ***fonction logique multiple*** (ou simultanée) est une application qui à toute combinaison X_i de B_2^n fait correspondre un point Y_i appartenant à B_2^q. Nous emploierons alors la notation Y=f(X).

Exemple1 : n=3, q=2

$y1 = a.b + c$

$y2 = a.c$

Exemple2 : Additionneur binaire n=3, q=2

$S = a \oplus b \oplus r$

$R_{sortante} = r . (a \oplus b) + a.b$

2.1.4. Fonctions logiques incomplètement spécifiées

Lors de la définition d'une fonction, la valeur de cette fonction peut être indifférente pour certaines combinaisons qui sont appelées combinaisons non-spécifiées. On conviendra de noter ϕ la valeur de la fonction en ces points.

Nous appellerons ***variable ϕ-booléenne*** une variable susceptible de prendre ses valeurs dans l'ensemble $B_2\phi=(0,1,\phi)$.

Une ***fonction incomplète*** est une application qui à toute combinaison X_i de B_2^n fait correspondre un élément de $B_2\phi^q$ ou $B_2\phi=(0,1,\phi)$.

On peut caractériser une fonction incomplète d'une façon qui en montre mieux sa nature. Pour cela définissons une relation d'ordre total sur $B_2\phi$ par $0 \leq \phi \leq 1$.

Cette relation d'ordre total permet de définir une relation d'ordre partiel sur l'ensemble des combinaisons en posant $X \leq Y$ si pour tout i $x_i \leq y_i$.

Exemple : $(0\phi1\phi01) \leq (0\phi11\phi1)$

Cette relation d'ordre partiel induit alors une relation d'ordre partiel sur l'ensemble des fonctions incomplètement spécifiées qui est : $f \leq g$ si pour tout X_i $f(X) \leq g(X)$.

Avec ces conventions on peut définir une fonction incomplètement spécifiée f à l'aide de deux fonctions complètes appelées « borne inférieure » (f_{inf}) et « borne supérieure » (f_{sup}).

La borne inférieure f_{inf} (supérieure) d'une fonction f est obtenue en remplaçant chacune des valeurs non spécifiées de f par la valeur 0 (1). On a donc la relation $f_{inf} \leq f \leq f_{sup}$.

2.2. Représentation des fonctions logiques

Il existe plusieurs manières de représenter les fonctions logiques. Parmi les plus courantes, nous noterons :

- la représentation algébrique,
- les représentations tabulaires,
- les représentations implicites,
- les représentations graphiques.

2.2.1. Représentation algébrique

Une fonction peut être définie à partir d'une expression logique comme étant la valeur de cette expression logique. A toute expression logique correspond donc une fonction logique et une seule.

Exemple : f(a,b,c) = a.b' + b.c

Cette représentation d'une fonction est appelée représentation algébrique.

Réciproquement on peut démontrer qu'il est toujours possible d'associer une expression logique à une fonction.

Exemple : Soit f / (0,0) -> 0
(1,1) -> 0
(0,1) -> 1
(1,0) -> 1

f(a,b) = ab' + a'b = a $\oplus$ b

Les termes a.b', a'.b, … sont les monômes booléens de la fonction. De manière plus générale, on appelle ***monôme booléen*** une quantité $M = A.x_1^*.x_2^*...x_p^*$

ou : « A » est le coefficient du monôme (variable booléenne)

« x_i » sont des variables booléennes

« x_i*» sont des littéraux (signifie que l'on ne précise pas si on a xi ou xi')

« $x_1^* . x_2^* ... x_p^*$ » est le corps du monôme

« p » est le degré du monôme

Si l'on raisonne dans un espace de dimension n, on peut avoir des monômes de dimension 1,2,...,n. D'autre part, un monôme n'a de signification que si son coefficient est égal à 1 (A=1).

On appellera ***monôme canonique*** ou ***minterme*** un monôme de dimension n (p=n).

La représentation algébrique d'une fonction incomplètement spécifiée requiert d'exprimer les bornes inférieures et supérieures de la fonction, c'est-à-dire d'exprimer la fonction d'une part pour $\phi = 0$ et d'autre part pour $\phi = 1$.

Exemple : Soit f / (0,0) -> 0
(1,1) -> ϕ
(0,1) -> 1
(1,0) -> 1

$f_{inf}(a,b) = ab' + a'b$

$f_{sup}(a,b) = ab' + a'b + ab$

2.2.2. Représentations tabulaires

2.2.2.a. Table de vérité

La table de vérité est l'écriture directe sous forme tabulaire de la correspondance entre B_2^n et B_2^p ou $(B_2^*)^p$ définie par la fonction. La liste des points de B_2^n est en général donnée dans l'ordre naturel de numération en base 2 (figure 2.3).

	abc	f	f_{inf}	f_{sup}
X_0	000	ϕ	0	1
X_1	001	1	1	1
X_2	010	1	1	1
X_3	011	0	0	0
X_4	100	1	1	1
X_5	101	ϕ	0	1
X_6	110	1	1	1
X_7	111	1	1	1

Figure 2.3. *Table de vérité d'une fonction incomplète f(a,b,c).*

2.2.2.b. Table de Veitch

La table de vérité possède une forme peu pratique qui la rend rapidement encombrante. On donne souvent les valeurs de la fonction sous la forme d'une table à deux entrées appelée table de Veitch.

Si n est le nombre de variables avec n=p+q, c'est à dire $B_2^n = B_2^p * B_2^q$, on porte sur la première entrée (colonnes) les éléments de B_2^p et sur la deuxième entrée les éléments de B_2^q avec les mêmes conventions que pour la table de vérité (ordre binaire naturel). On choisit en général p=partie entière de n/2.

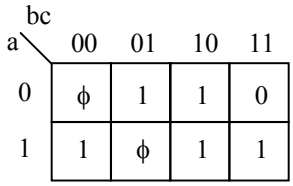


Figure 2.4. *Ecriture de la fonction f précédente sous forme de table de Veitch.*

2.2.2.c. Table de Karnaugh

Pour de nombreux problèmes de logique, en particulier pour la simplification des fonctions, on utilise la propriété d'adjacence.

Afin de mettre en évidence cette propriété, il est commode d'utiliser une table à deux entrées (comme la table de Veitch) avec un codage non plus binaire naturel mais binaire réfléchi (code de Gray). Une telle table est appelée table de Karnaugh.

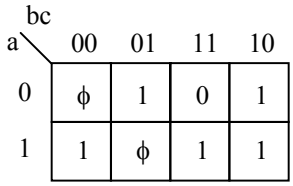


Figure 2.5. *Ecriture de la fonction f précédente sous forme de table de Karnaugh.*

Ainsi, les tables de Karnaugh sont telles que deux cases contiguës de la table correspondent à deux combinaisons adjacentes. Malheureusement la réciproque n'est pas vraie et il existe sur une telle table des cases non contiguës dont les codes sont adjacents.

Sur la table présentée sur la figure 2.5, le codage des cases « x » est adjacent au codage de la case X.

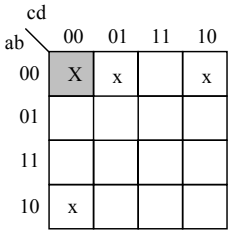


Figure 2.5. *Adjacences sur un table de karnaugh*

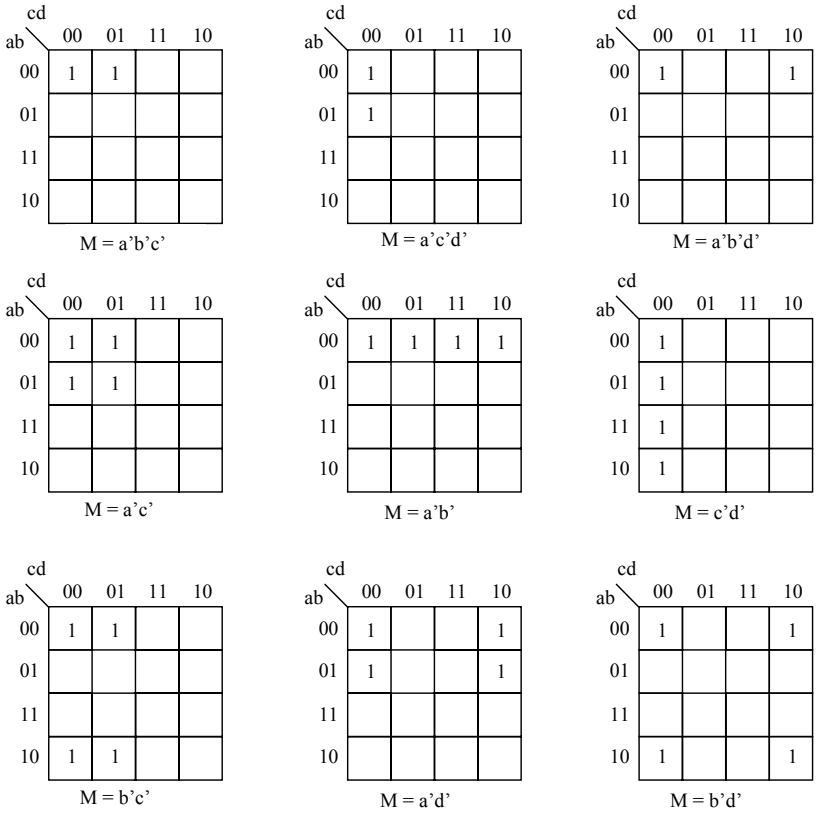


Figure 2.6. *Relation entre monômes et adjacences*

Les tables présentées sur la figure 2.6 montrent la relations entre monômes et adjacences sur la table de karnaugh.

2.2.3. Représentations implicites

2.2.3.a. Image caractéristique

L'image caractéristique est constituée des 2^n valeurs de la fonction, données sous forme d'une ligne. Pour retrouver la fonction, il est nécessaire de préciser les conventions utilisées. En général, et c'est ce que nous choisirons, les 2^n valeurs sont rangées dans l'ordre naturel de numération en base 2 croissant de la gauche vers la droite.

Exemple : L'image caractéristique de la fonction f précédente est :

$I_c[f(a,b,c)] = \phi 1101 \phi 11$

2.2.3.b. Image décimale

Une combinaison X_i peut être considérée comme une représentation en base 2 d'un nombre décimal N. Dans ce cas, il est nécessaire de d'établir une convention sur l'ordre des variables (poids fort à gauche : f (a,b,c,d) => a de poids fort).

On peut alors définir la fonction en précisant l'équivalent décimal des combinaisons X_i telles que f(X)=1. (L'ensemble des combinaisons X_i telles que f(X)=0 est précisé par défaut). Dans le cas des fonctions incomplètes il est nécessaire de préciser également les combinaisons X_i telles que $f(X)=\phi$.

Exemple : L'image décimale de la fonction f précédente est (a poids fort) :

$I_d[f(a,b,c)] = R_1(1,2,4,6,7) + R_\phi(0,6)$

2.2.4. Représentations graphiques

2.2.4.a. Hypercube

Une combinaison de B_2^n peut être représentée par un point dans un espace de dimension n. Ces points constituent les sommets d'un hypercube de dimension n.

Une fonction simple peut être représentée graphiquement en marquant les sommets de cet hypercube correspondant aux points vrais de la fonction (figure 2.7).

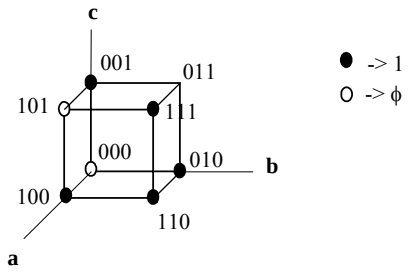


Figure 2.7. *Représentation d'une fonction dans l'hypercube*

Comme illustré sur la figure 2.8,

- un minterme est un sommet de l'hypercube
- un monôme est une "face" de l'hypercube.
- dans B_2^n, un monôme de degré p couvre 2^{n-p} points.

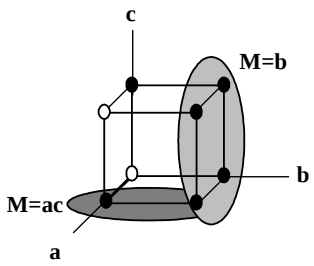


Figure 2.8. *Visualisation des monômes sur l'hypercube.*

Cette représentation qui met en évidence les distances entre les points de B_2^n n'est vraiment pratique que pour n<5.

2.2.4.b. Treillis

Une combinaison de B_2^n peut être représentée par une notation indicielle indiquant la position des 1. Par exemple, la combinaison 0101 peut être représentée par l'élément (2,4).

En utilisant cette notation, l'hypercube peut être transformé en un treillis dont la 1ère colonne ne comprend aucun indice (point 0), la seconde en comprend 1 (point ne comportant qu'un seul 1), la 3ème en comprend 2

(points comportant deux 1) et ainsi de suite. Cette représentation illustrée sur la figure 2.9 peut permettre de représenter des fonctions comportant plus de variables que la représentation hypercube tout en conservant la caractéristique de ce dernier à savoir faire apparaître les distances entre les points B_2^n (et notamment les points adjacents). Pour cela, il suffit de relier les points d'une colonne i aux points de la colonne suivante incluant les indices d'origine.

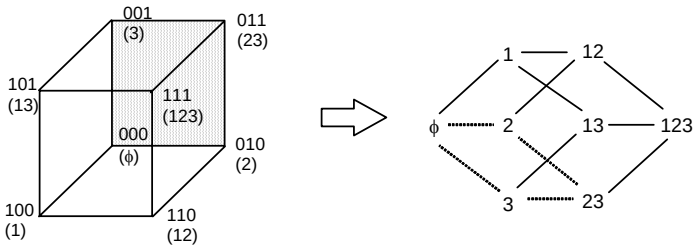


Figure 2.9. *Représentation d'une fonction sous forme de treillis.*

Dans ce cas, les mintermes sont les sommets du treillis et les monôme un sous-treillis composé de l'ensemble des liaisons reliant un point d'une colonne i à un point d'une colonne j. Le monôme correspondant à la face grisée de l'hypercube soit « a' » se retrouve dans le sous-treillis [(ϕ), (2,3)]

A titre d'exemple sur un espace de dimension 4 (figure 2.10) le point (12) représente le minterme « abc'd' » et le sous-treillis allant du point (1) au point (1234) représente le monôme « a ». Le sous-treillis allant du point (1) au point (123) représente quant à lui le monôme « ad' ».

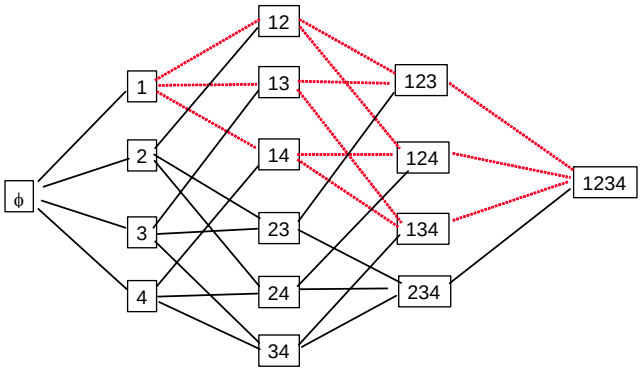


Figure 2.10. *Représentation d'une fonction de dimension 4*

2.2.4.c. Logigramme

C'est à la fois une représentation graphique et symbolique. Elle s'effectue à l'aide des représentations symboliques des opérateurs binaires reliées entre elles. Chaque opérateur est caractérisé par un ensemble d'entrées, une sortie et une règle opératoire précisée soit par un signe placé à l'intérieur de l'opérateur soit par sa forme même. La règle opératoire permet de calculer l'expression logique associée à la sortie d'un opérateur dès que l'on connaît les expressions logiques attachées à toutes ses entrées.

Une sortie d'opérateur est reliée à des entrées d'autres opérateurs. Les sorties libres sont appelées « sorties primaires », les entrées libres « entrées primaires ». A chaque entrée libre est associée une variable logique.

Un logigramme permet de représenter une expression logique et donc la fonction valeur de cette expression logique. Un exemple de logigramme est donné sur la figure 2.11.

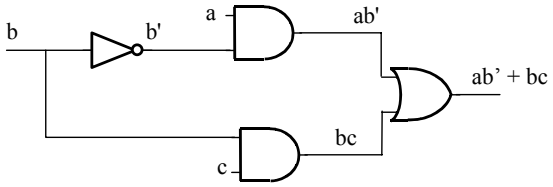


Figure 2.11. *Logigramme associé à la fonction f(a,b,c) = a.b' + b.c*

2.3. Fonctions remarquables

2.3.1. Fonctions linéaires

Définition : On dira qu'une fonction est ***linéaire*** si elle admet en notation galoisienne une expression qui s'écrit sans l'opérateur ET. Une fonction linéaire peut donc se mettre sous la forme d'une somme modulo 2 de variables. a⊕b⊕c et 1⊕a⊕b⊕c sont des fonctions linéaires.

Exemple : Soit f(a,b,c) = a'.b'.c' + a.b.c' + a.b'.c + a'.b.c

f(a,b,c) = c'.(a'.b' + a.b) + c.(a.b' + a'.b)

= c'.(a⊕b)' + c.(a⊕b)

= (c⊕(a⊕b))'

= 1 ⊕ c ⊕ a ⊕ b

=> f(a,b,c) est une fonction linéaire

2.3.2. Fonctions paires et impaires

Soit une fonction f(X). Nous noterons $f_d(X)$ la fonction duale de f(X) et f(X') la fonction obtenue à partir d'une expression quelconque de f(X) en remplaçant chacune des variables par son complément. On a :

$$f_d(X) = f'(X').$$

Exemple 1 : Soit f(X) = a + b f'(X') = (a'.b')
= a.b
= $f_d(X)$

Exemple 2 : Soit f(X) = a.b f'(X') = (a' + b')'
= a+b
= $f_d(X)$

Définition : Une fonction est ***impaire*** si elle est sa propre duale ce qui peut encore s'exprimer sous la forme :

$f(X)=f_d(X)$ ou $f(X)=f'(X')$ ou $f'(X)=f(X')$

Exemple : Soit f(a,b,c) = a.b + b.c + a.c

$f_d(a,b,c)$ = (a+b)(b+c).(a+c)

= (a.b + a.c + b + b.c).(a+c)

= (a.c + b).(a+c)

= (a.c + a.c + a.b + b.c)

= (a.c + a.b + b.c)

= f(a,b,c)

Remarque : Une fonction impaire de n variable a nécessairement 2^{n-1} sommets à 0 et 2^{n-1} sommets à 1. (Si un sommet X est à 0, le sommet X' est nécessairement à 1 et inversement).

Remarque : Si une fonction est impaire, sa fonction complémentaire est également impaire.

Définition : Une fonction est ***paire*** si elle vérifie la relation suivante : f(X')=f(X)

Exemple : Soit f(a,b,c) = a.b + b'.c + a'.c'

f(a',b',c') = a'.b' + b.c' + a.c

= a'.b'.c + a'.b'.c' + a.b.c' + a'.b.c' + a.b.c + a.b'.c

= a.b.(c+c') + b'.c.(a+a') + a'.c'.(b.b')

= a.b + b'.c + a'.c'

= f(a,b,c)

Remarque : Une fonction paire comporte nécessairement un nombre pair de sommets à 1 (et donc aussi un nombre pair de sommets à 0). Si une fonction est paire, sa fonction complémentaire est également paire. L'ensemble des fonctions paires est fermé pour la somme et le produit (la somme de deux fonctions paires est paire, le produit de deux fonctions paires est pair).

2.3.3. Fonctions symétriques

Définition : Une fonction est ***symétrique*** si elle est telle que toute permutation de variables laisse la fonction inchangée.

On démontre qu'une condition nécessaire et suffisante pour qu'une fonction soit symétrique est qu'elle prenne la valeur 1 si et seulement si k variables ont la valeur 1 ou k prend ses valeurs dans l'ensemble $(k_1,...,k_p)$ avec $0 \leq k_i \leq n$.

Exemple : la fonction de trois variables définie par l'expression f(a,b,c) = a.b.c + a'.b'.c + a.b'.c' + a'.b.c' est symétrique.

Elle prend la valeur 1 lorsque et seulement lorsque une ou trois variables prennent la valeur 1.

Définition : Une fonction est ***symétrique par rapport à un ensemble de littéraux*** (variables ou variables complémentées) si elle est telle que toute permutation des littéraux de l'ensemble laisse la fonction inchangée.

Exemple : soit f(a,b,c) = a'.b'.c' + a.b'.c + a'.b.c

La fonction f(a,b,c) n'est pas symétrique par rapport aux variables a,b,c mais l'est par rapport aux littéraux a,b,c'.

Les variables par rapport auxquelles la fonction est symétrique sont appelées « variables de symétrie ».

Remarque : L'ensemble des fonctions symétriques par rapport aux mêmes littéraux est fermé pour la somme et le produit (la somme de deux fonctions symétriques est symétrique, le produit de deux fonctions symétriques est symétrique).

2.4. Formes canoniques des fonctions logiques

2.4.1. Théorème d'expansion de Shannon

Le théorème d'expansion de Shannon est un théorème extrêmement important. C'est en effet sur ce théorème que s'appuient la plupart des techniques modernes de manipulation et d'optimisation des fonctions logiques. Il permet, en décomposant une fonction par rapport à ses variables, de traiter cette fonction de manière récursive.

Théorème : Soit $f(x_1,x_2,...,x_i,...,x_n)$ une fonction logique de n variables

(1) $f(x_1,x_2,...,x_i,...,x_n) = x_i' \cdot f(x_1,x_2,...,0,...,x_n) + x_i \cdot f(x_1,x_2,...,1,...,x_n)$

(2) $f(x_1,x_2,...,x_i,...,x_n) = [f(x_1,x_2,...,0,...,x_n) + x_i] \cdot [f(x_1,x_2,...,1,...,x_n) + x_i']$

$f(x_1,x_2,...,0,...,x_n)$ également noté $f_{xi'}$ est appelé ***cofacteur de f par rapport à x_i'***. Ce terme est obtenu en remplaçant dans f chaque occurrence de x_i par 0.

$f(x_1,x_2,...,1,...,x_n)$ également noté f_{xi} est appelé ***cofacteur de f par rapport à x_i***. Ce terme est obtenu en remplaçant dans f chaque occurrence de x_i par 1.

Preuve : Vérifions l'égalité en posant $x_i = 0$ et $x_i = 1$

Soit $x_i = 0$

(1) $\Rightarrow x_i' \cdot f(x_1,x_2,...,0,...,x_n) + x_i \cdot f(x_1,x_2,...,1,...,x_n) = f(x_1,x_2,...,0,...,x_n)$

(2) $\Rightarrow [f(x_1,x_2,...,0,...,x_n) + x_i] \cdot [f(x_1,x_2,...,1,...,x_n) + x_i'] = f(x_1,x_2,...,0,...,x_n)$

Soit $x_i = 1$

(1) $\Rightarrow x_i' \cdot f(x_1,x_2,...,0,...,x_n) + x_i \cdot f(x_1,x_2,...,1,...,x_n) = f(x_1,x_2,...,1,...,x_n)$

(2) $\Rightarrow [f(x_1,x_2,...,0,...,x_n) + x_i] \cdot [f(x_1,x_2,...,1,...,x_n) + x_i'] = f(x_1,x_2,...,1,...,x_n)$

En notant $f_{xi'}$ et f_{xi} les cofacteurs de f par rapport à 0 et 1 les relations (1) et (2) s'écrivent :

(1) $f = x_i'.f_{xi'} + x_i.f_{xi}$

(2) $f = [f_{xi'} + x_i] \cdot [f_{xi} + x_i']$

Exemple d'utilisation du théorème de Shannon :

$f(a,b,c) = a.b + a'.b.c + b'.c$

$f_a = b + b'.c$

$f_{a'} = b.c + b'.c$

(1) $\Rightarrow f(a,b,c) = a.(b + b'.c) + a'.(b.c + b'.c)$

(2) $\Rightarrow f(a,b,c) = [b.c + b'.c + a].[b + b'.c + a']$

Corollaire : Si une fonction est monoforme par rapport à une variable x alors le théorème de Shannon peut s'exprimer de la manière suivante :

Monoforme / x $\Rightarrow f = x \cdot f_x + f_{x'}$

$\Rightarrow f = (x + f_{x'}) \cdot f_x$

Monoforme / x' $\Rightarrow f = f_x + x'. f_{x'}$

$\Rightarrow f = f_{x'} \cdot (x' + f_x)$

Corollaire : Soit f et g deux fonctions logiques de variables x_i (i=1, ..., n) et Θ une fonction arbitraire.

$f \Theta g = x \cdot (f_x \Theta g_x) + x' \cdot (f_{x'} \Theta g_{x'})$

Corollaire : Soit f une fonction logique de variables x_i (i=1, ..., n)

$$f = x'.f_{x'} \oplus x.f_x$$

Corollaire : Soit f une fonction logique de variables x_i (i=1, ..., n)

$$f = 1 \Leftrightarrow f_x = 1 \text{ et } f_{x'} = 1$$

2.4.2. Formes de Lagrange

Soit une fonction logique de 2 variables f(x,y). Appliquons le théorème de Shannon par rapport à x puis par rapport à y.

$$\begin{aligned}(1) \Rightarrow f(x,y) &= x'.f(0,y) + x.f(1,y)\\ &= x'.[y'.f(0,0) + y.f(0,1)] + x.[y'.f(1,0) + y.f(1,1)]\\ &= x'.y'.f(0,0) + x'.y.f(0,1) + x.y'.f(1,0) + x.y.f(1,1)\end{aligned}$$

$$\begin{aligned}(2) \Rightarrow f(x,y) &= [x'+ f(1,y)].[x + f(0,y)]\\ &= \{x'+[y'+ f(1,1)].[y + f(1,0)]\}.\{x + [y' + f(0,1)].[y + f(0,0)]\}\\ &= [x'+ y'+ f(1,1)].[x' + y + f(1,0)].[x + y' + f(0,1)].[x + y + f(0,0)]\end{aligned}$$

En itérant le processus précédant sur une fonction de n variables les relations précédentes se généralisent de la manière suivante :

$$\begin{aligned}(1)\ f(x_1,x_2,...,x_n) &= [f(0,0,...,0)x_1'.x_2'...x_n'\ ] + [f(1,0,...,0)x_1.x_2'...x_n'] + ... +\\ &\quad [f(1,1,...,1)x_1.x_2...x_n]\end{aligned}$$

$$\begin{aligned}(2)\ f(x_1,x_2,...,x_n) &= [f(0,0,...,0) + x_1+x_2+...+x_n]\ .\ [f(1,0,...,0) + x_1'+x_2+...+x_n]\ .\ ...\ .\\ &\quad [f(1,1,...,1) + x_1' +x_2'+...+x_n']\end{aligned}$$

Ces deux expressions d'une fonction sont uniques. Elles sont connues sous les noms de « première et deuxième forme canonique » ou « première et deuxième forme de Lagrange ».

La première forme canonique est une forme ∑Π. Chaque intersection contient les n variables. Ces intersections sont appelées « mintermes ».

La deuxième forme canonique est une forme Π∑. Chaque réunion contient les n variables. Ces réunions sont appelées « maxtermes ».

Lorsqu'une fonction est donnée par l'une des représentations tabulaires ou numériques ses formes canoniques s'obtiennent directement par application du théorème d'expansion de Shannon.

- $1^{ère}$ forme canonique : Somme des mintermes vrais de la fonction
- $2^{ème}$ forme canonique : Produit des maxtermes complémentaires des points faux de la fonction.

Exemple : Les 2 formes canoniques de la fonction représentées sur table de vérité de la figure 2.12 sont les suivantes :

abc	f
000	0
001	1
010	1
011	0
100	1
101	0
110	1
111	1

f(a,b,c) = a'.b'.c + a'.b.c' + a.b'.c' + a.b.c' + a.b.c
f(a,b,c) = (a+b+c).(a+b'+c').(a'+b+c')

Figure 2.12. *1ère et 2ème formes canoniques*

La deuxième forme canonique d'une fonction f peut ainsi être obtenue en appliquant directement la relation f(X) = D[f'(X')]. Elle peut également être obtenue en complémentant la 1ère forme canonique de la fonction f' et en appliquant le théorème de De Morgan.

Exemple : Reprenons l'exemple précédent

f'(a,b,c) = a'.b'.c' + a'.b.c + a.b'.c

f(a,b,c) = (a'.b'.c' + a'.b.c + a.b'.c)'

f(a,b,c) = (a'.b'.c')'.(a'.b.c)'.(a.b'.c)'

f(a,b,c) = (a+b+c).(a+b'+c').(a'+b+c')

Les formes canoniques d'une fonction étant uniques, pour démontrer que deux fonctions sont identiques, il suffit de déterminer une de leur formes canoniques et d'en montrer l'identité.

Exemple : f1(a,b,c) = a.b + a'.b.c

f2(a,b,c) = b.c + a.b.c'

1ère forme canonique de f1(a,b,c) : a.b.c + a.b.c' + a'.b.c

1ère forme canonique de f2(a,b,c) : a.b.c + a'.b.c + a.b.c'

=> f1(a,b,c) = f2(a,b,c)

2.4.3. Formes de Reed-Muller

La ***notation galoisienne d'une fonction*** est une écriture de la fonction qui utilise uniquement les opérateurs OU exclusif et ET. Par exemple xy ⊕ xz et 1 ⊕ xy ⊕ xz sont des formes galoisiennes.

Rappel : pour écrire une fonction logique sous la forme galoisienne on peut utiliser les égalités suivantes:

$$x' = 1 \oplus x \qquad x + y = x \oplus y \oplus xy$$

Définition : La forme de Reed-Muller est la forme canonique de la fonction dans l'algèbre de Galois.

Cette forme de Reed-Muller peut être obtenue à partir de la 1ère forme canonique de la fonction transformée dans le champs de Galois.

Pour une fonction de deux variables, le processus est le suivant :

f(x,y) = x.f(1,y) + x'.f (0,y)

= x.f(1,y) ⊕ x'.f(0,y)

= x.[y.f (1,1) ⊕ y'.f (1,0)] ⊕ x'.[y.f (0,1) ⊕ y'.f (0,0)]

= x.y.f (1,1) ⊕ x.y'.f (1,0)] ⊕ x'.y.f (0,1) ⊕ x'.y'.f (0,0)]

= ..... en remplaçant x' par 1 ⊕ x et y' par 1 ⊕ y
= f(0,0) ⊕ x.[f(0,0)⊕f(1,0)] ⊕ y.[f(0,0) ⊕ f(0,1)] ⊕
x.y[f(0,0)⊕f(0,1)⊕f(1,0)⊕f(1,1)]

En généralisant ce développement à une fonction de n variables, la forme de Reed-Muller peut être obtenue à partir de la relation suivante :

$$f(x_1,x_2,...,x_n) = g_0 \oplus x_1\, g_1 \oplus x_2\, g_2 \ \oplus .. \oplus x_1.x_{2\ gn+1} \oplus \ .... \oplus x_1.x_2 \ldots x_n \ g_{2n-1}$$

avec g_i = somme exclusive des valeurs de f correspondant à toutes les lignes de la table de vérité pour lesquelles toutes les variables non impliquées ont la valeur 0

Exemple: La figure 2.13 présente le processus d'obtention de la forme de Reed Muller d'une fonction f de 3 variables a,b,c.

a b c	f
000	0
001	1
010	1
011	1
100	1
101	0
110	0
111	0

terme constant : $g_0 = f_0 = 0$ (a et b et c sont nuls)
en a : g_1 = somme des f_i pour lesquelles b et c sont nuls = 0 ⊕ 1= 1
en b : g_2 = somme des f_i pour lesquelles a et c sont nuls = 0 ⊕ 1= 1
en c : g_3 = somme des f_i pour lesquelles a et b sont nuls = 0 ⊕ 1= 1
en a.b : g_4 = somme des f_i pour lesquelles c est nul = 0 ⊕ 1 ⊕ 1 ⊕ 0 = 0
en a.c : g_5 = somme des f_i pour lesquelles b est nul = 0 ⊕ 1 ⊕ 1 ⊕ 0 = 0
en b.c : g_6 = somme des fi pour lesquelles a est nul = 0 ⊕ 1 ⊕ 1 ⊕ 1 = 1
en a.b.c : g_7 = somme des fi = 0 ⊕ 1⊕ 1⊕ 1⊕ 1⊕ 0⊕ 0⊕ 0= 0

Donc f(a,b,c) = a ⊕ b ⊕ c ⊕ b c : forme canonique de Reed-Muller

Figure 2.13. *Obtention de la forme de Reed Muller*

Notons que la forme de Reed-Muller peut également être obtenue simplement en exprimant la fonction dans le champ de Galois sous forme $\sum_{\oplus}\prod$ puis en faisant toutes les simplifications de type x ⊕ x = 0

Exemple : Soit f (a,b,c) = a' b' c + a' b c' + a' b c + a b' c'
= a' b' c ⊕ a' b c' ⊕ a' b c ⊕ a b' c'
= a' (b ⊕ c ⊕ b c) ⊕ a b' c'
= (1 ⊕ a) (b ⊕ c ⊕ b c) ⊕ a (1 ⊕ b) (1 ⊕ c)
= b ⊕ c ⊕ b c ⊕ ab ⊕ ac ⊕ ab c ⊕ a ⊕ ab ⊕ ac ⊕ ab c
= a ⊕ b ⊕ c ⊕ b c

2.4.4. Formes de Davio

Les formes de Davio (forme positive et forme négative) sont des écritures de la fonction dans le champs de Galois. Cet formes sont particulièrement intéressantes pour certaines applications. Elles sont obtenues à partir des théorèmes suivants :

Théorème : Soit $f(x_1,x_2,...,x_i,...,x_n)$ une fonction logique de n variables

(1) $f = f_{x'} \oplus x.(f_x \oplus f_{x'})$ (forme de Davio positive)
(2) $f = f_x \oplus x'.(f_x \oplus f_{x'})$ (forme de Davio négative)

Preuve : Soit $f(x_1,x_2,...,x_i,...,x_n)$ une fonction logique de n variables. F peut s'écrire sous la forme suivantes :

$f = x'.f_{x'} \oplus x.f_x$

(1) En appliquant le théorème de l'algèbre de Boole suivant : $a \oplus ab = ab'$

$f = (f_{x'} \oplus x.f_{x'}) \oplus x.f_x$

$f = f_{x'} \oplus x.(f_x \oplus f_{x'})$

(2) En appliquant le même théorème de l'algèbre de Boole sur le deuxième terme

$f = x'.f_{x'} \oplus (f_x \oplus x'.f_x)$

$f = f_x \oplus x'.(f_x \oplus f_{x'})$

2.5. Théorème d'inclusion

Savoir si un monôme est inclus dans l'expression de la fonction, c'est à dire couvert par une fonction, peut être un problème complexe. Ce problème peut en fait être réduit à un problème de vérification de tautologie en utilisant le théorème d'inclusion.

Théorème : Une expression F contient un monôme m si et seulement si le cofacteur de F par rapport à m (F_m) est une tautologie.

$$m \subset F \Leftrightarrow F_m = 1$$

Preuve : Remarquons que $m \subseteq F <=> F \,.\, m = m$

Supposons que $m \subseteq F$

$=> F \,.\, m = m$

Considérons les cofacteurs par rapport à m des 2 termes. Les termes étant égaux, les cofacteurs sont égaux.

$=> (F \,.\, m)_m = m_m$

$m_m = 1 => (F \,.\, m)_m = 1$

$=> F_m \,.\, m_m = 1$

$=> F_m = 1$

$=> F_m$ est une tautologie

Réciproquement, supposons que F_m soit une tautologie

$=> F_m = 1$

$=> F_m \,.\, m = m$

$=> F \,.\, m = [\,(m \,.\, F_m) + (m' \,.\, F_{m'})\,] \,.\, m = F_m \,.\, m = m$

$=> m \subseteq F$

Exemple : Soit la fonction $f(a,b,c) = a.b + a.c + a'$

Cherchons à savoir si le monôme "bc" est couvert par cette expression.

Le cofacteur de f par rapport à bc est : $f_{bc} = a + a' = 1$

Le monôme bc est donc inclus dans f comme le montre la figure 2.14

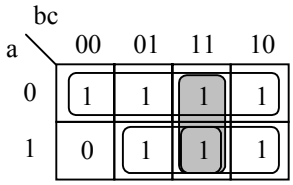


Figure 2.14. *Inclusion d'un monôme dans une fonction*

2.6. Equations logiques

Une équation logique est une égalité de la forme E1(X)=E2(X), ou E1 et E2 sont des expressions logiques de n variables $X=(x_1,...,x_n)$ et pour lesquelles on se propose de trouver les combinaisons de X qui satisfont cette égalité.

L'équation E1=E2 est équivalente à :

E1⊕ E2=0 c'est à dire E1.E2' + E1'.E2 = 0

(E1⊕ E2)'=1 c'est à dire E1.E2 + E1'.E2' = 1

On peut donc toujours mettre une équation logique sous des formes E(X)=0 ou E(X)=1.

2.6.1. Résolution explicite

Une première méthode de résolution consiste à écrire l'expression E(X) sous la forme d'une somme de produit.

$$E(X) = \sum m_i$$

On obtient alors toutes les solutions de l'équation sous la forme de familles de points définies par m_i.

Si E(X)=(E1⊕ E2)' les solutions sont directement données par mi=1

Exemple : E1(a,b,c,d) = a.b + d E2(a,b,c,d) = a.b.c

E1 = E2 => (E1⊕ E2)'=1

=>E1.E2 + E1'.E2' = 1

=> (a.b + d)(a'.b'.c) + (a' + b').d'.(a + b + c') = 1

=> a'.b'.c.d + (a'.d' + b'.d')(a + b + c') = 1

=> a'.b'.c.d + a'.b.d' + a'.c'.d' + a.b'.d' + b'.c'.d' =1

Les solutions sont donc les combinaisons qui vérifient l'une quelconque des égalités suivantes :

a'.b'.c.d = 1

a'.b.d' = 1

a'.c'.d' = 1

a.b'.d' = 1

b'.c'.d' = 1

La liste des solutions de l'équation est donc :

abcd = (0011), (01ϕ0), (0ϕ00), (10ϕ0), (ϕ000)

Pour montrer que deux expressions logiques sont identiques il vaut mieux utiliser la forme E(X)=E1⊕ E2. Si E(X)=0 quelles que soient les valeurs de X, les expressions E1 et E2 sont identiques.

Exemple : E1(X) = a.b + b'.c + a'.c'

E2(X) = a'.b' + b.c' + a.c

E1⊕ E2 = E1.E2' + E1'.E2

= (a.b + b'.c + a'.c').(a + b)(b' + c)(a' + c') +
(a' + b')(b + c').(a + c).(a'.b' + b.c' + a.c)

= (a.b + b'.c + a'.c').(a.b' + a.c + b.c).(a' + c') +
(a'.b + a'.c' + b'.c')(a + c).(a'.b' + b.c' + a.c)

= (a.b + b'.c + a'.c').(a.b'.c' + a'.b.c) +
(a'.b.c + a.b'.c').(a'.b' + b.c' + a.c)

= 0

=> E1=E2

On peut également montrer que deux fonctions sont identiques en établissant la forme canonique de chacune d'elles (la forme canonique est unique).

2.6.2. Résolution paramétrique

Une autre méthode de résolution consiste à éliminer successivement chacune des variables. Elle présente l'avantage d'aboutir à une écriture paramétrique de l'ensemble des solutions.

En effet, en vertu du théorème de Shannon , l'expression E(X)=1 peut se mettre sous la forme :

$E_1 = E(x_1,x_2,...,x_{n-1},1)x_n + E(x_1,x_2,...,x_{n-1},0)x_n' = 1$

La condition nécessaire pour que cette équation ait une solution en xn est que :

$E(x_1,x_2,...,x_{n-1},1) + E(x_1,x_2,...,x_{n-1},0) = 1$

On à donc une équation de n-1 variables. Si l'on poursuit la procédure, on aboutit à une équation d'une seule variable :

$E_n = a.x_1 + b.x_1' = 1$

Si a+b=1 (condition nécessaire pour que cette équation ait une solution en x_1), on montre que la solution générale de cette équation est :

$x_1 = b' + a.u$

ou u est une quantité logique pouvant prendre les valeurs 0 et 1. En reportant dans l'équation $E_{n-1}=1$, on peut alors résoudre en x_2 etc, jusqu'à obtenir la solution, si elle existe, de l'équation E=1.

Exemple : Soit à résoudre l'équation logique suivante par rapport aux variables x et y :

a.x'.y' + x.y = b

Mettons cette équation sous le forme E(X)=1.

a.b.x'.y' + b.x.y + b'(a' + x + y)(x' + y') = 1

a.b.x'.y' + b.x.y + b'(a'.x' + a'.y' + x.y' + x'.y) = 1

a.b.x'.y' + b.x.y + a'.b'.x' + a'.b'.y' + x.b'.y' + x'.b'.y = 1

Mettons cette équation sous le forme x.E + x'.E = 1.

x.(b.y + b'.y') + a'.b'.y' + x'.(a.b.y' + a'.b' + b'.y) = 1

x.(b.y + b'.y' + a'.b'.y') + x'.(a.b.y' + a'.b' + b'.y + a'.b'.y') = 1
x.(b.y + b'.y') + x'.(a.b.y' + a'.b' + b'.y) = 1 (1)

La condition d'existence d'une solution en x s'écrit donc :
(b.y + b'.y') + (a.b.y' + a'.b' + b'.y) = 1

Mettons cette équation sous le forme y.E + y'.E = 1.
y.(b + b' + a'.b') + y'.(a.b + b' + a'.b') = 1
y + y'.(a.b + b') = 1
y + y'.(a + b') = 1

La condition d'existence d'une solution en x s'écrit donc :
1 + (a + b') = 1

Cette condition est toujours remplie. La solution en y s'écrit :
y = a'.b + u

En reportant dans (1) on obtient :
x.[b.(a'b+u) + b'.(a'b+u)] + x'.[a.b.(a'b+u) + a'.b' + b'.(a'b+u)] = 1
x.[b.(a'b+u) + b'.(a+b')u')] + x'.[a.b.(a+b')u' + a'.b' + b'.(a'b+u)] = 1
x.[a'.b + u.b + u'.b'] + x'.[a.b.u' + a'.b' + u.b'] = 1

D'où la solution :
x = [a.b.u' + a'.b' + u.b'] + [a'.b + u.b + u'.b'].v = 1
x = [(a'+b'+u)(a+b)(u'+b)] + [a'.b + u.b + u'.b'].v = 1
x = [(a'b+ab'+au+bu)(u'+b)] + [a'.b + u.b + u'.b'].v = 1
x = (a'.b + a.b'.u' + b.u) + (a'.b + u.b + u'.b').v = 1

On aboutit donc à quatre expressions de x et y en fonction de a et b en remplaçant les paramètre u et v par les quatre combinaisons possibles.

uv=00 => x = a'.b + a.b'	y = a'.b
uv=01 => x = a'.b + b'	y = a'.b
uv=10 => x = b	y = 1
uv=11 => x = b	y = 1

2.6.3. Résolution de systèmes d'équation

Un système d'équations logiques se présente sous la forme de p équations de n variables X. Chacune d'elles est équivalente à une relation de la forme $E_i(X)=1$ pour i=1,2,...,p. On ramène se système à une seule équation logique qui est :

$$\sum_{i=1}^{i=p} E_i(X) = 1$$

Les procédures de résolution présentées précédemment sont alors applicables à cette équation.

Chapitre 3

Graphes de décision binaires

Pour répondre au problème posé par l'accroissement de la taille des fonctions logiques à traiter (grand nombre de fonctions, de monômes et de variables), de nouvelles techniques de représentation des fonctions booléennes ont été développées. Elles sont généralement basées sur la récursivité du théorème de Shannon et non plus sur les théorèmes habituels du calcul booléen. Les Graphes de Décision Binaires (BDD pour Binary Decision Diagram) sont des modèles de représentation particulièrement efficaces pour représenter et manipuler de telles fonctions logiques. Ils ont été introduits par Akers en 1978. En 1986, Bryant propose les OBDD (Ordered Binary Decision Diagram), une représentation canonique des BDD ainsi que des algorithmes pour calculer efficacement les opérations booléennes sur ces structures de données.

3.1. Arbres de décision binaire

Les arbres de décision binaires constituent un modèle de représentation des fonctions booléennes. Un arbre de décision binaire est un arbre orienté composé d'une racine, de sommets intermédiaires et de sommets terminaux valant 0 ou 1. La racine et les sommets intermédiaires sont indexés et possèdent deux sommets "fils", un fils gauche et un fils droit. Le fils gauche est atteint en empruntant la branche "0", le fils droit en empruntant la branche 1. Un arbre de décision binaire est obtenu en appliquant récursivement la première forme du théorème de Shannon sur l'ensemble des variables de la fonction.

Théorème de Shannon (rappel):

$$f(x_1,x_2,...,x_i,...,x_n) = x_i' \cdot f(x_1,x_2,...,0,...,x_n) + x_i \cdot f(x_1,x_2,...,1,...,x_n)$$

Si ce théorème est appliqué à f pour x_1, puis aux deux sous-fonctions obtenues pour x_2, et ainsi de suite jusqu'à x_n, on peut réaliser un arbre de décision binaire.

Exemple : f(a,b,c) = a'.b.c + a.c

f(a,b,c) = a'.f(0,b.c) + a.f(1,b,c)
f(0,b,c) = b.c
f(1,b,c) = c

f(0,b,c) = b'.f(0,0.c) + b.f(0,1,c)
f(0,0,c) = 0
f(0,1,c) = c
f(0,0,c) = c'.f(0,0,0) + c.f(0,0,1)
f(0,0,0) = 0
f(0,0,1) = 0
f(0,1,c) = c.'f(0,1,0) + c.f(0,1,1)
f(0,1,0) = 0

f(0,1,1) = 1
f(1,b,c) = b'.f(1,0.c) + b.f(1,1,c)
f(1,0,c) = c
f(1,1,c) = c
f(1,0,c) = c'.f(1,0,0) + c.f(1,0,1)
f(1,0,0) = 0
f(1,0,1) = 1
f(1,1,c) = c.'f(1,1,0) + c.f(1,1,1)
f(1,1,0) = 0
f(1,1,1) = 1

Ces relations peuvent être exprimées sous forme d'arbre de décision binaire de la façon suivante :

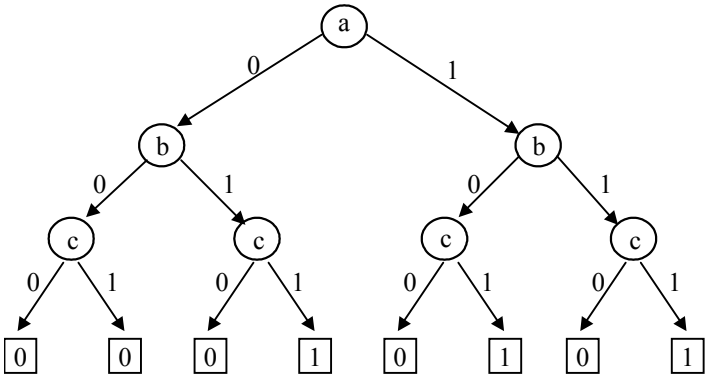


Figure 3.1. *Arbre de décision binaire associé à la fonction f = a'bc + ac*

Notons qu'il est possible de réduire la taille de l'arbre en s'arrêtant dans l'application du théorème de Shannon dès que l'on se trouve en présence d'une constante 0 ou 1. Ainsi, dans l'exemple précédent nous voyons que f(0,0,c)=0, c'est à dire que f=0 quelle que soit la valeur qui sera assignée à c. En s'arrêtant à ce niveau dans l'application du théorème de Shannon, l'arbre de décision binaire associé à la fonction f devient:

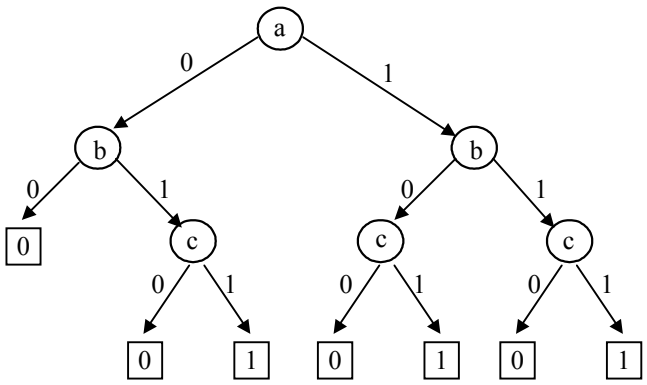


Figure 3.2. *Arbre de décision binaire simplifié associé à la fonction f = a'bc + ac*

3.2. Diagrammes de décision binaires (BDD)

Après sa construction, un arbre de décision binaire peut être réduit par transformation en un graphe acyclique orienté appelé graphe de décision binaire (BDD). En effet, nous avons pu remarquer qu'un arbre de décision

binaire peut posséder des feuilles, voir des sous-graphes identiques. La représentation peut être simplifiée en ne conservant qu'une seule représentation des feuilles et sous-graphes concernés, et en remplaçant les autres feuilles et sous-graphes par un arc vers le premiers. Le graphe de décision binaire ainsi obtenu peut également être réduit par élimination des sommets redondants, c'est à dire des sommets ayant un fils gauche et un fils droit identique.

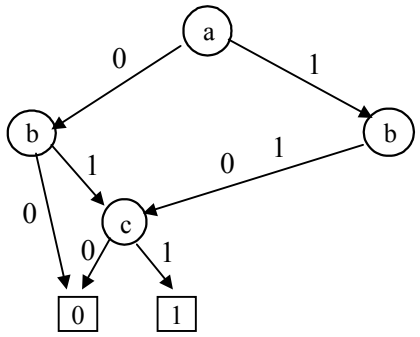


Figure 3.3. *Réduction par élimination des feuilles et sous-graphes identiques*

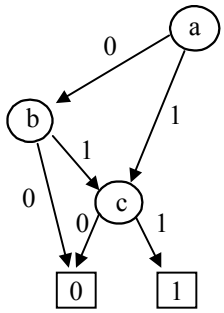


Figure 3.4. *Réduction par élimination sommets redondants*

En cherchant et remplaçant tous les sous-arbres équivalents d'un diagramme de décision binaire, on obtient le diagramme de décision binaire minimal pour l'ordre correspondant des variables (ROBDD pour Reduced Ordered BDD). Il est important de noter que pour un ordre donné de variables, le graphe de décision binaire minimal est **unique**. Cette unicité peut évidemment être utilisée pour montrer l'équivalence de deux expressions logiques.

Remarques :

- La complexité de la procédure de réduction d'un BDD est O(nlog(n)).
- La structure du BDD (et en particulier sa taille) dépend de l'ordre dans lequel sont considérées les variables. Ainsi, pour deux ordres différents, les BDD peuvent être de tailles très différentes.

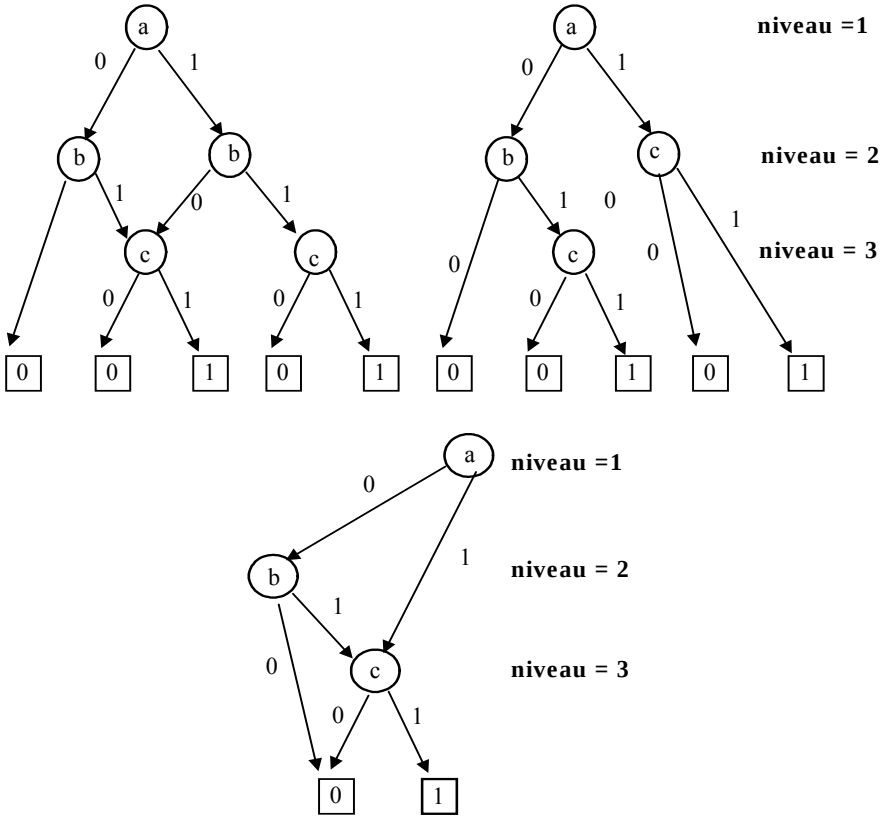


Figure 3.5. *OBDDs et ROBDD de la fonction f = (a+b).c*

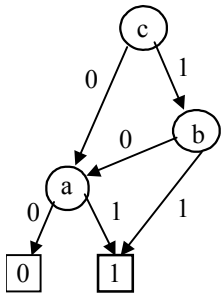


Figure 3.6. *ROBDD de la fonction f = (a+b).c sous l'ordre c, b, a.*

3.3. Construction des BDDs

Afin de garantir la canonicité de la représentation , les contraintes suivantes sont imposées :

- chaque variable ne peut apparaître qu'une fois au plus sur chaque chemin entre la racine et une feuille
- les variables sont ordonnées de telle façon que si un sommet de label x_i a un fils de label x_j alors ord(x_i) < ord (x_j)

Il y a 2 stratégies principales de construction des BDDs :

- de la racine vers les feuilles (top-down). Cette méthode est utilisée lorsque on part d'une formule algébrique
- des feuilles vers la racine (bottom-upp) lorsque l'on part d'une description structurelle du circuit.

3.3.1. Méthode top-down

On utilise la formule de Shannon.

Exemple : soit f(a,b,c)=a'bc' + ac avec ord(a)<ord(b)<ord(c)

- Si l'on fait l'expansion par rapport à a on obtient la Fig.3.7.a
- En faisant l'expansion par rapport à b on obtient la Fig.3.7.b
- En faisant l'expansion par rapport à c on obtient la Fig.3.7.c

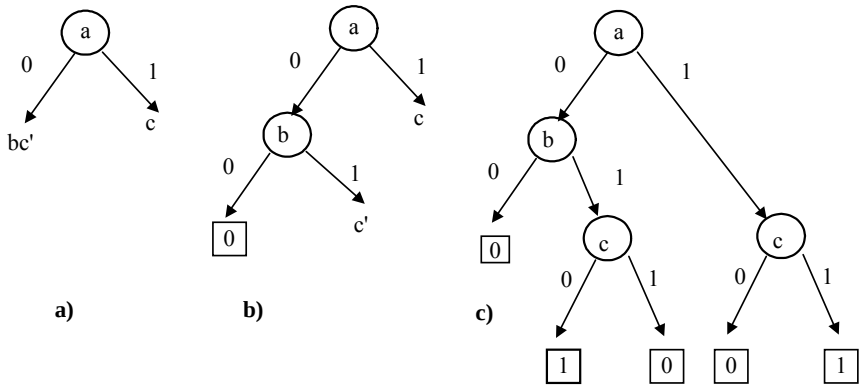


Figure 3.7. *Construction du BDD de la fonction f = a'bc' + ac*

3.3.2. Méthode bottom-up

Le ***niveau*** d'une formule est défini par :

- les variables d'entrée sont de niveau 0
- chaque sous-formule f=g<Op>h a un niveau égal à Max (niveau(g), niveau(h)) + 1

Méthode pour construire le BDD d'une fonction f de n variables :

1. construire les BDD des variables
2. construire les BDDs des formules de niveau 1 à l'aide de la fonction : appliquer (f1 : BDD, f2 : BDD, opérateur <Op>) : BDD
3. répéter l'étape 2 jusqu'à ce que tous les niveaux soient considérés.
   Par exemple si f = xy + z : on construit les BDD de x , y et z puis celui de x.y puis celui de x.y + z.

3.4. Opérations logiques entre deux fonctions représentées par BDD

Les opérations logiques applicables sur les BDD on été définies par Bryant. Ces opérations sont les opérations de complémentation, de test d'implication, de ET logique, de OU logique, et de OU Exclusif. L'algorithme permettant de déterminer le BDD résultant d'une opération logique entre deux fonctions est basé sur l'application récursive du théorème de Shannon.

Etant donné deux fonctions f et g, un opérateur Op et un ordre des variables on a:

$f<Op>g = x_i.(f/x_i=0 <Op> g/x_i=0) + x_i.(f/x_i=1 <Op> g/x_i=1)$

Le BDD résultant d'une opération logique entre deux fonctions peut être conçu à partir des deux BDD originaux en appliquant la procédure suivante :

Considérer les sommets racines des BDD. En fonction de la nature de ces deux sommets, appliquer récursivement l'une des règles suivantes. Itérer le processus jusqu'à la génération des sommets terminaux.

R1 : Si les deux sommets Sf et Sg sont des sommets terminaux alors le sommet résultant est un sommet terminal de valeur Valeur(Sf) <Op> Valeur(Sg).

R2 : Si le sommet Sf est un sommet terminal mais pas le sommet Sg, créer le sommet Sg en lui associant comme fils droit le résultat de la comparaison (Sg, fils droit de Sf) et comme fils gauche le résultat de la comparaison (Sg, fils gauche de Sf).

R3 : Si ni le sommet Sf ni le sommet Sg ne sont des sommet terminaux alors :

R3.1 : Si Sf et Sg représentent la même variable, créer un sommet représentant la variable en lui associant comme fils droit le résultat de la comparaison (fils droit de Sg, fils droit de Sf) et comme fils gauche le résultat de la comparaison (fils gauche de Sg, fils gauche de Sf).

R3.2 : Si Sf et Sg ne représentent pas la même variable, créer un sommet S représentant la variable intervenant la première dans l'ordonnancement considéré (S=min[ord(Sf), ord(Sg)] en lui associant comme fils droit (gauche) le résultat de la comparaison entre le fils droit (gauche) de min[ord(Sf), ord(Sg)] et l'autre sommet (max[ord(Sf), ord(Sg)]).

Remarque : Etant donné deux fonctions f et g dont les BDD respectifs ont n et m sommets. En termes d'appels récursif, la complexité de la procédure d'application d'une opération entre ces deux BDD est 2.n.m.

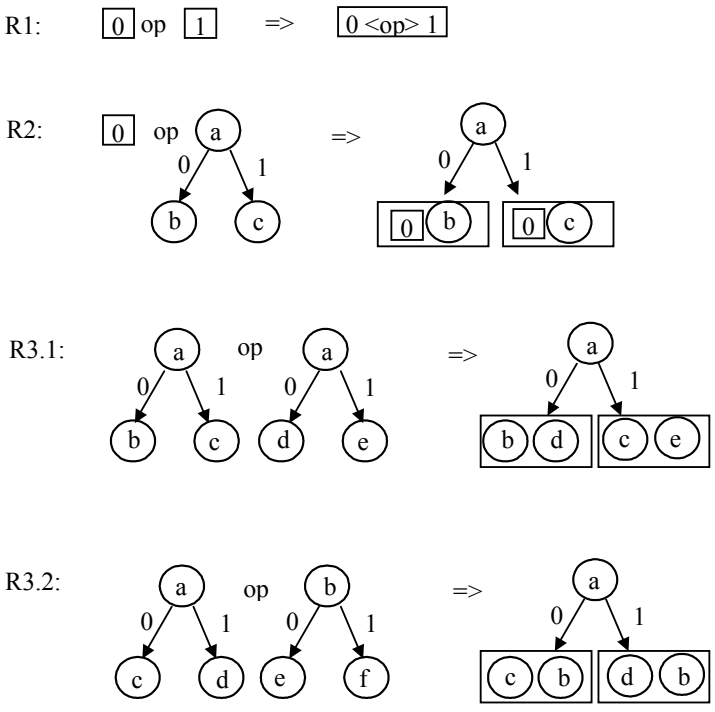


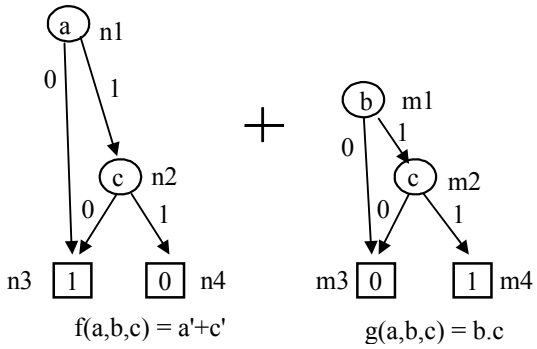
Figure 3.8. *Illustration des règles de combinaisons de BDD*

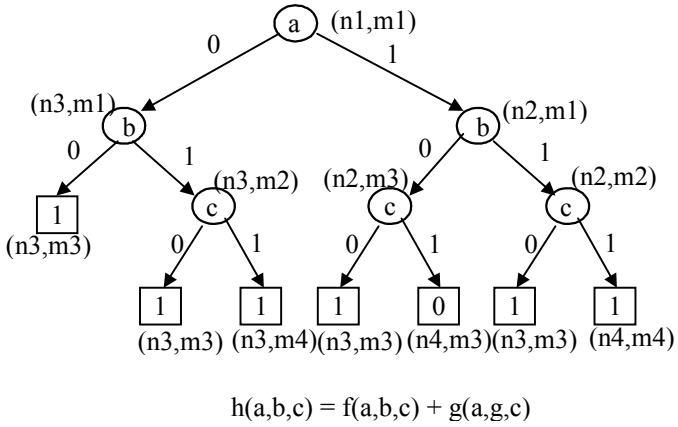
Remarque : Etant donné deux fonctions f et g dont les BDD respectifs ont n et m sommets. En termes d'appels récursifs, la complexité de la procédure d'application d'une opération entre ces deux BDD est 2.n.m.

Remarque **:** Cette méthode permet (outre les opérations) :

- de construire un BDD de façon ascendante,
- de calculer l'inverse d'une fonction en faisant "f ⊕ 1"
- de calculer une implication f1 => f2 <=> not (f1 . not(f2))

Exemple : Soit les fonctions f(a,b,c) = a'+c' et g(a,b,c) = b.c représentées par leurs BDD. Construire le BDD de la fonction h(a,b,c) = f(a,b,c) + g(a,b,c).





h(a,b,c) = f(a,b,c) + g(a,g,c)

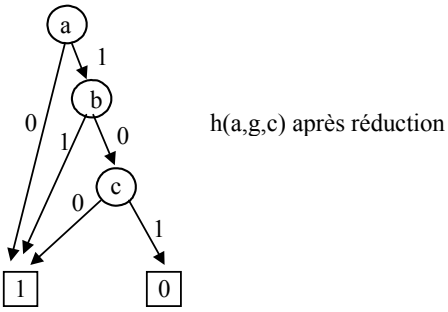


Figure 3.9. *Opération entre deux BDD*

3.5 Implantation informatique des BDD

La structure informatique permettant de représenter un BDD peut être un tableau tel que chaque ligne comporte 3 champs : un champs « Identificateur » permettant d'identifier le sommet dans le BDD, un champs Arc_0 donnant l'adresse du fils gauche du sommet en question et un champs Arc_1 donnant l'adresse du fils droit.

Identificateur	Arc_0	Arc_1
0	-	-
1	-	-
Variable sommet i	Adresse sommet gauche	Adresse sommet droit

Exemple : soit une fonction f(a,b,c) = a' + c' + b.c

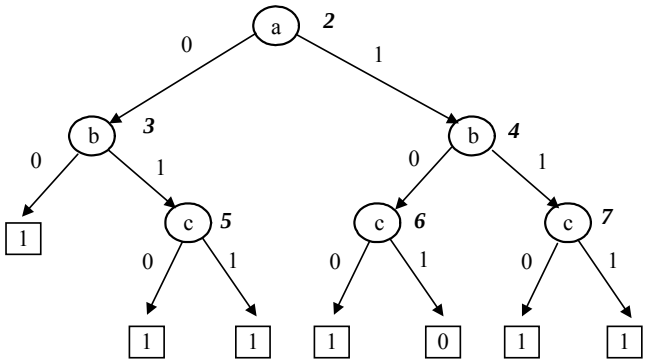


Figure 3.10. *BDD de la fonction f = a' + c' + b.c*

Le BDD de la fonction f peut se représenter informatiquement par le tableau suivant :

0	0	-	-
1	1	-	-
2	a	3	4
3	b	1	5
4	b	6	7
5	c	1	1
6	c	0	1
7	c	1	1

3.6. Algorithme de réduction des BDD

L'algorithme de réduction d'un BDD a pour objectif de supprimer les sommets redondants et les sous-graphes isomorphes. Le principe de algorithme est le suivant :

Faire id(v) = 0 pour chaque feuille v avec valeur (v) = 0 ;

Faire id(v) = 1 pour chaque feuille v avec valeur (v) = 1 ;
Initialiser le ROBDD avec 2 feuilles avec id =0 et id = 1 ;
Nextid = 1; /* Nextid = identificateur suivant disponible */
Pour (i = n to 1 avec i=i-1) {
V(i) = { v ∈ V / niveau(v) = i };
Pour chaque v ∈ V(i) {
if (id(gauche(v)) = id (droit(v)) { /* sommet redondant */
id(v) = id(gauche(v))
enlever v de V(i)}
else
cleF(v) = id (gauche(v)) , id (droit(v)) } /* la clef est définie comme la paire d'identificateurs des fils*/
ancienne_clef = 0 , 0;
Pour chaque v ∈ V(i) trié selon la clef {
if cleF(v) = ancienne_clef
id(v) = nextid /* le graphe de racine v est redondant */
else {
Nextid++;
id(v)=Nextid;
ancienne_clef = clef (v);
ajouter v au ROBDD avec des arcs allant aux sommets dont l'id est égale à ceux de gauche(v) et droit(v)) }}}

La figure 3.11 donne un exemple d'application de cet algorithme de réduction d'un BDD.

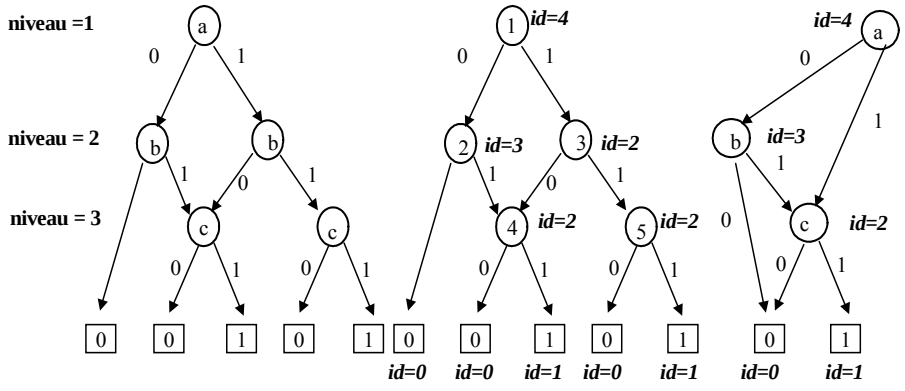


Figure 3.11. *Construction du ROBDD de la fonction f = (a+b)c*

3.7. Applications directes des BDD

3.7.1. Evaluation d'une fonction représentée sous forme de BDD

Etant donnée une assignation des variables d'entrées d'une fonction f, la valeur de la fonction f peut être trouvée en traversant le BDD depuis la racine jusqu'à une feuille en empruntant la branche gauche ou droite de

chaque sommet selon la valeur de la variable d'entrée correspondant. Dans le pire cas, le BDD correspondant à une fonction f de n variables est composé de 2^n-1 sommets. De plus, le nombre de branches entre la racine et une feuille est au plus égal à n. La complexité d'une procédure d'évaluation d'une fonction, c'est à dire de traversée d'un BDD depuis sa racine jusqu'à une feuille est O(n).

La représentation sous forme de BDD d'une fonction f permet de trouver rapidement une combinaison des entrées telles que f=0 ou telle que f=1. La procédure permettant de trouver une solution x de f(x) = a (a = 0 ou 1) est en O(n).

3.7.2. Equivalence de 2 fonctions

Puisque pour un ordre donné le ROBDD d'une fonction est unique, pour que 2 fonctions soit identiques, il suffit que leurs ROBBDs respectifs soient isomorphes.

3.7.3. Cofacteurs

Pour obtenir le cofacteur par rapport à x_i il suffit de supprimer le sommet x_i et de lier les sommets pointant sur x_i au fils droit de x_i.

Pour obtenir le cofacteur par rapport à x_i il suffit de supprimer le sommet x_i' et de lier les sommets pointant sur x_i au fils gauche de x_i

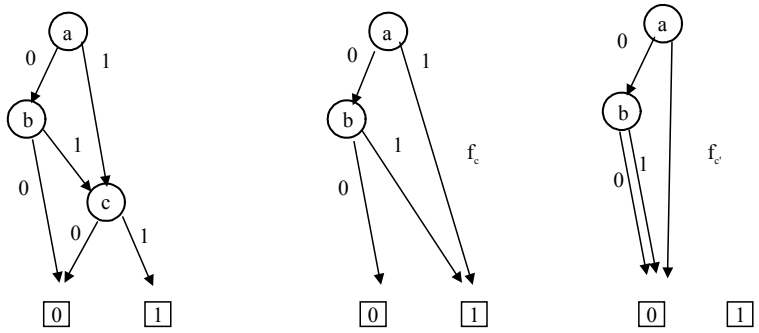


Figure 3.12. *BDD des cofacteurs*

3.7.4. Preuve de tautologie

Une fonction simple est une tautologie si elle est toujours égale à 1.

Démontrer qu'une fonction est une tautologie est relativement simple en élaborant l'arbre de décision binaire de la fonction. En effet, une condition nécessaire et suffisante pour qu'une fonction soit une tautologie est que tous les sommets terminaux (feuilles) de l'arbre de décision binaire vaillent 1 (aucun sommet terminal à 0).

Exemple : La fonction f(a,b,c) = a'.b.c + a.b.c + b.c' + b' est une tautologie car tous les sommets terminaux de l'arbre de décision binaire sont à 1.

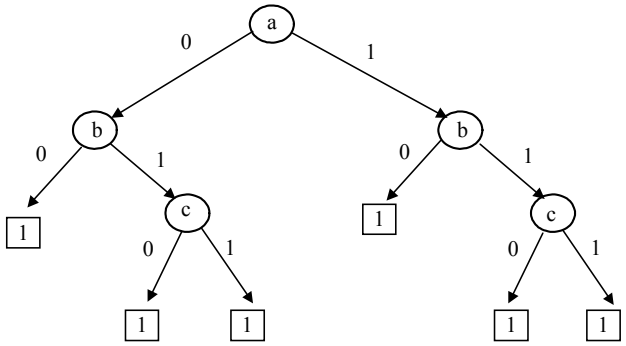


Figure 3.13. *Arbre de décision binaire de la fonction f = a.b.c + a.b.c + b.c + b*

Pour démontrer qu'une fonction est ou n'est pas une tautologie, on peut raisonner de la manière suivante:

Si l'un des cas suivants est détecté, il est possible d'arrêter le traitement, la fonction n'est pas une tautologie.

- Toutes les variables de F sont monoformes,
- Une variable monoforme est présente dans tous les monômes,
- La somme des tailles (dans le sens nombre de points couverts) des monômes est inférieure à 2n. (rappel : un monôme de p termes couvre 2(n-p) points)

Si aucune des conditions précédentes n'est remplie, réaliser l'arbre de décision binaire. Pour simplifier le traitement, on pourra appliquer le théorème suivant :

Théorème : Soit f une couverture monoforme par rapport à une variable x (f=x.g + h) , h le sous-ensemble de f ne dépendant pas de x. f est une tautologie si et seulement si h est une tautologie.

Preuve : Soit f monoforme en x: f=x.g + h

(1) : h = 1 => f = 1 donc h tautologie => f tautologie

(2) : f tautologie => f = 1 quelque soit la valeur de x

Si x=0, f = h => h =1

Ainsi, on pourra également considérer les variables de la manière suivante:

- Si des monômes sont composés d'une seule variable prendre ces variables en premier,
- Si une variable x apparaît toujours sous sa forme directe, ne pas continuer le graphe sur la branche 1 ($f = 1 <=> f_{x'} = 1$) puisque $f = x.f_x + f_{x'}$ avec $f_{x'}$ indépendant de x.
- Si une variable x apparaît toujours sous sa forme complémentée, ne pas continuer le graphe sur la branche 0 ($f = 1 <=> f_x = 1$) puisque $f = f_x + x'.f_{x'}$ avec f_x indépendant de x.

Exemple1 : Soit la fonction f(a,b,c) = a'.b.c + a.b.c + b.c' + b'.

- Il existe des variables biformes
- Aucune variable monoforme n'est présente dans tous les monômes
- La somme des tailles des monômes (1+1+2+4=8) n'est pas inférieure à 2^n.

L'arbre de décision binaire doit être réalisé. Un monôme est constitué d'une seule variable (b') => b sera pris en premier. Aucune variable n'apparaît uniquement sous sa forme normale ou uniquement sous sa forme complémentée.

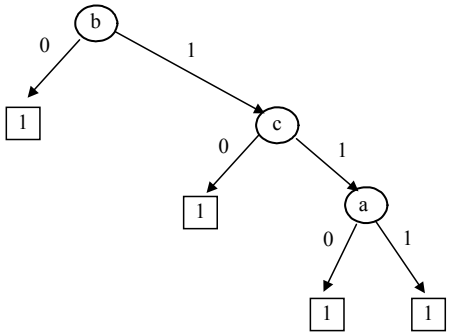


Figure 3.14. *Arbre de décision binaire de la fonction f = a'.b.c + a.b.c + b.c' + b'*

Exemple2 : Soit la fonction f(a,b,c) = a.b' + b.c + a.b.c' + a.c

- Il existe des variables biformes
- Aucune variable monoforme n'est présente dans tous les monômes
- La somme des taille des monômes (2+2+2+2=8) n'est pas inférieure à 2^n.

L'arbre de décision binaire doit être réalisé. La variable a est monoforme directe, ne pas développer suivant la branche 1.

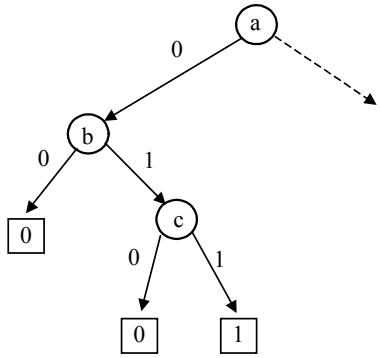


Figure 3.15. *Arbre de décision binaire la fonction f = a.b' + b.c + a.b.c' + a.c*

Exemple3 : Soit la fonction f(a,b,c) = a.b' + b' + b.c + b.c'

- Il existe des variables biformes.
- Aucune variable monoforme n'est présente dans tous les monômes.
- La somme des taille des monômes (1+1+2+4=8) n'est pas inférieure à 2^n.

L'arbre de décision binaire doit être réalisé. La variable a est monoforme directe, ne pas développer suivant la branche 1.

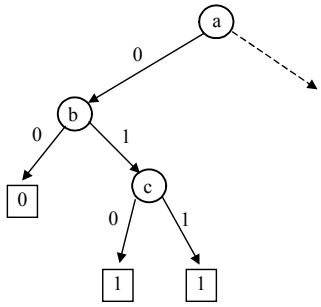


Figure 3.16. *Arbre de décision binaire de la fonction f = a.b' + b' + b.c + b.c'*

3.7.5. Test d'inclusion

Savoir si un monôme est inclus dans l'expression de la fonction, c'est à dire couvert par une fonction, peut être un problème complexe. Le théorème d'inclusion permet de ramener ce problème à une problème de preuve de tautologie sur des cofacteurs.

Rappel du théorème d'inclusion : Une expression F contient un monôme m si et seulement si le cofacteur de F par rapport à m (F_m) est une tautologie.

$$m \subset F \quad \Leftrightarrow \quad F_m = 1$$

Exemple : Soit la fonction f(a,b,c) = a.b + a.c + a'

Cherchons à savoir si le monôme "bc" est couvert par cette expression. La construction du BDD de la fonction f_{bc} représenté sur la figure 3.17 nous permet de conclure immédiatement à l'inclusion (comme confirmé par lz table de Karnaugh)

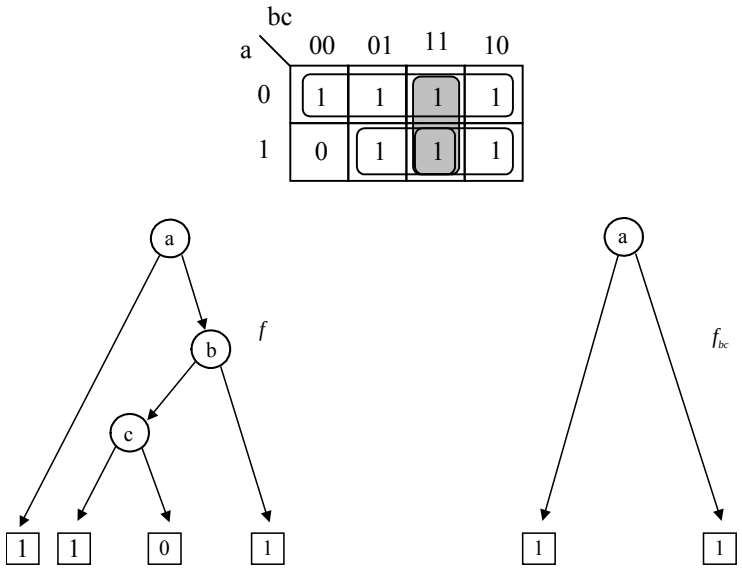


Figure 3.17. *Arbre de décision binaire de la fonction f_{bc}*

3.8. Représentation des multi-fonctions

La représentation des multi-fonctions peut se faire soit en réalisant un BDD par fonction, soit on partageant des sous-arbres. Cette dernière possibilité peut conduire à un gain de place conséquent.

Exemple :

f1(a,b) = a. b'

f2(a,b) = a ⊕ b

f3(a,b) = b'

f4(a,b) = a + b'

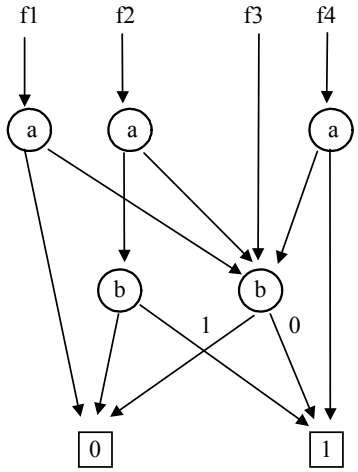


Figure 3.18. *BDD d'une multi-fonction*

La représentation informatique d'un tel BDD peut être :

	0	-	-	0
	1	-	-	1
2	b	0	1	
3	b	1	0	f3
4	a	0	3	f1
5	a	2	3	f2
6	a	3	1	f4

3.9. Manipulation des ROBDDs : fonction ITE

On peut manipuler simplement les ROBDD à l'aide de la fonction ITE (f,g,h) (if then else) définie par :

ITE (f,g,h) = f.g + f'.h (i.e. If f Then g Else h)

Soit x la première variable (i.e. Ord(x) = 1) de f,g,h.

Soit z = ITE (f,g,h). La fonction z est associée au sommet de variable x et dont les fils implantent :

ITE(f_x, g_x, h_x) et ITE ($f_{x'}$, $g_{x'}$, $h_{x'}$).

On a : $z = x.z_x + x'.z_{x'}$

$= x.(f .g + f' h)_x + x'.(f.g + f'h)_{x'}$

$= x.(f_x. g_x + f'_x h_x) + x' (f_{x'}.g_{x'} + f'_{x'}.h_{x'})$

$= ITE (x, ITE (f_x, g_x, h_x), ITE (f_{x'}, g_{x'}, h_{x'}))$

Identités remarquables :

ITE (f,1,0) = f

ITE (1,g,h) = g

ITE (0,g,h) = h

ITE (f,g,g) = g

ITE (f,0,1) = f'

De plus toutes les opérations habituelles peuvent être traduites en terme d'opérateur ITE :

0	=>	0
1	=>	1
f	=>	f
f'	=>	ITE (f,0,1)
f.g	=>	ITE (f,g,0)
f.g'	=>	ITE (f,g',0)
f'g	=>	ITE (f,0,g)
$f \oplus g$	=>	ITE (f,g',g)
f + g	=>	ITE (f,1,g)
(f+g)'	=>	ITE (f,0,g')
f + g'	=>	ITE (f,1,g')
f' + g	=>	ITE (f,g,1)
(f.g)'	=>	ITE (f,g',1)

Exemple : Construction du ROBDD à l'aide de la fonction ITE.

Soit f(a,b,c) = a.c + b.c et l'ordre (a,b,c). Soit les ROBDDs de a , b et c sur la figure suivante 3.19.

Le ROBDD de a.c est calculé comme ITE (a , c, 0).

Le ROBDD de b.c est obtenu par ITE (b, c, 0)

Le ROBDD de f est obtenu par :

f = ITE (ac, 1, bc)

= ITE (a, ITE(c, 1, b.c), ITE (0,1,b.c))

= ITE (a, c, b.c).

La variable "top" est a , la branche gauche est le ROBDD de b.c et la branche droite est le ROBDD de c. De plus puisque il y a dans le ROBBD de b.c un sommet c, ce sommet est utilisé pour noter la branche gauche. Le résultat est décrit sur la figure 3.19.

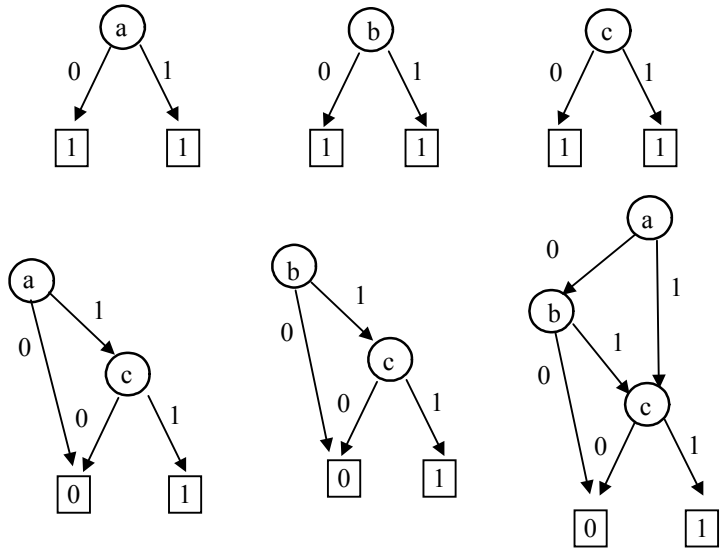


Figure 3.19. *ROBDDs de a , b et c. ROBDDs de a.c, de b.c et de f = a.c + b.c*

Algorithme :

```
ITE(f,g,h) {
  if (cas terminal)  /* c-àd 0 ou 1 figurant dans la table calculée */
        return ( r = résultat trivial)
  else    {
        if ( la table-calculee a une entrée {(f,g,h),r} )   /* déjà calculé */
              return (r à partir de la table-calculée)
        else {
              x = variable top de f, g, h ;
              t = ITE (fx, gx, hx);
              e = ITE (fx', gx', hx');
              if ( t == e)          /* fils gauche et droit isomorphes */
                    return (t);
              r = trouve-ou-ajoute-dans-Table (x, t, e); /* ajout r à table*/
              mise-a-jour-Table-calculee avec {(f,g,h), r};
              return (r);
              }
        }
  }
}
```

Cet algorithme utilise deux tables : Table et Table_calculee. Table est un tableau dont chaque ligne représente un sommet (c-à-d le tableau précédent). Ses colonnes sont l'identificateur du sommet, le nom de la variable associée , l'identificateur du fils gauche et l'identificateur du fils droit. Table-calculée est utilisée pour accélérer l'algorithme. Elle mémorise pour chaque triplet (f,g,h) le sommet implantant ITE (f,g,h).

Exemple : la Table correspondant à l'exemple de la Figure 3.20. est la suivante :

	Clef		
Identificateur	Variable	Fils gauche	Fils droit
4	a	3	2
3	b	0	2
2	c	0	1

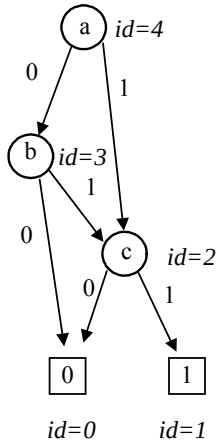


Figure 3.20. *ROBDD et table-unique*

3.10. Ordre des variables

L'ordre peut beaucoup influer sur le taille (nombre de nœuds) du BDD.

La recherche de l'ordre optimal est un problème NP-complet. En pratique on peut rechercher l'ordre optimal jusqu'à 20 variables. Pour plus de variables on utilise des heuristiques

Règles empiriques : Prendre en priorité

1. Les variables qui contrôlent le plus la fonction

- Variables apparaissant sous la même forme dans tous les monômes
- Variables constituant un monôme à elles seules
- Variables d'occurrence maximale

2. les groupes de variables ayant une relation "proche".

Exemple : $f = x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8$

Si on prend l'ordre $x_1, x_2, x_3, ..., x_8$ => 8 sommets pour 8 variables

Si on prend l'ordre $x_1, x_8, x_2, x_7, ...$ => 30 ($2^{n/2+1}-2$) sommets

Exemple : un multiplexeur 8 voies (a0,...,a7) avec 3 bits de contrôle (c0,c1,c2).

Ordre c0,c1,c2,a0,a1, ..., a7 => 7+8 sommets

Ordre a0,a1,...,a7,c0,c1,c2 => 2^{11} sommets

3.11. Comparaison avec d'autres représentations

Le tableau suivant donne la complexité de certains algorithmes en fonction du type de représentation des fonctions logiques.

	SIGMA - PI	*Galois*	*BDD*
non F	$O(\|f\|^{\|f\|1/2+1})$	O (1)	O (1)
f => g	$O((\|f\|+\|g\|)^{(\|f\|+\|g\|)\ 1/2+1})$	O (|f|*|g|) log(|f|*|g|))	O (|f|*|g|)
f <=> g	$O((\|f\|+\|g\|)^{(\|f\|+\|g\|)\ 1/2+1})$	O (|f|+|g|) log(|f|+|g|))	O (|f|*|g|)
f et g	O (|f|*|g|)	O (|f|*|g|) log(|f|*|g|))	O (|f|*|g|)
f ou g	O (|f|+|g|)	O (|f|*|g|) log(|f|*|g|))	O (|f|*|g|)
Satisfaction	*NP-complet*	O (n)	O (n)
Tautologie	*NP-complet*	O (1)	O (1)

Dans le cas d'une représentation par BDD, dans la majorité des cas, la borne supérieure du nombre de nœuds n'est pas atteinte. Ceci justifie en fait l'efficacité des BDD comparé à d'autres représentations.

Chapitre 4

Notation « cube de position »

La notation ”cube de position” constitue un modèle de représentation des fonctions logiques particulièrement intéressant pour représenter les fonctions en machine. C’est en effet une représentation largement utilisée par les outils modernes d’optimisation.

4.1. Notation ”cube de position”

La notation ”cube de position” est un codage binaire des monômes. Les différentes valeurs que peuvent prendre les entrées d’une fonction sont 0, 1, ϕ. La notation ”cube de position” code chacune de ces valeurs sur 2 bits de la façon suivante:

I => 00
0 => 10
1 => 01
ϕ => 11

Exemple : Le monôme a.b'.d' sera codé par le cube 01 10 11 10.

Définition : Le cube composé uniquement de 1 est appelé ***cube universel***.

Remarque : Le code 00 (I) est invalide. Si se code apparaît lors d’opérations entre monômes, le monômes correspondant est invalide et doit être éliminé.

La notation ”cube de position” d’une fonction est la liste des cubes de position de chaque monôme de la fonction.

Exemple : Soit la fonction f(a,b,c,d) = a'.d' + a'.b + a.b' + a.c'.d. Cette fonction peut être représentée par la liste de monômes suivants :

10 11 11 10
10 01 11 11
01 10 11 11
01 11 10 01

Pour les fonctions multiples un champ de sortie doit être rajouté à la table représentant les monômes en notation ”cube de position. Ce champ représente la validité du monôme sur chacune des fonctions.

Exemple : Soit les fonctions f1(a,b) = a'.b' + a.b , f2(a,b) = a.b, f3(a,b) = a.b' + a'.b. La notation ”cube de position” de cette couverture de la fonction est:

10 10 100
10 01 001
01 10 001
01 10 110

4.2. Opérations sur les monômes

La notation ”cube de position” facilite la manipulation et les opérations sur les monômes ou ensemble de monômes.

4.2.1. Opérations d'union et d'intersection de monômes

Les opérations d'union ($\cup$) et d'intersection ($\cap$)de deux monômes peuvent être réalisées simplement en appliquant l'opération « somme » ou l'opération « produit » sur les représentations codées des monômes.

Exemple : Soit les deux monômes α et β suivants:

α = a.b => 01 01 11

β = b.c' => 11 01 10

$m = \alpha \cap \beta$ = (01 01 11) . (11 01 10) = 01 01 10 => m = a.b.c'

$m = \alpha \cup \beta$ = (01 01 11) + (11 01 10) = 11 01 11 => m = b

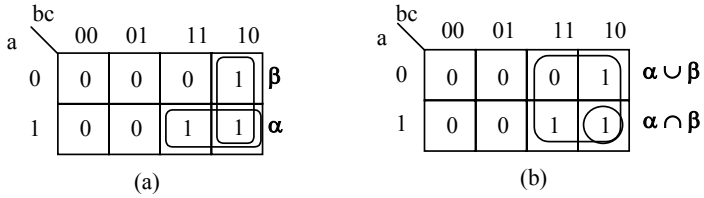


Figure 4.1. *Interprétation des opérations d'union et d'intersection*

L'intersection entre monômes peut être vide. Nous obtiendrons alors un cube invalide (avec au moins un champ à 00). L'union entre deux monômes peut être égale à 1. Dans ce cas, nous obtiendrons un cube composé uniquement de 1. Ceci signifie que la fonction est à 1 quelque soit la valeur des variables d'entrée (f = 11).

Le monôme obtenu par l'union des 2 monômes représente le plus petit monôme qui couvre tous les points couverts par α et β.

Exemple : Soit les deux monômes α et β suivants:

α = a.b => 01 01 11

β = b'.c' => 11 10 10

$\alpha \cap \beta$ = (01 01 11) . (11 10 10) = 01 00 10 => Intersection vide

$\alpha \cup \beta$ = (01 01 11) + (11 10 10) = 11 11 11 => Union = 1

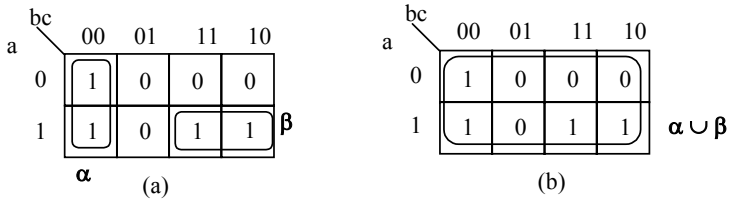


Figure 4.2. *Interprétation des opérations d'union et d'intersection*

4.2.2 Distance entre monômes

Définition : La distance entre deux monômes est le nombre de champs nuls (00) dans le monôme résultant de l'intersection (distance de Hamming entre les faces de l'hypercube).

Exemple : Soit les deux monômes α et β suivants:

α => 01 01 11 β => 11 01 10 $\alpha \cap \beta$ = 01 01 10 => $d[\alpha,\beta] = 0$

α => 01 01 11 β => 11 10 10 $\alpha \cap \beta$ = 01 00 10 => $d[\alpha,\beta] = 1$

Théorème : Lorsque la distance est égale à 0 les monômes s'intersectent, sinon ils sont disjoints.

Preuve : Si la distance est différente de 0, il existe un champ 00. Ce champ 00 ne peut être obtenu qu'à partir de l'intersection de deux champs valant 01 et 10. Ceci représente le fait qu'un des monôme dépende de la variable sous forme directe et l'autre sous forme complémentée. Ces deux monômes sont donc disjoints. Si la distance est égale à 0, il n'existe pas de champs 00. Il n'existe donc pas d'incompatibilité dans les deux monômes. Ces monômes ne peuvent donc pas être disjoints. Par conséquent ils s'intersectent.

Cette notion de distance permet notamment de savoir si la couverture d'une fonction est disjointe ou pas.

Exemple : Soit les fonctions f1(a,b) = a'.b' + a.b , f2(a,b) = a.b, f3(a,b) = a.b' + a'.b. La notation "cube de position" de cette couverture de la fonction est:

10 10 100
10 01 001
01 10 001
01 01 110

Aucun monôme ne s'intersecte. Cette expression constitue une couverture disjointe de la fonction.

4.2.3. Opération Dièse (#)

L'opération « Dièse » permet de déterminer (générer) tous les mintermes (monômes à n variables) couverts par un monôme α et non-couverts par un monôme β. $\alpha \# \beta = \alpha \,.\, \beta'$

Soit $\alpha = a_1\, a_2 \ldots a_n$ $\beta = b_1\, b_2 \ldots b_n$

$$\alpha \# \beta = \begin{cases} a_1.b_1' & a_2 & \ldots & a_n \\ a_1 & a_2.b_2' & \ldots & a_n \\ . & . & \ldots & \\ a_1 & a_2 & \ldots & a_n.b_n' \end{cases}.$$

Exemple 1 : Soit α = a.b => 01 01 11 et β = b.c' => 11 01 10

$\alpha \# \beta$ = 00 01 11 Invalide
01 00 11 Invalide
01 01 01

$\alpha \# \beta$ = a.b.c

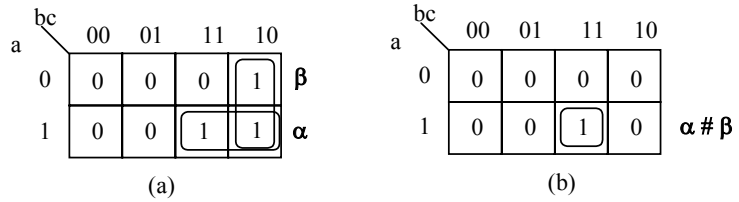


Figure 4.3. *Interprétation de l'opération « dièse »*

Exemple 2 : Soit α = a'.c' => 10 11 10 11 et β = b.c'.d => 11 01 10 01

$\alpha \# \beta$ = 00 11 01 11 Invalide
10 10 10 11
10 11 00 11 Invalide
10 11 10 10

$\alpha \# \beta$ = a'.b'.c' + a'.c'.d'

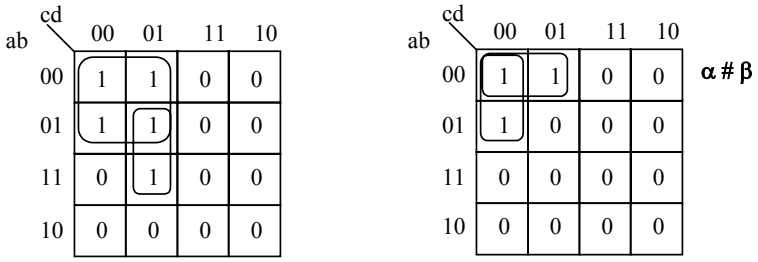


Figure 4.4. *Interprétation de l'opération « dièse »*

Cette opération permet également de déterminer le **complément** d'un monôme. En effet, le complément d'un monôme est l'ensemble des mintermes (cube universel) moins ceux couverts par le monôme en question.

Exemple : Soit α = a.b => 01 01 11

$\alpha' = U \# \alpha$ = (11 11 11) # (01 01 11)
= (10 11 11), (11 10 11)
= a'+ b'

L'opération "Dièse disjointe" notée $\#_d$ permet également de déterminer tous le mintermes couverts par un monôme α et non couverts par un monôme β mais produit une couverture disjointe. Cette opération est définie de la manière suivante:

Soit $\alpha = a_1\ a_2 \ldots a_n$ $\beta = b_1\ b_2 \ldots b_n$

$$\alpha \#_d \beta = \begin{cases} a_1.b_1' & a_2 & \dots & a_n \\ a_1.b_1 & a_2.b_2' & \dots & a_n \\ . & . & \dots & . \\ a_1.b_1 & a_2.b_2 & \dots & a_n.b_n' \end{cases}$$

Exemple : Soit α = a.b => 01 01 11

$\alpha' = U \#_d \alpha$ = (11 11 11) $\#_d$ (01 01 11)

= (10 11 11), (01 10 11)

= a' + a.b' Les 2 monômes a' et a.b' sont disjoints

4.2.4. Opération consensus

L'opération consensus entre deux monômes α et β notée Cons(α,β) est définie de la manière suivante:

Soit $\alpha = a_1\ a_2 \dots a_n$ $\beta = b_1\ b_2 \dots b_n$

$$Cons(\alpha,\beta) = \begin{cases} a_1+b_1 & a_2.b_2 & \dots & a_n.b_n \\ a_1.b_1 & a_2+b_2 & \dots & a_n.b_n \\ . & . & \dots & . \\ a_1.b_1 & a_2.b_2 & \dots & a_n+b_n \end{cases}$$

Rappel : x y + x' z = x y + x' z + y z (théorème des consensus)

Le résultat de l'opération de consensus est nul lorsque la distance entre les deux monômes est supérieure ou égale à 2. Lorsque la distance entre les deux monômes est égale à 1, l'opération de consensus conduit à un monôme unique.

Exemple : Considérons les trois monômes suivants:

α = a.b'.c => 01 10 01 d(α,β) = 1

β = a.c' => 01 11 10 d(α,χ) = 2

χ = a'.b.c => 10 01 01 d(χ,β) = 2

Cons (α,β) = 01 10 00 Invalide

01 11 00 Invalide

01 10 11 = a.b'

Cons (α,χ) = 11 00 01 Invalide

00 11 01 Invalide

01 00 01 Invalide

Cons (β,χ) = 11 01 00 Invalide

00 11 00 Invalide

00 01 11 Invalide

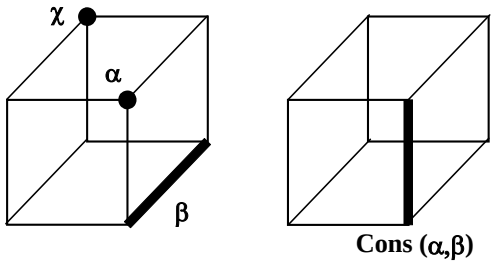


Figure 4.5. *Interprétation graphique du consensus*

4.3. Opérations sur les fonctions (ensemble de monômes)

Des opérations entre monômes ont été définies. Considérons maintenant ces opérations sur des ensembles de monômes.

L'union de deux ensembles de monômes, c'est à dire la somme de deux fonctions, est obtenue en fusionnant les deux ensembles (et en éliminant les duplications).

L'intersection de deux ensembles de monômes, c'est à dire le produit de deux fonctions, est obtenue en réalisant l'intersection de tous les couples de monômes. La complexité de cette opération est le produit des cardinalités des deux ensembles.

Les opérations Dièse (#) et Dièse disjointe ($\#_d$) peuvent être appliquées entre deux ensembles de monômes.

L'opération Dièse (disjointe) entre un monôme α et un ensemble de monômes M est obtenue en appliquant itérativement l'opérateur Dièse (disjoint) entre le monôme α et chaque monôme m_i de l'ensemble.

L'opération Dièse (disjointe) entre deux ensembles de monômes est l'intersection des résultats de l'opération Dièse appliquée sur le deuxième ensemble à partir de chaque élément du premier.

Démonstration : soit $f1 = m_1 + m_2 + ... + m_p$ et $f2 = n_1 + n_2 + ... + n_q$

$f1 \# f2 = f1 \,.\, f2'$

$= (m_1 + m_2 + ... + m_p) \,.\, (n_1 + n_2 + ... + n_q)'$

$= (m_1 + m_2 + ... + m_p) \,.\, n_1' \,.\, n_2' \,.... n_q'$

$= m_1 \,.\, n_1' \,.\, n_2' \,.... n_q' + m_2 \,.\, n_1' \,.\, n_2' \,.... n_q' + ... + m_p .\, n_1' \,.\, n_2' \,.... n_q'$

$= (m_1 \# n_1).(m_1 \# n_2.)...(m_1 \# n_q) + ...+ (m_p \# n_1).(m_p \# n_2)...(m_p \# n_q)$

Le complément d'une fonction peut être calculé par l'intermédiaire de l'opération Dièse (disjointe). Le complément est le résultat de l'application de l'opérateur Dièse (disjoint) entre le monôme U (représentant l'espace booléen) et l'ensemble des monômes de la fonction.

Exemple : Soit f(a,b,c) = a.b + a'.c'

α = a.b = 01 01 11

β = a'.c' = 10 11 10

$\alpha' = U \# \alpha$ = (11 11 11) # (01 01 11)

= (10 11 11), (11 10 11)

= a + b

$\beta' = U \# \beta$ = (11 11 11) # (10 11 10)

= (01 11 11), (11 11 01)
= a + c

$\alpha' \cap \beta'$ = (10 11 01), (01 10 11), (11 10 01)

=> f' = a'.c + a.b' + b'.c

4.4. Applications de la notation « cube de position »

4.4.1. Décomposition de fonction et cofacteurs

Le théorème de Shannon permet de décomposer les fonctions logiques par rapport aux différentes variables.

$f(x_1, x_2, ..., x_i, ..., x_n) = x_i \, . \, f_{xi} + x_i' \, . \, f_{xi}'$

$f(x_1, x_2, ..., x_i, ..., x_n) = (x_i + f_{xi}') \, . \, (x_i' + f_{xi})$

$f_{xi} = f(x_1, x_2, ..., 1, ..., x_n)$ cofacteur de f par rapport à la variable xi.

$f_{xi}' = f(x_1, x_2, ..., 0, ..., x_n)$ cofacteur de f par rapport à la variable xi'.

Cette décomposition est extrêmement intéressante car elle permet de traiter le fonctions de manière récursive.

Définition : Le cofacteur d'un monôme α ($\alpha = a_1 \, a_2 \, ... \, a_n$) par rapport à la variable x_i (x_i = 11 11 ...b_i 11 ...11) noté α_{xi} est obtenu par :

si $\alpha \cap x_i \neq \emptyset$. => $\alpha_{xi} = a_1 \, a_2 \, ... \, a_i + b_i' \, a_{i+1} \, ... \, a_n$

sinon vide

Définition : Le cofacteur d'un ensemble de monômes par rapport à la variable x_i est l'ensemble des cofacteurs des monômes par rapport à la variable x_i.

Développer une fonction revient à calculer les cofacteurs par rapport aux variables. Ces cofacteurs peuvent être déterminés par l'intermédiaire de la notation cube de position en appliquant les définitions suivantes:

Définition : Le cofacteur d'un monôme α ($\alpha = a_1 \, a_2 \, ... \, a_n$) par rapport à un monôme β ($\beta = b_1 \, b_2 \, ... \, b_n$) noté α_β est donné par:

si $\alpha \cap \beta \neq \emptyset$. => $\alpha_\beta = a_1 + b_1' \; a_2 + b_2' \; ... \; a_n + b_n'$

sinon vide

Le cofacteur d'un monôme par rapport à une variable peut être obtenu par la formule précédente sachant qu'une variable est un monôme particulier.

Définition : Le cofacteur d'un ensemble de monômes par rapport à un monôme α est l'ensemble des cofacteurs entre chacun des monômes de l'ensemble et le monôme α.

Exemple : f(a,b,c) = a.b + a'.b.c + b'.c

α = a.b => 01 01 11 a => 01 11 11

β = a.b.c => 10 01 01 a' => 10 11 11

χ = b.c => 11 10 01

α_a = 11 01 11 = b $\alpha_{a'}$ => vide ($\alpha \cap$ a => 00 01 11)

β_a => vide (b $\cap$ a => 00 01 01) $\beta_{a'}$ = 11 01 01 = b.c

χ_a = 11 10 01 = b'.c $\chi_{a'}$ = 11 10 01 = b'.c

=> f_a = b + b'.c

=> $f_{a'}$ = b.c + b'.c

=> f(a,b,c) = a.(b + b'.c) + a'.(b.c + b'.c)

4.4.2. Preuve de tautologie

La preuve de tautologie joue un rôle important dans les algorithmes d'optimisation logique. Malgré la difficulté du problème, savoir si une fonction est une tautologie peut être réalisé de manière efficace en utilisant une approche récursive.

Théorème : Une fonction f est une tautologie si et seulement si ses cofacteurs par rapport à toutes les variables et variables complémentées sont des tautologies.

Preuve : Ces deux implications sont des conséquences directes du théorème de Shannon

En conséquence, vérifier qu'une fonction est une tautologie peut être fait en développant récursivement la fonction par rapport aux variables et en vérifiant que les cofacteurs sont des tautologies.

Nous noterons que lorsqu'une fonction est monoforme pour une variable, il est suffisant de vérifier qu'un seul des cofacteurs est une tautologie. En effet, si un des cofacteurs est une tautologie l'autre l'est obligatoirement car $f_{xi'} \subseteq f_{xi}$ (xi monoforme positive) ou $f_{xi} \subseteq f_{xi'}$ (xi monoforme négative).

Exemple : f = a.b + c => f = a.(b + c) + a'.(c)

$c \subseteq b + c$ => si c est une tautologie, b+c est une tautologie

Théorème : Soit f une fonction monoforme par rapport à une variable x, g un sous-ensemble de f ne dépendant pas de x (11 dans le champs correspondant). f est une tautologie si et seulement si g est une tautologie.

Preuve : Soit f monoforme en x: f =x.g1 + g2

g2 = 1 => f = 1 donc g2 tautologie => f tautologie

f tautologie => f = 1 quelque soit la valeur de x

Si x=0, f = g2 => g2 =1

Théorème : Une couverture monoforme F est une tautologie si et seulement si un de ses monômes est le cube universel U = 11 11 ...

Preuve : Appliquons itérativement le théorème précédent sur toutes les variables. La couverture F est une tautologie si et seulement si le sous-ensembles G contenant les monômes ne dépendant pas de toutes les variables est une tautologie. Uniquement le cube universel peut être dans G. Par conséquent G est une tautologie si et seulement si F contient U.

A partir des théorèmes précédents, nous pouvons tirer 6 règles permettant d'arrêter ou de simplifier la procédure d'expansion récursive de vérification de tautologie:

1. Une couverture est une tautologie lorsqu'elle a une ligne de 1 (Cube tautologie).
2. Une couverture n'est pas une tautologie lorsqu'elle a une colonne de 0 (une variable ne prenant jamais soit la valeur 0, soit la valeur 1). c-à-d on une variable monoforme figurant dans tous les monômes.
3. Une couverture est une tautologie lorsqu'elle ne dépend que d'une variable et qu'il n'y a pas de colonne de 0 dans le champ correspondant. c-à-d si on a: x + x'
4. Une couverture n'est pas une tautologie lorsqu'elle est monoforme et qu'il n'y a pas de ligne de 1. c-à-d lorsqu'elle ne contient que des variables monoformes et qu'elle ne contient pas U.

5. Le problème peut être simplifié lorsque la couverture est monoforme sur certaines variables. Dans ce cas, la vérification de tautologie peut être faite en considérant uniquement les lignes ne dépendant pas des variables monoformes.
6. Lorsqu'une couverture peut s'écrire comme une union de deux sous-couvertures dépendant de sous-ensembles disjoints de variable, la vérification de tautologie peut être réduite en vérifiant la tautologie des deux sous-couvertures.

Lorsque la tautologie ne peut être décidée à partir des 6 règles précédentes, la fonction doit être expansée par rapport à une variable. Le choix de la variable à considérer est important mais ne sera pas traité ici, on prend en général la variable figurant dans le plus grand nombre de monômes.

Exemple 1 : Soit la fonction f(a,b,c) = a.b + a.c + a.b'.c' + a' . La question est de savoir si cette fonction est une tautologie.

La couverture F de cette fonction f peut s'exprimer ainsi:

01 01 11

01 11 01

01 10 10

10 11 11

Toutes les variables sont biformes. La première variable "a" apparaît le plus souvent dans les monômes, nous la considérerons en premier pour développer la fonction.

Cofacteur par rapport à C(a) = 01 11 11 => (11 01 11), (11 11 01), (11 10 10)

Cofacteur par rapport à C(a') = 10 11 11 => (11 11 11)

Le cofacteur par rapport à C(a') est une tautologie (règle 1). L'étude doit se poursuivre sur le cofacteur par rapport à C(a)

11 01 11

11 11 01

11 10 10

Intéressons nous maintenant au deux dernières colonnes. Les deux variables correspondantes sont biformes. Les deux variables intervenant le même nombre de fois dans les monômes (2), considérons "b" pour développer la fonction.

Cofacteur par rapport à C(b) = 11 01 11 => (11 11 11), (11 11 01)

Cofacteur par rapport à C(b') = 11 10 11 => (11 11 01), (11 11 10)

Le cofacteur par rapport à C(b) est une tautologie (règle 1). Le cofacteur par rapport à C(b') est également une tautologie (règle 3).

11 11 01

11 11 10

La fonction f est donc une tautologie

Exemple 2 : Soit la fonction f(a,b,c) = a.b + a.c + a' . La question est de savoir si cette fonction est une tautologie.

La couverture F de cette fonction f peut s'exprimer ainsi:

01 01 11

01 11 01

10 11 11

La première variable ”a” apparaît le plus souvent dans les monômes, nous la considérerons en premier pour développer la fonction.

Cofacteur par rapport à C(a) = 01 11 11 => (11 01 11), (11 11 01)

Cofacteur par rapport à C(a') = 10 11 11 => (11 11 11)

Le cofacteur par rapport à C(a') est une tautologie (règle 1). L’étude doit se poursuivre sur le cofacteur par rapport à C(a)

11 01 11

11 11 01

La couverture est monoforme et il n’y a pas de ligne de 1. f n’est donc pas une tautologie (règle 4).

4.4.3. Inclusion

Savoir si un monôme est inclus dans l’expression de la fonction, c’est à dire couvert par une fonction, peut être un problème complexe. Le théorème d’inclusion permet de ramener ce problème à une problème de preuve de tautologie sur des cofacteurs.

Rappel du théorème d’inclusion : Une expression F contient un monôme m si et seulement si le cofacteur de F par rapport à m (F_m) est une tautologie.

$$m \subset F \Leftrightarrow F_m = 1$$

Exemple : Soit la fonction f (a,b,c) = a.b + a.c + a' . La couverture F de cette fonction f peut s’exprimer ainsi:

01 01 11

01 11 01

10 11 11

Cherchons à savoir si le monôme ”bc” est couvert par par cette expression. Le cube de positionnement de ”bc” est C(bc) = 11 01 01. En calculant les cofacteurs de F par rapport au monôme ”bc” on obtient:

01 01 11 + 00 10 10 => 01 11 11

01 11 01 + 00 10 10 => 01 11 11

10 11 11 + 00 10 10 => 10 11 11

Cette expression est une tautologie car les monômes ne dépendent que d’une variable et, il n’y a pas de colonne de 0 (règle 3). le monôme ”bc” est donc couvert par f.

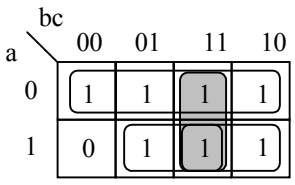


Figure 4.6. *Interprétation de l’opération du test d’inclusion*

4.4.4. Complémentation

La complémentation d'une fonction peut se faire en utilisant l'opération # par U#f mais également de manière très efficace en utilisant la récursivité et le théorème de Shannon.

$$f = (x + f_{x'}) \,.\, (x' + f_x) \quad \Rightarrow \quad f' = x' \,.\, f'_{x'} + x \,.\, f'_x$$

En effet, la complémentation d'une fonction peut être faite récursivement en utilisant l'expression complémentée des cofacteurs de f. Soit F' une expression de f', F'_x et $F'_{x'}$ des expressions de f'_x et $f'_{x'}$.

$$F' = [C(x') \cap F'_{x'}] \cup [C(x) \cap F'_x]$$

Théorème : Si une fonction est monoforme par rapport à une variable x alors le développement peut être simplifié.

Monoforme positive /x $\Rightarrow f' = x' \,.\, f'_{x'} + f'_x$

$\Rightarrow F' = [C(x') \cap F'_{x'}] \cup F'_x$

Monoforme négative /x $\Rightarrow f' = f'_{x'} + x \,.\, f'_{x'}$

$\Rightarrow F' = F'_{x'} \cup [C(x) \cap F'_x]$

Preuve : Théorème de Shannon $\Rightarrow f = (x + f_{x'}) \,.\, (x' + f_x)$

Fonction monoforme positive en x $\Rightarrow f = (x + f_{x'}) \,.\, f_x$

$\Rightarrow f' = x' \,.\, f'_{x'} + f'_x$

Fonction monoforme négative en x $\Rightarrow f = f_{x'} \,.\, (x' + f_x)$

$\Rightarrow f' = x \,.\, f'_x + f'_{x'}$

La procédure de calcul du complément est une procédure récursive qui se termine lorsque l'expression F satisfait une des conditions suivantes:

1. La couverture F est nulle. Alors son complément est le cube universel.
2. La couverture F a une ligne de 1. Alors, F est une tautologie et son complément est nul.
3. La couverture F est constituée d'un seul monôme. Alors le complément est calculé par l'intermédiaire du théorème de De Morgan (ou de l'opération #).
4. Tous les monômes de F ne dépendent que d'une seule variable et il n'existe pas de colonne de 0. Alors, F est une tautologie et son complément est nul.
5. La couverture a une colonne de 0 en position j. Soit α le cube composé uniquement de 1 sauf en position j. Alors, $F = \alpha \cap F_\alpha$ et $F' = \alpha' \cup F'_{\alpha'}$. Alors, calculer récursivement le complément de F'_α et de α et retourner leur union.

Si aucune des règles précédente n'est vérifiée, la procédure récursive d'inversion doit continuer. La fonction doit être développée par rapport à une variable. Le choix de la variable à considérer est important mais ne sera pas traité ici.

Exemple : Soit la fonction f(a,b,c) = a.b + a.c + a' . (ordre a, b, c). La couverture F de cette fonction f peut s'exprimer ainsi:

01 01 11
01 11 01
10 11 11

La seule variable biforme est "a". Elle sera prise en premier pour développer la fonction.

$F' = [C(a') \cap F'_{a'}] \cup [C(a) \cap F'_a]$

Cofacteur par rapport à C(a) = 01 11 11 => F_a = (11 01 11), (11 11 01)
Cofacteur par rapport à C(a') = 10 11 11 => $F_{a'}$ = (11 11 11)

La couverture F_a' a une seule ligne de 1. C'est une tautologie. Sont complément est nul ($F'_{a'} = 0$).

=> F' = [C(a') ∩ F' $_a$]

La couverture F_a ne répond à aucun critère d'arrêt. Continuons la procédure à partir de la couverture F_a:
11 01 11
11 11 01

Les deux variable sont monoformes. Développement par rapport à la variable b.

$F'_a = [C(b') \cap F'_{ab'}] \cup [C(b) \cap F'_{ab}]$

b est monoforme positive => on peut simplifier l'expression précédente.

$F'_a = [C(b') \cap F'_{ab'}] \cup F'_{ab}$
Cofacteur par rapport à C(b) = 11 01 11 => F_{ab} = (11 11 11)
Cofacteur par rapport à C(b') = 11 10 11 => $F_{ab'}$ = (11 11 01)

La couverture Fab a une seule ligne de 1. C'est une tautologie. Sont complément est nul ($F'_{ab} = 0$).

$F_{ab'}$ = (11 11 01) => un seul monôme
=> $F'_{ab'}$ = (11 11 10)

$F'_a = [C(b') \cap F'_{ab'}]$
= (11 10 11) ∩ (11 11 10)
= (11 10 10)

F' = [C(a) ∩ F'_a]
= (01 11 11) ∩ (11 10 10)
= (01 10 10)

f' = a.b'.c'

Des heuristiques permettent de choisir à priori le meilleur ordre des variables c-à-d celui qui permet de converger le plus rapidement. Ces heuristiques ne seront pas développées ici.

Remarque : l'expression de la couverture du complément dépend de l'ordre des variables.

Exemple : Soit la fonction f (a,b,c) = a'.b'.c' + a.b'.c' + a'.b.c' + a'.b'.c + a.b.c'. La couverture F de cette fonction f peut s'exprimer ainsi:

10 10 10
01 10 10
10 01 10
10 10 01
01 01 10

F' = [C(a') ∩ F'a'] ∪ [C(a) ∩ F'a]
Cofacteur par rapport à C(a) = 01 11 11 => F_a = (11 10 10), (11 01 10)
Cofacteur par rapport à C(a') = 10 11 11 => $F_{a'}$ = (11 10 10), (11 01 10), (11 10 01)

La couverture F_a a une colonne de 0. Soit α = 11 11 10,

$F_a = \alpha \cap F_{a\alpha}$ => $F'_a = \alpha' \cup F'_{a\alpha}$
$F_{a\alpha}$ = (11 10 11), (11 01 11)

Tous les monômes de $F_{a\alpha}$ ne dépendent que d'une seule variable et il n'y a pas de colonne de 0.

$F_{a\alpha}$ est donc un tautologie => $F'_{a\alpha} = 0$.
$F'_a = \alpha' \cup F'_{a\alpha}$ => (11 11 01)

La couverture $F_{a'}$ ne répond à aucun critère d'arrêt. Continuons la procédure à partir de la couverture $F_{a'}$:
11 10 10
11 01 10
11 10 01

$F'_{a'} = [C(b') \cap F'_{a'b'}] \cup [C(b) \cap F'_{a'b}]$
Cofacteur par rapport à C(b) = 11 01 11 => $F_{a'b}$ = (11 11 10)
Cofacteur par rapport à C(b') = 11 10 11 => $F_{a'b'}$ = (11 11 10),(11 11 01)

La couverture $F_{a'b}$ ne contient qu'un monôme => $F'_{a'b}$ = (11 11 01)

Tous les monômes de $F_{a'b'}$ ne dépendent que d'une seule variable et il n'y a pas de colonne de 0.

$F_{a'b'}$ est donc un tautologie => $F'_{a'b'} = 0$.
=> $F'_{a'} = [C(b) \cap F'_{a'b}] = (11\ 01\ 01)$
$F' = [C(a') \cap F'_{a'}] \cup [C(a) \cap F'_a] = (10\ 01\ 01),(01\ 11\ 01)$
f' = a'.b.c + a.c

Exemple : Considérons la même fonction que précédemment mais prenons les variables dans l'ordre b,a,c. f (a,b,c) = a'.b'.c' + a.b'.c' + a'.b.c' + a'.b'.c + a.b.c'.
10 10 10
01 10 10
10 01 10
10 10 01
01 01 10

$F' = [C(b') \cap F'_{b'}] \cup [C(b) \cap F'_b]$
Cofacteur par rapport à C(b) = 11 01 11 => F_b = (10 11 10), (01 11 10)
Cofacteur par rapport à C(b') = 11 10 11 => $F_{b'}$ = (10 11 10), (01 11 10), (10 11 01)

La couverture F_b a une colonne de 0.

Soit α = 11 11 10, $F_b = \alpha \cap F_{b\alpha}$ => $F'_b = \alpha' \cap F'_{b\alpha}$
$F_{b\alpha}$ = (10 11 11), (01 11 11)

Tous les monômes de $F_{b\alpha}$ ne dépendent que d'une seule variable et il n'y a pas de colonne de 0.

$F_{b\alpha}$ est donc un tautologie => $F'_{b\alpha} = 0$.
$F'_b = \alpha' \cup F'_{b\alpha}$ => (11 11 01)

La couverture $F_{b'}$ ne répond à aucun critère d'arrêt. Continuons la procédure à partir de la couverture $F_{b'}$:

10 11 10

01 11 10

10 11 01

$F'_{b'} = [C(a') \cap F'_{b'a'}] \cup [C(a) \cap F'_{b'a}]$

Cofacteur par rapport à C(a) = 01 11 11 => $F_{b'a}$ = (11 11 10)

Cofacteur par rapport à C(a') = 10 11 11 => $F_{b'a'}$ = (11 11 10),(11 11 01)

La couverture $F_{b'a}$ ne contient qu'un monôme => $F'_{b'a}$ = (11 11 01)

Tous les monômes de $F_{b'a'}$ ne dépendent que d'une seule variable et il n'y a pas de colonne de 0.

$F_{b'a'}$ est donc un tautologie => $F'_{b'a'} = 0$.

=> $F'_{b'} = [C(a) \cap F'_{b'a}] = (01\ 11\ 01)$

$F' = [C(b') \cap F'_{b'}] \cup [C(b) \cap F'_{b}] = (01\ 10\ 01),(11\ 01\ 01)$

f' = a.b'.c + b.c

La couverture obtenue est différente de celle obtenue en considérant les variables dans l'ordre a,b,c.

Exemple : Considérons la même fonction que précédemment mais prenons les variables dans l'ordre c,b,a. f(a,b,c) = a'.b'.c' + a.b'.c' + a'.b.c' + a'.b'.c + a.b.c'.

10 10 10

01 10 10

10 01 10

10 10 01

01 01 10

$F' = [C(c') \cap F'_{c'}] \cup [C(c) \cap F'_{c}]$

Cofacteur par rapport à C(c) = 11 11 01 => F_c = (10 10 11)

Cofacteur par rapport à C(c') = 11 11 10 => $F_{c'}$ = (10 10 11),(01 10 11),(10 01 11),(01 01 11)

La couverture F_c a un seul monôme. Pour simplifier le traitement, on peut appliquer le théorème de De Morgan (a'.b'= (a + b)').

=> F'_c = (01 11 11), (11,01,11)

La couverture $F_{c'}$ ne répond à aucun critère d'arrêt. Continuons la procédure à partir de la couverture $F_{c'}$:

10 10 11

01 10 11

10 01 11

01 01 11

$F'_{c'} = [C(b') \cap F'_{c'b'}] \cup [C(b) \cap F'_{c'b}]$

Cofacteur par rapport à C(b) = 11 01 11 => $F_{c'b}$ = (10 11 11),(01 11 11)

Cofacteur par rapport à C(b') = 11 10 11 => $Fc_{b'}$ = (10 11 11),(01 11 11)

Tous les monômes de $F_{c'b}$ ne dépendent que d'une seule variable et il n'y a pas de colonne de 0.

$F_{c'b}$ est donc un tautologie => $F'_{c'b} = 0$.

Tous les monômes de $F_{c'b'}$ ne dépendent que d'une seule variable et il n'y a pas de colonne de 0.

$F_{c'b'}$ est donc un tautologie => $F'_{c'b'} = 0$.

=> $F'_{c'} = 0$

=> $F' = [C(c) \cap F'_c] = (01\ 11\ 01),(11\ 01\ 01)$

$f' = a.c + b.c$

La couverture obtenue est encore différente de celle obtenue en considérant les variables dans l'ordre a,b,c ou b,a,c.

Chapitre 5

Minimisation de fonctions logiques

La minimisation de fonctions logiques à pour objectif de réduire le « coût » d'implantation de ces fonctions. Ce coût d'implantation représente généralement la surface occupée par le circuit ou le nombre de composants nécessaires à sa réalisation. En fonction du style d'implantation, ce coût peut être exprimé de diverses manières (nombre de littéraux, nombre de monômes, …). En effet, nous verrons par exemple que pour minimiser les fonctions implantées sur deux niveaux (Implantations de type PLA), le nombre de monômes intervenant dans l'expression logique est le paramètre à considérer alors que dans le cas des implantations multi niveaux c'est le nombre de littéraux qu'il faut privilégier.

Pour ce qui est du processus de minimisation proprement dit, nous verrons que dans le cas général il n'est pas possible d'obtenir directement une expression minimale d'une fonction. Aussi, les méthodes exactes de minimisation se décomposent-elles en deux étapes : tout d'abord, la recherche de la base première complète de la fonction qui contient tous les termes susceptibles d'intervenir dans une expression minimale puis, une procédure de choix sur l'ensemble des termes obtenus pour aboutir à une expression de la fonction qui minimise le coût.

Aujourd'hui, les fonctions de plus de 20 variables et de plus 500 monômes ne sont plus rares. Aussi, les méthodes exactes ne sont pratiquement plus utilisables pour de telles fonctions. Les méthodes permettant d'optimiser ces fonctions sont donc basées sur des heuristiques. En effet, elles ne cherchent pas à obtenir la solution optimale mais cherchent une solution approchante. Aussi, on ne cherchera pas la base première complète puis une base minimale, mais directement une base première irrédondante issue de la forme de départ.

5.1. Définitions générales

5.1.1. Coût d'une expression logique

Le coût d'une expression logique est une application qui à une expression logique quelconque fait correspondre un nombre réel positif. Cette notion de coût est rattaché au problème pratique traité : minimisation du matériel utilisé pour réaliser un circuit logique, minimisation de la place mémoire occupée, etc... On conçoit donc qu'il soit possible de définir plusieurs fonctions de coût.

Cette fonction coût dépend en général de plusieurs paramètres pouvant varier selon le style d'implantation (Implantation 2-niveaux ou multi-niveaux). Dans ce chapitre, nous considérons comme seul paramètre le nombre de monômes intervenant dans les expressions logiques. Ceci nous permettra de minimiser les fonctions implantées sur deux niveaux (Implantations de type PLA) et de faire une première réduction des fonctions implantées sur plusieurs niveaux.

A titre d'exemple, la figure 5.1 représente une implantation 2-niveaux de type PLA. Sur une telle implantation, chaque ligne correspond à un monôme. Minimiser le nombre de monômes revient donc à minimiser le nombre de lignes est par conséquent la surface de la structure.

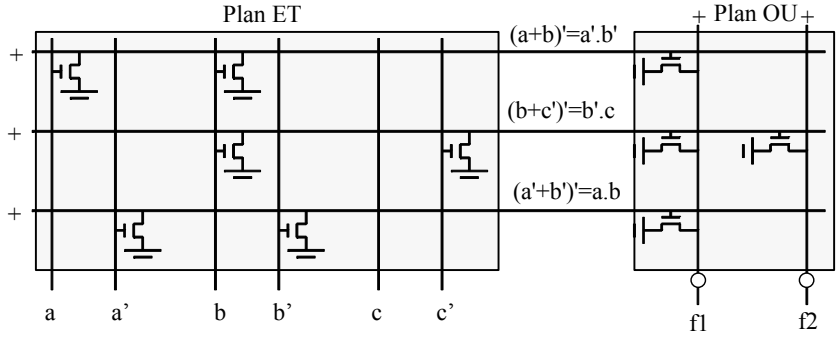


Figure 5.1. *Implantation PLA des fonctions f1 = a'.b' + b'.c + a.b , f2 = b'.c*

Pour les implantations multi-niveaux d'autres paramètres tels que le nombre de littéraux sont prépondérants. Les techniques d'optimisation d'implantations multi-niveaux sont basées sur la factorisation et seront développées dans le chapitre suivant.

5.1.2. Monômes premiers d'une fonction

Soit « m » un monôme d'une fonction. Notons « m_d » le monôme obtenu en supprimant un nombre quelconque de littéraux de « m ». « m_d » peut être considéré comme un des diviseurs de « m ».

Définition : On définit un ***monôme premier*** d'une fonction f(X) comme un monôme « m » de f(X) tel qu'aucun diviseur « m_d » de « m » ne soit inclus dans f(X).

Exemple : Considérons le monôme m = c'.d de f(X) définie par la table de la figure 5.2. Les diviseurs de « m » sont c', d et 1 aucun d'eux n'est inclus dans f(X) donc « m » est monôme premier. Par contre a.b.d n'est pas premier puisque bd est inclus dans f(x).

ab \ cd	00	01	11	10
00	0	1	0	1
01	1	1	0	1
11	1	1	1	1
10	1	1	1	0

Figure 5.2. *Monômes premier*

Remarque : il suffit manifestement de considérer les diviseurs par un seul littéral pour définir un monôme premier puisque tout diviseur contient l'un d'eux et donc, si aucun diviseur par un littéral n'est inclus dans la fonction, il en sera de même pour tout autre diviseur.

On voit encore ici qu'il est possible d'obtenir à partir d'une fonction ∑Π de f(X) une nouvelle expression ∑Π en remplaçant des monômes par les monômes premiers qui les contiennent.

Remarque : Si « m » est un monôme non premier de f(X), il existe toujours un monôme premier qui contient « m » et qui s'écrit avec moins de littéraux que « m ».

5.1.3. Base première d'une fonction

Définition : Une forme ∑Π d'une fonction f(X) est appelée ***base première*** si elle est composée uniquement de monômes premiers de f(X).

Définition : Une forme ∑Π d'une fonction f(X) est appelée ***base première complète*** si elle est composée de tous les monômes premiers de f(X).

Exemple :Soit la fonction Id(f) = 0,2,6,7,8,9,10,11,15. Sur la figure 5.3 sont représentées 2 bases premières de cette fonction. La première n'est pas complète (il manque des monômes premiers). La seconde est complète (elle contient tous les monômes premiers)

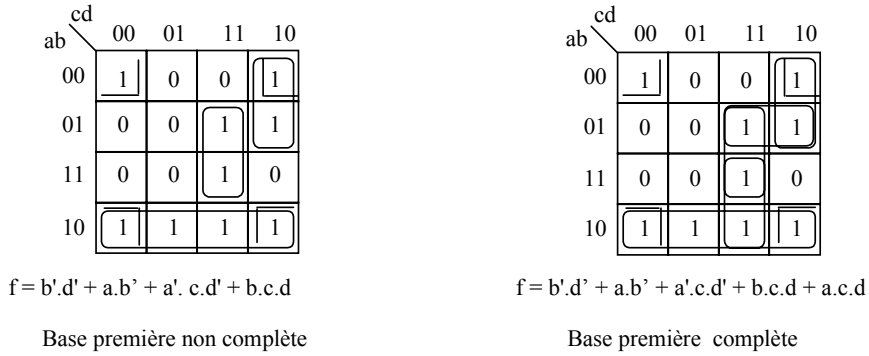


Figure 5.3. *Bases premières complète et non complète.*

Remarque : Il peut exister plusieurs bases premières d'une fonction mais la base première complète est évidemment unique.

Définition : Une base première est dite ***irrédondante*** si elle cesse d'être une base première lorsqu'on lui enlève un de ses monômes.

Exemple : Soit la fonction Id(f) = 0,2,6,7,8,9,10,11,15. Sur la figure 5.4 sont représentées 2 bases premières de cette fonction. La première est irredondantepas (on ne peut pas lui enlever de monôme). La seconde est redondante (le monôme acd peut être supprimé)

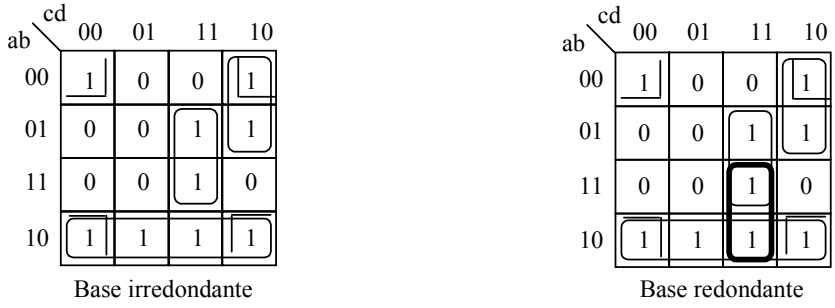


Figure 5.4. *Bases redondante et irrédondante.*

Définition : Un monôme premier est ***obligatoire*** dans une base s'il est le seul parmi tous les monômes de la base à couvrir au moins un point.

Remarque : Tous les monômes d'une base irrédondante sont obligatoires.

Définition : Un monôme premier est ***essentiel*** s'il est obligatoire dans la base première complète.

Remarque : Un monôme premier essentiel appartient forcément à toutes les bases.

5.1.4. Principe de minimisation

Sachant que tout monôme non premier d'une fonction f est inclus dans au moins un monôme premier de f, on voit bien que toute expression minimale de f s'écrit comme une somme de monômes premiers. On peut montrer dans le cas général qu'il n'est pas possible d'obtenir directement une expression minimale d'une fonction. Aussi, les méthodes de minimisation se décomposent-elles en deux étapes :

1. Tout d'abord, la recherche de la base première complète de la fonction qui contient tous les termes susceptibles d'intervenir dans une expression minimale.
2. Puis ensuite, une procédure de choix sur l'ensemble des termes obtenus pour aboutir à une expression de la fonction qui minimise le coût (nombre de monôme dans notre cas).

5.2. Recherche d'une base première

5.2.1. Méthode de Karnaugh

La méthode de Karnaugh est une méthode essentiellement visuelle qui consiste à appliquer la définition d'un monôme premier que nous avons donnée. Dans la pratique, elle n'est vraiment applicable que pour les fonctions d'un nombre réduit de variable (4 ou 5 au maximum).

5.2.1.a. Cas des fonctions simples complètement spécifiées

Dans un espace à n dimensions un monôme « m » écrit avec n-p variables contient 2^p points, chacun de ces points étant adjacent à p autres points de « m ».

Sur la table de quatre variables suivante est représenté en noir le monôme de trois variables m = a.c.d' (n=4, p=1). Il contient deux points (2^p=2) donc deux cases de la table. Chaque point de « m » est adjacent à un (p=1) autre point de « m ». Les diviseurs de « m » par une variable sont également représentés sur la table (a.c, a.d', c.d').

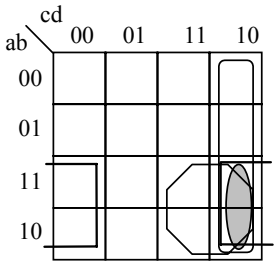


Figure 5.5. *Monômes premiers et base première*

La méthode consiste donc à rechercher pour chaque sommet vrai de f les monômes groupant le plus grand nombre possible de cases vraies avec ce sommet. La facilité d'application de cette méthode repose en grande partie sur la faculté de reconnaître, dans le quadrillage de la table, les formes possibles pour des monômes :

- de 4 variables (1 case),
- de 3 variables (2 cases adjacentes),
- de 2 variables (4 cases adjacentes),
- de 1 variable (8 cases adjacentes).

Exemple : Soit la fonction f de quatre variables (a de poids fort) définie par son image décimale :

Id[f(a,b,c,d)] = R1(0,2,6,7,8,9,10,11,12,15)

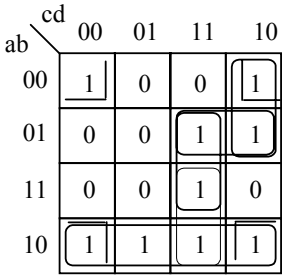


Figure 5.6. *Base première complète*

Sur la table précédente sont représentés les monômes premiers de f qui sont :

(0,2,8,10) -> b'.d'
(8,9,10,11) -> a.b'
(8,12) -> a.c'.d'
(11,15) -> a.c.d
(6,7) -> a'.b.c
(7,15) -> b.c.d
(2,6) -> a'.c.d'

La base première complète de f s'écrit donc :

Bp(f) = b'.d' + a.b' + a.c'.d' + a.c.d + a'.b.c + b.c.d + a'.c.d'

Les monômes suivants sont des monômes premiers essentiels :
b'.d'
a.b'
a.c'.d'

5.2.1.b. Cas des fonctions simples incomplètement spécifiées

Une fonction incomplètement spécifiée f* peut être définie par une borne inférieure f_{inf} et une borne supérieure f_{sup}. Le problème de la minimisation d'une telle fonction est la recherche de la fonction complètement spécifiée f comprise entre f_{inf} et f_{sup} et associée à une expression de coût minimal.

Définition **:** On appelle ***bases premières supérieures de f****, les bases premières de f_{sup} et bases premières inférieures de f*, les bases premières de f_{inf}.

Remarque **:** Tout monôme d'une fonction f avec $f_{inf} \leq f \leq f_{sup}$ est inférieur ou égal à au moins un monôme premier de f_{sup}.

L'expression minimale ∑Π de f* est donc une somme de monômes intervenant dans la base première complète supérieure. Les méthodes de minimisation s'appliquent donc en recherchant dans un premier temps la base première complète supérieure. D'autre part, f ≤ f*, il suffit donc pour que la fonction choisie vérifie les deux inégalités précédentes qu'elle couvre les points vrais de la borne inférieure f_{inf}. On appellera finalement base première complète d'une fonction non complètement définie, la base première complète supérieure restreinte au monômes couvrant au moins un point vrai de la fonction.

Exemple **:** Soit la fonction f de quatre variables (a de poids fort) définie par son image décimale : Id[f(a,b,c,d)] = R1(0,7,10,11,12,15) + Rϕ(2,6,8,9)

Sur les tables suivantes de la figure 5.7. sont représentés les monômes premier de f_{inf} (b) et les monômes premiers de f_{sup} (b). Les bases premières complètes inférieures et supérieures sont donc :

$Bp_{inf}(f)$ = b.c.d + a.c.d + a.b'.c + a'.b'.c'.d' + a.b'.c.d'

$Bp_{sup}(f)$ = b'.d' + a.b' + a.c'.d' + a.c.d + a'.b.c + b.c.d + a'.c.d'

Bp(f) = b'.d' + a.b' + a.c'.d' + a.c.d + a'.b.c + b.c.d

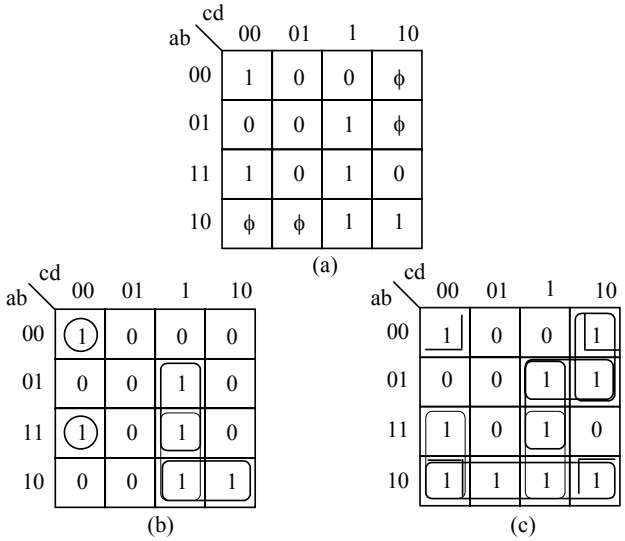


Figure 5.7. *Base première complète inférieure et supérieure*

5.2.1.c. Cas des fonctions multiples

Le problème posé ici est de minimiser une fonction de coût portant non pas sur une fonction, mais sur plusieurs.

Figure 5.8. *Schéma général d'une fonction multiple*

En général on ne peut pas traiter le problème de la minimisation de p fonctions simultanées F = (f1,...,fp) comme p problèmes de minimisation indépendants.

Exemple **:** Soit les fonctions f1 et f2 suivantes :

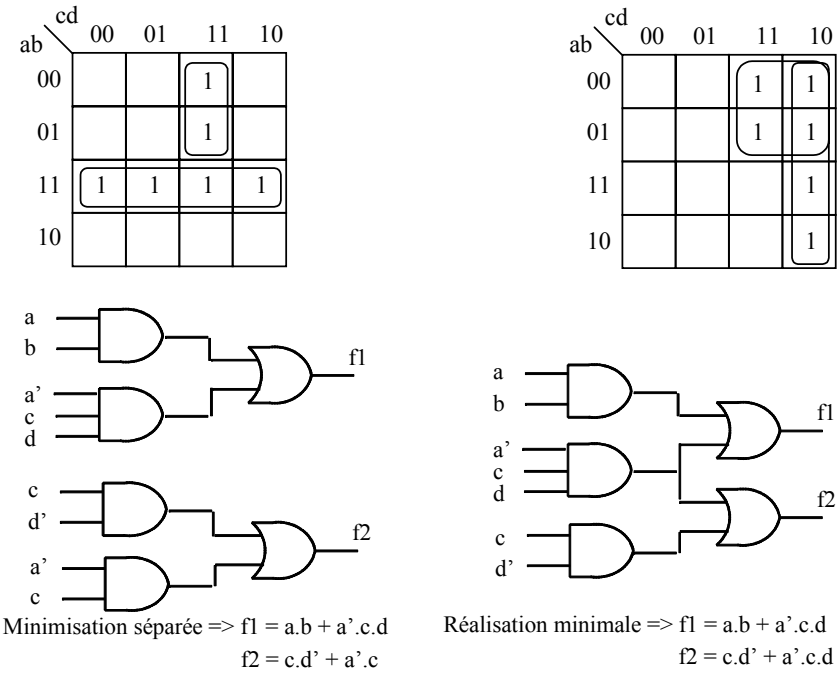


Figure 5.9. *Minimisation d'une fonction multiple*

Soit M = (m1,m2,...,mq-1,mq) un ensemble minimal de monômes distincts tel que toute composante de F puisse s'exprimer comme une somme de termes de M. Remarquons qu'un même terme appartenant à M peut intervenir dans l'expression de plusieurs composantes de F.

Supposons que le coût d'une expression de F soit une fonction du nombre de ses monômes et considérons l'expression particulière de F obtenue en utilisant les éléments de M. Soit un monôme « mi » quelconque de M apparaissant dans p composantes de F, fi1,fi2,...,fip. Le monôme « mi » est donc un monôme de la fonction produit fp = fi1 . fi2 . fip.

Si « mi » n'est pas un monôme premier de fp alors il existe une solution de coût inférieur ou égal au coût de cette expression de F obtenue en remplaçant « mi » par le monôme premier de fp qui contient « mi ».

Nous appellerons fonction produit de F=(f1,...fn) toute fonction obtenue en faisant le produit des composantes de F. Un monôme premier de F est un monôme premier d'une quelconque de ses fonctions produit.

Toute expression minimale de F=(f1,...,fn) s'écrit à l'aide de monômes premiers de F.

Exemple **:** Soit à rechercher les monômes premiers de la fonction F=(f1,f2,f3) de quatre variables (a,b,c,d).

Id(f1) = R1(3,9,10,11,12,13,14,15)

Id(f2) = R1(1,3,5,9,10,13,15)

Id(f3) = R1(1,3,5,6,12,14)

Sur les tables présentées sur la figure 5.10, on fait apparaître les fonctions f1, f2, f3, f1.f2, f1.f3, f2.f3, f1.f2.f3.

Dans la liste des monômes premiers, certains sont trouvés plusieurs fois. Par exemple, abcd est un monôme premier de f1.f2.f3, f1.f2, et f1.f3. Il est évidemment inutile de les considérer comme des monômes premiers différents.

La liste des monômes premiers de F est donnée en marquant chaque terme par la fonction produit dont il est monôme premier. On obtient pour cet exemple :

A = a' b' c d	(3)	-> f1 f2 f3
B = a' b' d	(1,3)	-> f2 f3
C = a' c' d	(1,5)	-> f2 f3
D = a b d'	(12,14)	-> f1 f3
E = a b d	(13,15)	-> f1 f2
F = a c' d	(9,13)	-> f1 f2
G = a b' c d'	(10)	-> f1 f2
H = b c d'	(6,14)	-> f3
I = c' d	(1,5,9,13)	-> f2
J = a b	(12,13,14,15)	-> f1
K = a c	(10,11,14,15)	-> f1
L = a d	(9,11,13,15)	-> f1
M = b' c d	(3,11)	-> f1

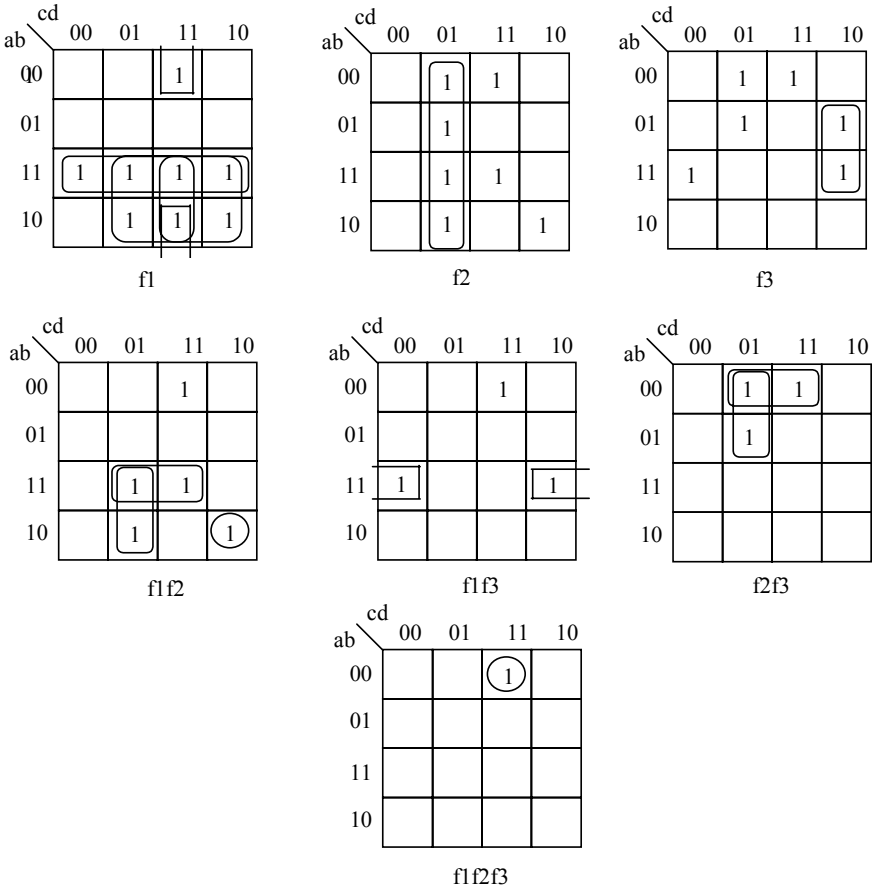


Figure 5.10. *Minimisation d'une fonction multiple*

5.2.2. Méthode de Mc Cluskey

5.2.2.a. Exposé de la méthode

Cette méthode possède l'avantage d'être systématique et donc programmable. Elle consiste, partant de la première forme canonique (∑Π) de la fonction, en l'application itérée de la relation :

(1) $Ax + Ax' = A$ où A est un produit de littéraux.

Soit f une fonction de n variables. Si l'on applique la relation (1) à tous les couples de points vrais de f on obtient alors tous les groupements de deux points contenus dans f, c'est à dire un ensemble de termes contenant n-1 littéraux. Les points de f n'ayant pu être groupés avec d'autre points sont des monômes premiers de f.

Reprenons les termes obtenus à la fin de cette première étape et contenant n-1 littéraux. En leur appliquant la même procédure on obtient l'ensemble des groupements de quatre points contenus dans f et l'on à déterminé les monômes premiers de n-1 littéraux (termes non regroupés).

Au bout d'un nombre fini d'étapes (inférieur à n) il n'existe plus de groupement possible entre les termes restants. Ces termes sont donc aussi de monômes premiers de f.

La base première complète de la fonction est donc la somme des monômes premiers déterminés à chaque étape et des termes restant à la dernière étape.

5.2.2.b. Cas des fonctions simples

Pour effectuer les calculs on peut utiliser une représentation algébrique ou numérique (base 10). La notation algébrique est souvent utilisée en remplaçant les variables sous la forme normale par 1, celles complémentées par 0, et celles absentes par -. Avec cette notation, les calculs débutent avec l'ensemble des combinaisons binaires des points vrais de la fonction.

Tous les points vrais de la fonction sont classés dans un tableau par classes comportant le même nombre de 1 dans leur représentation binaire (ou de variables sous la forme normale). La classe C0 contient le point 0, si ce point est un point vrai de la fonction, la classe C1, toutes les combinaisons ne comportant qu'un seul 1, etc...

Présentation des calculs :

1. Seules deux classes voisines peuvent contenir des combinaisons adjacentes
2. A une étape k quelconque les termes obtenus sont rangés en classes. Deux termes appartiennent à la même classe s'il sont formés à partir de termes provenant de classes dans l'étapes précédentes.
3. Les deux termes qui donnent naissance à un groupement sont marqués comme n'étant pas des monômes premiers de f.

Règles de combinaisons en représentation binaire :

1. On recombine les termes qui n'ont qu'un seul bit différent.
2. On recombine les termes qui ont le tiret au même endroit et n'ont qu'un seul bit différent.
3. On recombine les termes qui ont deux tirets au même endroit et n'ont qu'un seul bit différent.
4. etc ...

Règles de combinaisons en représentation décimale :

1. On recombine les termes $a \in C_i$ et $b \in C_{i+1}$ si la différence a-b est négative et puissance de 2.
2. On recombine les termes $a,b \in C_i$ et $c,d \in C_{i+1}$ si les différences a-b et c-d sont égales, négatives et puissance de 2 et si la différence a-c est négative et puissance de 2.

3. On recombine les termes a,b,c,d $\in C_i$ et e,f,g,h $\in C_{i+1}$ si les différences b-c et f-g sont égales, négatives et puissance de 2 et si la différence a-e est négative et puissance de 2.
4. etc ...

Exemple **:** Soit à rechercher la base première complète de la fonction de quatre variables définie par son image décimale.

Id[f(a,b,c,d)] = R_1 (0,2,6,7,8,9,10,11,12,15)

C0	0	0000	x	C01	0,2	00-0	x	C012	0,2,8,10	-0-0
C1	2	0010	x		0,8	-000	x		0,8,2,10	-0-0
	8	1000	x	C12	2,6	0-10		C123	8,9,10,11	10--
C2	6	0110	x		2,10	-010	x		8,10,9,11	10--
	9	1001	x		8,9	100-	x			
	10	1010	x		8,10	10-0	x			
	12	1100	x		8,12	1-00				
C3	7	0111	x	C23	6,7	011-				
	11	1011	x		9,11	10-1	x			
C4	15	1111	x		10,11	101-	x			
				c34	7,15	-111				
					11,15	1-11				

La base première complète est donc constituée des termes couvrant les sommets (8,9,10,11),(0,2,8,10), (11,15),(7,15),(6,7),(8,12),(2,6) soit :

Bp(f) = a.b' + b'.d' + a.c.d + b.c.d + a'.b.c + a.c'.d' + a'.c.d'

5.2.2.c. Cas des fonctions multiples

Pour traiter les fonctions multiples, on construit la table de Mc Cluskey (rangement par classes) en prenant comme ensemble de départ la réunion des points vrais des fonctions simples.

On rajoute une colonne « étiquette » permettant de marquer l'appartenance ou la non appartenance des monômes aux différentes fonctions (0 si le monôme appartient à la fonction fi et 1 dans le cas contraire).

La procédure de formation des monômes premiers est identique à celle présentée dans le cas des fonction simples. D'autre part, toutes les fois que deux termes sont recombinés, on associe au terme obtenu une étiquette qui est le produit composante à composante des étiquettes des termes originaux. Ces étiquettes permettent de marquer les termes qui ne sont pas des monômes premiers. On marque un terme origine s'il a la même étiquette que le terme qu'il produit.

Remarque **:** Il peut arriver que l'on forme des termes dont l'étiquette est composée uniquement de 0, ils seront évidemment abandonnés car contenus dans aucune des fonctions simples.

Exemple **:** Soit à rechercher les monômes premier de la fonction F=(f1,f2,f3) de quatre variables a,b,c,d (a de poids fort).

Id[f1] = R1(3,9,10,11,12,13,14,15)

Id[f2] = R1(1,3,5,9,10,13,15)

Id[f3] = R1(1,3,5,6,12,14)

				f1f2f3
C1	1	0001	011	x
C2	3	0011	111	
	5	0101	011	x
	6	0110	001	x
	9	1001	110	x
	10	1010	110	
	12	1100	101	x
C3	11	1011	100	x
	13	1101	110	x
	14	1110	101	x
C4	15	1111	110	x

			f1f2f3	
C12	1,3	00-1	011	
	1,5	0-01	011	
	1,9	-001	010	x
C23	3,11	-011	100	
	5,13	-101	010	x
	6,14	-110	001	
	9,11	10-1	100	x
	9,13	1-01	110	
	10,11	101-	110	x
	10,14	1-10	100	x
	12,13	110-	100	x
	12,14	11-0	101	
C34	11,15	1-11	100	x
	13,15	11-1	110	
	14,15	111-	100	x

			f1f2f3
C123	1,5,9,13	--01	010
C234	9,13,11,15	1--1	100
	10,11,14,15	1-1-	100
	12,13,14,15	11--	100

Les monômes premiers de F sont donc :

A = a' b' c d	(3)	-> f1 f2 f3
B = a' b' d	(1,3)	-> f2 f3
C = a' c' d	(1,5)	-> f2 f3
D = a b d'	(12,14)	-> f1 f3
E = a b d	(13,15)	-> f1 f2
F = a c' d	(9,13)	-> f1 f2
G = a b' c d'	(10)	-> f1 f2
H = b c d'	(6,14)	-> f3
I = c' d	(1,5,9,13)	-> f2
J = a b	(12,13,14,15)	-> f1
K = a c	(10,11,14,15)	-> f1
L = a d	(9,11,13,15)	-> f1
M = b' c d	(3,11)	-> f1

5.2.3. Méthode des consensus (Tison)

Les méthodes précédentes partent de la forme canonique de la fonction. Aussi, elles sont inutilisables pour des fonctions de taille importante. La méthode des consensus permet de ne pas redescendre aux points élémentaires de la fonction. Cette méthode est une application directe du théorème des consensus.

5.2.3.a. Définition des consensus

Il existe un consensus entre deux monômes m1 et m2 si une seule des variables apparaissant à la fois dans m1 et m2 est biforme (complémentée dans un monôme, non complémentée dans l'autre). Le consensus est alors égal au produit des variables monoformes de m1 et m2.

Exemple **:**

m1 = a.b.c	m2 = a'.b.d	Consensus (m1,m2) = b.c.d
m1 = a.b.c	m2 = a'.b'.d	Consensus (m1,m2) inexistant

5.2.3.b. Interprétation de consensus

Le théorème des consensus (a.b + a'.c = a.b + a'.c + b.c) peut être interprété de la manière suivante : Lorsque qu'il existe un consensus entre deux monômes ce consensus peut être ajouté à la fonction sans que cela ne change la fonction.

Les exemple suivants permettent d'illustrer ce qu'est effectivement un consensus.

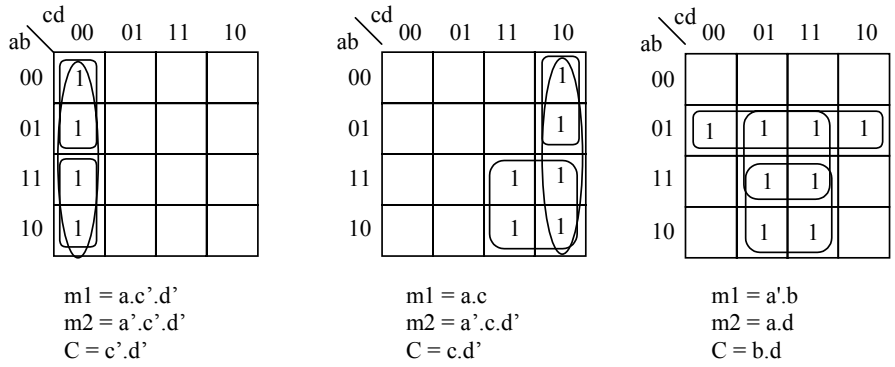


Figure 5.11. *Interprétation des consensus*

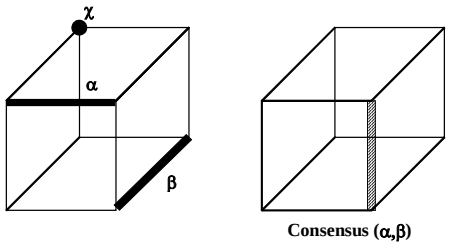


Figure 5.12. *Interprétation des consensus dans l'hypercube*

5.2.3.c. Exposé de la méthode

Soit une somme de monômes (quelconque) représentant une fonction logique. En déterminant tous les consensus possible entre les monômes, en les rajoutant à la fonction et en ne conservant que les monômes premiers (monômes non inclus dans d'autres monômes), on obtient toutes les possibilités de couverture de la fonction avec des monômes premiers, c'est à dire la base première complète. Deux algorithmes permettent de trouver ainsi une base première complète.

Algorithme 1 :

1. Suppression des multiples et des monômes inclus dans d'autres par application des théorèmes d'idempotence (a+a=a) et d'absorbtion (a.b+a=a)
2. Pour chaque monôme, on forme les consensus avec les monômes suivants, on rajoute ces consensus à la fonction et on supprime les multiples et monômes inclus.

Algorithme 2 :

Cet algorithme est une variante du précédent. La recherche s'effectue par étapes successives, chaque étape étant relative à une variable biforme dans les monômes dont on dispose.

On considère une variable biforme, on a et on forme des monômes

1. Suppression des multiples et des monômes inclus dans d'autres par application des théorèmes d'idempotence (a+a=a) et d'absorbtion (a.b+a=a)
2. Pour chaque variable biforme faire l'étape 3
3. Pour chaque monôme, on forme avec les monômes suivants les consensus par rapport à la variable biforme en cours de traitement, on rajoute ces consensus à la fonction et on supprime les multiples et monômes inclus.

Exemple **:** Id[f(a,b,c,d)] = R_1 (0,2,6,7,8,9,10,11,12,15)

f = a'b'c'd' + a'b'c d' + a'b c d' + a'b c d + a b'c'd' + a b'c'd + a b'c d' + a b'c d + a b c'd' + a b c d

Consensus par rapport à a => a'b'c'd' et a b'c'd' donnent b'c'd'
a'b'c d' et a b'c d' donnent b'c d'
a b c d et a'b c d donnent b c d

On rajoute b'c'd' b'c d' b c d
On élimine a'b'c'd' a b'c'd' a'b'c d' a b'c d' a'b c d a b c d
f= a'b c d' + a b'c'd + a b'c d + a b c'd' + b'c'd' + b'c d' + b c d
Consensus par rapport à b => a'c d' + a c d + a c'd'
f = a b'c'd + b'c'd' + b'cd' + b c d + a'cd' + a c d + a c'd'
Consensus par rapport à c => a b'd + b'd' + a'b'd' + a b'd'
f = b c d + a'c d' + a c d + a c'd' + a b'd + b'd'
Consensus par rapport à d => a'b c + a b'c + a b'c' + a b'
f = b c d + a'c d' + a c d + a c'd' + b'd' + a'b c + a b'

5.2.3.d. Cas des fonctions multiples.

Pour traiter les fonctions multiples, on construit une fonction qui est la somme des fonctions de départ.

On rajoute à chaque monôme une « étiquette » permettant de marquer l'appartenance ou la non appartenance des monômes aux différentes fonctions (0 si le monôme appartient à la fonction fi et 1 dans le cas contraire).

La procédure de formation des monômes premiers est identique à celle présentée dans le cas des fonction simples. D'autre part, toutes les fois que deux termes sont recombinés, on associe au consensus obtenu une étiquette qui est le produit composante à composante des étiquettes des termes originaux. La seule modification à apporter à la procédure concerne l'élimination des monômes identiques ou inclus. En effet, un monôme ne peut être éliminé que s'il est identique ou inclu dans un monôme de même étiquette ou s'il a une étiquette composée uniquement de 0.

Exemple **:** Soit la fonction logique F de 4 variables a,b,c,d définies par :
Id[f1] = R_1 (3,7,12,13,14,15)
Id[f2] = R_1 (2,3,6,7,10,14)

F = a'b'cd' + a'b'cd + a'bcd' + a'bcd + ab'cd' + ab'cd + abc'd + abcd' + abcd
01 11 01 11 01 10 10 11 10
B2/a => b'cd' + bcd' + bcd
01 01 10
F = a'b'cd + a'bcd + abc'd + abc'd + abcd' + b'cd' + bcd' + bcd
11 11 10 10 11 01 01 10

B2/b => a'cd + a'cd + acd' + cd'
11 10 01 01
F = abc'd' + abc'd + abcd' + bcd + a'cd + cd'
10 10 11 10 11 01
B2/c => abd' + abd' + abd
10 00 10
F = abcd' + bcd + a'cd + cd' + abd' + abd
11 10 11 01 10 10
B2/d => abc + abc + bc + abc + a'c + abc + ab
10 10 00 10 01 00 10
F = abcd' + bcd + a'cd + cd' + a'c + ab
11 10 11 01 01 10

5.2.4. Obtention d'une base première complète par amélioration de la méthode des consensus

Cette méthode de recherche d'une base première complète s'appuie sur la méthode de Tison à base de consensus.

Algorithme **:** On part d'une base première de la fonction. On forme le consensus du monôme courant avec les monômes qui le suivent dans la liste courante. Si le consensus est inclus dans un monôme présent dans la liste, on le supprime, sinon on l'agrandit au maximum jusqu'à obtenir un monôme premier et on le met en bout de liste. Ce processus est répété jusqu'à ce qu'il n'y ait plus de nouveau monôme créé par consensus.

Exemple : f = R1 (0,2,6,7,8,9,10,11,12,15)

ab \ cd	00	01	11	10
00	1	0	0	1
01	0	0	1	1
11	1	0	1	0
10	1	1	1	1

Partons d'une expression quelconque de la fonction.

f = b'd' + a'cd' + bcd + ab' + ac'd'
Consensus (2,3) = a'bc pas d'agrandissement ; monôme premier
Consensus (2,4) = b'cd' inclus dans (1)
Consensus (3,4) = acd pas d'agrandissement ; monôme premier

La base première complète est :

f (a,b,c,d) = b'd' + a'cd' + bcd + ab' + ac'd' + a'bc + acd

Remarque : Dans cette méthode, contrairement à celle de Tison, le nombre de monômes présents au cours de l'algorithme ne dépasse jamais le nombre de monômes de la base première complète. De plus, après la génération d'un consensus, il n'est pas nécessaire de tester l'inclusion d'un des monômes présents dans ce nouveau monôme puisque tous les monômes présents sont déjà premiers. Cette méthode pose par contre le problème du test d'inclusion (déterminer si un monôme est inclus dans la fonction).

5.3. Recherche de bases irrédondantes et minimales

La détermination des monômes premiers d'une fonction f étant achevée, on aboutit à un ensemble de termes M=(m1,...mq). Chaque terme « mi » est inclus dans f et couvre un sous ensemble des points de la fonction. On se pose alors le problème de choisir le (ou les) sous-ensemble de M qui permet d'exprimer la fonction et qui minimise son coût. Ces expressions sont appelées bases minimales de la fonction. Les méthodes proposée dans ce chapitre permettent de déterminer ces bases minimales (ou bases irrédondantes lorsqu'elles ne sont pas appliquées à partir de la base première complète, mais d'une base première quelconque).

Remarque **:** Une base minimale est une base irrédondante. L'inverse n'est pas nécessairement vrai.

Exemple **:** Sur la figure 5.13 sont présentées deux couvertures d'une même fonction. La première est une base irrédondante mais qui n'est pas minimale.

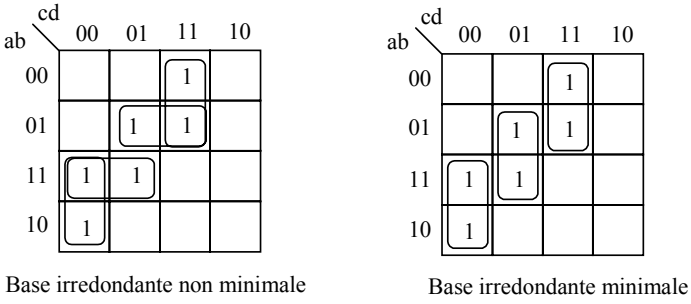


Figure 5.13. *Base Irredondante / Base minimale*

5.3.1. Table de choix

5.3.1.a. Cas des fonctions simples

Le problème peut se mettre sous la forme d'une table à deux entrées. On porte sur les colonnes les sommets vrais de la fonction (représentés par leur équivalent décimal) et sur les lignes les monômes premiers de la fonction (base première complète supérieure lorsque la fonction n'est pas complètement définie).

A l'intersection de la ligne M et du sommet C, on porte une marque (X) si le sommet C est couvert par le monôme M.

Exemple **:** Reprenons l'exemple traité dans la partie 5.2.1

Id[f(a,b,c,d)] = R1(0,7,10,11,12,15) + Rϕ(2,6,8,9)

Bp(f) = b'.d' + a.b' + a.c'.d' + a.c.d + a'.b.c + b.c.d

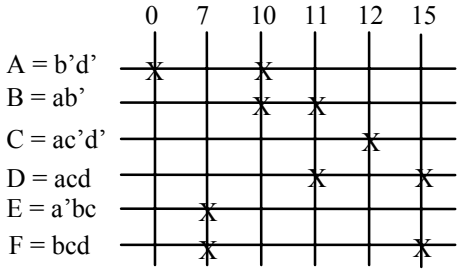


Figure 5.14. *Table de choix*

Le problème que l'on se pose ici est de trouver un ensemble minimum de monômes qui couvre l'ensemble des sommets vrais de la fonction.

Pour cela, nous identifierons en premiers lieu les monômes premiers essentiels. Un monôme est premier essentiel s'il est seul à couvrir un sommet de la fonction (une seule croix dans une colonne). Ces monômes apparaissent nécessairement dans toute écriture de la fonction et en particulier dans toute base minimale de la fonction.

Dans l'exemple précédent les monômes premiers essentiels sont b.d et a.c.d.

Les monômes premiers essentiels étant déterminés, on élimine les sommets couverts par ces monômes.

Pour couvrir les sommets restant il existe plusieurs possibilités, des choix sont donc à réaliser. Ces choix peuvent être fait de manière complètement aléatoires ou menés par des heuristiques plus ou moins performantes selon les cas. Quoi qu'il en soit, on ne peut être sûr qu'un choix fait à un instant donné ne réagisse pas sur la solution finale, c'est à dire que rien ne permet d'affirmer que la solution obtenue à partir des choix réalisés est optimale.

Dans tous les cas, la procédure de recherche de cet ensemble peut être menée de la façon suivante :

1. Prendre un sommet non encore couvert (choix aléatoire ou dirigé par une heuristique),
2. Prendre un monôme couvrant ce sommet (choix aléatoire ou dirigé par une heuristique),
3. Éliminer tous les sommets couverts par ce monôme,
4. Réitérer la procédure jusqu'à avoir couvert tous les sommets.

Exemple **:** Reprenons l'exemple précédent.
Monômes premiers essentiels : b'.d' et a.c'.d'.
Sommets couverts par ce monôme : 0,10,12

Sommets restant à couvrir : 7,11,15
Monômes couvrant le sommet 7 (1er de la liste) : a'.b.c, b.c.d
Sommets couverts par le monôme a b c (1er de la liste) : 7

Sommets restant à couvrir : 11,15
Monômes couvrant le sommet 11 (1er de la liste) : a.b', a.c.d
Sommets couverts par le monôme a b (1er de la liste) : 11

Sommets restant à couvrir : 15
Monômes couvrant le sommet 15 : a.c.d, a'.b.c
Sommets couverts par le monôme a.c.d (1er de la liste) : 15

Sommets restant à couvrir : aucun
Bm = b'.d' + a.c'.d' + a'.b.c + a.b' + a.c.d

Remarque **:** Cette méthode n'est pas une méthode exacte dans le sens ou elle ne permet pas d'assurer que l'expression obtenue est minimale. Toutefois cette approche est utilisée dans de nombreuses applications ne nécessitant pas nécessairement l'obtention du minimum vrai de la fonction.

D'autre part, cet exemple à été traité sans critères particuliers sur les choix effectués. Les choix se sont toujours portés sur le premier élément apparaissant dans la liste des éléments (monômes ou sommets à choisir). Il est évident qu'une telle approche n'est pas optimale quand à l'expression obtenue. Afin d'améliorer le résultat,

sans toutefois affirmer que la solution sera minimale, les choix peuvent être guidés par des critères plus judicieux qu'un simple choix aléatoire.

5.3.1.b. Critères de choix

Plusieurs critères peuvent être utilisés pour guider les choix à effectuer. A titre d'exemple, nous présentons une heuristique qui s'avère relativement bonne dans la plupart des cas même si elle ne conduit pas toujours à la solution optimale.

Cette procédure de recherche d'une couverture minimale s'exprime de la façon suivante :

1. Affecter un poids à chaque sommet (nombre de monômes couvrant chacun des sommets),
2. Affecter un poids à chaque monôme (nombre de sommets que couvre chaque monôme),
3. Prendre les sommets de poids minimum (choisir le premier de la liste s'il y en a plusieurs),
4. Prendre le monôme de poids maximum couvrant ce sommet (choisir le premier de la liste s'il y en a plusieurs),
5. Éliminer tous les sommets couverts par ce monôme,
6. Mettre à jours le poids des monômes,
7. Réitérer la procédure jusqu'à avoir couvert tous les sommets.

Exemple **:** Reprenons l'exemple précédent

Sommet de poids minimum (poids 1) : 0,12
Monômes couvrant le sommet 0 (1er de la liste) : b'.d'
Sommets couverts par ce monôme : 0,10
Mise à jours du poids des monômes (décrémentation du poids de a.b')

Sommets restant à couvrir de poids minimum (poids 1) : 12
Monômes couvrant le sommet 12 : a.c'.d'
Sommets couverts par ce monôme : 12
Mise à jours du poids des monômes (rien)

Sommets restant à couvrir de poids minimum (poids 2) : 7,11,15
Monômes couvrant le sommet 7 (1er de la liste) : a'.b.c, b.c.d
Monôme de poids maximum : b.c.d
Sommets couverts par le monôme b.c.d : 7,15
Mise à jours du poids des monômes (décrémentation du poids de a'.b.c et a.c.d)

Sommets restant à couvrir de poids minimum (poids 2) : 11
Monômes couvrant le sommet 11 : a.b', a.c.d
Monôme de poids maximum : a.b', a.c.d
Sommets couverts par le monôme a b' (1er de la liste) : 11

Sommets restant à couvrir : aucun
Bm = b'.d' + a.c'.d' + b.c.d + a.b'

5.3.1.c. Cas des fonctions multiples

Dans le cas des fonctions multiples, la table de choix est également une table à deux entrées.

On porte sur les colonnes les sommets vrais de chacune des fonctions (représentés par leur équivalent décimal) et sur les lignes les monômes premiers de la fonction multiple auxquels on joint un marquage indiquant la fonction produit dont il est monôme premier.

A l'intersection de la ligne M et du sommet C de la fonction Fi, on porte une marque (X) si le monôme M intervient sur la fonction Fi et couvre le sommet C.

La procédure de recherche d'une base minimale est alors identique à celle développée dans le cas des fonctions simples.

Exemple **:** Reprenons l'exemple de fonction multiple présenté au paragraphe 5.2..

Id(f1) = R1(3,9,10,11,12,13,14,15)

Id(f2) = R1(1,3,5,9,10,13,15)

Id(f3) = R1(1,3,5,6,12,14)

La base première complète est constituées des monômes premiers suivants :

A = a' b' c d	(3)	-> f1 f2 f3
B = a' b' d	(1,3)	-> f2 f3
C = a' c' d	(1,5)	-> f2 f3
D = a b d'	(12,14)	-> f1 f3
E = a b d	(13,15)	-> f1 f2
F = a c' d	(9,13)	-> f1 f2
G = a b' c d'	(10)	-> f1 f2
H = b c d'	(6,14)	-> f3
I = c' d	(1,5,9,13)	-> f2
J = a b	(12,13,14,15)	-> f1
K = a c	(10,11,14,15)	-> f1
L = a d	(9,11,13,15)	-> f1
M = b' c d	(3,11)	-> f1

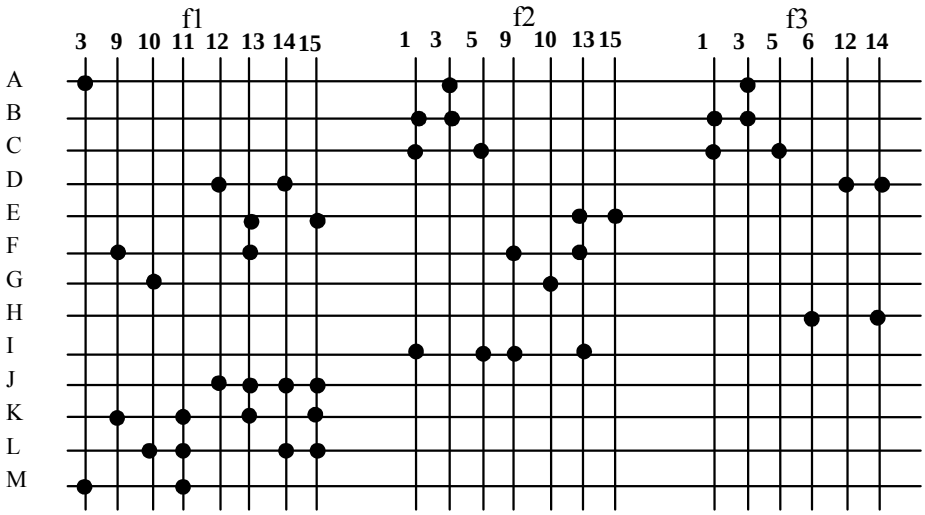


Figure 5.15. *Table de choix multiple*

Sommet de poids minimum (poids 1) : f2(10,15),f3(5,6,12)
Monômes couvrant ces sommets (monômes 1er essentiels) : G,E,C,H,D
Sommets couverts par ces monômes :
f1(10,12,13,14,15),f2(1,5,10,13,15),f3(1,5,6,12,14)
Mise à jours du poids des monômes

Sommets restant à couvrir de poids minimum (poids 2) : f1(3,9),f2(3,9),f3(3)
Monômes couvrant le sommet f1(3) (1er de la liste : A(3),M(2)
Monôme de poids maximum : A
Sommets couverts par le monôme A : f1(3),f2(3),f3(3)
Mise à jours du poids des monômes

Sommets restant à couvrir de poids minimum (poids 2) : f1(9),f2(9)
Monômes couvrant le sommet f1(9) (1er de la liste) : F(2),K(2)
Monôme de poids maximum : F (poids identique =>1er de la liste)
Sommets couverts par le monôme F : f1(9),f2(9)
Mise à jours du poids des monômes

Sommets restant à couvrir de poids minimum (poids 3) : f1(11)
Monômes couvrant le sommet f1(11) : K(1),L(1),M(1)
Monôme de poids maximum : K (poids identique =>1er de la liste)
Sommets couverts par le monôme K : f1(11)
Mise à jours du poids des monômes

Sommets restant à couvrir : aucun
Bm = G+E+C+H+D+A+F+K

Les fonctions f1,f2,f3 sont constituées des monômes suivants :
f1 = A+D+E+F+G+K = a'.b'.c.d + a.b.d' + a.b.d + a.c'.d + a.b'.c.d' + a.c
f2 = A+C+E+F+G = a'.b'.c.d + a'.c'.d + a.b.d + a.c'.d + a.b'.c.d'
f3 = A+C+D+H = a'.b'.c.d + a'.c'.d + a.b.d' + b.c.d'

5.3.2. Résolution algébrique

En remarquant que pour réaliser une fonction il faut couvrir tous les sommets vrais et que pour couvrir chaque sommet, on peut prendre l'un des monômes premiers couvrant ce sommet, il est possible de déterminer les bases minimales d'une fonction par un simple calcul algébrique. Ce calcul est présenté sur l'exemple suivant :

Exemple **:** Reprenons l'exemple du paragraphe 5.2.1
Id[f(a,b,c,d)] = R1(0,7,10,11,12,15) + Rϕ(2,6,8,9)
Bp(f) = b'.d' + a.b' + a.c'.d' + a.c.d' + a'.b.c + b.c.d

A = b'.d' -> 0,2,8,10
B = a.b' -> 8,9,10,11
C = a.c'.d' -> 8,12
D = a.c.d -> 11,15
E = a'.b.c -> 6,7

F = b.c.d -> 7,15

Sommets à couvrir : 0,7,10,11,12,15

Pour couvrir le sommet 0 il faut prendre le monôme A, et pour couvrir le sommet 7, il faut prendre le monôme E ou (inclusif) le monôme F, etc... En répétant le même raisonnement pour chaque sommet vrai de la fonction, on obtient :

F = (A).(E+F).(A+B).(B+D).(C).(D+F)

Développons cette expression.
F = AC.(E+F).(B+D).(D+F)
F = AC.(E+F).(BD+BF+D+DF)
F = AC.(E+F).(D+BF)
F = AC.(ED+EBF+FD+FB)
F = AC.(ED+FD+FB)
F = ACED + ACFD + ACFB

Pour couvrir la fonction, il faut donc prendre les monômes ACED, ACFD ou ACFB.

Il existe donc trois bases minimales (composées de 4 monômes) qui sont :
Bm1 = b'.d' + a.c'.d' + a'.b.c + a.c.d
Bm2 = b'.d' + a.c'.d' + b.c.d + a.c.d
Bm3 = b'.d' + a.c'.d' + b.c.d + a.b'

Remarque **:** Cette approche par résolution algébrique est une solution qui permet d'obtenir toutes les bases minimales d'une fonction mais qui peut s'avérer irréaliste dès que le nombre de monômes et de sommets est élevé.

5.3.3. Méthode des consensus

Une base minimale peut être déterminée en affectant un identificateur (lettre) à chaque monôme de la base première et en réappliquant la méthode des consensus définie précédemment.

Exemple **:** Id[f(a,b,c,d)] = R_1 (0,2,6,7,8,9,10,11,12,15)

f = a'b'c'd' + a'b'cd' + a'bcd' + a'bcd + ab'c'd' + ab'c'd + ab'cd' + ab'cd + abc'd' + abcd

La base première complète de F est :

f = bcd + a'cd' + acd + ac'd' + b'd' + a'bc + ab'

On considère la fonction :

f = A bc d + B a'cd' + C acd + D ac'd' + E b'd' + F a'bc + G ab'

Consensus par rapport à d => AB a'bc + CE ab'c

a.b'.c n'est pas un des monômes premier (ab'c $\subset$ ab') => CE ab'c n'a pas à être ajouté à la fonction.

f = A bcd + B a'cd' + C acd + D ac'd' + E b'd' + (F+AB) a'bc + G ab'

Consensus par rapport à c => DCE ab'd'

ab'd' $\subset$ b'd' =>DCE ab'd' n'a pas à être ajouté à la fonction.

f = A bcd + B a'cd' + C acd + D ac'd' + E b'd' + (F+AB) a'bc + G ab'

Consensus par rapport à b => AG acd + E(F+AB) a'cd'

f = A bcd + (B+EF) a'cd' + (C+AG) acd + D ac'd' + E b'd' + (F+AB) a'bc + G ab'

Consensus par rapport à a => (F+AB)(C+AG) bcd + G(B+EF) b'cd'

b'cd' $\in$ b'd' => G(B+EF) b'cd' n'a pas à être ajouté à la fonction.

f = (A+FC) bcd + (B+EF) a'cd' + (C+AG) acd + D ac'd' + E b'd' + (F+AB) a'bc + G ab'

Interprétation :

Le monôme bcd est couvert par A mais peut aussi être couvert par F et C

Le monôme a'cd' est couvert par B mais peut aussi être couvert par E et F

Le monôme acd est couvert par C mais peut aussi être couvert par A et G

Le monôme ac'd' est couvert par D

Le monôme b'd' est couvert par E

Le monôme a'bc est couvert par F mais peut aussi être couvert par A et B

Le monôme ab' est couvert par G

Pour trouver toutes les bases minimales, on considère les identificateurs comme des variables booléennes et on développe le produit des parenthèses.

(A+FC) (B+EF) (C+AG) D E (F+AB) G

DEG (A+FC) (B+F) (C+A) (F+AB)

DEG (AB + AF + CF) (CF + AF + AB)

DEG (AB + AF + CF)

Il y a trois bases minimales :

F = ac'd' + b'd' + ab' + bcd + a'cd'

F = ac'd' + b'd' + ab' + bcd + a'bc

F = ac'd' + b'd' + ab' + acd + a'bc

5.3.4. Algorithme de type « branch and bound »

Dans la plupart des cas, disposer de toutes les bases minimales n'est pas nécessaire, une solution suffit. Une approche basée sur un algorithme de type « branch and bound » permet de déterminer une base minimale en résolvant le problème de manière exacte.

On affecte une variable logique Mi à chaque monôme premier. Mi=1 indique la présence du monôme correspondant dans l'expression de la fonction. Mi=0 indique l'absence de ce monôme.

Minimiser la fonction logique revient alors à minimiser une fonction de coût Z = $\sum$ Mi sous contraintes. Les contraintes représentent le fait que chaque sommet doit être couvert.

Exemple **:** Reprenons l'exemple du paragraphe 5.2.1

Id[f(a,b,c,d)] = R1(0,7,10,11,12,15) + Rϕ(2,6,8,9)

Bp(f) = b'.d' + a.b' + a.c'.d' + a.c.d + a'.b.c + b.c.d

A = b'.d' -> 0,2,810
B = a.b' -> 8,9,10,11
C = a.c'.d' -> 8,12
D = a.c.d -> 11,15
E = a'.b.c -> 6,7
F = b.c.d -> 7,15

La table de recouvrement peut être présentée par une matrice d'éléments a_{ij} (0 ou 1) de la façon suivante :

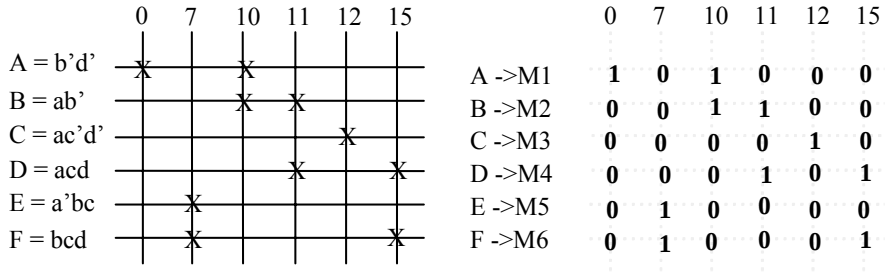


Figure 5.16. *Table de choix et matrice de recouvrement*

Minimiser la fonction logique revient alors à minimiser une fonction Z = M1 + M2 + M3 + M4 + M5 + M6 sous la contrainte suivante :

$$\forall j \text{ (colonne de la matrice)}, \ \sum_i a_{ij}.M_i \geq 1$$

Ces contraintes peuvent s'exprimer de la manière suivante :

M1 ≥ 1
M5 + X6 ≥ 1
M1 + M2 ≥ 1
M2 + M4 ≥ 1
M3 ≥ 1
M4 + M6 ≥ 1

Ces contraintes deviennent :

M1 = 1
M3 = 1
M5 + X6 ≥ 1
M2 + M4 ≥ 1
M4 + M6 ≥ 1

Les monômes M1 et M2 sont des monômes premiers essentiels. Minimiser la fonction logique revient alors à minimiser la fonction Z = M2 + M4 + M5 + M6 sous les contraintes suivantes :

M5 + X6 ≥ 1
M2 + M4 ≥ 1
M4 + M6 ≥ 1

Cette opération peut être réalisée à l'aide d'un graphe de la manière suivante :

On initialise la fonction Z à 0

On a 2 possibilités : - prendre la première variable (0)

- ne pas la prendre (1)

Si l'on ne prend pas la première variable Z reste à 0 (valeur précédente)
Si l'on prend la première variable Z passe à 1 (est incrémenté)

On doit alors effectuer deux test :

- Un test d'arrêt « Stop » lorsqu'on ne prend pas la variable (permet de vérifier qu'il y a toujours une solution sans cette variable)
- Un test de fin « OK » lorsqu'on prend la variable (permet de vérifier si une solution est trouvée, c'est à dire si toute les contraintes sont remplies)

Si le test de fin n'est pas satisfait, on continue l'opération à partir du noeud ou la fonction Z à la valeur minimum.

Les conditions d'arrêt (Stop) et de fin (OK) peuvent s'exprimer de la manière suivante :

$$\text{Stop} : \exists\, j \,/\, \sum_{i=1}^{N} aij.Mi + \sum_{i=N+1}^{n} aij = 0$$

$$\text{OK} : \forall\, j \,/\, \sum_{i=1}^{N} aij.Mi \ .$$

i indice correspondant aux monômes (lignes)
j indice correspondant aux sommets (colonne)

Exemple : Une base minimale est ainsi obtenue en rajoutant aux monômes premiers essentiels A et C correspondant à M1 et M3, les monômes D et E correspondant à M4 et M5.

Bm -> A, C, D, E

Bm1 = b'.d' + a.c'.d' + a.c.d + a'.b.c

Remarque **:** L'ordre dans lequel sont considérées les variables intervient sur la complexité du graphe et sur la solution obtenue. Toutefois, il n'intervient en aucun cas sur la « qualité » de la solution obtenue (nombre de monômes).

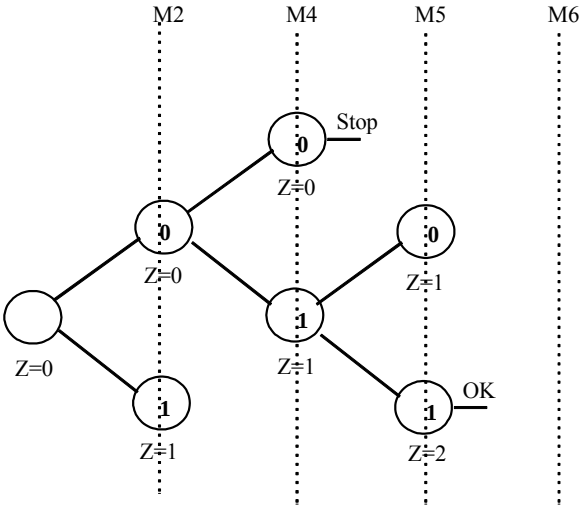


Figure 5.17. *Graphe de décision*

5.4. Méthodes heuristiques basées sur le test d'inclusion

La complexité (problème np-complet) des méthodes précédemment développées fait que ces méthodes ne permettent pas de traiter les « grosses » fonctions logiques. Les méthodes développées pour ces fonctions sont basées sur le principe suivant :

- Prendre un monôme et le rendre premier en l'agrandissant au maximum,
- Supprimer tous les monômes inclus dans ce monôme.

Agrandir un monôme consiste à essayer de lui « enlever » une ou plusieurs variables. Le monôme ainsi agrandi doit être encore inclus dans la fonction. Une fois le monôme ainsi agrandi dans toutes les directions possibles, nous sommes sûr d'avoir un monôme premier.

Algorithme **:**

1. Marquer les monômes de départ avec l'indicateur « non-premier »
2. S'il existe un monôme m marqué « non-premier », le prendre ; Sinon terminer, la liste des monômes restant formant une base de la fonction.
3. S'il existe une des variables de m que l'on a pas encore essayée de supprimer alors, générer le diviseur m* en enlevant cette variable de m. Sinon aller à l'étape 5.
4. Tester l'inclusion de m* dans l'ensemble des monômes. Si le test est positif alors remplacer m par m*. Revenir en 3.
5. Marquer m avec l'indicateur « premier ».
6. Supprimer les monômes marqués « non-premier » inclus dans m*. Revenir à l'étape 2.

Exemple : Id[f(a,b,c,d)] = R1 (0,2,6,7,8,9,10,11,12,15)

ab \ cd	00	01	11	10
00	1	0	0	1
01	0	0	1	1
11	1	0	1	0
10	1	1	1	1

f = a'b'c'd' + a'b'cd' + a'bcd' + a'bcd + ab'c'd' + ab'c'd + ab'cd' + ab'cd + abc'd' + abcd

m = a'b'c'd' Elimination de a' => m* = b'c'd' est inclus dans f
m = b'c'd' Elimination de b' => m* = c'd' n'est pas inclus dans f
m = b'c'd' Elimination de c' => m* = b'd' est inclus dans f
m = b'd' Elimination de d' => m* = b' n'est pas inclus dans f

On remplace a'b'c'd' par b'd'. On supprime a'b'cd' , ab'c'd' , a'b'cd'

f = b'd' + a'bcd' + a'bcd + ab'c'd + ab'cd + abc'd' + abcd

m = a'bcd' Elimination de a' => m* = bcd' n'est pas inclus dans f
m = a'bcd' Elimination de b => m* = a'c'd' est inclus dans f
m = a'cd' Elimination de c => m* = a'd' n'est pas inclus dans f
m = a'cd' Elimination de d' => m* = a'c n'est pas inclus dans f

On remplace a'bcd' par a'cd' .

f = b'd' + a'cd' + a'bcd + ab'c'd + ab'cd + abc'd' + abcd

m = a'b'cd Elimination de a' => m* = bcd est inclus dans f
m = bcd Elimination de b => m* = cd n'est pas inclus dans f
m = bcd Elimination de c => m* = bd n'est pas inclus dans f
m = bcd Elimination de d => m* = bc n'est pas inclus dans f

On remplace a'bcd par bcd. On élimine abcd.

f = b'd' + a'cd' + bcd + ab'c'd + ab'cd + abc'd'

m = ab'c'd Elimination de a => m* = b'c'd n'est pas inclus dans f
m = ab'c'd Elimination de b' => m* = ac'd n'est pas inclus dans f
m = ab'c'd Elimination de c' => m* = ab'd est inclus dans f
m = ab'd Elimination de d => m* = ab' est inclus dans f

On remplace ab'c'd par a b'. On élimine ab'cd.

f = b'd' + a'cd' + bcd + ab' + abc'd'

m = abc'd' Elimination de a => m* = bc'd' n'est pas inclus dans f
m = abc'd' Elimination de b => m* = ac'd' est inclus dans f
m = ac'd' Elimination de c' => m* = ad' n'est pas inclus dans f
m = ac'd' Elimination de d' => m* = ac' n'est pas inclus dans f

On remplace abc'd' par ac'd' .

f = b'd' + a'cd' + bcd + ab' + ac'd'

Dans cet algorithme, le nombre de monômes n'est jamais augmenté. On limite ainsi beaucoup la complexité du processus de minimisation. Par contre, la solution obtenue n'est pas nécessairement la solution optimale. En effet, le résultat peut dépendre de l'ordre dans lequel sont considérés les monômes et les variables. Des heuristiques adaptées permettent toutefois de faire des choix conduisant à des résultats proches de l'optimum.

La principale difficulté de cet algorithme est de déterminer si un monôme est inclus dans une fonction sachant que ce problème peut se ramener, par le théorème d'inclusion, à un problème de calcul de cofacteurs et de preuve de tautologie

Rappel du théorème d'inclusion : Une expression F contient un monôme m si et seulement si le cofacteur de F par rapport à m (F_m) est une tautologie.

$$m \subset F \Leftrightarrow F_m = 1$$

5.5. Un exemple de minimiseur : « ESPRESSO »

ESPRESSO est l'outil de minimisation développé à Berkeley [DeMicheli]. Il exploite les opérations présentées précédemment en réalisant une optimisation itérative de la fonction. L'expression obtenue en sortie d'ESPRESSO est une base première irrédondante de cardinalité minimale dans la plupart des cas.

Les fonctions seront exprimées dans la notation « cube de position ». Nous considérerons qu'elles peuvent être incomplètement définies et nous noterons :

f^1 l'ensemble des monômes tels que f=1, et F^1 une couverture de f^1

f^0 l'ensemble des monômes tels que f=0, et F^0 une couverture de f^0

f^ϕ l'ensemble des monômes tels que f=ϕ, et F^ϕ une couverture de f^ϕ

ESPRESSO part des expressions de F^1 et F^ϕ et calcule tout d'abord le complément F^0. Il applique alors l' opération « Expand » de manière à obtenir une base première, puis l'opération « Irrédondant » afin d'obtenir une base première irrédondante. L'opération « Essentiel » est alors appliquée afin d'extraire les monômes premiers essentiels. ESPRESSO applique alors de manière itérative les opérations « Réduction », « Expansion », « Irrédondant » de façon à chercher une base première irrédondante de moindre cardinalité. Lorsque le gain est nul, ESPRESSO applique les opérations « Réduction » et « Expansion » avec d'autres heuristiques (opération Nouvelle_heuristiques) qui ne seront pas détaillées ici. Ainsi l'algorithme général de ESPRESSO peut s'exprimer de la façon suivante :

```
ESPRESSO (F^1 , F^ϕ)  {
    F^0 = Complément (F^1 ∪ F^ϕ) ;
    F = Expansion (F^1 , F^0) ;
    F = Irrédondant (F , F^ϕ) ;
    E = Essentiel (F , F^ϕ) ;
    F = F - E ;
    F^ϕ = F^ϕ ∪ E ;
    repeat{
            Φ2 = coût (F) ;
            repeat{
                    Φ1 = coût (F) ;
                    F = Réduction (F , F^ϕ) ;
                    F = Expansion (F , F^0) ;
                    F = Irrédondant (F , F^ϕ) ;
            }until coût(F)<Φ1 ;
            F=Nouvelles_heuristiques(F , F^ϕ , F^0) ;
    }until coût(F)<Φ2 ;
    F = F ∪ E ;
    F^ϕ = F^ϕ - E ;
}
```

5.5.1. Opération « Expansion »

L'opération d'expansion est utilisé par la plupart des optimiseurs actuels. Son but est d'accroître la taille de chaque monôme d'une expression de f de manière à couvrir le plus de point de la fonction. L'extension maximum des monômes correspond aux monômes premiers. Le résultat de l'expansion sera donc une base première de la fonction.

L'expansion d'un monôme est réalisée en changeant successivement les 0 du « cube de position » en 1 (ce qui revient à accroître sa taille d'un facteur 2) et en vérifiant à chaque fois que le monôme ainsi obtenu est toujours valide, c'est à dire qu'il est toujours un monôme de f. Cette vérification peut se faire de deux manières différentes :

1 : Intersection du monôme expansé avec F^0.
Intersection vide => monôme valide
Intersection non-vide => monôme non-valide

2 : Vérifier que le monôme étendu couvre $F^1 \cup F^\phi$, cette vérification pouvant se réduire à un problème de preuve de tautologie.

La première approche impose de calculer F^0. Elle est toutefois utilisée dans la plupart des minimiseurs actuel (MINI, ESPRESSO). La seconde approche ne nécessite pas le calcul de F^0 mais conduit à un calcul plus complexe que l'opération d'intersection. Elle est utilisée dans le programme PRESTO.

Exemple : Soit la fonction f(a,b,c) telle que :
$f^1(a,b,c) = a'.b'.c' + a.b'.c' + a'.b.c' + a'.b'.c$
$f^\phi(a,b,c) = a.b.c'.$

a \ bc	00	01	11	10
0	1	1	0	1
1	1	0	0	ϕ

La couverture F^1 de cette fonction f^1 peut s'exprimer ainsi :
10 10 10
01 10 10
10 01 10
10 10 01

La couverture F^ϕ de f^ϕ est :
01 01 10

La couverture F^0 de f^0 peut être calculée à partir de F^1 et F^ϕ. Cette couverture est (voir complémentation) :
01 11 01
11 01 01

Prenons le premiers monôme et expansons le.

10 10 10 => 11 10 10 => (11 10 10) $\cap$ F^0 = Φ =>monôme valide
=> 11 11 10 => (11 11 10) $\cap$ F^0 = Φ =>monôme valide
=> 11 11 11 => (11 11 11) $\cap$ F^0 = F^0 =>monôme non-valide

Le monôme étendu est donc 11 11 10. On remplace (10 10 10) par (11 11 10) et on élimine les monômes de F couvert par ce nouveau monôme, ce qui donne une nouvelle couverture :

11 11 10
10 10 01

Le second monôme et maintenant traité.

10 10 01=> 11 10 01 => (11 10 01) $\cap$ F^0 = (01 10 01) =>monôme non-valide
=> 10 11 01 => (10 11 01) $\cap$ F^0 = (10 01 01) =>monôme non-valide
=> 10 10 11 => (10 10 11) $\cap$ F^0 = Φ =>monôme valide

La couverture F devient donc ;
11 11 10
10 10 11

f = c' + a'.b' est une base première.

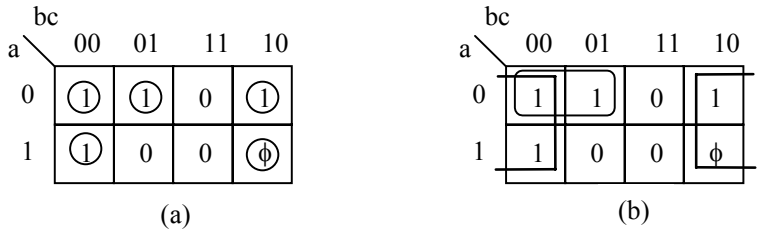


Figure 5.18. *(a) Fonction f. (b) Base première de f(a,b,c).*

Heuristique : Quelque soit l'ordre dans lequel sont considérés les monômes, l'expression obtenue est identique. Toutefois, elle peut être obtenue plus ou moins rapidement en fonction de l'ordre dans lequel sont considérés les monômes. L'heuristique proposée dans ESPRESSO est la suivante : on calcule un vecteur ou chaque élément est la somme de chacune des colonnes de la matrice F représentant la couverture de f, on effectue le produit entre la matrice et ce vecteur transposé. Ceci donne un poids à chaque monôme. Dans l'opération d'expansion les monômes de moindre poids sont considérés en priorité.

Exemple : Soit la couverture d'une fonction f(a,b,c):
10 10 10 (m1)
01 10 10 (m2)
10 01 10 (m3)
10 10 01 (m4)
V = [313131]
F * V^T => (m1) -> 9
(m2) -> 7
(m3) -> 7
(m4) -> 7

Le monôme m2 sera traité en premier par le programme ESPRESSO.

Remarque : L'expression obtenue dépend par contre de l'ordre dans lequel les 0 sont transformées en 1. Dans ESPRESSO une heuristique particulière a été proposée. Elle ne sera pas développée ici.

5.5.2. Opération « Réduction »

L'opération de réduction est utilisé par la plupart des optimiseurs actuels. Son but est de diminuer la taille de chaque monôme d'une expression de f de manière à ce qu'une nouvelle application de l'opération d'expansion conduise à une couverture contenant moins de monômes (couverture de moindre cardinalité).

Remarques : Un monôme réduit reste valide tant que la fonction continue à être couverte. La couverture réduite a le même nombre de monômes que la couverture originale. Les monômes réduits ne sont pas premiers. Un monôme redondant peut être réduit jusqu'à disparaître (mais l'opération de réduction ne garantie pas l'irrédondance de la couverture).

Pour réduire un monôme α de F, on doit enlever de α tous les mintermes couverts par F-{α}. Ceci peut en principe ce faire en calculant l'intersection entre m et le complément de F-{α}, mais cette opération peut conduire à plusieurs monômes. Pour que cela ne conduise qu'à un seul monôme, nous réaliserons l'intersection non pas avec le complément de F-{α}, mais avec le complément du supercube contenant F-{α}. Ce supercube est obtenu en faisant l'union des monômes de F-{α} (opération d'union entre monômes en représentation cube de position).

Exemple : Reprenons la fonction f(a,b,c) telle que :

$f^1(a,b,c)$ = a'.b'.c' + a.b'.c' + a'.b.c' + a'.b'.c

$f^0(a,b,c)$ = a.b.c'.

Après l'opération d'expansion, la couverture F de la fonction f est ;

11 11 10 (α)

10 10 11 (β)

Essayons de réduire le monôme α.

F-{α} => 10 10 11

(F-{α})' => 01 11 11

=> 11 01 11

Supercube [(F-{α})']=> 11 11 11

$\alpha \cap$ Supercube [(F-{α})'] = α => le monôme α ne peut être réduit

Essayons de réduire le monôme β.

F-{β} => 11 11 10

(F-{β})' => 11 11 01

Supercube [(F-{β})']=> 11 11 01

$\beta \cap$ Supercube [(F-{β})'] = 10 10 01 => le monôme β peut être réduit

La couverture F devient donc ;

11 11 10

10 10 01

f = c' + a'.b'.c

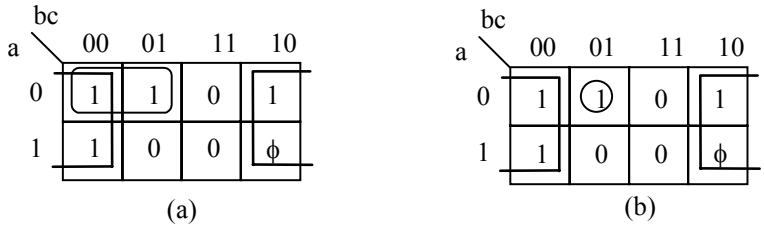


Figure 5.19. *(a) Base première de f. (b) Fonction f après réduction.*

Heuristique : L'expression obtenue dépend de l'ordre dans lequel les monômes sont réduits. Dans ESPRESSO, un poids est affecté à chaque monômes (voir opération d'expansion). Pour l'opération de réduction, les monômes de plus fort poids sont considérés en priorité.

5.5.3. Opération « Irrédondant »

L'opération « Irrédondant » permet d'éliminer les monômes redondants d'une expression de la fonction.

Supposons que la couverture de départ F soit une base première (obtenue par l'opération « Expansion »). La couverture F est séparée en trois sous-ensembles :

- Er : Ensemble *« Essentiel relatif »*. Er contient les monômes couvrant des sommets de la fonction non couvert par d'autres monômes. Ces monômes sont essentiels relativement à la couverture considérée (pas nécessairement la base première complète).
- Rt : Ensemble *« Redondant totalement »*. Rt contient les monômes couvert par Er.
- Rp : Ensemble « *Redondant partiellement »*. Rp contient les monômes restants

La détermination de Er peut se faire simplement. En effet, un monôme α appartenant à F, appartient à Er s'il n'est pas inclus dans $F \cup F^{\phi} - \{\alpha\}$. Pour montrer cette inclusion, il suffit de montrer que $[F \cup F^{\phi} - \{\alpha\}]_{\alpha}$ n'est pas une tautologie.

La détermination de Rt peut se faire manière semblable. Un monôme α appartenant à F, appartient à Rt s'il est inclus dans $Er \cup F^{\phi}$. Pour montrer cette inclusion, il suffit de montrer que $[Er \cup F^{\phi}]_{\alpha}$ est une tautologie.

Rp est alors égal à $F - \{Er \cup Rt\}$.

Exemple : Considérons la fonction f(a,b,c) = a'.b' + b'.c + a.c + a.b + b.c'

La couverture F de la fonction f est ;

10 10 11 (α)
11 10 01 (β)
01 11 01 (χ)
01 01 11 (δ)
11 01 10 (ε)

$[F \cup F^{\phi} - \{\alpha\}]_{\alpha}$ = 11 11 01	Ce n'est pas une tautologie => $\alpha \in$ Er
$[F \cup F^{\phi} - \{\beta\}]_{\beta}$ = 10 11 11 , 01 11 11	C'est une tautologie => $\beta \notin$ Er
$[F \cup F^{\phi} - \{\chi\}]_{\chi}$ = 11 10 11, 11 01 11	C'est une tautologie => $\chi \notin$ Er
$[F \cup F^{\phi} - \{\delta\}]_{\delta}$ = 11 11 01, 11 11 10	C'est une tautologie => $\delta \notin$ Er
$[F \cup F^{\phi} - \{\varepsilon\}]_{\varepsilon}$ = 01 11 11	Ce n'est pas une tautologie => $\varepsilon \in$ Er

L'ensemble Er des monômes essentiels relatifs est :

10 10 11 (α)

11 01 10 (ε)

$[Er \cup F^{\phi}]_{\beta}$ = 10 11 11 C'est n'est pas une tautologie => $\beta \notin Rt$

$[Er \cup F^{\phi}]_{\chi}$ = {} C'est n'est pas une tautologie => $\chi \notin Rt$

$[Er \cup F^{\phi}]_{\delta}$ = 11 11 10 C'est n'est pas une tautologie => $\delta \notin Rt$

L'ensemble Rt des monômes totalement redondants est vide.

L'ensemble Rp des monômes partiellement redondants est :

11 10 01 (β)

01 11 01 (χ)

01 01 11 (δ)

Le problème est maintenant de trouver un sous-ensemble Nr (non redondant) de monômes qui couvre F^1. Cet ensemble Nr contient nécessairement les monômes de Er. Reste à déterminer les autres monômes composant Nr. Pour cela, on peut prendre chaque monôme α de Rp et regarder s'il est contenu dans $[Er \cup Rp \cup F^{\phi} - \{\alpha\}]$ (pour montrer cette inclusion, il suffit de montrer que $[Er \cup Rp \cup F^{\phi} - \{\alpha\}]_{\alpha}$ est une tautologie). Dans l'affirmative, le monôme est éliminé de Rp. En itérant cette opération sur tous les monômes de Rp une couverture non redondante peut être construite ($Nr = Er \cup Rp$).

Exemple : Reprenons l'exemple précédent

Er : 10 10 11(α)
11 01 10(ε)

Rp : 11 10 01(β)
01 11 01(χ)
01 01 11(δ)

$[Er \cup Rp \cup F^{\phi} - \{\beta\}]_{\beta}$ = 10 11 11 , 01 11 11
C'est une tautologie => β est éliminé de Rp

Rp : 01 11 01(χ)
01 01 11(δ)

$[Er \cup Rp \cup F^{\phi} - \{\chi\}]_{\chi}$ = 11 11 01
Ce n'est pas une tautologie => χ est conservé

$[Er \cup Rp \cup F^{\phi} - \{\delta\}]_{\delta}$ = 11 11 10 , 11 11 01
C'est une tautologie => δ est éliminé de Rp

Rp : 01 11 01(χ)

Nr : 10 10 11(α)
11 01 10(ε)
01 11 01(χ)

f = a'.b' + b.c' + a.c

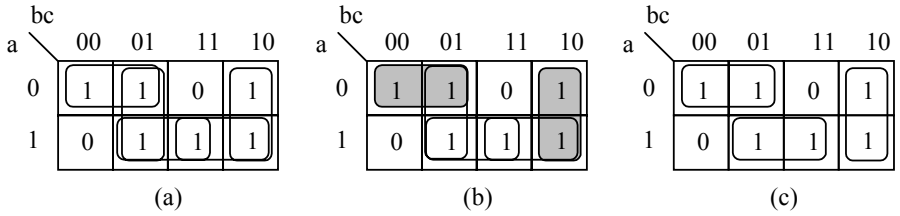


Figure 5.20. *(a) Base première redondante. (b) Monômes essentiels relatifs (c) Base première irrédondante.*

Remarque : L'expression obtenue dépend de l'ordre dans lequel sont considéré les monômes de Rp. Dans ESPRESSO une heuristique particulière a été proposée. Elle ne sera pas développée ici.

5.5.4. Opération « Essentiel »

L'opération « Essentiel » permet de déterminer les monômes premiers essentiels. Une fois déterminé, ces derniers peuvent être retirés du traitement, ce qui simplifie le processus d'optimisation.

La détection des monômes premier essentiels peut se faire par l'intermédiaire du théorème suivant et de son corollaire :

Théorème : Soit $F = G \cup \alpha$ ou α est un monôme premier disjoint de G ($G \cap \alpha = 0$). α est un monôme premier essentiel si et seulement si Consensus(G,α) ne couvre pas α.

Corollaire : α est un monôme premier essentiel si et seulement si $H \cup F^{\phi}$ ne couvre pas α avec :

$$H = \text{Consensus } (((F^{1} \cup F^{\phi})\#\alpha),\alpha)$$

Preuve : La preuve de ces théorèmes ne sera pas faite ici. On peut la trouver dans [DeMicheli].

Le corollaire permet de réduire l'opération de détection des monômes essentiel à un test de contenance qui peut être résolu par un test de tautologie. En effet, vérifier qu'un monôme α est contenu dans un ensemble E revient à vérifier que le cofacteur de E par rapport à α est égal à 1.

Exemple : Considérons la fonction f (a,b,c) telle que :

$f^{1}(a,b,c)$ = a'.b' + b'.c + a.c + a.b + b.c'

$f^{\phi}(a,b,c)$ = {}

La couverture F de la fonction f est :

10 10 11 (α)
11 10 01 (β)
01 11 01 (χ)
01 01 11 (δ)
11 01 10 (ε)

Cherchons à savoir α si est un monôme premier essentiel

$\beta \# \alpha$ => 01 10 01
$\chi \# \alpha$ => 01 11 01
01 01 01
$\delta \# \alpha$ => 01 01 11

01 01 11
ε # α => 01 01 10
11 01 10

Après réduction on obtient (test d'inclusion) :
F # α => 01 11 01
01 01 11
11 01 10

H = Consensus ((F#α),α)
H => 11 10 01
10 11 10

H_α => 11 11 01
11 11 10

H_α est une tautologie => n'est pas un monôme premier essentiel

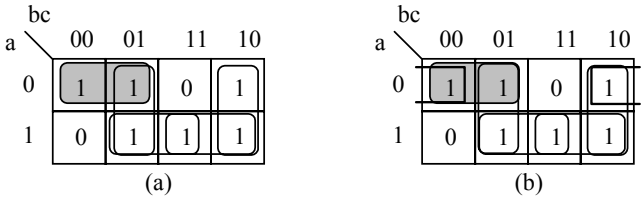


Figure 5.21. *(a) Fonction f. (b) Base première complète de f (monômes α non essentiel).*

Exemple : Considérons la fonction f (a,b,c) telle que :
f^1 = a'.b' + b'.c + a.c + a.b
f^ϕ = {}

La couverture F de la fonction f est :
10 10 11 (α)
11 10 01 (β)
01 11 01 (χ)
01 01 11 (δ)

Cherchons à savoir α si est un monôme premier essentiel
β # α => 01 10 01
χ # α => 01 11 01
01 01 01
δ # α => 01 01 11
01 01 11

Après réduction on obtient (test d'inclusion) :
F # α => 01 11 01
01 01 11

H = Consensus ((F#α),α)

H => 11 10 01

H_α => 11 11 01

H_α n'est pas une tautologie => α est un monôme premier essentiel

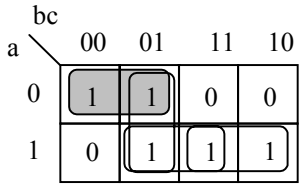


Figure 5.22. *Monôme premier essentiel.*

Chapitre 6

Factorisation et minimisation des fonctions multi-niveaux

Selon le type de circuit visé, des critères autres que le nombre de monômes peuvent être utilisés pour optimiser l'expression d'une fonction logique. Il peut être important de reconnaître dans une expression les facteurs communs à plusieurs monômes et plus généralement les sous-expressions communes à plusieurs fonctions simples. Pour certains types d'implantations (par exemple sur cellules standards), la surface des cellules est proportionnelle au nombre de littéraux. Le but de la factorisation est de minimiser le nombre de littéraux intervenant dans l'expression d'un ensemble de fonctions.

6.1. Définitions

6.1.1. Produit algébrique

Définition : Soient F et G deux expressions algébriques de deux fonctions booléennes. Le ***produit algébrique*** F.G est définit par:

$$F.G = \sum m_i.m_j \quad \text{pour } i = 1,...,n \text{ et } j = 1,...,m$$

avec $F = \sum m_i$ pour $i = 1,...,n$

$G = \sum m_j$ pour $j = 1,...,m$

Exemple : $F = a + b$

$G = c + d'.e$

$F.G = = a.c + a.d'.e + b.c + b.d'.e$

Remarque : Dans le produit algébrique une variable ne doit pas apparaître sous les deux formes. Les termes de la forme x.x' ou x.x. n'existent pas

6.1.2. Division algébrique

Définition : Soient F et G deux expressions algébriques de deux fonctions booléennes. G est un ***diviseur algébrique*** de F si F = G.Q + R.

ou: G.Q est le produit algébrique,

Q est la plus grande expression possible non nulle, c'est à dire qu'il n'existe pas Q* tel que :

$Q < Q^*$ avec $F = G.Q^* + R$ (R étant une expression algébrique).

Remarque : Le quotient Q (noté aussi F/G) et le reste R de cette division sont unique.

Exemple : F = a.b + a.d + c.b + d.c + e
G = a + c
G est un diviseur de F.
Le quotient est Q = b + d
Le reste est R = e.

Définition : Si F = GQ (c'est à dire R=0) avec les mêmes conditions, G est un ***facteur algébrique*** de F.

L'exemple suivant illustre la différence entre division algébrique et division booléenne.
F = a' b + b.c + a.c et Q = a+b
Division algébrique : F = c (a+ b) + a' b
Division booléenne : F = (a+ b) (a' + c)

La division algébrique est une division sur la forme de l'expression. En pratique, on n'utilise pas la division booléenne car trop compliquée, on se contente de la division algébrique.

6.1.3. Expression libre

Définition : Une expression algébrique F est une ***expression libre*** s'il n'existe pas de monôme m (avec m différent de 1) qui soit un facteur algébrique de F.

Exemple :

F = a.b + a.c	n'est pas libre
F = a.b.c	n'est pas libre
F = a.b + c	est libre

6.1.4. Noyau

Définition : K est un ***noyau*** d'une expression algébrique F si K = F / m (quotient) ou m est un monôme et K une expression libre. "m" est appelé le ***co-noyau*** de K.

Remarque : Un noyau comporte forcément plus d'un monôme puisqu'un monôme n'est pas une expression libre.

Remarque : Si F est une expression libre, F est un noyau d'elle même (F = F/1).

Définition : Un noyau est de degré 0 s'il n'admet pas d'autre noyau que lui-même. ***Le degré d'un noyau*** est défini de manière récursive. Si k(F) représente l'ensemble des noyaux de F et ki(F) représente l'ensemble des noyaux de degré i de F,

k0 = {K $\in$ k(F) / k(K) = {K} }

k1 = {K $\in$ k(F) / il existe K' $\in$ k(K) et K' $\neq$ K et k(K') = {K'} }

etc ...

Exemple : F = a'.b'.c'.d' + a'.b.d + a'.c.d
F = a'.(b'.c'.d' + b.d + c.d)
K = b'.c'.d' + b.d + c.d est un noyau de F, a' est son co-noyau
Il est de degré 1 car (b + c) est un noyau de K, d est son co-noyau

Remarque : Les noyaux d'une expression algébrique représentent toutes les factorisation maximales (en nombre de variables) possibles.

6.1.5. Gain associé à un noyau

Définition : On peut associer un ***gain*** G en nombre de littéraux à un noyau. Ce gain est le nombre de littéraux du co-noyau multiplié par le nombre de monômes du noyau moins un.

G = (nombre de littéraux du co-noyau) * (nombre de monômes du noyau - 1)

Exemple : F = a'.b'.c'.d' + a'.b.d + a'.c.d => 10 littéraux
F = a'.(b'.c'.d' + b.d + c.d) => 8 littéraux

Le gain associé au noyau (b'.c'.d' + b.d + c.d) est G = 1 * 2 = 2.

6.1.6. Algorithme de division (en notation cubes de position)

Soit à diviser f par g.

Soit A = l'ensemble des monômes de f = $\{C_j^A , j= 1,2 ....,l\}$

Soit B = l'ensemble des monômes de g = $\{C_i^B , i= 1,2 ....,n\}$

```
DIVISION ( A, B) { /* renvoie Q et R */
for (i = 1 to n) {                          /* prendre un monôme de G */
    D = {C_j^A tels que C_j^A ⊇ C_i^B} /* monômes du dividende qui incluent C_i^B  */
    if ( D ==  Ø ) return ( Ø , A);    /* quotient nul */
    D_i = D dans lequel on a supprimé toutes les variables figurant dans C_i^B
    if (i == 1)
        Q = D_i ;
    Else
        Q = Q ∩ D_i ;}
R = A - Q X B;
return ( Q , R);
}
```

Exemple 1: f = ac + ad + bc + bd + e
g = a + b

A = { ac , ad , bc , bd , e}

B = { a , b }

pour i = 1 , C_1^B = a , D ={ ac , ad } => D1 = { c , d } .

Donc Q = { c , d }.

pour i = 2 , C_2^B = b , D ={ bc , bd } => D2 = { c , d } .

Donc Q = { c , d } ∩ { c , d }={ c , d}

Finalement Q = { c , d } et R = {e} c-à-d f = (c+d) g + e

Exemple 2: f = axc + axd + bc + bxd + e

g = ax + b

A = { axc , axd , bc , bxd , e}

B = { ax , b }

pour i = 1, C_1^B = a, D ={ axc , axd } => D1 = { c , d }.

Donc Q = { c , d }.

pour i = 2, C_2^B = b, D ={ bc , bxd } => D2 = { c , xd }.

Donc Q = { c , d }∩{ c , xd } = { c }.

Il s'agit de l'intersection d'ensembles où les monômes sont les éléments.

Finalement Q = { c } et R = {axd, bxd, e}

Théorème: Soient f et g deux expressions algébriques. f / g est vide si l'une des conditions suivantes est vraie :

1. g contient une variable qui n'appartient pas à f
2. g contient un cube dont le support (ensemble de variables mises en jeu) n'est contenu dans celui d'aucun cube de f.
3. g a plus de termes que f
4. le nombre de variables de g est plus grand que celui de f

Toutes ces règles soit simplifient l'algorithme ci-dessus (les 2 premières), soit permettent de détecter quand l'algorithme est inutile.

6.2. Factorisation d'une fonction simple

La factorisation d'une fonction simple peut être obtenue par l'application de l'un des 2 algorithmes suivants :

6.2.1. Algorithme 1

1. Calculer tous les noyaux de degré 0 de F,
2. Classer les noyaux par gain croissant,
3. Opérer les divisions successives de la fonction par les noyaux.

Exemple : F = a'.b'.c'.d' + a'.b.d + a'.c.d + a.b'.c'.d

Les noyaux et co-noyaux de degré 0 de F sont:

K^0_1 = (a'.d' + a.d) (Co-noyau = b'.c') => Gain 2

K^0_2 = (b + c) (Co-noyau = a'.d) => Gain 2

F/K^0_1 = b'.c'. (a'.d' + a.d) + a'.b.d + a'.c.d

$(F/K^0_1)/K^0_2$ = b'.c'. (a'.d' + a.d) + a'.d. (b + c)

Remarque : Attention, une fois une factorisation effectuée, il est possible que certaines des factorisations suivantes ne soient plus possibles.

Exemple : F = a.b + a.c.d + c.e

Les noyaux et co-noyaux de degré 0 de F sont:

K^0_1 = (b + c.d) (Co-noyau = a) => Gain 1

K^0_2 = (a.d + e) (Co-noyau = c) => Gain 1

$F/K^0_1 = a.(b + c.d) + c.e$

On ne peut plus diviser par le second noyau c.(a.d + e).

6.2.2. Algorithme 2

1. Calculer tous les noyaux de degré 0 de F,
2. Diviser par le noyau de gain maximal et renommer ce noyau par une sous-fonction,
3. Recommencer en 1 jusqu'à ce qu'il n'y ait plus de noyau,
4. Réinjecter toutes les sous fonctions pour obtenir une forme factorisée de f.

Exemple : F = a'.b.c + b.c.e + b'.c.d

Un seul noyau de degré 0.

$K^0_1 = (a' + e)$ (Co-noyau = b.c) => Gain 2

F = b.c. (G1) + b'.c.d avec G1 = (a' + e)

Un seul noyaux de degré 0.

$K^0_1 = (b.G1 + b'.d)$ (Co-noyau = c) => Gain 1

F = c. (G2) avec G2 = $(b.G_1 + b'.d)$

En réinjectant les sous fonctions G1 et G2 dans F on obtient une forme factorisée.

F = c. (b.G1 + b'.d)

F = c. (b.(a' + e) + b'.d)

6.3. Factorisation d'une fonction multiple

Dans le cas de fonctions multiples, il est intéressant de trouver les sous-expressions communes à plusieurs fonctions. On va donc essayer de détecter:

- Les noyaux ou parties de noyaux communs,
- Les monômes ou parties de monômes communs.

6.3.1. Recherche de noyaux ou parties de noyaux communs

6.3.1.a. Algorithme de recherche de noyaux ou parties de noyaux communs

1. On calcule tous les noyaux de chaque fonction Fi
2. On associe à chaque monôme de tous les noyaux une variable Tj
3. Pour chaque noyau on associe le monôme Π Tk qui compose ce noyau
4. On calcule tous les noyaux de la fonction F =∑ (Π Tk)
5. Chaque co-noyau d'un noyau de degré 0 de la fonction F correspond à une partie de noyau commune.

Exemple : F1 = a.c.d + a.d.e + a.i

F2 = b.c.d + b.d.e + b.h

F3 = e.c.d + e.i

Noyau de F1: $K1_1 = (c.d + d.e + i)$ (Co-noyau = a) Gain = 1*(3-1) = 2

$K1_2 = (c + e)$ (Co-noyau = a.d) Gain = 2*(2-1) = 2

Noyau de F2: $K2_1 = (c.d + d.e + h)$ (Co-noyau = b) Gain = 1*(3-1) = 2

$K2_2$ = (c + e) (Co-noyau = b.d) Gain = 2*(2-1) = 2

Noyau de F3: $K3_1$ = (c.d + i) (Co-noyau = e) Gain = 1*(2-1) = 1

On associe une variable à chaque monôme de ces trois noyaux:

T1 = c.d

T2 = d.e

T3 = i

T4 = c

T5 = e

T6 = h

On forme la fonction F = T1.T2.T3 + T4.T5 + T1.T2.T6 + T4.T5 + T1.T3

Les noyaux et co-noyaux des noyaux de degré 0 de F sont: T1.T2 , T1.T3 et T4.T5

Les parties de noyaux communs sont donc: (c.d + d.e) , (c.d + i) et (c + e)

6.3.1.b. Introduction d'une sous fonction commune

Il est possible de prendre une des parties commune, de la renommer et de la réinjecter dans les fonctions ou elle apparaît.

Exemple : Reprenons l'exemple précédent.

Soit G1 = c.d + d.e

F1 = a.G1 + a.i

F2 = b.G1 + b.h

F3 = e.c.d + e.i

6.3.1.c. Gain

Pour chaque fonction i ou la partie commune est renommée, le gain G_i est égal à:

G_i = [(Nombre de littéraux du co-noyau i) * (Nombre de monômes de la partie cummune -1)]

Le gain global G (en nombre de littéraux) associé à l'introduction d'une sous-fonction commune est:

G = ∑ [Gain pour chaque fonction i ou la partie commune sera renommée]
+ [(Nombre de littéraux de la sous-fonction -1) * (Nombre d'occurences de la sous-fonction)] - [Nombre de littéraux de la sous-fonction]

Dans l'exemple précédent, le gain associé à G1 = c.d + d.e est:

G = [1*(2 - 1) + 1*(2 - 1)] + [(4 - 1) * 2] - [4]

G = 4

Le nombre de littéraux de la forme initiale de F1 et F2 et F3 est égal à 21 (8 pour F1, 8 pour F2, 5 pour F3). Le nombre de littéraux après renommage est égal à 21 - 4 = 17 (4 pour F1, 4 pour F2, 5 pour F3 et 4 pour G1).

6.3.2. Recherche de monômes ou parties de monômes communs

6.3.2.a. Algorithme de recherche de monômes ou parties de monômes communs

1. On rajoute à chaque monôme de chaque fonction Fi une variable v_i

2. On calcule tous les noyaux de degré 0 et co-noyaux associés de la fonction F composée de tous les monômes précédemment construits
3. Les co-noyaux de noyaux de degré 0 ou aucune variable v_i n'apparaît correspondent à des parties de monômes communs. Si plusieurs co-noyaux sont associés au même noyau, ils forment une somme de monômes communs.

Exemple : F1 = a.b.c.d + d.e + h

F2 = a.b.c.e + d.e + h

$F = a.b.c.d.v_1 + d.e.v_1 + h.v_1 + a.b.c.e.v_2 + d.e.v_2 + h.v_2$

Les noyaux de degré 0 et les co-noyaux correspondants ou v_1 et v_2 n'apparaissent pas sont:

$K^0_1 = (d.v_1 + e.v_2)$ (Co-noyau = a.b.c)

$K^0_2 = (v_1 + v_2)$ (Co-noyau = d.e)

$K^0_3 = (v_1 + v_2)$ (Co-noyau = h)

a.b.c est une partie de monômes commune à F1 et F2

(d.e + h) est une somme de monômes commune à F1 et F2

6.3.2.b. Introduction d'une sous fonction commune

Il est possible de prendre une des parties commune, de la renommer et de la réinjecter dans les fonctions ou elle apparaît.

Exemple : Reprenons l'exemple précédent.

Soit G1 = a.b.c

F1 = d.G1 + d.e + h

F2 = e.G1 + d.e + h

6.3.2.c. Gain

Le gain G (en nombre de littéraux) associé à l'introduction d'une sous-fonction commune est:

G = [(Nombre de littéraux de la sous-fonction -1) * (Nombre d'occurences de la sous-fonction)] - [Nombre de littéraux de la sous-fonction]

Dans l'exemple précédent, le gain associé à G1 = a.b.c est:

G = [(3 - 1) * 2] - [3]

G = 1

Le nombre de littéraux de la forme initiale de F1 et F2 est égal à 14 (7 pour F1, 7 pour F2). Le nombre de littéraux après renommage est égal à 14 - 1 = 13 (5 pour F1, 5 pour F2 et 3 pour G1).

6.3.3. Algorithme général de factorisation de fonctions multiples

1. Calculer tous les noyaux de chaque fonction et déterminer leur gain
2. Calculer toutes les intersection de noyaux communes à plusieurs fonctions et déterminer leur gain
3. Calculer toutes les parties de monômes et sommes de monômes commun à plusieurs fonctions et déterminer leur gain
4. Renommer la sous-fonction de gain maximal et l'ajouter à la liste des fonctions.
5. Retourner en 1 tant qu'il existe un noyau, une partie de monôme commune ou une somme de monômes communs dont le gain est positif.

6. Réinjecter toutes les sous-fonctions.

Exemple : F1 = a'.b'.c'.d' + a'.b.d + a'.c.d + a.b'.c'.d
F2 = a'.b'.c'.d + a.b.d' + a.c.d'

1 - Calculer tous les noyaux de chaque fonction

Noyau de F1:

$K1_1$ = (a'.d' + a.d) (Co-noyau = b'.c') => Gain = 2*(2-1) = 2

$K1_2$ = (b'.c'.d' + b.d + c.d) (Co-noyau = a') => Gain = 1*(3-1) = 2

$K1_3$ = (a'.b + a'.c + a.b'.c') (Co-noyau = d) => Gain = 1*(3-1) = 2

$K1_4$ = (b + c) (Co-noyau = a'.d) => Gain = 2*(2-1) = 2

Noyau de F2:

$K2_1$ = (b + c) (Co-noyau = a.d') => Gain = 2*(2-1) = 2

2 - Calculer toutes les intersection de noyaux communes à plusieurs fonctions

T1 = b'.c'.d'
T2 = b.d
T3 = c.d
T4 = a'.d'
T5 = a.d
T6 = a'.b
T7 = a'.c
T8 = a.b'.c'
T9 = b
T10 = c

F = T1.T2.T3 + T4.T5 + T6.T7.T8 + T9.T10 + T9. T10
Noyau de F : K1(F) = 1 (Co-noyau T9.T10)
T9.T10 correspond à une partie de noyau commune
Intersection de noyaux communes à F1 et F2 => T9.T10 => (b + c)
Gain = [2+2] + [(2-1)*2] - [2]
Gain = 4

3 - Calculer toutes les parties de monômes et sommes de monômes commun à plusieurs fonctions

F = a'.b'.c'.d'.v_1 + a'.b.d.v_1 + a'.c.d.v_1 + a.b'.c'.d.v_1 + a'.b'.c'.d.v_2 + a.b.d'.v_2 + a.c.d'.v_2

Noyau de degré 0 de F:

K^0_1 = d'.v_1 + d.v_2 (Co-noyau a'.b'.c')

Parties de monômes communes à F1 et F2 => a'.b'.c' Gain = [(3-1)*2] -[3] = 1

Somme de monômes commune à F1 et F2 => Aucune (1 seul co-noyau)

4 - Renommer la sous-fonction de gain maximal et l'ajouter à la liste des fonctions.
On choisit la sous-fonction de gain maximal que l'on renomme: G1 = (b + c)

F1 = a'.b'.c'.d' + a'.d.G1 + a.b'.c'.d
F2 = a'.b'.c'.d + a.d'.G1
G1 = b + c

5 - Retourner en 1 tant qu'il existe un noyau, une partie de monôme commune ou une somme de monômes communs dont le gain est positif.

1 - Calculer tous les noyaux de chaque fonction

Noyau de F1: $K1_1$ = (b'.c'.d' + d.G1) (Co-noyau = a') => Gain 1

$K1_2$ = (a'.G1 + a.b'.c') (Co-noyau = d) => Gain 1

$K1_3$ = (a'.d' + a.d) (Co-noyau = b'.c') => Gain 2

Noyau de F2: Aucun

Noyau de G1: Aucun

2 - Calculer toutes les intersection de noyaux communes à plusieurs fonctions

Intersection de noyaux communes => Aucun

3 - Calculer toutes les parties de monômes et sommes de monômes commun à plusieurs fonctions

Parties de monômes communes => a'.b'.c' Gain 1

Somme de monômes commune => Aucune

4 - Renommer la sous-fonction de gain maximal et l'ajouter à la liste des fonctions.

On choisit la sous-fonction de gain maximal que l'on renomme: G2 = (a.d + a.d)

F1 = b'.c'.G2 + a'.d.G1

F2 = a'.b'.c'.d + a.d'.G1

G1 = b + c

G2 = a'.d' + a.d

5 - Retourner en 1 tant qu'il existe un noyau, une partie de monôme commune ou une somme de monômes communs dont le gain est positif.

1 - Calculer tous les noyaux de chaque fonction

Noyau de F1: Aucun

Noyau de F2: Aucun

Noyau de G1: Aucun

Noyau de G2: Aucun

2 - Calculer toutes les intersections de noyaux communes à plusieurs fonctions

Intersection de noyaux communes => Aucun

3 - Calculer toutes les parties de monômes et sommes de monômes commun à plusieurs fonctions

Parties de monômes communes => b'.c' Gain 0

=> a'.d Gain 0

Il n'existe plus de sous-fonction possible de gain positif (b'.c' et a'.d ont un gain nul)

6 - Réinjecter toutes les sous-fonctions.

On réinjecte G2

F1 = b'.c'.(a'.d' + a.d) + a'.d.G1

F2 = a'.b'.c'.d + a.d'.G1

G1 = b + c

On réinjecte G1

F1 = b'.c'.(a'.d' + a.d) + a'.d.(b + c)

F2 = a'.b'.c'.d + a.d'.(b + c)

Remarque : Cet algorithme est très coûteux et ne peut être appliqué directement sur de gros exemples. Certaines heuristiques doivent alors être appliquées. On peut par exemple se limiter aux noyaux de degré 0 de chaque fonction de départ.

Chapitre 7

Eléments d'arithmétique binaire

La plupart des calculateurs actuels utilisent la base 2 pour représenter les nombres. Les opérations entre nombres sont donc basées sur l'arithmétique binaire. Bien que le but de ce cours ne soit pas de détailler le traitement arithmétique dans les calculateurs, il est intéressant d'en préciser les bases élémentaires.

Dans la première partie de ce chapitre, nous rappellerons les notions élémentaires de codage et de représentation des nombres. Nous définirons ensuite les principes élémentaires de l'arithmétique binaire. Le principe des opérations de base (addition, soustraction, multiplication, division) sera présenté.

7.1. Représentation des nombres

Un code constitue une correspondance entre des symboles et des objets à désigner. Les codes utilisés pour représenter des nombres sont des codes pondérés : dans une base de travail donnée, la valeur d'un rang donné est un multiple par la base de celle du rang inférieur.

7.1.1. Représentation des entiers naturels

De manière générale, un nombre entier naturel N exprimé dans une base b est un ensemble ordonné de n chiffres chacun d'eux prenant une valeur comprise entre 0 et b-1.

$$N_b = a_{n-1}\ a_{n-2}\ \ldots.\ a_1\ a_0$$

Les nombres exprimés dans la base 10 sont appelés nombres décimaux. Les nombres exprimés dans la base 2 sont appelés nombres binaires. Pour les calculateurs électroniques ces nombres ont un intérêt tous particulier du fait qu'ils ne font intervenir que deux valeurs (0,1).

La valeur d'un nombre exprimé dans une base b est la somme pondérée de ces n chiffres, les poids de chacun d'eux étant des puissances de la base de numération. Cette valeur est comprise entre 0 est b^n-1.

$$N = a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \ldots.. + a_1b^1 + a_0b^0 \quad => \quad N = \sum_{i=0}^{n-1} a_i b^i$$

Exemple : Le nombre binaire $N_2 = 1011$ représente le nombre décimal $N_{10}=13$ ($2^3+2^2+2^0$).

7.1.2. Représentation des entiers relatifs

Un entier relatif est un entier pouvant être négatif. Soit un nombre de n bits s'écrivant de la manière suivante :

$$N_b = a_{n-1}\ a_{n-2}\ \ldots..\ a_1\ a_0$$

Par convention, le bit de poids fort (a_{n-1}), appelé bit de signe est utilisé pour représenter le signe. Les autres bits (a_{n-2} ….. a_1 a_0) sont utilisés pour représenter la valeur du nombre. Ainsi, un nombre signé prend ses valeurs dans l'intervalle $[-(2^{n-1}-1)\ ,\ 2^{n-1}-1]$

7.1.2.a. Codage signe et valeur absolue

Dans ce code, les chiffres (bits) a_{n-2} ….. a_1a_0 représentent la valeur absolue du nombre et le bit de signe a_{n-1} prend les valeurs suivantes :

$a_{n-1} = 0$ si $N \geq 0$
$a_{n-1} \neq 0$ si $N < 0$

un nombre N varie entre $-(2^{n-1}-1)$ et $+(2^{n-1}-1)$

Exemple : Interprétation de nombres binaires en représentation Signe et Valeur absolue (n=3)

2^2	2^1	2^0	N
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	-0
1	0	1	-1
1	1	0	-2
1	1	1	-3

Remarque : Un des problèmes induits par cette représentation réside dans le processus d'addition/soustraction de tels nombres. En effet, lorsque les 2 nombres à additionner sont de signes opposés, il est nécessaire de comparer la valeur des 2 nombres pour effectuer l'opération de soustraction et pour déterminer le signe du résultat. Un autre problème lié à cette représentation est la double représentation du 0.

7.1.2.b. Codage signe et complément

Le codage en complément à été introduit pour faciliter les opérations d'addition/soustraction de nombre rationnels. Dans ce code, le bit de signe a_{n-1} prend les valeurs suivantes :

$a_{n-1} = 0$ si $N \geq 0$
$a_{n-1} = b-1$ si $N < 0$

Les chiffres (bits) a_{n-2} ….. a_1a_0 représentent la valeur du nombre si le nombre est positif ($a_{n-1}=0$) ou la valeur codée en complément si le nombre est négatif ($a_{n-1}=b-1$). Le complément des nombres est un codage particulier. Les deux codes en complément les plus couramment utilisés sont le complément à b-1 (complément à la base – 1) et le complément à b (complément à la base).

7.1.2.b.1. Complément à b-1

Le complément à b-1 d'un nombre N exprimé sur n chiffres (bits), est obtenu en soustrayant le nombre N du radical R diminué d'une unité. Le radical R d'un nombre N exprimé en base b sur n chiffres (ou bits) est la puissance de b immédiatement supérieure à la valeur maximum de N ($R = b^n$).

$$C_{b-1}(N) = b^n - 1 - N$$

Exemple **:**

$C_9(5230)_{10} = 10^4 - 1 - (5230)_{10} = (4769)_{10}$

$C_1(1100)_2 = 2^4 - 1 - (1100)_2 = (16 - 1 - 12)_{10} = (3)_{10} = (0011)_2$

Nous pouvons constater que le complément à b-1 de N peut s'obtenir en complémentant à b-1 tous les chiffres. Dans le cas de la base 2 cela se traduit par une simple inversion de tous les bits.

Exemple : Interprétation de nombres binaires en représentation Signe et Valeur absolue (n=3)

2^2	2^1	2^0	N
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	-3
1	0	1	-2
1	1	0	-1
1	1	1	-0

Remarque : Un des problèmes induits par cette représentation réside dans la double représentation du 0.

7.1.2.b.2. Complément à b

Le complément à b d'un nombre N exprimé sur n chiffres (bits), est obtenu en soustrayant le nombre N du radical R.

$$C_b(N) = b^n - N$$

Exemple **:**

$C_{10}(5230)_{10} = 10^4 - (5230)_{10} = (4770)_{10}$

$C_n(1100)_2 = 2^4 - (1100)_2 = (16 - 12)_{10} = (4)_{10} = (0100)_2$

Nous pouvons constater que le complément à b de N peut s'obtenir à partir du complément à b-1 : $C_b(N) = C_{b-1}(N)+1$ mais également par la procédure de scrutation / complémentation suivante :

- Scruter le nombre à partir de la droite
- Tant que les bits rencontrés sont à 0, les conserver
- Complémenter à b le premier chiffre non nul
- Complémenter à b-1 tous les suivants

Dans le cas de la base 2 cela se traduit par :

- Scruter le nombre à partir de la droite
- Tant que les bits rencontrés sont à 0, les conserver
- Conserver le premier 1
- Inverser tous les bits suivants

Du fait de la plus grande simplification apportée au processus de soustraction, la représentation en complément la plus usuelle est la représentation en complément à b (complément à 2 en binaire). Avec cette représentation en complément à b, la valeur décimale d'un nombre peut être obtenue par la relation suivante :

$$(N)_{10} = -[(a_{n-1})/(b-1)]b^{n-1} + \sum_{i=0}^{n-2} a_i b^i$$

Cette valeur est comprise entre $-b^{n-1}$ et $b^{n-1}-1$.

Pour des nombres binaires, cette relation devient :

$$(N)_{10} = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Exemple : Interprétation de nombres binaires signé exprimés en complément à 2 (n=3)

2^2	2^1	2^0	N
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	-4
1	0	1	-3
1	1	0	-2
1	1	1	-1

Remarque : Outre son intérêt pour dans le cadre des opérations d'addition/soustraction, le codage en complément à b permet d'éviter la double représentation du 0 ce qui n'est pas le cas en codage complément à b-1.

7.1.3. Représentation des nombres rationnels

7.1.3.a. Représentation en virgule fixe

Les représentations précédentes peuvent s'étendre aux nombres rationnels. Pour ces nombres, la partie fractionnaire est séparée de la partie entière, par une virgule ou un point. Ce type de codage des nombre rationnels est appelé représentation en virgule fixe.

$$(N)_b = a_{n-1} \ldots. \ a_0 \ , \ a_{-1} \ldots. a_{-m}$$

La valeur décimale d'un tel nombre est :

$$(N)_{10} = a_{n-1}b^{n-1} + ... + a_0 b^0 + a_{-1}b^{-1} + \ldots. + a_{-m}b^{-m} \quad => \quad (N)_{10} = \sum_{i=-m}^{n-1} a_i b^i$$

Exemple : Le nombre binaire N = 1101,11 représente la valeur 13,75 ($2^3+2^2+2^0+2^{-1}+2^{-2}$).

7.1.3.b. Représentation en virgule flottante

Le codage en virgule fixe sur *n* bits ne permet de représenter qu'un intervalle de 2^n valeurs. Pour un grand nombre d'applications, cet intervalle de valeurs est trop restreint. La représentation à virgule flottante (*floating-point*) a été introduite pour répondre à ce besoin et améliorer la précision des calculs.

Cette représentation consiste à représenter un nombre binaire sous la forme $1,M * 2^E$ ou M et la mantisse et E l'exposant.

La représentation en virgule flottante a été normalisée (norme IEEE 754) afin que les programmes aient un comportement identique d'une machine à l'autre. La norme IEEE754 défini la façon de coder en virgule flottante un nombre binaire sur 32 bits ou 64 bits (double précision) en définissant 3 composantes :

- le signe S est représenté par un seul bit : le bit de poids fort (le plus à gauche)
- l'exposant E est codé sur les 8 bits consécutifs au signe en excès de 127 (simple précision) ou 1023 (double précision)
- la mantisse M (bits situés après la virgule) est représentée sur les 23 (simple précision) ou 55 (double précision) bits restants.

Remarque : les exposants peuvent aller de -126 à +127 (simple précision) ou de -1022 à +1023 (double précision).

Ainsi le codage sur un mot de 32 bits se fait sous la forme suivante:

seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm

- le **s** représente le bit relatif au signe
- les **e** représentent les bits relatifs à l'exposant avec un excès de 127
- les **m** représentent les bits relatifs à la mantisse.

Dans cette représentation la valeur d'un nombre sur 32 bit est donné par l'expression :

$N = (-1)^s * 1,M * 2^{(E-127)}$

$N = (-1)^s * (1 + M*2^{-23}) * 2^{(E-127)}$

Exemple : Soit à coder la valeur 525,5.

525,5 s'écrit en base 2 de la façon suivante: 1000001101,1

on veut l'écrire sous la forme 1.0000011011×2^9. Par conséquent:

- le bit s vaut 1
- l'exposant vaut 9+127 = 136, soit 10010000
- la mantisse est 0000011011

La représentation du nombre 525.5 en binaire avec la norme IEEE754 est:

1 10010000 00000110110000000000000

Remarque : Le zéro est le seul nombre qui ne peut pas se représenter dans le système à virgule flottante à cause de la contrainte de non nullité du premier digit ($1,M * 2^E$). Le zéro est donc représenté par une convention de représentation. La norme IEEE754 précise cette convention de représentation (E=M=0) ainsi que d'autre conditions particulières :

E	M	Valeur représentée
Max	0	∞
Max	≠0	NaN (*Not a number*)
0	0	0
0	≠0	Nombre dénormalisé

7.1.4. Conversion de bases

Compte tenu de la facilité de conversion entre les base 2, 8 et 16 (puissances de 2), les bases 8 (octal) et 16 (hexadécimal) sont également utilisées pour représenter les nombres binaires sous forme plus synthétique. Un nombre octal ou hexadécimal peut effectivement être convertit en nombre binaire par conversion de chacun de ses coefficients individuellement dans sa représentation binaire équivalente.

Exemple : $(3D)_{hex}$ => $(00111101)_{bin}$

Cette propriété est vraie pour les bases puissance de 2. Pour les bases non puissance de deux, cette conversion est plus complexe.

Une solution permettant d'effectuer ces conversions est d'utiliser directement les expressions polynomiales. Cette technique d'évaluation directe d'expressions polynomiales de nombre est une méthode générale de conversion d'une base arbitraire b1 en une autre base arbitraire b2. Cette méthode est appelée méthode polynomiale.

7.1.4.a. Méthode polynomiale :

- Exprimer le nombre $(N)_{b1}$ en polynôme avec dans le polynôme des nombres exprimés dans la base b2,
- Evaluer le polynôme en utilisant l'arithmétique de la base b2.

A titre d'exemple considérons le nombre binaire (1011,101). L'expression polynomiale est la suivante:

$$A = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$$

$(A)_{10}$= 1*8 + 0*4 + 1*2 + 1*1 + 1*1/2 + 0*1/4 + 1*1/8
= 11 + 5/8
= 11,625

$(A)_8$ = 1*10 + 0*4 + 1*2 + 1*1 + 1*1/2 + 0*1/4 + 1*1/10
= 13 + 5/10
= 13,5

$(A)_3$ = 1*22 + 0*11 + 1*2 + 1*1 + 1*1/2 + 0*1/11 + 1*1/22
= 102 + 12/22
= 102,121212 ...

Ce type de conversion est en fait très utilisé pour passer d'une base quelconque à une expression décimale. Dans la pratique un changement de base entre deux bases quelconques se fait généralement par l'intermédiaire de l'expression décimale:

base b1 => expression décimale (base 10) => base b2

Nous avons vu que le passage d'une base quelconque à la base décimale par la méthode algorithmique est simple l'inverse l'étant moins. Une solution pour réaliser ce passage (base 10 à base quelconque) est d'appliquer une méthode itérative.

7.1.4.b. Méthode itérative :

Soit A la représentation d'un nombre entier positif dans la base 10 Pour obtenir sa représentation dans la base b quelconque il suffit de diviser A par b, puis le quotient par b, jusqu'à ce que le quotient devienne nul. Les restes successifs lu de bas en haut sont la représentation de A dans la base b.

$A = a_3b^3 + a_2b^2 + a_1b^1 + a_0$

$= b(a_3b^2 + a_2b^1 + a_1) + a_0 = bQ1 + a_0$

$Q1 = b(a_3b^1 + a_2b^0) + a_1 = bQ2 + a_1$

$Q2 = ba_3 + a_2 = bQ3 + a_2$

$Q3 = a_3$

Exemple **:** Ecrire 88 en base 2 et 3

88 = (44*2) + 0
44 = (22*2) + 0
22 = (11*2) + 0
11 = (5*2) + 1
5 = (2*2) + 1
2 = (1*2) + 0
1 = (0*2) + 1

$(88)_{10} = (1011000)_2$

88 = (29*3) + 1
29 = (9*3) + 2
9 = (3*3) + 0
3 = (1*3) + 0
1 = (0*3) + 1

$(88)_{10} = (10021)_3$

Exemple **:** Ecrire 23 en base 2 et 8

23 = (11*2) + 1
11 = (5*2) + 1
5 = (2*2) + 1
2 = (1*2) + 0
1 = (0*2) + 1

$(23)_{10} = (10111)_2$

23 = (2*8) + 7
2 = (0*3) + 2

$(23)_{10} = (27)_8$

Remarque **:** Cet exemple illustre les relations simples qui existent entre la base 2 et la base 8. Les digits binaires (appelés bits) sont pris 3 par 3 et exprimés en décimal pour obtenir le nombre octal. Ainsi l'équivalent du nombre binaire 10111 en base 8 est 27.

010 111
2 7

Il en est de même pour toutes les bases puissance de 2 et notamment pour la base 16 (hexadécimal). Dans le cas de la base 16, le nombre de bits à considérer est 4.

7.1.4.c. Cas particulier de la base 2 :

La conversion du système décimal au système binaire peut s'effectuer en remarquant que le reste de la division est 0 ou 1 selon que le dividende est pair ou impair.

La conversion décimale binaire peut donc être représentée plus simplement en écrivant les quotients de droite à gauche; on écrit « 1 » sous chaque quotient impair et « 0 » sous chaque quotient pair.

Exemple **:** Conversion de $(53)_{10}$ en binaire

1 3 6 13 26 53
1 1 0 1 0 1 => $(53)_{10} = (110101)_2$

7.1.4.d. Cas des nombres rationnels :

Considérons maintenant le cas d'un nombre $(N)_{10}$ ayant une partie fractionnaire et dont nous voulons l'expression dans la base b.

Pour le radical, la conversion s'effectue comme nous venons de le voir.

Pour la partie fractionnaire on opère de la façon suivante :

- On multiplie son expression en base 10 par b, on obtient un résultat $(n1)_{10}$.
- Le premier chiffre de la partie fractionnaire exprimé en base b est la partie entière de $(n1)_{10}$.
- On calcule $(n2)_{10} = (n1)_{10}$ - partie entière de $(n1)_{10}$
- La valeur du second chiffre de la partie fractionnaire exprimée en base b est obtenue en utilisant $(n2)_{10}$ comme nous avons utilisé $(n1)_{10}$ pour calculer le premier chiffre.

Exemple **:** Soit à convertir $(88,23)_{10}$ en base 2

Nous savons que $(88)_{10} = (1011000)_2$. Effectuons la conversion de la partie fractionnaire.

0,23 * 2 = 0,46 0,0
0,46 * 2 = 0,92 0,00
0,92 * 2 = 1,84 0,001
0,84 * 2 = 1,68 0,0011
0,68 * 2 = 1,36 0,00111
0,36 * 2 = 0,72 0,001110
d'ou $(88,23)_{10} = (1011000,001110...)_2$

On voit bien que cette conversion peut ne pas se terminer et donc que l'on obtient lorsque l'on s'arrête à un nombre fini de positions, une approximation de la représentation du nombre.

Le cas particulier des bases multiples de 2 s'applique également dans le cas des nombre fractionnaires. Dans le cas de la base 8 par exemple, les bits sont pris 3 par 3 de chaque coté de la virgule. Ainsi $(010111,011101)_2 = (27,35)_8$

7.2. Comparateurs binaires

7.2.1. Comparateur égalité

Pour savoir si deux nombres binaires A ($A= a_n...a_0$) et B ($B = b_n ... b_0$) sont égaux, il faut que trous les bits de ces deux nombres soient identiques. On a donc :

$A = B$ si et seulement si $(a_0 = b_0)$ et $(a_1 = b_1)$ et $(a_2 = b_2)$ et ...

Soit S la sortie du circuit comparateur égalité (S=1 si les 2 nombres sont identiques)

$S = (a_0 \oplus b_0)' . (a_1 \oplus b_1)' . (a_2 \oplus b_2)' ....$

Un tel circuit peux également être déterminé en raisonnant à partir d'une structure itérative (en tranches) telle que celle présentée sur la figure 7.1. Ainsi, il suffit de déterminer la structure d'une cellule et de cascader cette cellule.

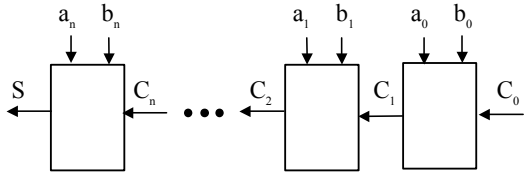


Figure 7.1. *Structure en tranche*

Pour déterminer la structure interne des cellules, considérons un étage i

Supposons que $C_i = 1$ si et seulement si $\forall j$ compris entre 0 et i-1, aj = bj

La même information (C_{i+1}) doit être fournie à l'étage suivant

=> $C_{i+1} = C_i . (a_i \oplus b_i)'$ (avec $C_0 = 1$)

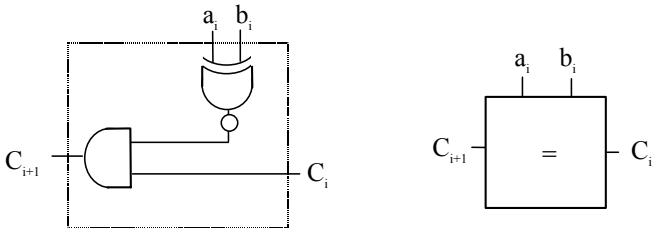


Figure 7.2. *Cellule du comparateur égalité*

7.2.2. Comparateur supériorité

Pour savoir si un nombre binaire A ($A= a_n...a_0$) est supérieur ou pas à un nombre binaire B ($B = b_n ... b_0$) il faut scruter ce nombre à partir des poids forts. On aura :

$A > B$ si et seulement si $(a_n > b_n)$ ou $[(a_n = b_n)$ et $(a_{n-1} > b_{n-1})]$ ou ...

Soit S la sortie du circuit comparateur égalité (S=1 si les 2 nombres sont identiques)

$S = (a_n.b_n') + (a_n \oplus b_n)' . (a_{n-1}.b_{n-1}') +$ ....

Un tel circuit peux également être déterminé en raisonnant à partir d'une structure itérative (en tranches) telle que celle présentée sur la figure 7.1. Pour déterminer la structure interne des cellules, considérons un étage i

Supposons que $C_i = 1$ si et seulement si, en ne considérant que les bits de poids plus faible on a A > B.

La même information (C_{i+1}) doit être fournie à l'étage suivant

$C_{i+1} = a_i.b_i' + (a_i \oplus b_i)'. C_i$ (avec $C_0 = 0$)

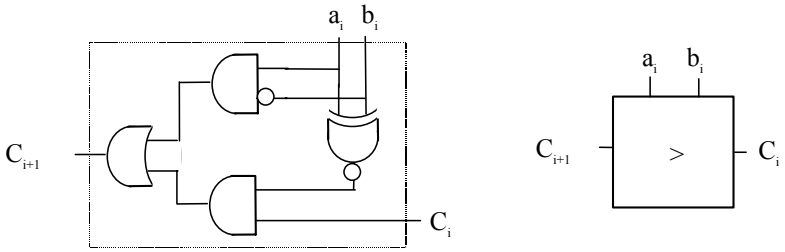


Figure 7.3. *Cellule du comparateur supériorité*

7.3. Addition binaire

7.3.1. Structure de l'additionneur binaire

Ce paragraphe présente la structure de base permettant de réaliser l'addition entre deux nombre binaires de n bits.

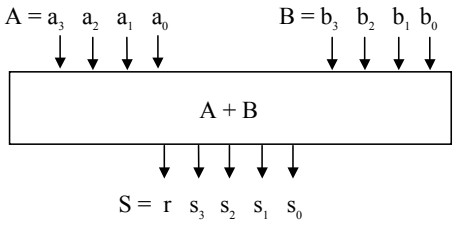


Figure 7.4. *Additionneur binaire*

La table d'addition binaire sur 2 bits est 1a suivante:

a b s	r	$s = a \oplus b$
0 + 0 = 0	0	r = a.b
0 + 1 = 1	0	
1 + 0 = 1	0	
1 + 1 = 0	1	

L'opération d'addition de deux représentation binaires s'effectue de façon similaire à l'addition décimale c'est à dire en additionnant digit par digit les deux représentations alignées sur la virgule et en reportant une éventuelle retenue sur la colonne suivante.

Exemple **:**

```
Retenues -> 10011 11
              1001,011  = ( 9,375)10
            + 1101,101  = (13,625)10
              ----------  ----------
              10111,000  = (23,000)10
```

Pour réaliser un additionneur, le module de base doit donc réaliser l'addition sur 3 bits. L'addition étant une opération associative, on peut aisément déduire la table d'addition binaire sur 3 bits de la table d'addition binaire sur 2 bits.

Soit deux nombres A ($A= a_n ... a_0$) et B ($B= b_n ... b_0$).

a_i	b_i	r		s_i	r_{i+1}	$s_i = a_i \oplus b_i \oplus r_i$
0	0	0	=>	0	0	$r_{i+1} = a_i.b_i + a_i.r_i + b_i.r_i$
0	0	1	=>	1	0	
0	1	0	=>	1	0	
0	1	1	=>	0	1	
1	0	0	=>	1	0	
1	0	1	=>	0	1	
1	1	0	=>	0	1	
1	1	1	=>	1	1	

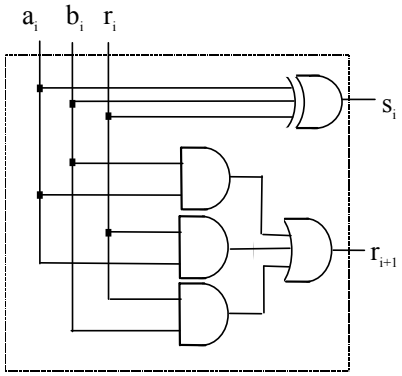


Figure 7.5. *Cellule additionneur 2 fois 1 bit plus retenue*

Les fonctions s_i et r_i peuvent également s'exprimer de la manière suivante:

$s_i = (a_i \oplus b_i) \oplus r_i$

$r_{i+1} = r_i.(a_i \oplus b_i) + a_i.b_i$

Démonstration :

$r_{i+1} = r_i.(a_i \oplus b_i) + a_i.b_i$

$= r_i.a_i'.b_i + r_i.a_i.b_i' + a_i.b_i + a_i.b_i$

$= a_i.(b_i + r_i.b_i') + b_i.(a_i + r_i.a_i')$

$= a_i.(b_i + r_i) + b_i.(a_i + r_i)$

$= a_i.b_i + a_i.r_i + b_i.r_i$

Ceci conduit à la structure suivante:

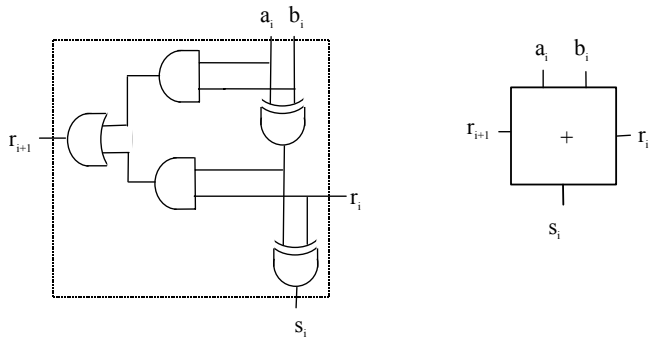


Figure 7.6. *Cellule additionneur 2 fois1 bit plus retenue*

La structure permettant d'additionner deux nombres de n bits est obtenue en cascadant cette structure de base.

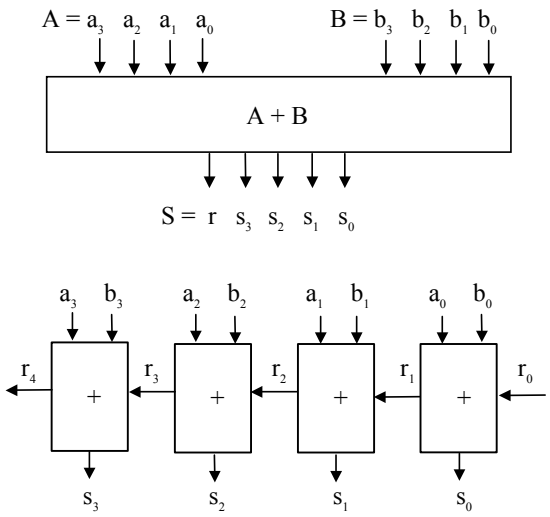


Figure 7.7. *Structure de l'additionneur binaire*

7.3.2. Incrémenteur

La structure d'un incrémenteur peut être déterminée à partir de celle de l'additionneur en considérant qu'un des 2 mots est à 0 et que la retenue entrante vaut 1.

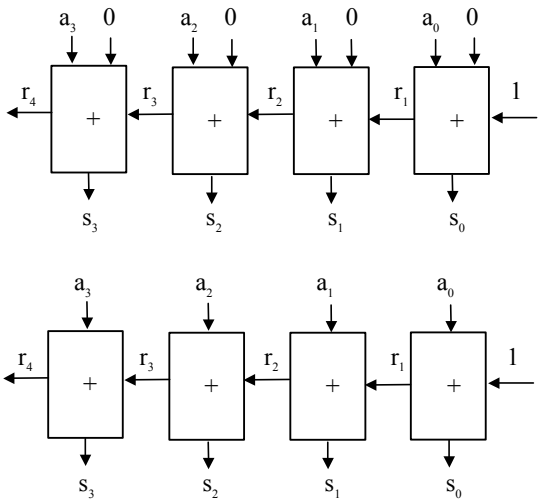


Figure 7.8. *Structure de l'incrémenteur*

$s_i = (a_i \oplus b_i) \oplus r_i$ => $s_i = a_i \oplus r_i$

$r_{i+1} = r_i.(a_i \oplus b_i) + a_i.b_i$ => $r_{i+1} = r_i.a_i$

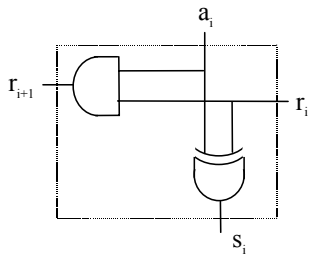


Figure 7.9. *Cellule élémentaire de l'incrémenteur*

7.3.3. Additionneur à propagation anticipée

L'inconvénient de la structure précédente est le temps nécessaire à la réalisation de l'addition. Ce temps est en effet conditionné par la propagation de la retenue à travers tous les additionneurs élémentaires. Dans un additionneur à carry anticipée on évalue en même temps la retenue de chaque étage. Pour cela on détermine pour chaque étage les quantités Pi et Gi suivantes:

$P_i = a_i \oplus b_i$ (propagation d'une retenue)
$G_i = a_i b_i$ (génération d'une retenue)

La retenue entrante à l'ordre i vaut 1 si :

- soit l'étage i-1 a généré la retenue ($G_{i-1} = 1$)
- soit l'étage i-1 a propagé la retenue générée à l'étage i-2 ($P_{i-1}=1$ et $G_{i-2}=1$)
- soit les étages i-1 et i-2 ont propagé la retenue générée à l'étage i-3 ($P_{i-1}=P_{i-2}=1$ et $G_{i-3}=1$)
- .....
- soit tous les étages inférieurs ont propagé la retenue entrante dans l'additionneur ($P_{i-1}=P_{i-2}=...=P_0=r_0=1$).

Donc $r_i = G_{i-1} + P_{i-1}.G_{i-2} + P_{i-1}.P_{i-2}.G_{i-3} +.....+ P_{i-1}.P_{i-2}.P_{i-3}....P_0.r_0$
$r_1 = G_0 + P_0.r_0$
$r_2 = G_1 + P_1.G_0 + P_1.P_0.r_0$
$r_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.r_0$
$r_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.r_0$

Dans un additionneur à retenue anticipée, on évalue en parallèle:

- les couples (G_i,P_i)
- les retenues r_i
- les bits de somme $s_i=a_i \oplus b_i \oplus r_i = P_i \oplus r_i$

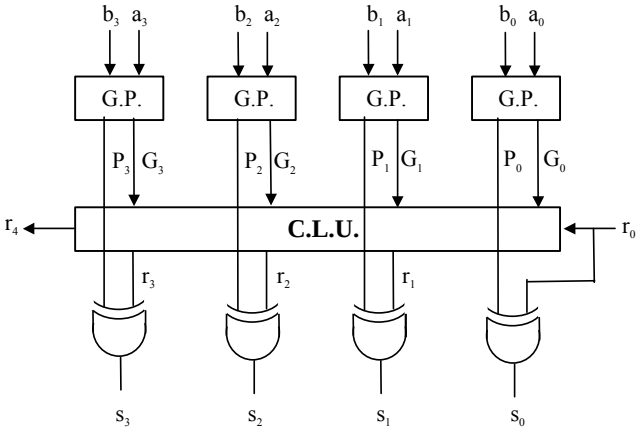


Figure 7.10. *Additionneur à retenue anticipée*

La structure du bloc CLU peut être déterminée à partir des équation donnant les retenues r_i.

7.4. Soustraction binaire

7.4.1. Structure du soustracteur binaire

Ce paragraphe présente la structure de base permettant de réaliser la soustraction entre deux nombre binaires de n bits.

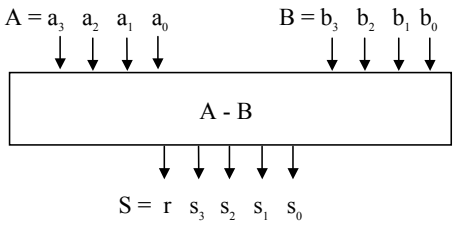


Figure 7.11. *Soustracteur binaire*

La table de soustraction sur 2 bits est la suivante:

a b s	s	
0 - 0 = 0	0	$s = a \oplus b$
0 - 1 = 1	1	$s = a' . b$
1 - 0 = 1	0	
1 - 1 = 0	0	

La procédure de soustraction de nombre binaire est semblable à celle utilisée en décimal c'est à dire :

- le signe du résultat est le signe du nombre le plus grand en valeur absolue.
- le résultat est obtenu en soustrayant le nombre le plus petit en valeur absolue du nombre le plus grand.

Exemple **:**

```
Retenues ->     111                          011
                101,0    ( 5  )              1100    ( 12)
              - 011,1    (-3,5)            - 0011    (- 3)
                ------   -----                ----    ----
                001,1    ( 1,5)              1001    (  9)
```

Pour réaliser un soustracteur, le module de base doit donc donc travailler sur 3 bits. Les relations donnant la somme et la retenue sur trois bits sont les suivantes.

Soit deux nombres A (A= a_n ... a_0) et B (B= b_n ... b_0) et soit à réaliser l'opération A-B.

a_i	b_i	r_i		s_i	r_{i+1}	$s_i = a_i \oplus b_i \oplus r_i$
0	0	0	=>	0	0	$r_{i+1} = a_i'.b_i + a_i'.r_i + b_i.r_i$
0	0	1	=>	1	1	
0	1	0	=>	1	1	
0	1	1	=>	0	1	
1	0	0	=>	1	0	
1	0	1	=>	0	0	
1	1	0	=>	0	0	
1	1	1	=>	1	1	

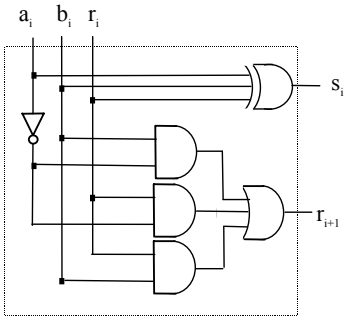


Figure 7.12. *Cellule soustracteur 2 fois 1 bit plus retenue*

Les fonctions S_i et R_i peuvent aussi s'exprimer de la manière suivante:

$s_i = (a_i \oplus b_i) \oplus r_i$

$r_{i+1} = r_i.(a_i \oplus b_i)' + a_i'.b_i$

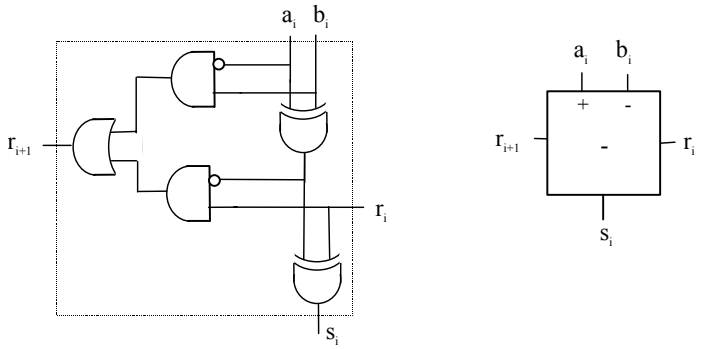


Figure 7.13. *Cellule soustracteur 2 fois 1 bit plus retenue*

$s_i' = (a_i' \oplus b_i) \oplus r_i$

$r_{i+1} = r_i.(a_i' \oplus b_i) + a_i'.b_i$

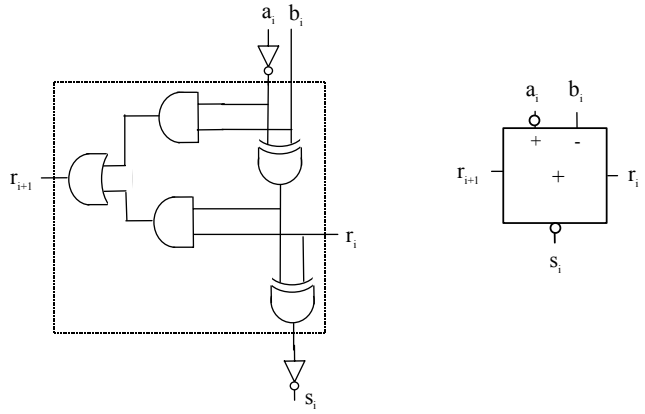


Figure 7.14. *Cellule soustracteur 2 fois 1 bit plus retenue*

Comme pour l'additionneur, un soustracteur peut être réalisé en cascadant les cellules de base.

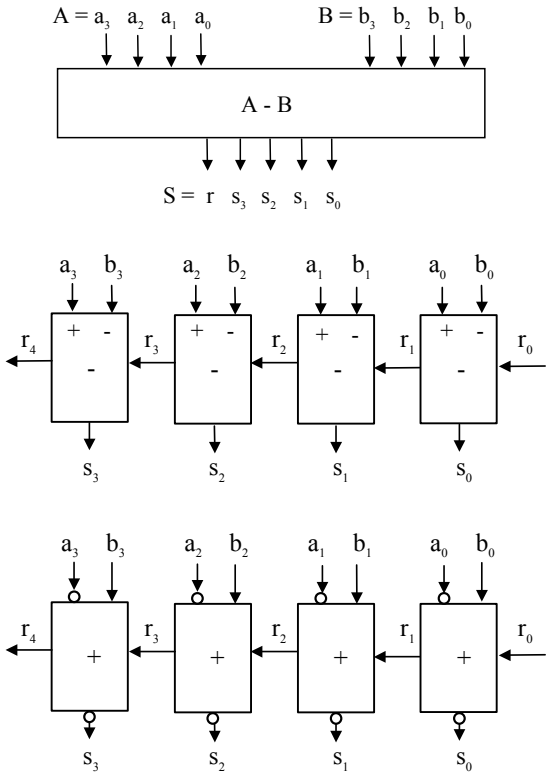


Figure 7.15. *Structure du soustracteur binaire*

Remarque **:** Relation entre la structure et le complément à 1.

$C_1 (C_1(A) + B) = C_1 (2^n - 1 - A + B)$

$= 2^n - 1 - 2^n + 1 + A - B$

$= A - B$

Remarque **:** La non-symétrie du soustracteur impose un dispositif amont de comparaison et d'orientation des entrées.

7.4.2. Décrémenteur

La structure d'un décrémenteur peut être déterminée à partir de celle du soustracteur en considérant que le mot à retrancher est nul et que la retenue entrante vaut 1. La cellule élémentaire de l'additionneur peut

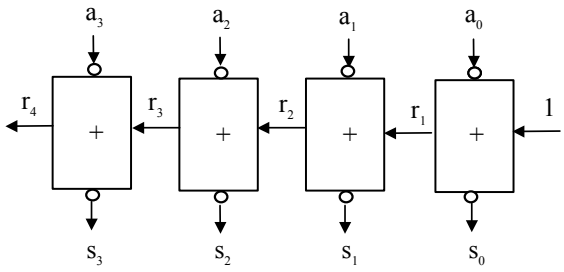


Figure 7.16. *Structure de l'incrémenteur*

Comme pour l'incrémenteur, la cellule élémentaire de l'additionneur peut être simplifiée en considérant b = 0.

7.4.3. Additionneur / Soustracteur

La structure d'un opérateur assurant soit l'addition soit la soustraction de deux nombres A et B en fonction d'une commande Op peut être conçu à partir des structures d'additionneurs et soustracteurs précédemment présentées. La structure d'un tel opérateur est représenté sur la figure 7.17.

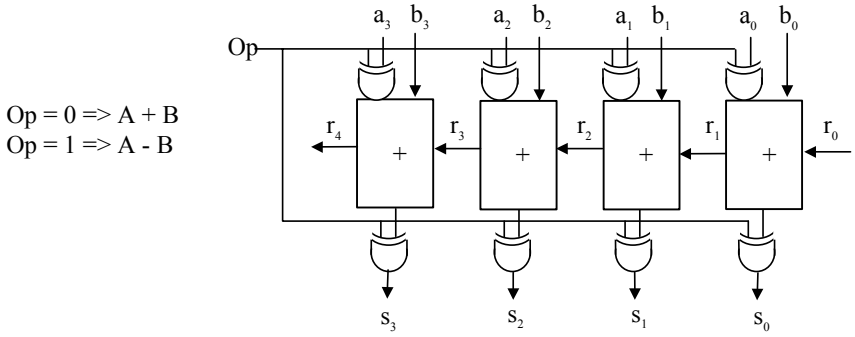


Figure 7.17. *Structure du additionneur / soustracteur binaire*

Remarque **:** Tout comme pour le soustracteur, la non-symétrie de cette cellule en mode soustraction impose un dispositif amont de comparaison et d'orientation des entrées.

Notons qu'une cellule assurant les opérations incrémentation / décrémentation en fonction d'une commande C peut être réalisée le même principe.

7.5. Addition algébrique

Un nombre entier relatif peut être représenté par un nombre binaire dont le bit de poids fort (a_{n-1}) indique le signe (0 correspond à un nombre positif, 1 à un nombre négatif) et les autres bits (a_{n-2} ….. a_1a_0) représentent la partie significative du nombre. Si la partie significative du nombre est représentée en valeur absolue, l'opération de soustraction est relativement complexe. Elle nécessite effectivement la connaissance du plus grand des deux nombres, la modification de ce nombre au cours de la soustraction, etc.... La structure réalisant l'opération d'addition sur des nombres signés exprimés en valeur absolue est représentée sur la figure 7.18.

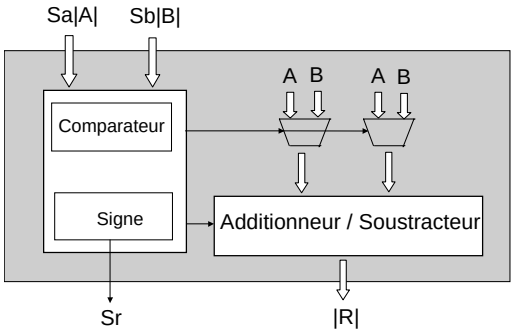


Figure 7.18. *Additionneur de nombre signés*

Un moyen d'éviter ce processus pour effectuer une soustraction est d'utiliser un codage en complément pour représenter les nombres négatifs. Il existe deux formes de compléments: le complément à 2 (complément à b) et le complément à 1 (complément à b-1). Avec de tels codages, toute opération de soustraction se transforme en une opération d'addition entre nombre codés. La structure générale de l'additionneur algébrique est la suivante:

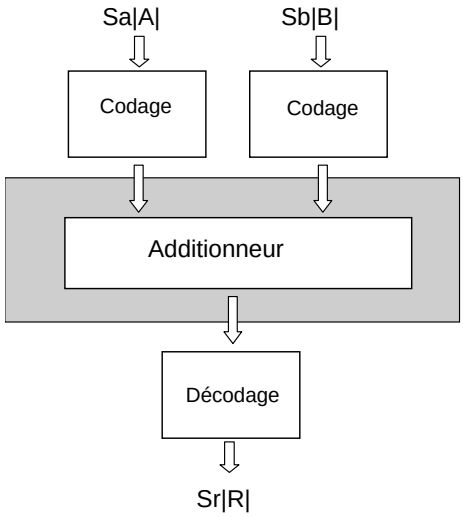


Figure 7.19. *Structure d'un additionneur algébrique*

7.5.1. Processus d'addition / soustraction en complément à 2

En utilisant un codage en complément à 2 pour les nombres négatifs, l'opération de soustraction se transforme en une simple opération d'addition binaire [A-B = A+(-B)].

7.5.1.a. Principe

1. Le bit de retenue de l'addition doit être ignoré dans tous les cas. Le résultat de l'addition est la valeur attendue. Ce résultat peut être positif ou négatif

2. Si le bits de signe du résultat vaut 0, le nombre obtenu est positif et codé en binaire naturel sur les bits significatifs. Si le bits de signe du résultat vaut 1, le nombre obtenu est négatif et codé en complément à 2.

Exemple **:**

```
 14 = 01110       14  =  01110
-13 =-01101    +(-13) =  10011
                          -----
                          00001     Signe=0 =>R=(00001)2=(1)10

 13 =  01101      13  =  01101
-14 = -01110   +(-14) =  10010
                          -----
                          11111     Signe=1 => R=-C2(11111)2
                                               =-(00001)2
                                               =-(1)10
```

Remarque **:** Le résultat de l'opération doit être inférieure à 2^n, c'est à dire qu'en valeur absolue sa représentation binaire comporte n bits au plus. Les cas qui peuvent poser problèmes sont ceux ou le signe des deux opérandes de l'addition est identique. Dans ce cas, il peut y avoir dépassement de capacité et l'addition de deux nombres positifs peut donner un résultat négatif.

```
   15 =   01111       15 = 01111
- (-2)= + 00010     + 2 = 00010
                          -----
                          10001     Signe=1 => R=-C2(10001)2
                                               =-(01111)2
                                               =-(15)10
```

Un bit de signalisation de dépassement de capacité peut être généré (Overflow)

$$Overflow = S(M).S(N).S'(R) + S'(M).S'(N).S(R)$$

7.5.1.b. Démonstration

Afin de démontrer le processus d'addition/soustraction en complément à 2, trois cas sont à envisager.

Cas 1 **:** M>0 et N>0 => signe(M)=signe(N)=0
=> retenue = 0
=> Résultat = M + N

Cas 2 **:** M>0 et N<0

N est représenté par son complément à 2 soit (2^n - N).

On effectue donc l'addition M + (2^n - N).

$$S = (M - N) + 2^n$$
$$= M - N \quad (\text{sur n bits})$$
$$C_2(S) = 2^n - M + N$$
$$= N - M \quad (\text{sur n bits})$$

=> Résultat: R = S si bit de signe = 0
= $-C_2(S)$ si bit de signe = 1

Cas 3 **:** M<0 et N<0

M et N sont représentés par leur complément à 2.

On effectue donc l'addition $(2^n - M) + (2^n - N)$.

$S = (- M - N) + 2^n + 2^n$

$= - M - N$ (sur n bits)

$C_2(S) = 2^n - S$

$= M + N + 2^n$

$= M + N$ (sur n bits)

=> Résultat: $R = -C_2(S)$

7.5.1.c. Additionneur / Soustracteur en complément à 2

Le changement de signe d'une opérande est obtenu en prenant son complément à 2. L'opération A-B se transforme donc en une opération $A+C_2(B)$. La structure d'un opérateur assurant soit l'addition soit la soustraction de deux nombres A et B en fonction d'une commande C peut donc être déterminée très simplement à partir d'un additionneur et d'un bloc multiplexé calculant le complément à 2.

Sachant que C2(N)=C1(N)+1, un tel opérateur peut être optimisé en jouant sur la retenue entrante pour réaliser l'opération +1 et sur le fait que l'opérateur complément à 1 est une simple inversion. Une telle structure est présentée sur la figure 7.20:

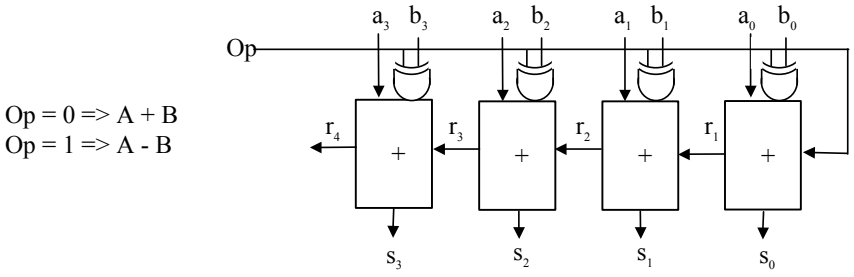


Figure 7.20. *Structure du additionneur / soustracteur en complément à 2*

7.5.2. Processus d'addition / soustraction en complément à 1

En utilisant un codage en complément à pour les nombres négatifs, l'opération de soustraction se transforme en une simple opération d'addition binaire [A-B = A+(-B)], mais dans ce cas, le résultat final n'est obtenu qu'après addition de la retenue au résultat partiel. Cette addition supplémentaire fait que le codage en complément à 1 ne présente que peu d'intérêt par rapport au codage en complément à 2. Le processus d'addition/soustraction en complément à 1 ne sera donc présenté ici qu'à titre pédagogique.

7.5.2.a. Principe

1. Le bit de retenue de l'addition doit être ignoré. Si ce bit vaut 0, le résultat de l'addition est la valeur attendue. Si ce bit vaut 1, le résultat attendu est le résultat de l'addition +1. Le résultat final peut être positif ou négatif.
2. Si le bit de signe du résultat final vaut 0, le nombre obtenu est positif et codé en binaire naturel sur les bits significatifs. Si le bit de signe du résultat final vaut 1, le nombre obtenu est négatif et codé en complément à 1.

Exemple **:**

```
 14 = 01110          14 = 01110
-13 =-01101        +(-13)= 10010
                           -----
                         100000
                       +      1
                          -----
                          00001   Signe=0 => R=(00001)2=(1)10

 13 =  01101          13 = 01101
-14 = -01110        +(-14)= 10001
                           -----
                         011110
                       +      0
                          -----
                          11110     Signe=1 => R=-C1(11110)2
                                               =-(00001)2
                                               =-(1)10
```

Remarque : Le résultat de l'opération doit être inférieure à 2^n, c'est à dire qu'en valeur absolue sa représentation binaire comporte n bits au plus. Les cas qui peuvent poser problèmes sont ceux ou le signe des deux opérandes de l'addition est identique.

```
   15 =   01111      15 =  01111
- (-2)= + 00010     + 2 =  00010
                           -----
                           10001
                         +     0
                           -----
                           10001    Signe=1 => R=-C1(10001)2
                                               =-(01111)2
                                               =-(15)10
```

Un bit de signalisation de dépassement de capacité peut être généré (Overflow)

$$Overflow = S(M).S(N).S'(R) + S'(M).S'(N).S(R)$$

7.5.2.b. Démonstration

Afin de démontrer le processus d'addition/soustraction en complément à 1, trois cas sont à envisager.

Cas 1 : M>0 et N>0 => signe(M)=signe(N)=0
=> retenue = 0
=> Résultat = M + N

Cas 2 : M>0 et N<0

N est représenté par son complément à 1 (2^n-1-N).

On effectue donc l'addition M + (2^n-1-N).

Si le résultat S est inférieur à 2^n (bit de retenue à 0) on obtient:

$S1 = M + (2^n-1 - N)$

$S2 = S1 + 0$

$C1(S2) = 2^n-1 - (M - N + 2^n-1)$

$= N – M$

=> Résultat: $R = -C_1(S2)$

Si le résultat est supérieur à 2^n (bit de retenue à 1) et si l'on ignore le bit de retenue (résultat - 2^n) on obtient :

S1 = M + (2n-1 - N) - 2n
= M - N – 1
S2 = S1 + 1
= M – N
=> Résultat: R = S2

Pour avoir le résultat exact, il faut donc rajouter le bit de retenue au résultat de l'addition.

Cas 3 : M<0 et N<0 => signe(M)=signe(N)=1
=> retenue = 1

N et M sont représentés par leur complément à 1.
On effectue donc l'addition $(2^n-1-M) + (2^n-1-N)$.
Le résultat est supérieur à 2^n (bit de retenue à 1). Si l'on ignore le bit de retenue (résultat - 2^n) on obtient :

$S1 = (2^n-1 - M) + (2^n-1 - N) - 2^n$
$= - M - N + 2^n - 1 - 1$
$S2 = S1 + 1$
$= - M - N + 2^n - 1$
$C1(S2) = 2^n-1 - (- M - N + 2^n-1)$
$= M + N$
=> Résultat: $R = -C_1(S2)$

7.6. Mutiplication binaire

7.6.1. Multiplication par une puissance de 2

Décaler un nombre écrit en base b de k positions vers la droite reviens à multiplier ce nombre par b^{-k}, et le décaler de k positions vers la gauche reviens à le multiplier par b^k.

Soit un nombre N écrit en base b :

$$(N)_b = \sum_{i=-m}^{n} p_i b^i = (p_n\ p_{n-1} \cdots p_1\ p_0\ ,\ p_{-1}\ p_{-2} \cdots p_{-m})_b$$

Ce nombre décalé de k position vers la gauche s'écrit :

$$(p_n\ p_{n-1} \cdots p_1\ p_0\ p_{-1} \cdots p_{-k}\ ,\ p_{-k-1} \cdots p_{-m})_b = \sum_{i=-m}^{n} p_i b^{i+k}$$

$$\sum_{i=-m}^{n} p_i b^{i+k} = b^k * \sum_{i=-m}^{n} p_i b^i = b^k * (N)_b$$

Décaler un nombre binaire de k positions (k positif pour un décalage à gauche et négatif pour un décalage à droite) reviens donc à multiplier ce nombre par 2^k.

Exemple :
$(110,101)_2 = (6,625)_{10}$
$(1,10101)_2 = 2^{-2} * (6,625)_{10} = (6,625 / 4)_{10} = (1,65625)_{10}$
$(11010,1)_2 = 2^{+2} * (6,625)_{10} = (6,625 * 4)_{10} = (26,5)_{10}$

7.6.2. Multiplieur cablé

Le processus de multiplication binaire est en fait beaucoup plus simple que le processus de multiplication décimale dans la mesure les tables de multiplications des chiffres se limitent au ET logique. La table de multiplication binaire est la suivante :

0 * 0 = 0

0 * 1 = 0

1 * 0 = 0

1 * 1 = 1

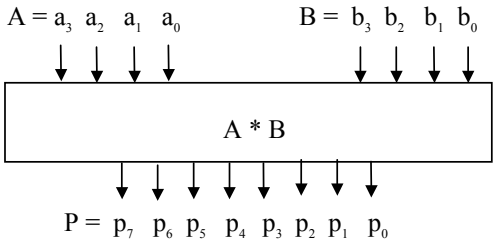
Pour chaque digit du multiplieur égal à 1, un produit partiel non nul est formé. Ce produit est constitué du multiplicande décalé d'un certain nombre de positions de manière à aligner son digit de poids le plus faible avec le 1 correspondant du multiplieur. Le produit final est obtenu par addition de tous les produits partiels.

Le processus de multiplication binaire est illustré par l'exemple suivant:

```
  11010   Multiplicande
    101   Multiplieur
 ------
  11010   Produit partiel
 00000    Produit partiel
11010     Produit partiel
-------
10000010
```

Cette opération de multiplication binaire se résume donc à une somme de produits partiels. Elle peut être réalisée par une structure cablée cascadant des additionneurs (Figure 7.21.).

				A_3	A_2	A_1	A_0	Multiplicande
				B_3	B_2	B_1	B_0	Multiplieur
				---	---	---	---	
				A_3B_0	A_2B_0	A_1B_0	A_0B_0	Produit partiel
			A_3B_1	A_2B_1	A_1B_1	A_0B_1		Produit partiel
		A_3B_2	A_2B_2	A_1B_2	A_0B_2			Produit partiel
	A_3B_3	A_2B_3	A_1B_3	A_0B_3				Produit partiel
---	---	---	---	---	---	---	---	
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	



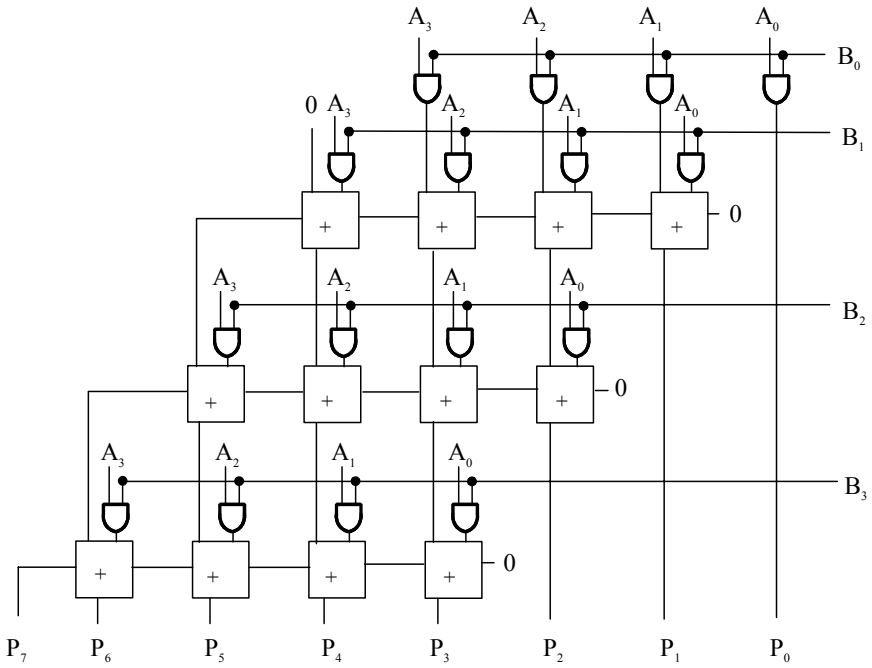


Figure 7.21. *Structure d'un multiplieur cablé*

Remarque : Si les 2 opérandes A et B s'écrivent sur n bits le ($A<2^n$ et $B<2^n$), le produit P s'écrit sur 2n bits ($P<2^{2n}$)

La multiplication des nombre radicaux peut s'effectuer sur le même principe. La virgule est placée en utilisant les mêmes règles que pour la multiplication décimale. Le nombre de digits à droite de la virgule est égal à la somme du nombre de digits à droite de la virgule du multiplicande et du multiplieur.

```
  110.10   Multiplicande
    10.1   Multiplieur
  ------
   11010   Produit partiel
  00000    Produit partiel
 11010     Produit partiel
 -------
10000.010
```

Pour les nombre signés, la technique la plus simple pour effectuer la multiplication de nombres négatifs est d'utiliser le processus précédent avec la valeur absolue du nombre et de déterminer le signe séparément (Signe(R) = Signe(A) ouex Signe(B)). Il est également possible de réaliser la multiplication directement avec des nombres négatifs représentés par leur complément à 2. Ceci est fait couramment en utilisant un transcodage appelé algorithme de BOOTH.

7.6.3. Multiplieur séquentiel

Supposons que l'on dispose de deux registres de n bits A et B pour stocker les opérandes et de deux registres R1 et R2 pour stocker le résultat final. La multiplication de deux nombres A et B peut être réalisée selon l'algorithme suivant :

```
          (A)  1110
          (B)  1011
          ----------
R2R1      0000 0000        Initialisation de R1 et R2
+A*2^0         1110        b0=1 => on ajoute A*2^0
          ----------
R2R1      0000 1110        Décalage de B => B=(0101)
+A*2^1       1 110         b0=1 => on ajoute A*2^1
          ----------
R2R1      0010 1010        Décalage de B => B=(0010)
+0*2^2                     b0=0 => on ajoute 0*2^2
          ----------
R2R1       0010 1010       Décalage de B => B=(0001)
+A*2^3      111 0          b0=1 => on ajoute A*2^3
          ----------
R2R1       1001 1010       Fin
```

D'après cet algorithme, il faut faire des additions de 2n bits entre R1R2 et A, A étant décalé au maximum de (n-1) positions vers la gauche. En réalité, les additions ne portent que sur des opérandes de n bits (résultat sur n+1 bits), le reste du résultat étant la recopie d'une partie de R1R2. Il est donc possible de n'utiliser qu'un additionneur de n bits.

A devant être décalé de R1 à R2, il faudrait donc le stocker dans un registre de 2n bits. En fait il reviens au même de fixer A et de décaler R1R2 vers la droite, à condition de stocker le résultat de l'addition dans R2. On doit donc introduire une bascule R3 (registre 1 bit) pour la retenue éventuelle de l'addition (R2 + A -> R3R2). Cette approche permet de n'utiliser pour A qu'un registre n bits.

```
          (A) 1110
          (B) 1011
            ------------
R3R2R1      0 0000 0000     Initialisation de R3 R2 R1
+A            1110          b0=1 => (R2 + A -> R3R2)
            ------------
R3R2R1      0 1110 0000     Décalage de R3R2R1
R3R2R1      0 0111 0000     Décalage de B => B=(0101)
+A            1110          b0=1 => (R2 + A -> R3R2)
            ------------
R3R2R1      1 0101 0000     Décalage de R3R2R1
R3R2R1      0 1010 1000     Décalage de B => B=(0010)
+0            0000          b0=0 => (R2 + 0 -> R3R2)
            ------------
R3R2R1      0 1010 1000     Décalage de R3R2R1
R3R2R1      0 0101 0100     Décalage de B => B=(0001)
+A            1110          b0=1 => (R2 + A -> R3R2)
            ------------
R3R2R1      1 0011 0100     Décalage de R3R2R1
R3R2R1      0 1001 1010     Fin
```

Si on observe l'évolution de R1 et de B au cours des décalages successifs, on constate que l'on a toujours n zéros qui ne sont d'aucune utilité. On peut encore économiser un registre en plaçant B dans R1, d'ou le schéma minimal de la multiplication.

```
          (A) 1110
              ------------
R3R2R1        0 0000 1011       Initialisation de R3R2R1 (B->R1)
+A              1110            b0(R1)=1 => (R2 + A -> R3R2)
              ------------
R3R2R1        0 1110 1011       Décalage de R3R2R1
R3R2R1        0 0111 0101       b0(R1)=1 => (R2 + A -> R3R2)
+A              1110
              ------------
R3R2R1        1 0101 0101       Décalage de R3R2R1
R3R2R1        0 1010 1010       b0(R1)=0 => (R2 + 0 -> R3R2)
+0              0000
              ------------
R3R2R1        0 1010 1010       Décalage de R3R2R1
R3R2R1        0 0101 0101       b0(R1)=1 => (R2 + A -> R3R2)
+A              1110
              ------------
R3R2R1        1 0011 0101       Décalage de R3R2R1
R3R2R1        0 1001 1010        Fin
```

7.7. Division binaire

La division est la plus complexe des 4 opérations arithmétiques. Le principe de la division décimale classique que l'on apprend à l'école est basée sur le principe d'essais successifs. Par exemple, pour diviser 11 par 4, on doit d'abord s'apercevoir que 11 est supérieur à 4 et ce demander combien de fois 4 va dans 11. Si l'on fait un essai initial avec 3, la multiplication 3*4=12 nous indique que le choix est mauvais (12>11). Il faut recommencer avec 2 etc...

Compte tenu du moins grand nombre de possibilité d'essais, ce principe d'essais successifs et en fait beaucoup plus simple dans la division binaire.

Exemple **:**

```
11  | 4
30  | 2,75
 20
  0
```

```
1011    | 100
-100    | 10,1100
 ----
 0011
 -000
  ----
  0110
  -100
   ----
   0100
   -100
    ----
    0000
```

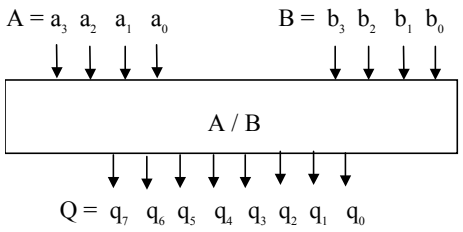
7.7.1. Diviseur cablé

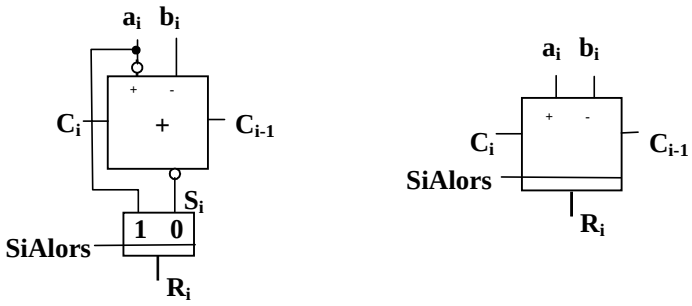
Soit deux nombre binaires A et B. Q représente le quotient de la division de A par B et R les restes partiels. Le principe général de la division binaire peut se mettre sous la forme suivante:

```
Si R>B     Alors Q=1 et R-B
           Sinon Q=0 et R
```

```
                1 0 1 1
0 <-  - 0 1 0 0 |
      -------   |
          1 0   |
0 <-  - 0 1 0 0 |
      -------   |
          1 0 1 |
1 <-  - 0 1 0 0 |
      -------   |
          0 0 1 1
0 <-  - 0 1 0 0 |
      -------   |
        0 0 1 1 0
1 <-  - 0 1 0 0 |
      -------   |
        0 0 1 0 0
1 <-  - 0 1 0 0 |
      -------   |
        0 0 0 0 0
0 <-  - 0 1 0 0 |
      -------   |
        0 0 0 0 0
0 <-  - 0 1 0 0
```

La structure du diviseur binaire peut être directement déduite du principe exposé précédemment.





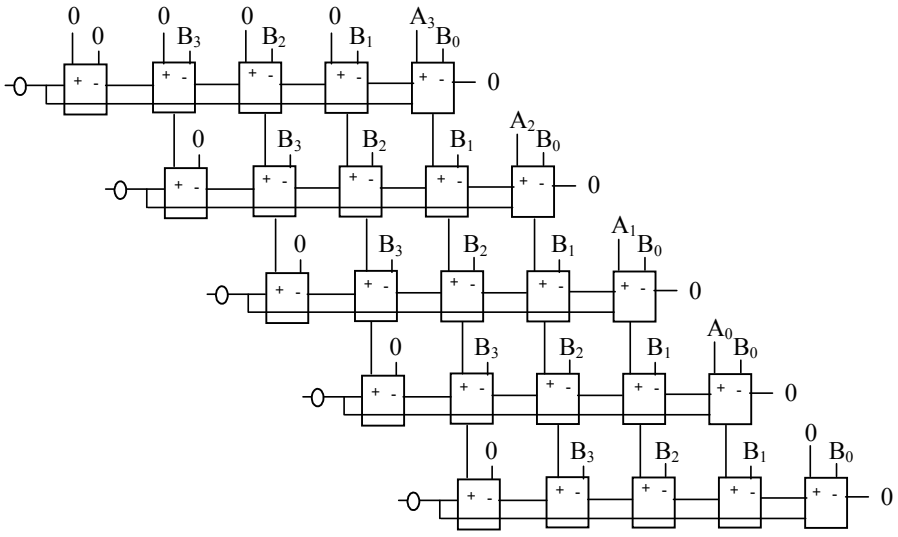


Figure 7.22. *Structure d'un diviseur combinatoire*

7.7.2. Diviseur séquentiel

Tout comme l'opération de multiplication, l'opération de division peut être réalisée de manière séquentielle selon les algorithmes suivants :

7.7.2.a. Algorithme de division avec restauration du dividende

1: Soustraire le diviseur du dividende avec le bits de poids fort du diviseur aligné sur le bit de poids fort du dividende.

```
                    (12/2)          (2/12)
Dividende            1100            10
Diviseur           - 10            - 1100
                   -------         --------
                     0100          - 0100
```

2: Si le résultat est positif (ou nul) un 1 apparaît dans le quotient. Le poids de ce digit est identique à celui du premier digit droit du dividende sur lequel un digit du diviseur a été soustrait.

```
Dividende            1100
Diviseur       - 10
               -------
                0100   => Quotient 1..,
```

Le diviseur est décalé d'un bit vers la droite et est soustrait au dividende partiel (résultat de la soustraction)

```
Dividende partiel     0100
Diviseur            -   10
                    --------
                      0000
```

Le processus est réitéré à partir de là.

2: Si le résultat est négatif, un 0 apparaît dans le quotient. Le poids de ce digit est identique à celui du premier digit droit du dividende sur lequel un digit du diviseur a été soustrait.

```
Dividende      10
Diviseur     - 1100
             --------
             - 0100    => Quotient .,.0
```

Le diviseur est alors rajouté au résultat de manière à restaurer le dividende initial.

```
Diviseur       1100
Résultat     - 0100
             --------
               1000
```

Le diviseur est décalé d'un bit vers la droite et est soustrait au dividende restauré.

```
Dividende            1000
Diviseur       - 1100
               --------
```

00100

Le processus est réitéré à partir de là.

Exemple 1 : (12/5)

```
Dividende               1100
Diviseur              - 101
                      ------
Dividende partiel       0010     => Quotient 1.,
Diviseur décalé       -  101
                      -------
Dividende partiel     - 0011     => Quotient 10,
Ajout du diviseur     +  101
                      -------
Dividende restauré      0010
Diviseur décalé       -   101
                      -------
Dividende partiel     - 00001    => Quotient 10,0
Ajout du diviseur     +   101
                      -------
Dividende restauré      00100
Diviseur décalé       -    101
                       --------
Dividende partiel       000011   => Quotient 10,01
Diviseur décalé       -     101
                      ---------
                        0000001   => Quotient 10,011
```

etc... => Résultat = (1*2 + 0*1 + 0*0,5 + 1*0,25 + 1*0,125 + ...

Exemple 2 **:** (5/12)

```
Dividende                 101
Diviseur                - 1100
                        ------
Dividende partiel       - 0010     => Quotient 0,0
Ajout du diviseur       + 1100
                        -------
Dividende restauré        101
Diviseur décalé         -  1100
                        -------
Dividende partiel       + 01000    => Quotient 0,01
Diviseur décalé         -   1100
                         -------
Dividende partiel       + 000100   => Quotient 0,011
Diviseur décalé         -    1100
                         --------
Dividende partiel       - 0000100    => Quotient 0,0110
Ajout du diviseur          + 1100
                         --------
Dividende  restauré       0001000
Diviseur décalé         -     1100
                          --------
                        - 00000100    => Quotient 0,01100
```

etc... => Résultat = (0*1 + 0*0,5 + 1*0,25 + 1*0,125 + 0*0,0625 +...

7.7.3.b. Algorithme de division sans restauration du dividende

Dans l'algorithme de division sans restauration du dividende, l'étape d'addition du diviseur à un résultat négatif qui permettait de restaurer le dividende initial est supprimée et le diviseur décalé d'une position vers la droite est rajouté au résultat négatif. Cette étape d'addition du diviseur remplace donc les deux étape de rajout du diviseur et de soustraction du diviseur décalé.

Ceci peut être justifié de la manière suivante: Si R représente le dividende partiel négatif et Y le diviseur alors 1/2Y représente le diviseur décalé d'une position vers la droite.

Alg. avec restauration		Alg. sans restauration
R + Y - 1/2Y	=	R + 1/2Y

Exemple **:** (12/5)

```
Dividende              1100
Diviseur             - 101
                     ------
Dividende partiel      0010     => Quotient 1.,
Diviseur décalé      -  101
                     -------
Dividende partiel    - 0011     => Quotient 10,
Diviseur décalé      +   101
                     -------
Dividende partiel    - 00001    => Quotient ..10,0
Diviseur décalé      +    101
                     --------
Dividende partiel      000011   => Quotient ..10,01
Diviseur décalé      -     101
                     ---------
                       0000001  => Quotient ..10,011
```

etc... => Résultat = (1*2 + 0*1 + 0*0,5 + 1*0,25 + 1*0,125 + ...

Chapitre 8

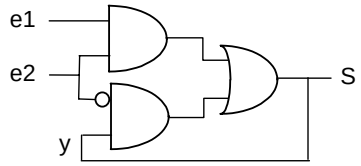
Circuits séquentiels élémentaires

Dans un circuit combinatoire, les valeurs des sorties à un instant donné sont directement imposées par celles des entrées. Ce type de circuits ne permet en fait de traiter qu'une classe restreinte de problèmes ne nécessitant pas de besoin de mémorisation. Les circuits séquentiels ont quant à eux la capacité de mémoriser des informations et par conséquent de traiter des séquences de données. L'étude des circuits séquentiels élémentaires (bascules, registres, mémoires, compteurs) fera l'objet de ce chapitre.

8.1. Bascules

8.1.1. Notions de cicuit séquentiel et de point mémoire

A l'inverse des circuits combinatoires, l'apparition d'un même vecteurs d'entrée sur un circuit séquentiel n'entraîne pas nécessairement des valeurs identiques sur les sorties. A titre d'exemple, considérons le circuit présenté sur la figure 8.1. Les combinaisons (01, 00) et (11, 00) appliquées sur les entrées conduisent à une valeur de sortie différente sur la sortie et ceci, malgré le fait que le dernier vecteur soit identique (00).



e1	e2	S
0	0	S
0	1	0
1	0	S
1	1	1

Figure 8.1. *Circuit séquentiel*

De manière générale, les circuits séquentiels font apparaître des **boucles de rétroaction** qui permettent de mémoriser des informations relatives aux stimuli antérieurs appliqués sur le circuit. La sortie d'un circuit séquentiel est ainsi fonction de variables internes (y) également appelées variables d'état.

Un système séquentiel peut être synchrone ou asynchrone selon qu'il est commandée ou pas par un signal d'horloge. En d'autre terme, un système séquentiel est ***asynchrone*** si à partir de l'instant ou on applique un vecteur d'entrée, son évolution est incontrôlable de l'extérieur. Il est ***synchrone*** si son évolution est contrôlable de l'extérieur par un signal d'horloge.

Les bascules sont les circuits séquentiels élémentaires permettant de mémoriser une information binaire (bit) sur leur sortie. Elles constituent le **point mémoire** élémentaire. Elles peuvent être synchrones ou asynchrones mais toutes ont au minimum trois modes de fonctionnement (et par conséquent au moins 2 commandes):

positionnement de la sortie à 0, positionnement de la sortie à 1 et mémorisation de l'information portée par la sortie.

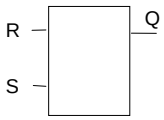
8.1.2. Bascule RS

La bascule RS est le dispositif de mémorisation élémentaire. Cette bascule est asynchrone. Toutes les bascules, y compris les bascules synchrones, ne sont en fait que des évolutions de cette bascule.

La bascule RS est un dispositif à deux entrées R (pour Reset) et S (pour Set) et une sortie Q présentant la propriété suivante :

- lorsque S et R sont à 0, Q conserve sa valeur (Etat mémoire)
- une apparition (même fugitive) de S entraîne durablement Q=1
- une apparition (même fugitive) de R entraîne durablement Q=0

Appelons Q_n, la sortie à l'instant n et Q_{n+1} la sortie à l'instant n+1. Nous pouvons dresser la table de vérité et la table de Karnaugh définissant Q_{n+1}.



Q_n	S	R	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	X
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	X

Q \ SR	00	01	11	10
0	0	0	x	1
1	1	0	x	1

Q_{n+1}

Figure 8.2. *Définition de Q_{n+1}*

On s'aperçoit que l'énoncé du problème est incomplet : les combinaisons telles que SR=11 (3 et 7) ne sont pas définies. Elles correspondent à des ordres d'enclenchement (SET) et de déclenchement (RESET) simultanés. En laissant le problème incomplètement spécifié, on peut obtenir plusieurs équations de la bascule.

$Q_{n+1} = S.R' + Q_n.R' = (S + Q_n)\ R' = ((S + Q_n)' + R)'$ (1)

$Q_{n+1} = S + Q_n.R' = (S' . (Q_n.R)')'$ (2)

Les schémas correspondant à ces équations sont donnés sur la Figure 8.3.

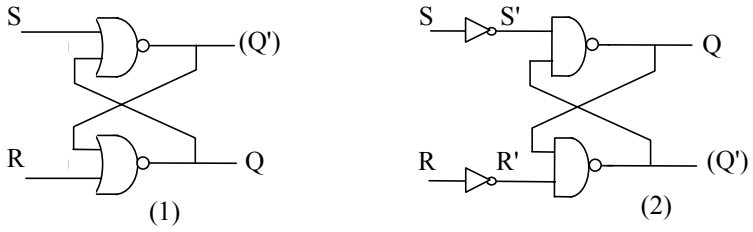


Figure 8.3. *Bascules RS*

Dans le cas (1) les valeurs indéterminées sont affectées à 0. La bascule est dite à déclenchement prioritaire. Dans le cas (2) les valeurs indéterminées sont affectées à 1. La bascule est dite à enclenchement prioritaire.

Sur ce type de bascule, la combinaison (R,S)=(1,1) doit être interdite car elle peut introduire une indétermination. En effet, le passage de la combinaison (R,S)=(1,1) à (R,S)=(0,0) entraîne deux valeurs possibles sur Q selon que R ou S commute en premier. Si l'on interdit la combinaison (R,S)=(1,1) on remarque

que sur les deux structures, la connexion symétrique de la sortie Q porte la valeur Q'. La table de vérité de la bascule RS est représentée sur la Figure 8.4.

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	Interdit

Figure 8.4. *Table de vérité de la bascule RS*

L'avantage de cette bascule est sa simplicité. Ces inconvénients sont le fait qu'elle soit purement asynchrone, qu'elle soit sensible aux parasites (tout événement sur une des entrées affecte la sortie), et qu'il existe un état interdit.

Cette bascule RS est toutefois utilisée dans un certain nombre de procédés tels que les systèmes anti-rebond. La Figure 8.5 montre que la fermeture d'un bouton poussoir peut être soumise à des rebonds et qui engendrer des commutations parasites indésirables.

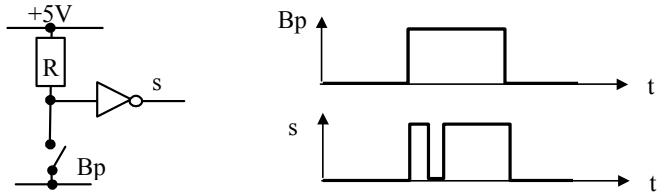


Figure 8.5. *Bouton poussoir sans système anti-rebond*

La Figure 8.5 montre comment à l'aide d'une bascule RS, ces commutations parasites peuvent être éliminées. L'état mémoire permet en effet de filtrer ces transitions.

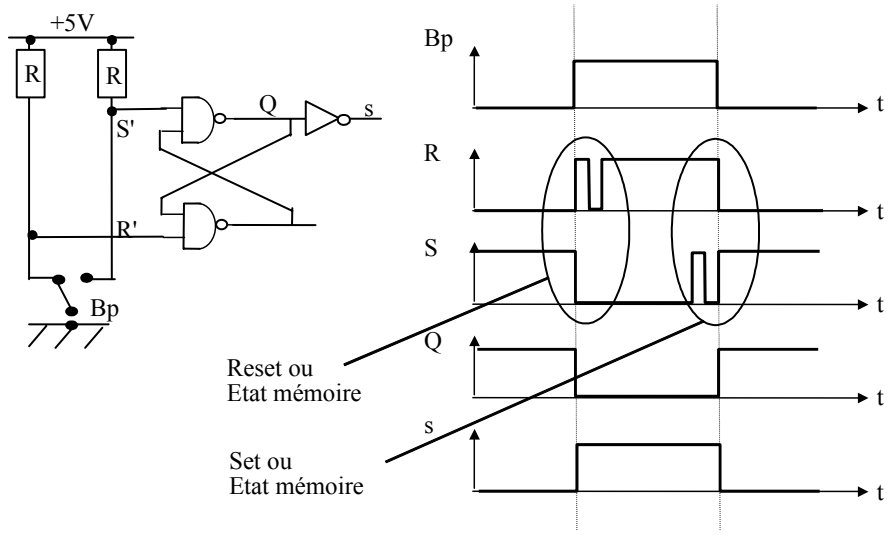


Figure 8.6. *Dispositif anti-rebond*

8.1.3. Bascule RSH

La bascule RSH est une bascule RS synchronisée par un signal d'horloge H. Lorsque H=0, la bascule est dans l'état mémoire. Lorsque H=1, la bascule fonctionne comme une bascule RS. Le schéma de cette bascule est donné sur la figure 8.7.

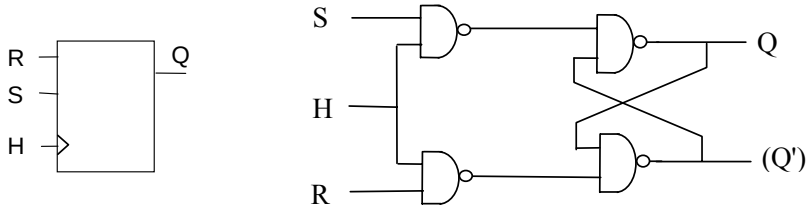
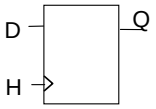


Figure 8.7. *Bascule RSH*

Cette bascule a toujours un état interdit et fonctionne sur les niveaux d'horloge. Tout en restant sensible aux parasites elle l'est moins que la bascule RS puisqu'elle est uniquement sensible sur le niveau haut de l'horloge (plus le niveau haut de l'horloge est réduit, moins cette bascule est sensible aux parasites).

8.1.4. Bascule D-latch

La bascule D-Latch est une bascule ayant une entrée D et une sortie Q synchronisée par un signal d'horloge H. Sa table de vérité est la suivante (Figure 8.8).



D	$Q_{(n+1)}$
0	0
1	1

Si H=1 : $Q_{n+1} = D_n$

Si H=0 : $Q_{n+1} = Q_n$

Figure 8.8. *Table de vérité de la D-latch*

La bascule D-Latch est une bascule qui peut être conçue sur le même principe que la RSH. Dans ce cas, elle est obtenue à partir d'une bascule RSH en ne considérant que les deux combinaisons (R,S) = (0,1) et (1,0). Cette bascule n'a donc pas d'état interdit. Elle est transparente sur le niveau haut de l'horloge (Q=D) et mémorise la valeur de sortie sur le niveau bas. Ce dispositif est en fait l'élément mémoire de base.

La structure d'un tel dispositif peut être obtenue à partir de la structure de la bascule RSH mais également directement en repartant du cahier des charges. Deux schémas à portes de cette bascule sont donnés sur la Figure 8.9.

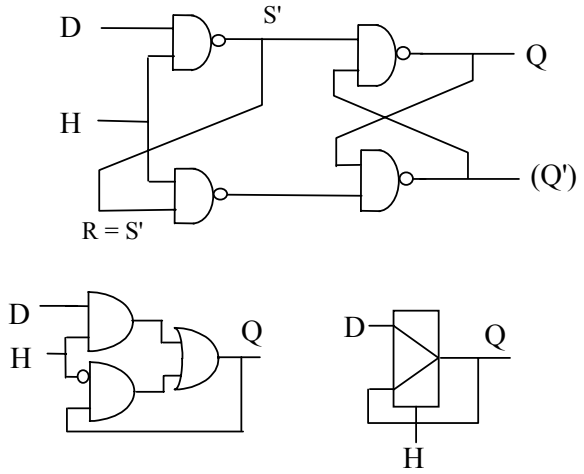


Figure 8.9. *Bascule D-Latch*

Remarque : Bien qu'il ne soit pas question ici d'implantation technologique, nous pouvons souligner que l'intérêt d'un tel dispositif, outre le fait qu'il constitue l'élément mémoire de base, réside dans la compacité de son implantation en logique 3 états et notamment en technologie CMOS. La figure 8.10 présente la structure symbolique d'une telle bascule dans ces technologies.

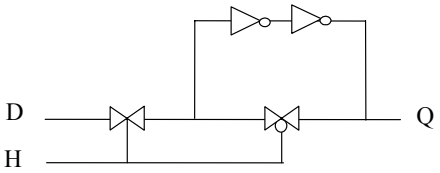
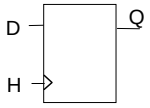


Figure 8.10. *Bascule D-Latch en logique 3 états*

8.1.5. Bascule D

La bascule D est la cellule mémoire fondamentale utilisée dans la grande majorité des applications. Ce dispositif fonctionne sur un front d'horloge (front montant ou front descendant). La table de vérité d'une bascule D est la suivante (Figure 8.11).



D	$Q_{(n+1)}$
0	0
1	1

Si front(montant)sur H : $Q_{n+1} = D_n$

Sinon : $Q_{n+1} = Q_n$

Figure 8.11. *Table de vérité de la bascule D*

Cette bascule peut être réalisée en cascadant 2 bascules D-Latch (dispositif maître esclave) comme indiqué sur la Figure 8.12.

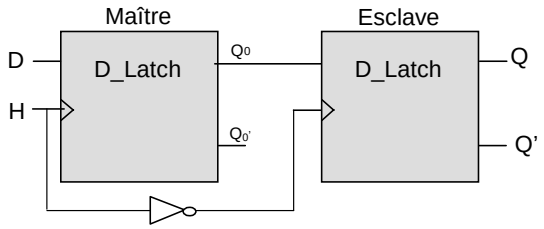
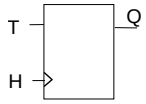


Figure 8.12. *Bascule D*

Un des gros avantages de ce dispositif est sont immunité aux transitoires ou parasites. En effet, la mise en cascade de 2 D-latch fonctionnant sur des niveaux opposés de l'horloge permet d'éviter la propagation de signaux transitoires à travers la bascule.

8.1.6. Bascule T

La bascule T (T pour Toggle) est un élément qui interprète son entrée de commande T, non comme une entrée à mémoriser, mais comme un ordre de changement d'état. Cette bascule est particulièrement intéressante à utiliser pour certaines applications et notamment pour la réalisation de compteurs. Sa table de vérité est la suivante (Figure 8.13).



T	$Q_{(n+1)}$
0	Q_n
1	Q_n'

Si front (montant) sur H et si T=1: $Q_{n+1} = Q_n'$

Sinon : $Q_{n+1} = Q_n$

Figure 8.13. *Table de vérité de la bascule T*

Une telle bascule peut être réalisée à partir d'une bascule D avec :
$D_n = T'.Q_n + T.Q_n'$

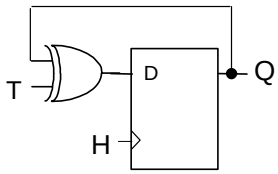
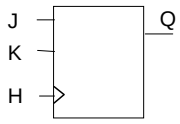


Figure 8.14. *Bascule T réalisée avec une bascule D*

8.1.7. Bascule JK

La bascule JK est une bascule fonctionnement sur front d'horloge (maître-esclave) comportant 2 entrées (J et K). La table de vérité de cette bascule est donnée sur la Figure 8.15.



JK	$Q_{(n+1)}$
00	Qn
01	0
10	1
11	Qn'

Si front(montant)sur H : $Q_{n+1} = J.Q_n' + K'.Q_n$

Sinon : $Q_{n+1} = Q_n$

Figure 8.15. *Table de vérité de la bascule JK*

Comme la bascule T, cette bascule peut être réalisée à partir d'une bascule D de la manière suivante (Figure 8.16) :

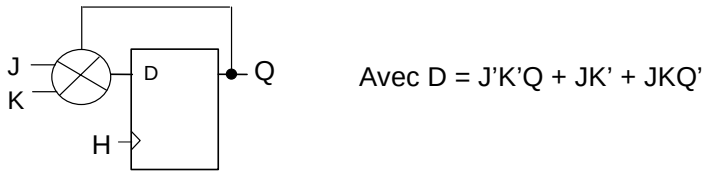


Figure 8.16. *Bascule JK réalisée avec une bascule D*

8.1.8. Initialisation des bascules

Une bascule, quel que soit son type, peut être initialisée par l'intermédiaire d'une combinaison de ses entrées et d'un coup d'horloge. Sur la plupart des bascules, il existe également des signaux spécifiques d'initialisation asynchrone, c'est à dire ne nécessitant pas de coup d'horloge. Ces entrées d'initialisation sont généralement appelées « Clear » pour la remise à 0 de la sortie et « Preset » pour la remise à 1. Elle agissent sur l'étage esclave des bascules. Un exemple de bascule avec entrées d'initialisation asynchrone « Clear » et « Preset » est donné sur la Figure 8.17.

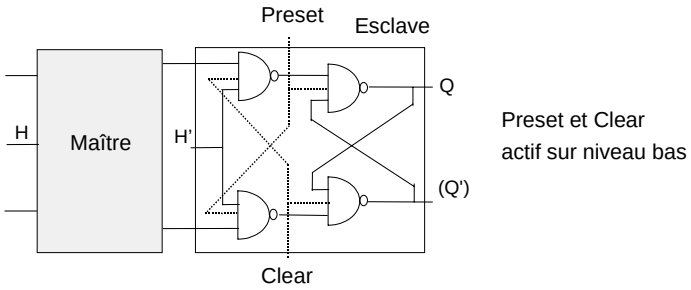


Figure 8.17. *Bascule avec Clear et Preset*

L'initialisation d'une bascule peut également être envisagée de manière synchrone. Pour cela il faut prévoir un dispositif intervenant directement sur les entrées synchrones des bascules comme présenté sur la figure 8.18 dans le cas d'une bascule D.

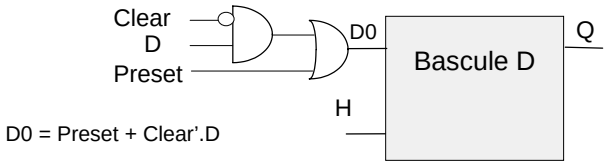


Figure 8.18. *Initialisation synchrone d'une bascule D*

8.1.9. Inhibition du fonctionnement des bascules

Pour certaines applications utilisant des bascules, le fonctionnement des bascules doit pouvoir être inhibé (conservation de l'état quoi qu'il arrive notamment sur l'horloge) par un signal d'inhibition (Inib). Pour cela plusieurs solution peuvent être envisagées mais certaines sont à proscrire absolument sus peine de sérieux ennuis.

Par exemple, le blocage, par une porte, du signal d'horloge pour maintenir l'état d'une bascule, est une erreur fondamentale que l'on rencontre parfois. Cette approche provoque d'une part, des décalages temporels entre les signaux d'horloge *(clock skew)* qui peuvent conduire à comportements imprévisibles des bascules concernés et d'autre part, des possibilités d'impulsions parasites sur les signaux d'horloge qui peuvent également conduire à des fonctionnement imprévisibles.

Ainsi, la seule solution viable pour inhiber le fonctionnement d'une bascule est d'agir sur les entrées synchrones des bascules comme présenté sur la figure 8.19 dans le cas d'une bascule D.

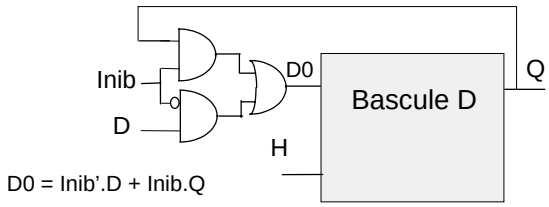


Figure 8.19. *Inhibition du fonctionnement d'une bascule D*

8.1.10. Paramètres temporels des bascules

Pour qu'une bascule fonctionne correctement, il est nécessaire que le signal présent sur les entrées de la bascule (D ou JK) soit stabilisé depuis un certain temps lorsque le front d'horloge actif intervient (temps de « setup ») et reste stable pendant un certain temps après ce front d'horloge (temps de « hold » ou de maintien).

D'autre part, la commutation des sorties d'une bascule se fait avec un certain temps de retard par rapport au signal qui a produit cette commutation (Horloge, Reset ou Preset). Ces retards peuvent être différents selon le signal qui a produit la commutation, mais également selon que la commutation du signal de sortie est montante on descendante. Ces retards seront notés TpLH et TpHL pour « Temps de Propagation Low High » et « Temps de Propagation High Low ».

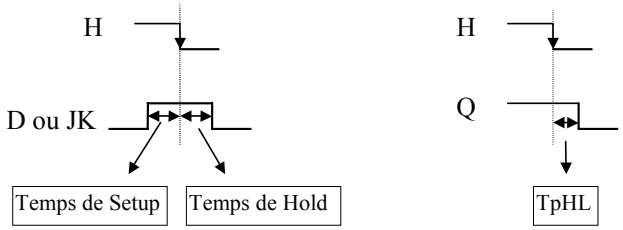


Figure 8.20. *Paramètres temporels des bascules*

8.2. Registres

Les registres sont des associations de bascules permettant de mémoriser et de réaliser certaines opérations sur des mots logiques.

8.2.1. Registre de mémorisation

Le registre de mémorisation est le registre élémentaire. Il est constitué d'une juxtaposition de bascules permettant de mémoriser un mot binaire (Figure 8.21). Ce registre est également appelé registre à entrées parallèles.

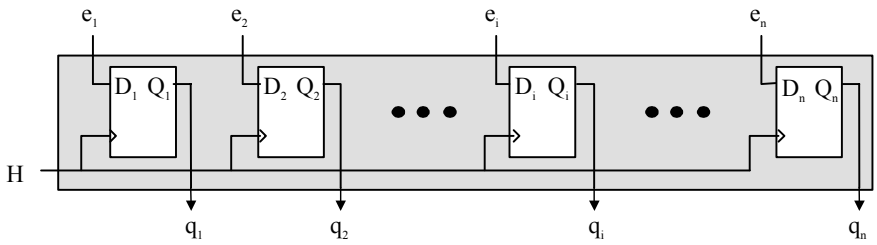


Figure 8.21. *Registre de mémorisation*

8.2.2. Registre de mémorisation avec signal d' inhibition

Le registre de mémorisation présenté précédemment n'est réellement utilisable dans la pratique que s'il est muni d'une entrée de contrôle permettant d'inhiber son chargement lorsque celui ci n'est pas souhaité (cf §.8.4.). Le registre présenté sur la Figure 8.22 est un registre de mémorisation commandé par un signal d'inhibition C.

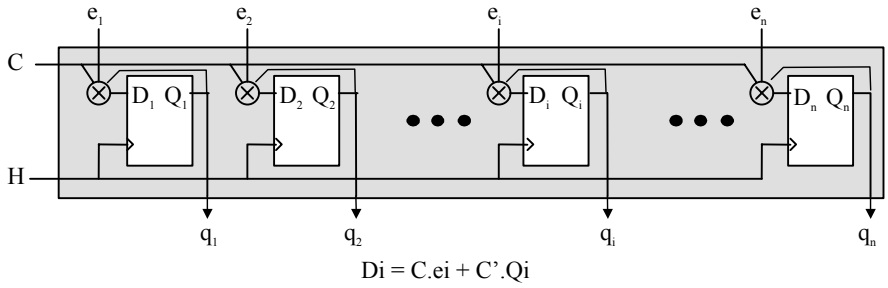


Figure 8.22. *Registre de mémorisation avec signal d'inhibition*

8.2.3. Registre à décalage

Le registre à décalage est une association de bascules permettant de décaler un mot binaire. L'entrée d'un mot dans le registre peut se faire, en fonction d'une commande C, soit par chargement parallèle comme précédemment soit par décalage à partir d'une entrée série. Le registre présenté sur la Figure 8.23 est un registre à décalage à droite. L'entrée e_1 joue à la fois le rôle d'entrée parallèle et d'entrée série.

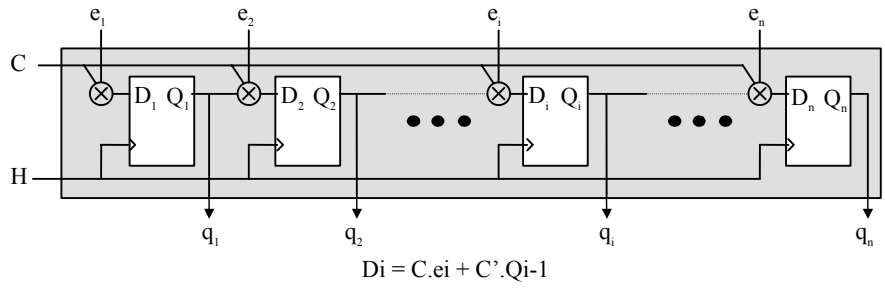


Figure 8.23. *Registre à décalage à droite*

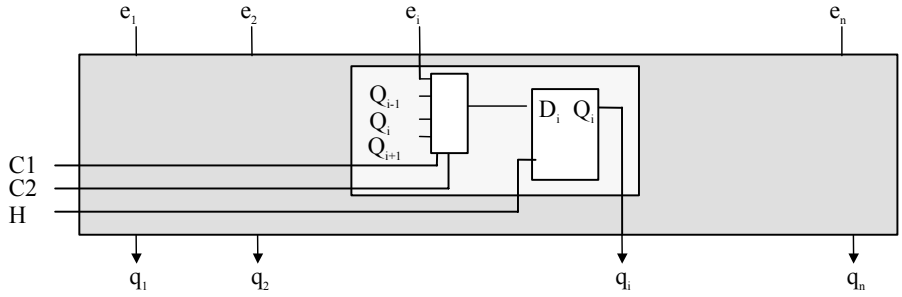
Remarque : Un registre à décalage à droite peut être utilisé comme un diviseur par 2 alors qu'un registre à décalage à gauche peut être utilisé comme un multiplieur par 2.

8.2.4. Registre universel

Le registre universel est une association de bascules permettant quatre modes de fonctionnement commandés par deux variables C1 et C2.

C1C2 = 00	Chargement parallèle
C1C2 = 01	Décalage à droite
C1C2 = 10	Décalage à gauche
C1C2 = 11	Inhibition de l'horloge.

Pour permettre ces quatre modes de fonctionnement, chacune des bascules est précédée d'un multiplexeur. L'entrée D de chaque bascule est ainsi fonction du mode de fonctionnement désiré (Figure 8.24).



Di = C1'.C2' .ei + C1'.C2.Qi-1 + C1.C2'.Qi+1 + C1.C2.Qi

Figure 8.24. *Registre universel*

Remarque : L'entrée e_1 peut jouer simultanément le rôle d'entrée parallèle et d'entrée série gauche. L'entrée e_n peut jouer simultanément le rôle d'entrée parallèle et d'entrée série droite.

8.3. Mémoires

Dans un ordinateur, les informations (programmes, données provenant de l'extérieur, résultats intermédiaires, données à transférer à l'extérieur, ...) doivent en général être conservées pendant un certain temps pour permettre leur exploitation. La conservation de ces informations est dévolu soit à des registres, soit à des structures plus adaptées aux grandes capacités de stockage : les mémoires vives (RAM).

8.3.1. Mémoires vives

Les mémoires vives ou RAM sont des mémoires à lecture et écriture qui permettent d'enregistrer des informations, de les conserver et de les restituer. RAM signifie Random Access Memory. Littéralement cela se traduit par mémoire à accès aléatoire. En fait, l´accès à une RAM n´a rien d´aléatoire. Ce qu´il faut comprendre, c´est qu´on peut accéder à n´importe quelle partie de la mémoire directement, sans obligation technique particulière.

Dans une mémoire vive, l'information élémentaire, ou bit, est mémorisée dans une cellule ou point mémoire. Ces cellules sont groupées en mots de n bits, c'est-à-dire que les n bits sont traités (écrits ou lus) simultanément. Les cellules sont arrangées en bloc mémoire. L'organisation matricielle des blocs mémoires permet d'optimiser la structure tant d'un point de vue surface (adressage des mots) que temps d'accès. (éviter des pistes trop longues pour la distribution des différents signaux aux cellules).

Extérieurement, et en ne tenant compte que des signaux logiques, un bloc mémoire peut être représenté comme sur la figure 8.25. Pour pouvoir identifier individuellement chaque mot on utilise k lignes d'adresse

(signal adr). La taille d'un bloc mémoire est donc 2^k, le premier mot se situant à l'adresse 0 et le dernier à l'adresse 2^k - 1. Une commande (R/W) indique si la mémoire est accédée en écriture (l'information doit être mémorisée) ou en lecture (l'information doit être restituée). Une commande (CS) permet d'activer le fonctionnement de la mémoire (en lecture ou en écriture) ou de l'inhiber. Sur ce schéma on distingue deux canaux de n lignes en entrée et en sortie, mais dans d'autres cas les accès en entrée et en sortie peuvent être confondus en un seul canal bidirectionnel.

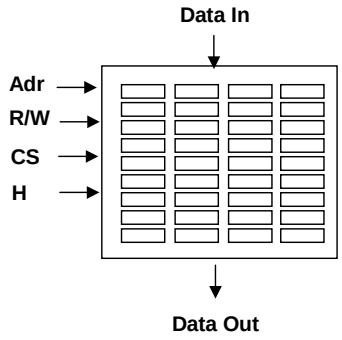


Figure 8.25. *Architecture générale d'une mémoire*

Parmi les caractéristiques d'une mémoire nous trouvons la capacité et le format. La capacité représente le nombre total de bits et le format correspond à la longueur des mots. Le nombre de bits d'adresse k définit le nombre total de mots de la mémoire, si n est le nombre de bits par mot, la capacité de la mémoire est donnée par :

$$\text{Capacité} = 2^k \text{ mots} = 2^k \text{ x n bits}$$

Cette capacité est exprimée en multiple de 1024 ou kilo. La table suivante résume la valeur des autres préfixes utilisés pour exprimer les capacités des mémoires :

Symbole	Préfixe	Capacité
1 k	(kilo)	$2^{10} = 1024$
1 M	(méga)	$2^{20} = 1048576$
1 G	(giga)	$2^{30} = 1073741824$
1 T	(tera)	$2^{40} = 1099511627776$

La figure 8.26, présente une organisation logique possible pour une mémoire de 16 mots de n bits. Ici chaque mot est stocké dans une case de n bits, tel un registre. Cette case reçoit en entrée n lignes de données et une ligne de chargement. Elle dispose de n lignes de sortie fournissant le contenu du registre. Chacune de ces lignes est commandée par une porte "3 états". Ces cases sont organisées en une matrice de 4 lignes et 4 colonnes. Les 4 bits d'adresse (Adr) sont séparés en deux groupes, 2 bits pour identifier la ligne (A_L) et 2 bits pour la colonne (A_C). Les décodeurs de ligne et de colonne permettent de sélectionner les connexions à activer pour adresser la cellule souhaitée.

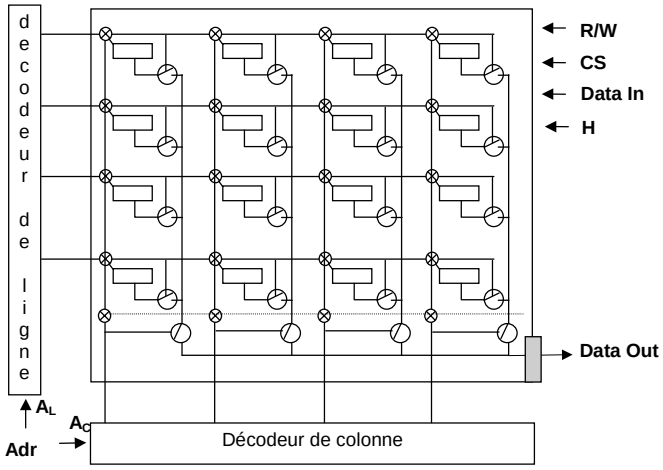


Figure 8.26. *Architecture interne d'une mémoire*

En lecture, les quatre cases de la ligne sélectionnée fournissent leur contenu sur les quatre bus verticaux. Une seule des quatre portes "3 états", au bas du schéma, est connectée à la sortie S du boîtier. Cette porte "3 états" fournit une amplification des signaux (registre de sortie).

En écriture, le mot à charger doit être présent sur l'entrée « Data In » du circuit. Ces données sont distribuées simultanément sur toutes les cellules de n bits. La ligne désignée par l'adresse A_L est à 1. Le signal de chargement est transmis à la seule colonne identifiée par l'adresse A_C. Seul le registre à l'intersection de cette ligne et de cette colonne est donc chargé.

8.3.2. Mémoires RAM Statiques / Dynamiques

Il existe deux grandes familles de mémoires RAM : les RAM statiques (SRAM) ou les RAM dynamiques (DRAM).

Dans le cas des RAM statique, le point mémoire élémentaire est une bascule (Figure 7.27). Les bascules garantissent la mémorisation de l'information aussi longtemps que l'alimentation électrique est maintenue sur la mémoire.

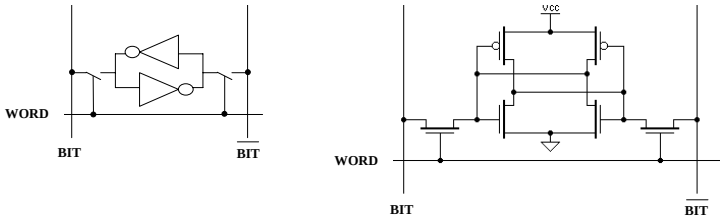


Figure 8.27. *Cellule mémoire SRAM*

Dans le cas des mémoires dynamiques (DRAM), l'élément de mémorisation est un condensateur (capacité) commandée par un transistor (Figure 8.28). L'information est mémorisée sous la forme d'une charge électrique stockée dans le condensateur. Cette technique permet une plus grande densité d'intégration, car un point mémoire nécessite environ deux à quatre fois moins de place que dans une mémoire statique. Par contre, du fait des courants de fuite le condensateur a tendance à se décharger. C'est pourquoi ces mémoires doivent être rafraîchies régulièrement pour entretenir la mémorisation : il s'agit de lire l'information avant qu'elle n'ait

totalement disparu et de la recharger. Ces mémoires sont dites RAM dynamique (DRAM) du fait de cette opération de rafraîchissement.

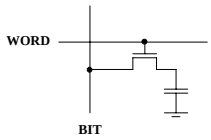


Figure 8.28. *Cellule mémoire DRAM*

Ce rafraîchissement indispensable a plusieurs conséquences. Tout d'abord il complique la gestion des mémoires dynamiques car il faut tenir compte des actions de rafraîchissement qui sont prioritaires. D'autre part, la durée de ces actions augmente le temps d'accès aux informations. Le temps d'attente des données est variable selon que la lecture est interrompue ou non par des opérations de rafraîchissement et la quantité de cellules à restaurer. Il faut donc se placer dans le cas le plus défavorable pour déterminer le temps d'accès à utiliser en pratique.

En général les mémoires dynamiques, qui offrent une plus grande densité d'information et un coût par bit plus faible, sont utilisées pour la mémoire centrale, alors que les mémoires statiques, plus rapides, sont utilisées pour les caches.

8.4. Compteurs / décompteurs

8.4.1. Définitions

Un compteur est une association de n bascules permettant de décrire, au rythme d'une horloge, une séquence déterminée qui peut avoir au maximum 2^n combinaisons différentes. Les combinaisons apparaissent toujours dans le même ordre.

Définition : Une combinaison de sortie d'un compteur est appelée ***état***. Le nombre d'états différents pour un compteur est appelé le ***modulo*** ~ de ce compteur.

Un compteur modulo N démarrant à 0 et comptant dans l'ordre binaire naturel compte de 0 à N-1. Le graphe présenté sur la figure 29 est le graphe d'un compteur binaire modulo 8.

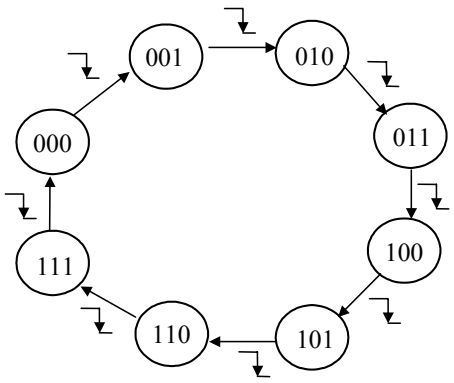


Figure 8.29. *Graphe d'un compteur modulo 8*

8.4.2. Compteurs « asynchrones »

Ce type de structure est présenté ici pour des raisons pédagogiques, mais comme nous le verrons au cours de cet exposé il n'est absolument pas à recommander car pouvant être la source de nombreux ennuis.

Pour construire un compteur, nous pouvons remarquer qu'une bascule T dont l'entrée est à 1 fonctionne en diviseur de fréquence. Il en est de même que lorsque la sortie complémentée d'une bascule D est rebouclée sur l'entrée D ou que les entrées d'une bascule JK sont égales à 1, ces bascules (Figure 8.30).

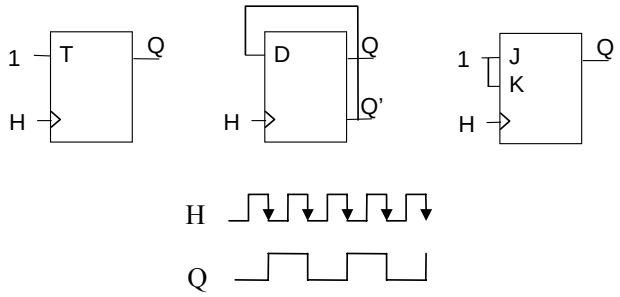


Figure 8.30. *Bascules montées en diviseur de fréquence*

8.4.2.a. Compteurs « asynchrones » modulo 2^n

En cascadant les bascules selon le schéma de la Figure 8.31, on réalise un dispositif répondant au cahier des charges fonctionnel d'un compteur modulo 2^n (n étant le nombre de bascules) . Ce compteur est appelé compteur « asynchrone » du fait que toutes les bascules ne sont pas commandées par le même signal d'horloge. Le chronogramme correspondant est présenté sur la Figure 8.32.

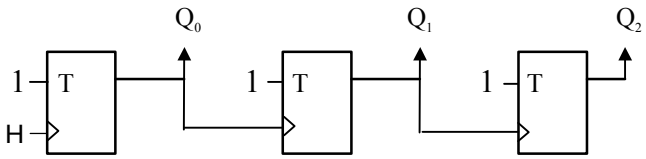
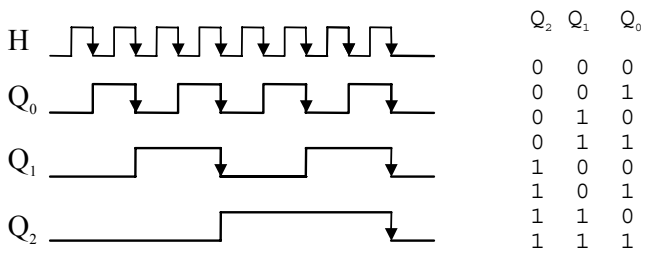


Figure 8.31. *Compteur modulo 8 asynchrone*



Q_2	Q_1	Q_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Figure 8.32. *Chronogramme du compteur par 8*

Pour réaliser un décompteur il suffit de considérer sur les sorties Q' des bascules (Figure 8.33) ou de réaliser le même montage avec des bascules fonctionnant sur front montant (Figure 8.34).

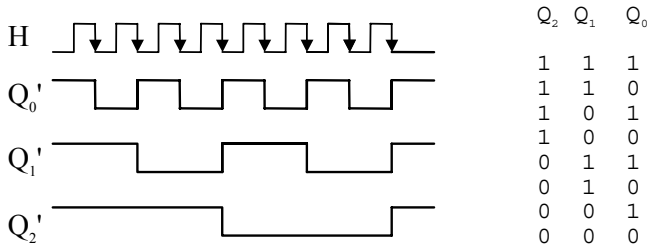


Figure 8.33. *Chronogramme du décompteur par 8*

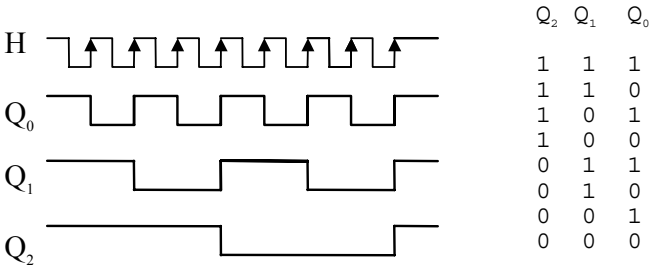


Figure 8.34. *Décompteur par 8 avec bascules fonctionnant sur fronts montants*

8.4.2.b. Compteurs « asynchrones » modulo différent de 2^n

Pour réaliser un compteur ou un décompteur dont le modulo n'est pas une puissance de 2, une solution qui pourrait être envisagée est d'agir sur l'entrée « Clear » lorsque la combinaison correspondant au modulo du compteur ce produit sur les sorties de celui ci.

Exemple : La Figure 8.35 présente la structure d'un compteur asynchrone par 6 ainsi que le chronogramme associé. Pour réaliser un compteur par 6, il suffit de détecter la combinaison $Q_2Q_1Q_0 = 110$ et de la renvoyer sur le signal « Clear »

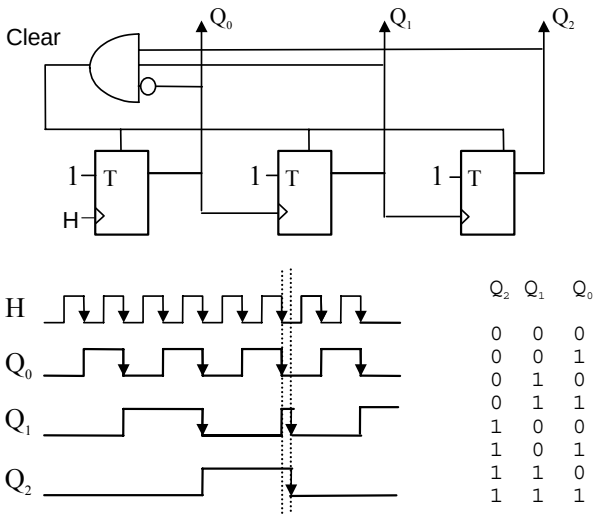


Figure 8.35. *Compteur asynchrone par 6*

Si la démarche suivie pour construire ces compteurs (décompteurs) peut paraître attrayante, les structures obtenues présentent des inconvénients majeurs qui vont jusqu'à remettre en cause le fonctionnement des structures précédemment présentées.

Les bascules ne commutant pas sur le même signal d'horloge, les retards de commutation se cumulent sur chacune des bascules du compteur. En effet, c'est la commutation de la première bascule qui entraîne l'activation de la seconde qui elle même entraîne l'activation de la troisième, etc. Ainsi, la fréquence maximum de fonctionnement F_H d'un compteur modulo n, constitué de n bascules de délai de propagation Dp dépend du nombre de bascules du compteur et donc du modulo du compteur. Cette fréquence peut être établie comme suit :

$Tmax = Dp * n$	Délai de propagation du compteur
$T_H \geq Tmax$	Période de l'horloge
$F_H \leq 1/(Tmax) = 1/(n * Dp)$	Fréquence de l'horloge

D'autre part, ces retards de commutation introduisent des états transitoires relativement conséquents (Figure 8.36).

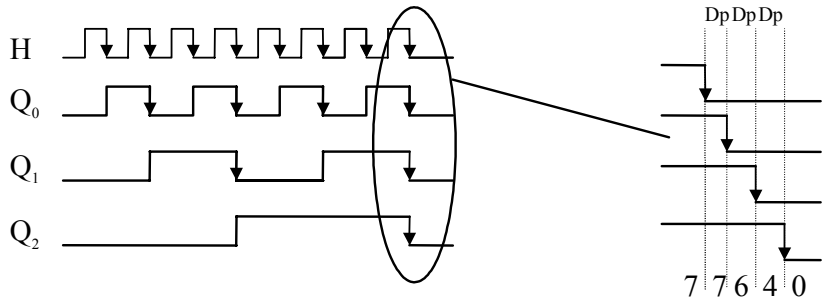


Figure 8.36. *Etats transitoires*

Mais l'inconvénient majeur est lié au fait que cette structure nécessite de la logique sur des signaux asynchrones (Horloge générée par une bascule et Clear généré par une structure combinatoire). Cette logique peut engendrer (ou propager) des transitoires ou des courses critiques entre signaux d'Horloge et de Clear qui peuvent entraîner des dysfonctionnements du compteur. Ce type de structures, qui a été présenté pour raisons pédagogiques, est donc à proscrire impérativement.

8.4.3. Compteurs synchrones

Un compteur synchrone est une structure ou toutes les bascules reçoivent le même signal d'horloge (Figure 8.37). La fonction comptage ou décomptage est réalisée par l'intermédiaire des fonctions appliquées sur les entrées synchrones des bascules.

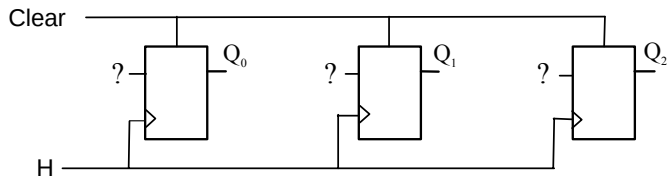


Figure 8.37. *Structure générale d'un compteur par 8 synchrone réalisé avec des bascules JK*

8.4.3.a. Compteurs « synchrones » modulo 2^n

Pour que le compteur décrive une séquence déterminée, il faut commander les entrées des bascules (T, D ou JK) de façon adéquate. Pour cela, on peut remarquer sur la table de vérité du compteur (Figure 8.38) que le bit de poids faible change à tous les coups d'horloge et que qu'un bit quelconque change lorsque tous les bits de droite sont égaux à 1.

Q_2	Q_1	Q_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Figure 8.38. *Table de vérité d'un compteur par 8*

Sachant que sur une bascule T (JK), il y a inversion de la sortie pour T = 1 (JK = 11), on peut en déduire les entrées de chacune des bascules et par conséquent la structure des compteurs synchrones (Figure 8.39).

$T_0 = 1$

$T_1 = Q_0$

$T_2 = Q_0.Q_1$

$T_n = Q_0.Q_1....Q_{n-1}$

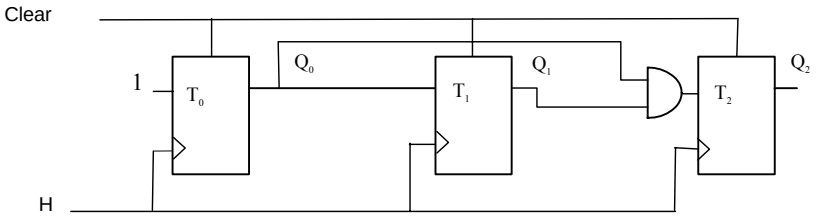


Figure 8.39. *Compteur par 8 synchrone*

Le résonnement fait précédemment avec des bascules T (JK) peut être mené à l'identique avec des bascules D sachant que les deux structures présentées sur la figure 8.40 sont fonctionnellement parfaitement équivalentes.

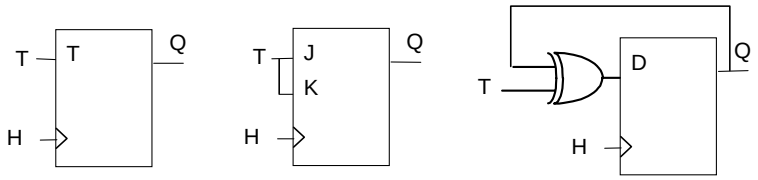


Figure 8.40. *Bascules T, JK, D*

La fréquence maximum de fonctionnement F_H d'un compteur modulo n, constitué de n bascules de délai de propagation Dp peut être établie comme suit :

Tmax = Dp	Délai de propagation du compteur
$T_H \geq$ Tmax	Période de l'horloge

$F_H \leq 1/(Tmax) = 1/Dp$ Fréquence de l'horloge

Le compteur synchrone est donc plus rapide que le compteur asynchrone puisque les délais de propagation des bascules ne sont pas cumulés. D'autre part, si l'on suppose que toute les bascules on le même délai de propagation il n'y a pas d'état transitoire sur la sortie. Dans la pratique, ce n'est bien évidemment pas le cas car les délais de propagation de bascules peuvent être différents (temps de montée et de descente différents, charges différentes etc.). Toutefois, la durée de ces transitoires et réduite à la différence de fonctionnement des bascules et en aucun cas n'est aussi importante qu'en asynchrone.

De la même manière que dans le cas asynchrone, un décompteur peut être obtenu en sortant sur les sorties Q' du compteur. On peut également réaliser un décompteur en remarquant sur la table de vérité (Figure 8.41) que le bit de poids faible change à tous les coups d'horloge et que qu'un bit quelconque change lorsque tous les bits de droite sont égaux à 0.

Q_2	Q_1	Q_0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

Figure 8.41. *Table de vérité d'un décompteur par 8*

Sachant que sur une bascule T, il y a inversion de la sortie pour T=1, on peut en déduire les entrées de chacune des bascules et par conséquent la structure des décompteurs synchrones (Figure 8.42).

$T_0 = 1$

$T_1 = Q_0'$

$T_2 = Q_0'.Q_1'$

$T_n = Q_0'.Q_1'....Q_{n-1}'$

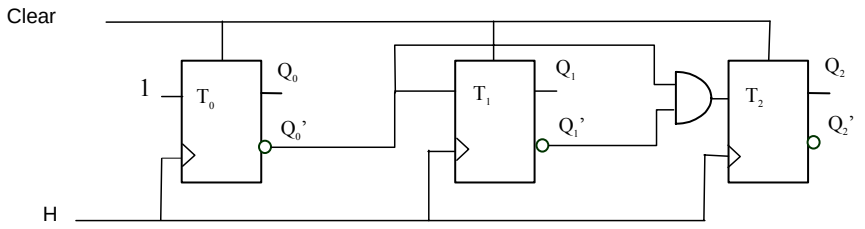


Figure 8.42. *Décompteur par 8 synchrone*

Par le même raisonnement, on peut déterminer la structure d'un compteur / décompteur synchrone (Figure 8.43) dont le mode comptage ou décomptage est commandé par une commande C (C=0 => Comptage, C=1 => Décomptage).

$T_0 = 1$

$T_1 = C'.Q_0 + C.Q_0' = C \oplus Q_0$

$T_2 = C'.Q_0.Q_1 + C.Q_0'.Q_1'$

$T_n = C'.Q_0.Q_1...Q_{n-1} + C.Q_0'.Q_1'....Q_{n-1}'$

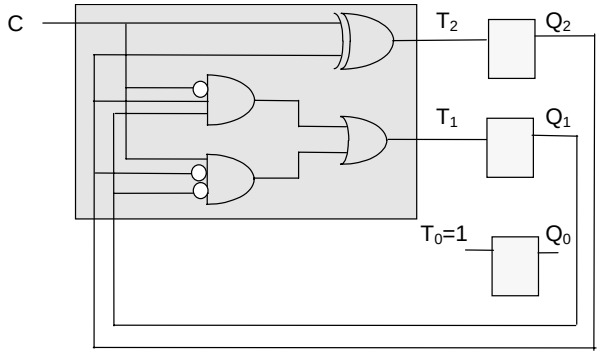


Figure 8.43. *Schéma logique du compteur / décompteur par 8*

Remarque : Le système précédent permet de basculer du mode comptage au mode décomptage et inversement sans modification de l'état courant. Ce ne peut être le cas d'un système basé sur fonctionnement asynchrone ou d'un système basé sur un compteur synchrone avec sortie Q et Q' multiplexées.

8.4.3.b. Compteurs « synchrones » modulo différent de 2^n

Pour réaliser un compteur, un décompteur ou un compteur / décompteur dont le cycle n'est pas une puissance de 2, on pourrait envisager, comme en asynchrone, d'agir sur l'entrée « Clear » lorsque la combinaison correspondant au modulo du compteur ce produit sur les sorties de celui ci. Mais, comme il a été précisé précédemment, cette solution est à proscrire absolument car les transitoires produits ou transmis par la logique sur le signal asynchrone de Clear risquent d'entraîner un dysfonctionnement de la structure.

Une autre solution permettant de réaliser un compteur, un décompteur ou un compteur / décompteur dont le cycle n'est pas une puissance de 2, est de redéfinir les fonctions d'entrée des bascules pour réaliser la fonction souhaitée.

Exemple 1 : Compteur par 6.

Le fonctionnement du compteur par 6 doit rester identique à celui du compteur par 8 tant que la combinaison 5 n'est pas présente sur les sorties du compteur. Lorsque la combinaison 5 est présente, les fonctions d'entrée des bascules doivent être modifiées. Au lieu de passer de 101 à 110, il faut passer de 101 à 000.

Soit C_5 un flag prévenant qu'on est sur la combinaison 5. $C_5 = Q_2.Q_1'.Q_0$

En reconsidérant le fonctionnement des bascules lorsque C_5 vaut 1 on obtient :

$T_0 = 1$ (Même fonctionnement que C_5 vaille 0 ou 1)

$T_1 = C_5'.Q_0 + C_5.0 = C_5'.Q_0$ (Conservation de Q_1 lorsque C_5=1)

$T_2 = C_5'.Q_0.Q_1 + C_5.1 = C_5'.Q_0.Q_1 + C_5$ (Inversion de Q_2 lorsque C_5=1)

Exemple 2 : Compteur / décompteur par 6.

Pour le mode comptage (C=0), le fonctionnement doit rester identique au compteur par 8 tant que la combinaison 5 n'est pas présente sur les sorties du compteur. Lorsque la combinaison 5 est présente, les fonctions d'entrée des bascules doivent être modifiées. Au lieu de passer de 101 à 110, il faut passer de 101 à 000.

Pour le mode décomptage (C=1), le fonctionnement doit rester identique au décompteur par 8 tant que la combinaison 0 n'est pas présente sur les sorties du compteur. Lorsque la combinaison 0 est présente, les fonctions d'entrée des bascules doivent être modifiées. Au lieu de passer de 000 à 111, il faut passer de 000 à 101.

Soit C_0 un flag prévenant qu'on est sur la combinaison 0. $C_0 = Q_2'.Q_1'.Q_0'$

Soit C_5 un flag prévenant qu'on est sur la combinaison 5. $C_5 = Q_2.Q_1'.Q_0$

Les équations des entrées de bascules peuvent s'exprimer de la manière suivante :

$T_0 = C'[1\ ] + C[1] = 1$

$T_1 = C'[C_5'.Q_0] + C[C_0'.Q_0']$

$T_2 = C'[C_5'.Q_0.Q_1 + C_5] + C[C_0'.Q_0'.Q_1' + C_0]$

8.4.3.c. Compteurs « synchrones » avec signal d'inhibition

La prise en compte d'un signal d'inhibition (Inib) du compteur peut se faire de la même manière, c'est à dire en intervenant sur les entrées synchrones (T_i) des bascules.

Exemple : Compteur / décompteur par 6 avec signal d'inhibition (Inib)

Les équations des entrées des bascules T peuvent s'exprimer de la manière suivante :

$T_0 = Inib'$

$T_1 = Inib'\{C'[C_5'.Q_0] + C[C_0'.Q_0']\}$

$T_2 = Inib'\{C'[C_5'.Q_0.Q_1 + C_5] + C[C_0'.Q_0'.Q_1' + C_0]\}$

8.4.3.d. Comparaison bascules T / bascules D

Les bascules T sont particulièrement intéressantes pour la réalisation de compteurs car contrairement aux bascules D, elle ne nécessitent pas d'exprimer les condition de maintien de des sorties de bascules. En effet, seules sont exprimées les conditions de commutation, ce qui simplifie de manière notoire les fonctions d'entrée de bascule.

A titre de comparaison, le compteur/décompteur par 6 avec signal d'inhibition (Inib) précédent conduirait, avec des bascules D, aux équations suivantes :

$D_0 = Inib'\{C[C_5'.Q_0' + C_5.Q_0'] + C'[C_0'.Q_0' + C_0.Q_0']\} + Inib.Q_0$

$= Inib'Q_0' + Inib.Q_0$

$= Inib \oplus Q_0'$

$D_1 = Inib'\{C[C_5'.(Q_0.Q_1'+Q_0'.Q_1)+C_5.0] + C'[C_0'.(Q_0'.Q_1'+Q_0.Q_1)+C_0.0]\} + Inib.Q_0$

$= Inib'\{C[C_5'.(Q_0 \oplus Q_1)] + C'[C_0'.(Q_0 \oplus Q_1')]\} + Inib.Q_1$

$D_2 = Inib'\{C[C_5'.(Q_0.Q_1 \oplus Q_2)] + C'[C_0'.(Q_0+Q_1 \oplus Q_2')]\} + Inib.Q_2$

Bien qu'il soit probable que, dans ce cas particulier, les équations puissent encore se simplifier elles restent de toute manière plus complexes que celles obtenues avec des bascules T.

Remarque : Ces équations sont obtenues par un raisonnement généralisable à des compteurs de modulo différent.

8.5. Règles de conception

Une plaie de trop de réalisations rencontrées est le mélange, dans une même unité fonctionnelle, des commandes asynchrones et synchrones. En effet, la présence de commandes asynchrones dans une conception synchrone induit une sensibilité aux phénomènes transitoires et/ou parasites qui peut être source de nombreux ennuis. D'autre part, la conception d'une application complexe n'est généralement envisageable qu'au travers d'un processus initial de partitionnement du cahier des charges.

8.5.1. Signaux de forçage asynchrones

Les signaux de forçage à action directe (Clear ou Preset), c'est à dire indépendants de l'horloge (asynchrone) peuvent servir à initialiser une application par une commande spécifique mais toute autre utilisation notamment pour réaliser une fonction particulière (commande par de la logique) peut être la source de nombreux ennuis. En effet, toute logique sur ces signaux peut engendrer des impulsions transitoires de durées inconnues, souvent très faibles, mais suffisante pour conduire à un résultat qui, s'il peut être instructif dans un contexte d'enseignement, est catastrophique dans une réalisation. Dans ce cas, une carte qui semble donner toute satisfaction quand on l'observe avec un oscilloscope, peut par exemple cesser de fonctionner dès que l'on retire l'appareil de mesure. Ce phénomène provient généralement de la modification de la durée ou de l'amplitude d'impulsions transitoires sur les signaux asynchrones induite par la charge capacitive supplémentaire apportée par la sonde de mesure. Le dépannage d'un tel système relève alors plus de la divination que d'une méthodologie raisonnée. Ce type de pratique est donc à condamner sans appel.

8.5.2. Les signaux d'horloges

Le blocage, par exemple par une porte, des signaux d'horloge pour maintenir l'état d'un registre, est une autre erreur que l'on rencontre parfois. Cette faute, qui provoque des décalages temporels entre les signaux d'horloge *(clock skew)* appliqués aux différentes parties d'une carte, ou d'un circuit, risque de conduire à des violations de temps de maintien ou de pré-positionnement, d'où des comportements imprévisibles des registres concernés.

Un autre effet pervers des circuits combinatoires de « calcul » des signaux d'horloge, est la génération, difficile à contrôler, d'impulsions parasites sur ces signaux. La recherche de ces impulsions, suffisamment larges pour faire commuter les circuits actifs sur des fronts, mais suffisamment étroites pour ne pas être vues lors d'un examen rapide avec un oscilloscope, est un passe temps dont on se lasse très vite.

Quand il est nécessaire d'appliquer à différentes parties d'un ensemble des signaux d'horloges différents, il est indispensable de traiter à part, et de façon méticuleuse, la réalisation du distributeur d'horloge correspondant. Notons, en passant, que pour ces fonctions il convient de surveiller de très près les modifications apportées par les optimiseurs ; ces derniers ont la fâcheuse tendance d'éliminer les portes inutiles d'un point de vue algébrique, même si elles sont utiles d'un point de vue circuit.

8.5.3. Conception synchrone

S'il y a une règle de conception à retenir des deux paragraphes précédent pour éviter tout désagrément du aux phénomènes temporels et transitoires c'est d'éviter à tout prix d'intervenir sur les signaux asynchrones que sont l'horloge (H) et les signaux de forçage (Clear, Preset). Ainsi, les seuls signaux sur lesquels ont doit intervenir pour réaliser la fonction souhaitée sont les entrées synchrones (D) des bascules. Cette règle de conception est illustrée sur la figure 8.44.

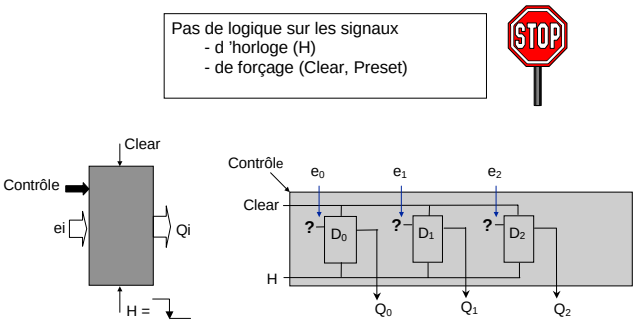


Figure 8.44. *Conception synchrone*

Tout manquement à cette règle peut conduire à de sérieux désagréments ou en tout cas faire l'objet d'une attention toute particulière.

8.5.4. Diviser pour régner

L'erreur de méthode la plus fréquente, et la plus pénalisante, que commettent beaucoup de débutants dans la conception des systèmes électroniques, qu'ils soient analogiques ou numériques, est sans doute de dessiner des schémas, voire de les câbler, avant même d'avoir une vision claire de l'ensemble de la tâche à accomplir.

Le travail de réflexion sur la structure générale d'une application est primordial. Ce que l'on appelle traditionnellement la méthode descendante *(top down desig*n), n'est rien d'autre que l'application de cette règle simple : quand on conçoit un ensemble, *on va du général au particuli*er, on ne s'occupe des détails que quand le cahier des charges a été mûrement réfléchi, et que le plan général de la solution a été établi. Si, au cours de la descente vers les détails, on découvre qu'une difficulté imprévue apparaît, il faut revenir au niveau général pour voir comment la réponse à cette difficulté s'insère dans le plan d'ensemble.

Le premier réflexe à avoir, face à un problème, un tant soit peu complexe à résoudre, est de le couper en deux. La démarche précédente est répétée, pour chaque demi-problème, jusqu'à obtenir des sous-ensembles dont la réalisation tient en quelques circuits élémentaires, en quelques lignes de code source dans un langage ou dans un diagramme de transitions qui ne dépasse pas une dizaine d'états différents.

Chapitre 9

Synthèse des systèmes séquentiels synchrones

Ce chapitre est consacré à la présentation de la méthode d'Huffman-Mealy pour la synthèse des systèmes séquentiels synchrones. Cette méthode permet de passer du cahier des charges décrivant un système au circuit logique correspondant. Elle est particulièrement utilisée pour la réalisation de contrôleurs.

9.1. Définitions générales

9.1.1. Circuits séquentiels synchrones et asynchrones

Les systèmes séquentiels peuvent être différenciés en fonction de leur mode de fonctionnement qui peut être synchrone ou asynchrone. Dans le mode synchrone, les éléments de mémorisation sont des bascules. Les modifications d'état du système ne peuvent donc intervenir qu'à des instants très précis déterminés par des signaux d'horloge. Par contre, dans le mode asynchrone, la fonction de mémorisation est réalisée par de simples boucles de rétroaction. L'évolution des états ne dépend donc que des modifications intervenant sur les entrées E_i de la machine.

Comme illustré sur la Figure 9.1, ces systèmes séquentiels peuvent donc être représentés par deux modèles différents.

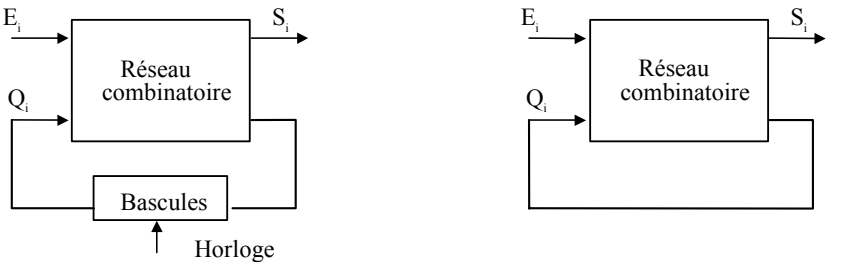


Figure 9.1. *Systèmes séquentiels synchrones et asynchrones*

Ce chapitre est consacré à l'étude des systèmes séquentiel synchrones. Les systèmes séquentiels asynchrones feront l'objet du chapitre suivant.

9.1.2. Modèle des systèmes séquentiels synchrones

Dans un système séquentiel, la présence d'un même vecteurs d'entrée n'entraîne pas nécessairement l'apparition d'une même combinaison sur les sorties. La combinaison de sortie dépend également de données internes caractérisant l'état du système.

L'état d'un système séquentiel est en fait une image des événements antérieurs s'étant produits sur les entrées du système. Cet état, qui évolue lorsque de nouveaux événements se produisent sur les entrées, est mémorisé dans des éléments internes spécifiques appelés éléments mémoires. Un circuit séquentiel contient r éléments de mémoire élémentaire q_1, q_2, ..., q_r, le vecteur $Q=(q_1, q_2, ..., q_r)$ caractérisant l'état interne du circuit séquentiel.

Les vecteurs E_i (Entrées), S_i (Sorties), Q_i (Etat) évoluent à des instants déterminés (phases) que l'on peut discrétiser en notant n l'instant présent et n+1 l'instant suivant. Le fonctionnement d'un système séquentiel peut alors être exprimé par des équations récurrentes et un état initial. Le modèle général (modèle de Mealy) de ces équations est le suivant :

$$Q_i(n+1) = F\ \{E_i(n)\ ,\ Q_i\ (n)\}$$
$$S_i\ (n) = G\ \{E_i(n)\ ,\ Q_i\ (n)\}$$

Un cas particulier concerne les systèmes dont la sortie ne dépend que de l'état interne (Modèle de Moore). Dans ce cas les équations s'expriment de la manière suivante :

$$Q_i\ (n+1) = F\ \{E_i\ (n)\ ,\ Q_i\ (n)\}$$
$$S_i\ (n) = G\ \{Q_i\ (n)\}$$

Ces équations peuvent être représentées de manière symbolique par le modèle présenté sur la Figure 9.2.

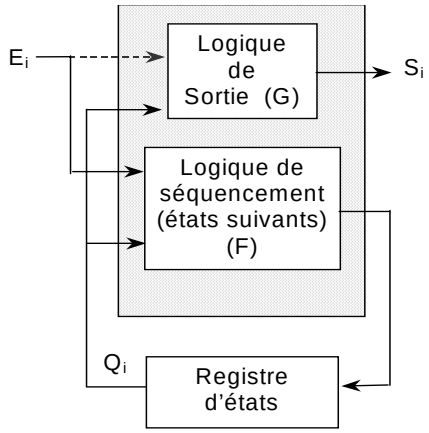


Figure 9.2. *Schéma général d'un système séquentiel synchrone*

Quel que soit le type de machine (Moore ou Mealy) l'état suivant du système $Q_i\ (n+1)$ dépend de l'état actuel $Q_i\ (n)$ et des entrées $E_i\ (n)$. La différence entre machine de Mealy et machine de Moore n'intervient que sur les sorties. Dans une machine de Mealy, les sorties $S_i\ (n)$ dépendent de l'état $Q_i\ (n)$ et des entrées $E_i\ (n)$. Par contre dans une machine de Moore, les sorties $S_i\ (n)$ ne dépendent que de l'état $Q_i\ (n)$.

Le registre d'état de ces machines synchrones est composé de bascules maître-esclaves synchronisée par le signal d'horloge. Ainsi, les modifications d'état du système c'est à dire des sorties de ces bascules ne peuvent donc intervenir qu'à des instants très précis déterminés par le signal d'horloge. Divers types de bascules peuvent être utilisés pour réaliser ce registre d'état (D, T, JK). Dans la pratique, se seront essentiellement des bascule D.

9.2. Synthèse de séquenceurs

Avant de développer la méthode générale de synthèse des systèmes séquentiels synchrone, nous nous proposons d'en aborder le principe de manière simplifiée en considérant le cas des séquenceurs.

Un séquenceur est un système séquentiel délivrant une séquence de combinaisons prédéterminées. Un tel système un en fait cas particuliers de systèmes séquentiels synchrone dans la mesure dans la mesure ou les sorties peuvent être directement portées par les sorties des bascules. Un compteur est par exemple un séquenceur particulier, mais autant la synthèse d'un compteur peut s'envisager de manière intuitive du fait des caractéristiques qui émanent de l'évolution des états (au moins dans le cas d'un compteur en binaire naturel), autant une méthode plus systématique est nécessaire lorsque la séquence à générer ne dispose pas de propriétés particulières.

A titre d'exemple nous considérerons un séquenceur délivrant une séquence de 8 états successifs correspondant à ceux d'un compteurs binaire par 8. Supposons également que ce séquenceur dispose d'une entrée C permettant à tout moment de revenir de manière synchrone dans l'état initial (000). Ce système peut se mettre sous la forme générale d'un système séquentiel synchrone pour lequel les sorties sont directement les sorties S_i sont directement les variables d'état Q_i (sorties des bascules) (Figure 9.3).

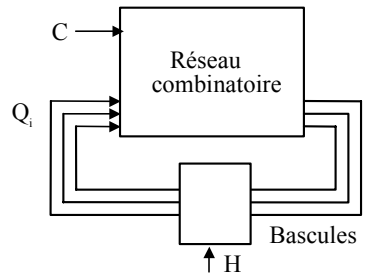


Figure 9.3. *Schéma général d'un séquenceur*

D'autre part, la sortie ne dépendant pas directement de l'entrée C, il s'agit d'une machine de Moore.

La fonction réalisée par tout système séquentiel synchrone peut être représentée par un graphe appelé graphe d'état. Le graphe du compteur est représenté sur la Figure 9.4.

Chaque noeud du graphe représente un état du compteur et chaque arc indique l'état suivant que prendra le compteur après un coup d'horloge. Cet état suivant est fonction de l'entrée C.

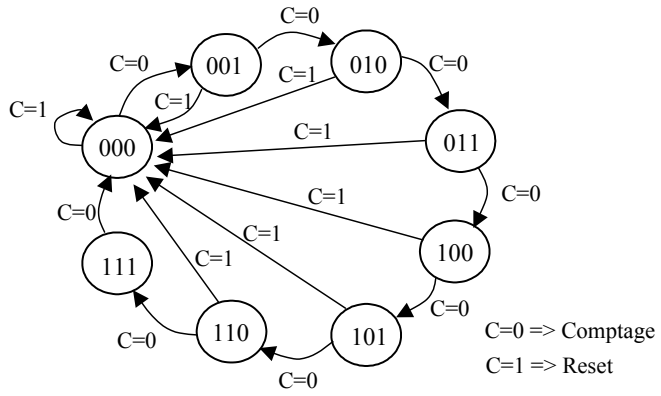


Figure 9.4. *Graphe de d'état*

Ce graphe peut également s'exprimer sous une forme tabulaire. Cette représentation, appelée table d'état (Figure 9.5).

Etats	Etats Suivants	
	C = 0	C = 1
000	001	000
001	010	000
010	011	000
011	100	000
100	101	000
101	110	000
110	111	000
111	000	000

Figure 9.5. *Table d'état*

Supposons maintenant que pour réaliser ce système nous prenions des bascules D. La table de transition d'une telle bascule est la suivante (Figure 9.6) :

D(n)	Q(n+1)
0	0
1	1

Figure 9.6. *Tables de transition des bascules D*

Pour chaque bascule i nous connaissons l'état suivant $Q_i(n+1)$ (après le coup d'horloge) en fonction de l'état présent $Q_i(n)$ et de l'entrée C. Il reste donc à déterminer les entrées D_i de chaque bascule à partir des tables de transition correspondantes (Figures 9.7).

Etats $Q_2Q_1Q_0(n)$	Etats Suivants $Q_2Q_1Q_0(n+1)$		C = 0			C = 1		
	C = 0	C = 1	$D_2(n)$	$D_1(n)$	$D_0(n)$	$D_2(n)$ $D_0(n)$		$D_1(n)$
000	001	000	0	0	1	0	0	0
001	010	000	0	1	0	0	0	0
010	011	000	0	1	1	0	0	0
011	100	000	1	0	0	0	0	0
100	101	000	1	0	1	0	0	0
101	110	000	1	1	0	0	0	0
110	111	000	1	1	1	0	0	0
111	000	000	0	0	0	0	0	0

Figure 9.7. *Entrées des bascules*

Remarque : Avec les bascules D, on a D(n) = Q(n+1). Avec de telle bascules, on peut ainsi s'affranchir de l'écriture des colonne donnant les $D_i(n)$ sachant qu'elles sont identiques à celle donnant les états suivants $Q_i(n+1)$.

Les expressions des entrées de bascules $D_i(n)$ peuvent maintenant être déterminées (Figure 9.8).

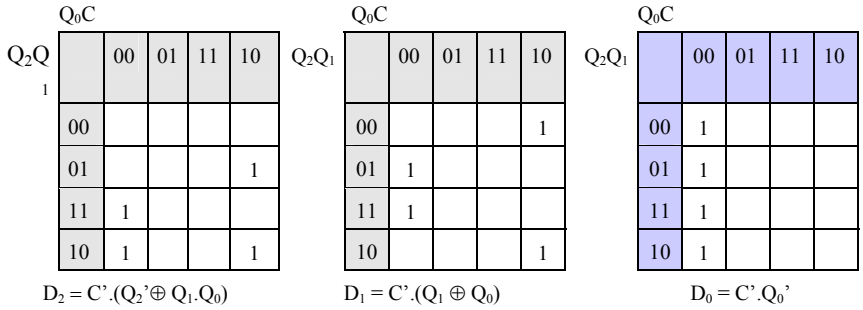


Figure 9.8. *Synthèse des entrées de bascules*

Ayant les équations des entrées des bascules le circuit peut être réalisé (Figure 9.9).

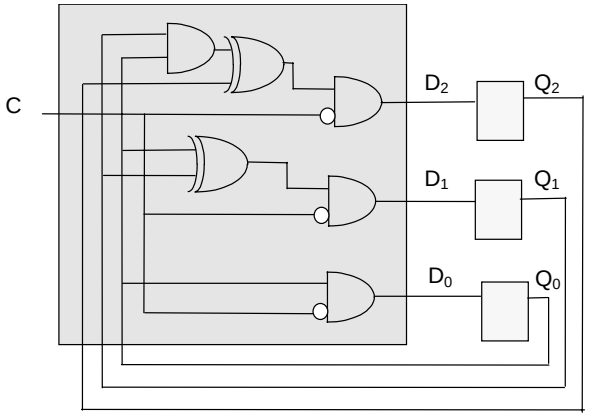


Figure 9.9. *Schéma logique*

9.3. Méthode de synthèse d'Huffman-Mealy

Dans cette partie, nous généraliserons les concepts évoqués précédemment afin de permettre le passage d'un cahier des charges quelconque au circuit correspondant. De plus nous établirons des règles de minimisation permettant d'optimiser le nombre de bascules utilisées pour la réalisation du circuit.

La méthode proposée, connue sous le nom de méthode d'Huffman-Mealy se décompose en plusieurs étapes. Ces étapes sont les suivantes :

1 : Modélisation du cahier des charges

- Graphe d'état
- Table d'état

2 : Minimisation du nombre d'états

- Règles de minimisation
- Détermination du nombre de bascules minimum

3 : Codage

- Codage des états
- Codage des entrées de bascules

4 : Synthèse

- Synthèse des entrées de bascules et des sorties de la machine
- Implantation technologique (mapping)

9.3.1. Modélisation du cahier des charges

Le cahier des charges d'un système est généralement donné en langage courant.

Exemple : Le système considéré a une entrée (E) et une sortie (S). Il reçoit sur son entrée des bits arrivant en série. La sortie (S) doit passer à 1 chaque fois qu'une séquence 010 apparaît sur l'entrée (E) puis repasser à 0 sur le bit suivant quel que soit sa valeur.

Pour faire la synthèse d'un tel cahier des charges, la première étape est de le modéliser.

9.3.1.a. Graphe d'état

Le modèle généralement utilisé pour représenter le cahier des charges d'un système est un graphe appelé graphe d'état ou graphe de fluence. Les nœuds de ce graphe représentent les états, un nom symbolique étant affecté à chacun des états. Les arcs du graphe sont orientés. Ils représentent les possibilités de passage entre états. Ces changements d'états se font sur un front d'horloge en fonction des valeurs d'entrée. La structure générale du graphe représentant l'évolution des états d'une machine ayant une entrée E est représentée sur la Figure 9.10.

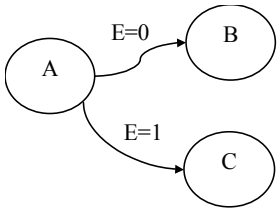


Figure 9.10. *Structure générale du graphe d'état d'un système séquentiel synchrone*

Les caractéristiques de ces graphes sont les suivantes :

- Chaque état (Q_i) est représenté par un cercle,
- A chaque état est associé un nom symbolique
- Le passage d'un état à un autre se fait au coup d'horloge,
- L'état atteint dépend de l'état de départ et de la valeur des d'entrées (E_i),
- De chaque état part au plus 2^n arcs, n étant le nombre d'entrées (E_i),
- Ce graphe est connexe.

Si l'on considère maintenant la sortie, il faut différencier les deux types de machines que sont machines de Moore et machines de Mealy. En effet, dans une machine de Moore, les sorties ne dépendent que des états et par conséquent peuvent être consignées à l'intérieur des cercles. Dans une machine de Mealy, les sorties dépendent des états mais également des entrées. Ces sorties doivent donc être consignées sur les arcs du graphe. Ainsi, selon le type de machine, un modèle différent de graphe d'état doit être considéré (Figure 9.11).

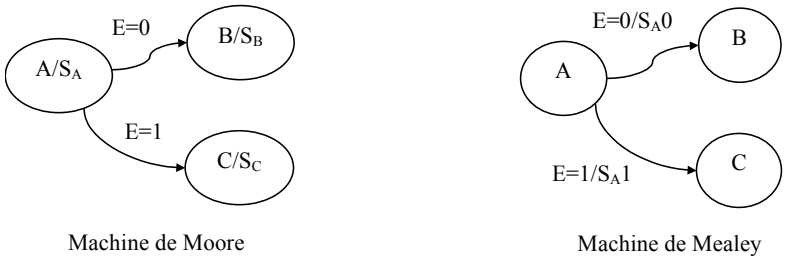


Figure 9.11. *Structure des graphes d'état de Moore et de Mealy*

Exemple : Selon que le système sera réalisé sous forme de machine de Moore ou sous forme de Machine de Mealy, le graphe d'état représentant le cahier des charges précédent est représenté sur la Figure 9.12.

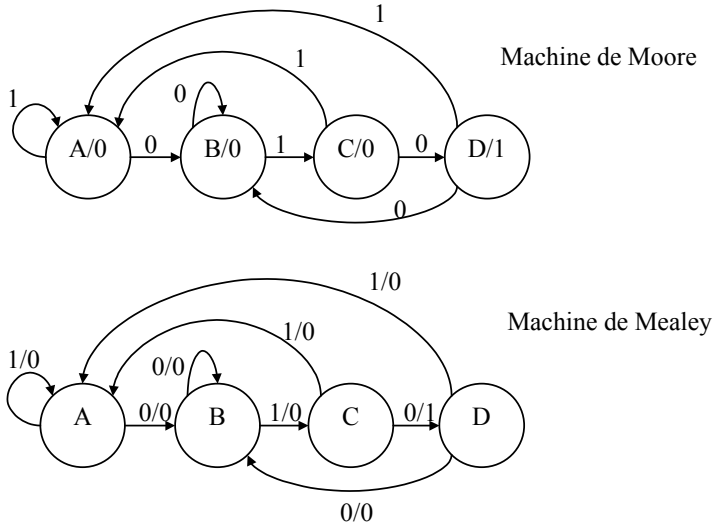


Figure 9.12. *Graphes d'état de la machine détectant les séquences 010*

Remarque : La structure des deux graphes paraît identique. En fait, il est toujours possible de passer d'un graphe de Moore à un graphe de Mealy. Pour cela il, suffit de reporter les sorties associées à chaque état (Moore) sur les arcs arrivant à chacun de ces états. Nous verrons par la suite que l'inverse, c'est à dire passer d'un graphe de Mealy à un graphe de Moore n'est pas toujours possible.

Ce modèle permet de représenter le cahier des charges sous une forme exploitable, mais permet également de figer le fonctionnement du circuit dans tous les cas particuliers non prévus dans le cahier des charges initial. Dans le cas de cet exemple, le graphe correspond à une machine qui détecte les séquences 010 en mots disjoints (Arc 1 de D à A) et non en mots imbriqués.

A ce niveau de la synthèse, ce qui est important c'est de s'assurer que le graphe correspond bien au cahier des charges. Le nombre d'état utilisés pour représenter ce cahier des charges importe peu. Il sera minimisé par la suite.

9.3.1.b. Table d'état

Le cahier des charges d'un système peut également être modélisé sous une forme tabulaire qui est plus facile à manipuler qu'une représentation sous forme de graphe. Cette représentation tabulaire, appelée table d'état, est directement déductible du graphe d'état. Elle représente les différentes possibilités d'états suivants de chacun des états du système et ceci en fonction des entrées. Les sorties associées à chaque état sont également représentées

sur cette table. Elles dépendent ou non des entrées selon qu'il s'agit d'une machine de Mealy ou d'une machine de Moore.

Exemple : Les tables d'état correspondant au graphe de la figure 1.12 sont présentées sur la Figure 9.13.

Etats	Etats Suivants		Sortie
	E=0	E=1	
A	B	A	0
B	B	C	0
C	D	A	0
D	B	A	1

Machine de Moore

Etats	Etats Suivants		Sortie	
	E=0	E=1	E=0	E=1
A	B	A	0	0
B	B	C	0	0
C	D	A	1	0
D	B	A	0	0

Machine de Mealy

Figure 9.13. *Tables d'état de la machine détectant les séquences 010*

9.3.2. Minimisation du nombre d'états

Le nombre d'états de la machine influe directement sur le nombre de bascules nécessaires pour réaliser ce système. Or, le nombre d'états utilisés pour représenter le cahier des charges, que ce soit sur le graphe d'état ou sur la table d'état, n'est pas nécessairement minimum.

9.3.2.a. Règles de minimisation

Deux règles permettent de déterminer les états équivalents et par conséquent de minimiser le nombre d'états nécessaires à la réalisation du circuit.

- Règle R1 : Deux états sont équivalents si pour chaque combinaison d'entrée, ils ont mêmes sorties et mêmes états suivants.
- Règle R2 : Les états sont regroupés en différentes classes selon les valeurs de sorties associées. Ainsi, deux états ayant mêmes sorties (pour chaque combinaison d'entrée) sont dans la même classe. Les états appartenant à une même classe sont équivalents s'il ne peuvent être séparés. Or les états appartenant à une même classe doivent être séparés si les états suivants associés à chacun d'eux sont dans des classes différentes.

Lorsque plusieurs états sont équivalents, il suffit de garder qu'un seul représentant par classe d'équivalence et de renommer les états suivants en conséquence.

Exemple 1: Les règles de minimisation appliquées à la table d'état de la machine de Mealy précédente donnent :

R1 : A et D on mêmes sorties et mêmes états suivants, ils sont donc équivalents. L'état D peut par exemple être éliminé. En renommant les états suivants en conséquence, c'est à dire en remplaçant D par A, la table d'état devient :

Etats	Etats Suivants		Sortie	
	E=0	E=1	E=0	E=1
A	B	A	0	0
B	B	C	0	0
C	A	A	1	0

Figure 9.14. *Table d'état réduite*

Au sens de la règle R1 il n'y a pas d'autres états équivalents.

R2 : les états peuvent être regroupés en deux classes (classe 1 et classe 2).

(1) (2)	Classes
(A , B) (C)	Etats
BA BC	Etats suivants
11 12	Classes des états suivants

Les états A et B doivent être séparés. Il y a maintenant qu'un seul état par classe. Il n'y a donc plus d'états équivalents. Cette machine peut être réalisée avec 3 états.

Remarque : s'il est toujours possible de passer du graphe représentant une machine de Moore à un graphe représentant la même machine en Mealy, l'exemple précédent montre que l'inverse n'est pas toujours possible. En effet, une machine de Mealy peut comporter moins d'état qu'une machine de Moore.

Le nombre d'états nécessaire à la réalisation d'une machine de Mealy pouvant être inférieur à celui nécessaire à la réalisation d'une machine de Moore, le nombre de bascules peut l'être également. D'où l'avantage qu'il peut y avoir à réaliser une machine de Mealy plutôt qu'une machine de Moore. Ceci dit, les machines de Mealy peuvent avoir des inconvénients liés au fait que les sorties dépendent directement des entrées. En effet, lors du passage d'un état à un autre, les entrées ne doivent pas varier. Il se produit donc un instant entre le changement d'état et le changement d'entrée ou le système se trouve dans le nouvel état mais en présence de l'entrée ayant conduit à cet état, c'est à dire de l'entrée précédente. Puisqu'en machine Mealy, les sorties dépendent directement de l'état et des entrées, elles peuvent donc être soumise à des commutations parasites.

9.3.2.b. Détermination du nombre de bascules minimum

Le nombre minimum d'états "q" étant déterminé, on peut en déduire le nombre minimum "n" de variables d'état et par conséquent de bascules nécessaires au codage de ces états à partir de la double inéquation suivante :

$$2^{n-1} < q < 2^n$$

Exemple : Pour la machine de Mealy précédente, le nombre minimum d'état étant de 3, le nombre de variables d'état nécessaire au codage de ces états est 2. Deux bascules sont donc nécessaires pour réaliser ces systèmes.

9.3.3. Codage

9.3.3.a. Codage des états

Chaque état peut être codé par une combinaison de ces n variables d'état. Chaque état doit avoir un code différent des autres mais le codage des états peut être quelconque. Notons toutefois que le codage influence la structure de la future machine et peut donc influencer sa complexité. Le problème de l'optimisation de la

machine résultante passe donc par un choix judicieux du codage des états. Ce problème ne sera pas développé ici.

Une fois les états codés, la table d'état peut être exprimée en fonction de ces codes.

Exemple : Nous appellerons les variables d'état (sorties des 2 bascules) de la machine détectant la séquence 010, Q_1 et Q_2. En considérant un codage donné (celui indiqué dans la colonne "Etats"), la table d'état codée de la machine de Mealy correspondant à la table d'état de la Figure 9.14 est représentée sur la Figure 9.15.

Etats $Q_1Q_2(n)$	Etats Suivants $Q_1Q_2(n+1)$		Sortie	
	E=0	E=1	E=0	E=1
00 (A)	01	00	0	0
01 (B)	01	11	0	0
11 (C)	00	00	1	0

Figure 9.15. *Tables d'état codées de la machine de Mealy*

9.3.3.b. Codage des entrées de bascules

Le code des états étant déterminé, les entrées de bascules peuvent l'être. Pour chaque bascule i nous connaissons l'état suivant $Q_i(n+1)$ (après le coup d'horloge) en fonction de l'état présent $Q_i(n)$ et des entrées. Pour réaliser ce système il reste à déterminer les entrées de chaque bascule.

Avec des bascules D, les entrées D_i peuvent être déterminée directement à partir de la relation :

$$D_i(n) = Q_i(n+1)$$

Exemple : Reprenons la machine de Mealy précédente. Les entrées D_1 et D_2 des bascules Q_1 et Q_2 sont représentées sur la Figure 9.16.

Etats $Q_1Q_2(n)$	Etats Suivants $Q_1Q_2(n+1)$		Sortie		Entrées Bascules $D_1D_2(n)$	
	E=0	E=1	E=0	E=1	E=0	E=1
00 (A)	01	00	0	0	01	00
01 (B)	01	11	0	0	01	11
11 (C)	00	00	1	0	00	00

Figure 9.16. *Détermination des entrées de bascules*

9.3.4. Synthèse

9.3.4.a. Synthèse des entrées de bascules et des sorties de la machine

Sur la table précédente on dispose des sortie et entrées de bascules exprimées en fonction des entrées et des variables d'état (sorties des bascules). Il suffit donc maintenant d'exprimer les fonction logiques relatives aux sorties et entrées de bascules.

Exemple : Reprenons la machine de Mealy précédente et déterminons les équations de la sortie S et des entrée de bascules D_1 et D_2 (Figure 9.17)

Q_1Q_2 \ E	0	1
00	0	0
01	0	1
11	0	0
10	ϕ	ϕ

$D_1 = E.Q_1 + E.Q_2$

Q_1Q_2 \ E	0	1
00	1	0
01	1	1
11	0	0
10	ϕ	ϕ

$D_2 = E'.Q_2' + Q_1'Q_2$

Q_1Q_2 \ E	0	1
00	0	0
01	0	0
11	1	0
10	ϕ	ϕ

$S = E'.Q_1$

Figure 9.17. *Synthèse de D1 et D2*

9.3.4.b. Implantation technologique

L'implantation technologique de fonctions logiques est un problème en soit qui sera traité dans un chapitre spécifique. Nous pouvons nous contenter ici, d'une implantation à portes obtenue directement à partir de ces équations déterminée précédemment (Figure 9.18).

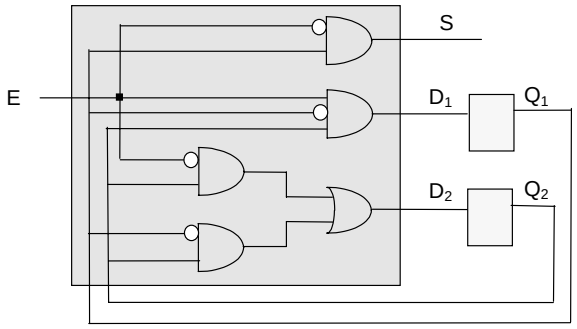


Figure 9.18. *Schéma du détecteur de séquence 010*

9.4. Traitement de « mots binaires »

Lorsque les bits d'entrée correspondent à des mots binaires, la synthèse de la machine analysant ces mots peut être réalisée par le processus décrit précédemment mais également par un processus spécifique ou les temps d'arrivée des bits sur la machine sont exprimés de manière explicite.

9.4.1. Graphes en arbre

Lorsque la machine à réaliser doit traiter des mots binaire arrivant en série sur ces entrées, le graphe d'état représentant la machine peut être réalisé sous forme d'arbre, sachant qu'à la fin de l'analyse d'un (ou plusieurs) mot(s) il faut revenir à l'état initial pour traiter le(s) suivant(s).

Exemple : On désire réaliser une machine recevant sur sont entrée E des mots de 4 bits en série (poids faible en tête). Sa sortie S doit passer à 1 chaque fois qu'un mot représente un nombre supérieur à 9 et revenir à 0 sur le premier bit du mot suivant. Ce cahier des charges peut se modéliser par le graphe en arbre représenté sur la figure 9.19.

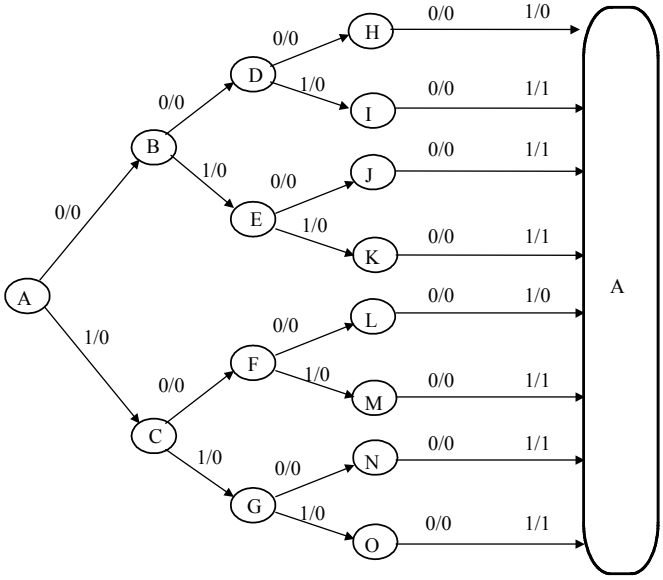


Figure 9.19. *Graphe d'état de la machine détectant les nombre supérieurs à 9*

Ce graphe n'est évidemment pas optimum d'un point de vue nombre d'état, mais à ce stade de la synthèse, ce n'est absolument pas le but. L'essentiel est que le cahier des charges soit fidèlement représenté.

La table d'état correspondant à ce graphe est la suivante (figure 9.20) :

Etats	Etats Suivants		Sortie	
	0	1	0	1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	I	0	0
E	J	K	0	0
F	L	M	0	0
G	N	O	0	0
H	A	A	0	0
I	A	A	0	1
J	A	A	0	1
K	A	A	0	1
L	A	A	0	0
M	A	A	0	1
N	A	A	0	1
O	A	A	0	1

Figure 9.20. *Table d'état de la machine détectant les nombre supérieurs à 9*

- En appliquant la première règles de minimisation, on trouve les équivalences suivantes (I,J,K,M,N,O) (H,L). On conserve H et I et on élimine J, K, M, N, O, L (Figure 9.21.a).
- En ré-appliquant la première règle de minimisation, on trouve les équivalences suivantes (D,F) (E,G). On conserve D et E et on élimine F et G (Figure 9.21.b).
- En ré-appliquant la première règles de minimisation, on trouve les équivalences suivantes (B,C). On conservant B et on élimine C (Figure 9.21.c).

Etats	Etats Suivants		Sortie	
	0	1	0	1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	I	0	0
E	I	I	0	0
F	H	I	0	0
G	I	I	0	0
H	A	A	0	0
I	A	A	0	1

(a)

Etats	Etats Suivants		Sortie	
	0	1	0	1
A	B	C	0	0
B	D	E	0	0
C	D	E	0	0
D	H	I	0	0
E	I	I	0	0
H	A	A	0	0
I	A	A	0	1

(b)

Etats	Etats Suivants		Sortie	
	0	1	0	1
A	B	B	0	0
B	D	E	0	0
D	H	I	0	0
E	I	I	0	0
H	A	A	0	0
I	A	A	0	1

(c)

Figure 9.21. *Réduction de la table d'état*

A ce stade, il n'y a plus de minimisation possible avec la règle R1. On peut également vérifier que la règle R2 n'amène pas de réduction supplémentaires. Le nombre d'états minimum permettant de réaliser cette machine étant de 6, le nombre de bascules nécessaires est donc 3. Nous ne développerons pas ici la suite de la synthèse de cette machine sous cette forme.

Remarque : Dans le cas ou l'on a à analyser des mots binaires, le cahier des charges peut toujours être modélisé par un graphe en arbre. Un tel graphe n'est évidemment pas optimum d'un point de vue nombre d'état, mais une telle approche facilite grandement l'interprétation du cahier des charge sachant qu'à ce stade de la synthèse, l'essentiel est que le cahier des charges soit fidèlement représenté. Réaliser directement un graphe réduit (voir optimum) est toujours possible mais peut conduire à une interprétation plus « risquée » du cahier des charges. Le graphe optimum peut d'ailleurs toujours être obtenu à l'issu du processus de minimisation du nombre d'états. A titre d'exemple et de comparaison avec le graphe en arbre, le graphe optimum décrivant le comportement de la machine précédente est présenté sur la figure 9.22. Ce graphe est issu de la table présentée sur la figure 9.21.c.

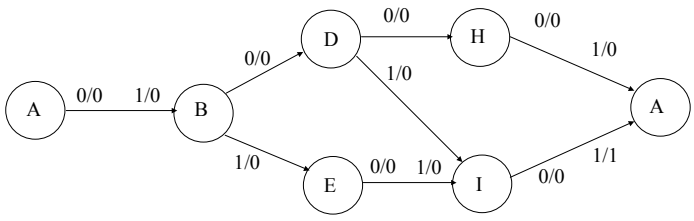


Figure 9.22. *Graphe d'état optimum de la machine détectant les nombre supérieurs à 9*

9.4.2. Machines à « temps explicite »

Supposons maintenant qu'il existe dans le circuit un dispositif (compteur) synchronisé sur la même horloge fournisse une information sur le bit présent en entrée de la machine à un instant donné (Figure 9.23).

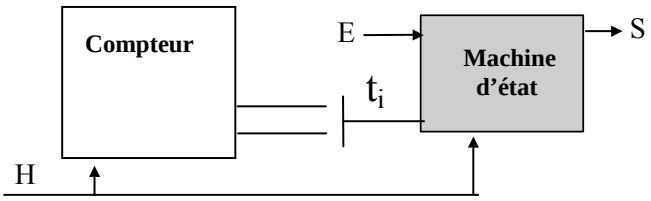


Figure 9.23. *Machine à temps explicite*

En reprenant l'exemple précédent, supposons que les quatre sorties du compteur (t1,t2,t3,t4) soient telles que :

t1 t2 t3 t4 = 1000 lorsque le premier bit est présent sur les entrées de la machine,
t1 t2 t3 t4 = 0100 lorsque le deuxième bit est présent sur les entrées de la machine,
t1 t2 t3 t4 = 0010 lorsque le troisième bit est présent sur les entrées de la machine,
t1 t2 t3 t4 = 0001 lorsque le quatrième bit est présent sur les entrées de la machine.

Ces informations complémentaires peuvent être utilisées comme entrées de la machine d'état, machine comporte maintenant, dans le cas de notre exemple, non plus une seule entrée mais 5 qui sont : E, t1, t2, t3, t4. Notons par ailleurs qu'uniquement 4 combinaisons de t1, t2, t3, t4 sont possibles.

Le graphe de la machine peut rester identique à celui réalisé précédemment. Il suffit simplement d'indiquer les valeurs de t1 t2 t3 t4.

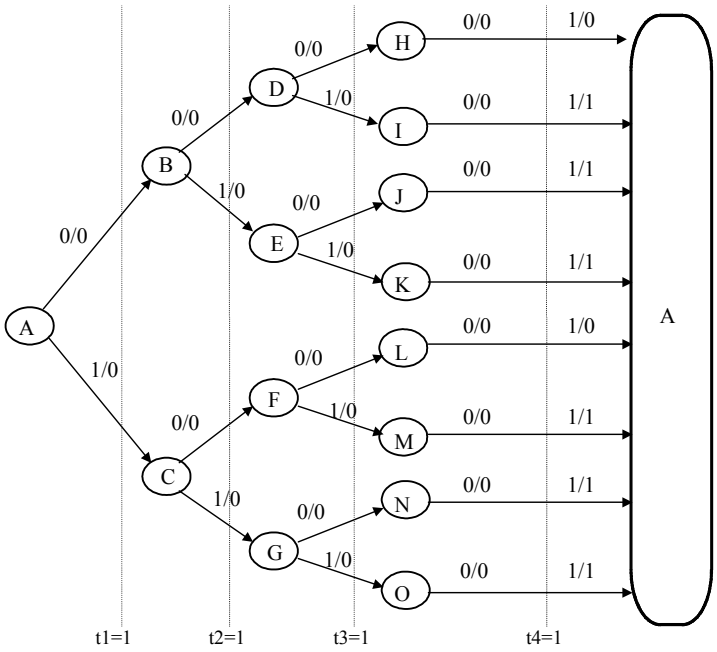


Figure 9.24. *Graphe d'état de la machine à temps explicite.*

La table d'état peut maintenant s'exprimer de la manière suivante (Figure 9.25).

Etats	Etats Suivants								Sortie			
	t1=1		t2=1 t4=1		t3=1				t1	t2	t3	t4
	0	1	0	1	0	1	0	1	01	01	01	01
A	B	C	-	-	-	-	-	-	00	--	--	--
B	-	-	D	E	-	-	-	-	--	00	--	--
C	-	-	F	G	-	-	-	-	--	00	--	--
D	-	-	-	-	H	I	-	-	--	--	00	--
E	-	-	-	-	J	K	-	-	--	--	00	--
F	-	-	-	-	L	M	-	-	--	--	00	--
G	-	-	-	-	N	O	-	-	--	--	00	--
H	-	-	-	-	-	-	A	A	--	--	--	00
I	-	-	-	-	-	-	A	A	--	--	--	01
J	-	-	-	-	-	-	A	A	--	--	--	01
K	-	-	-	-	-	-	A	A	--	--	--	01
L	-	-	-	-	-	-	A	A	--	--	--	00
M	-	-	-	-	-	-	A	A	--	--	--	01
N	-	-	-	-	-	-	A	A	--	--	--	01
O	-	-	-	-	-	-	A	A	--	--	--	01

Figure 9.25. *Table d'état de la machine à temps explicite.*

En appliquant la première règles de minimisation sans affecter les états indéterminés nous revenons à 6 états comme précédemment (Figure 9.26).

Etats	Etats Suivants								Sortie			
	t1=1		t2=1		t3=1		t4=1		t1	t2	t3	t4
	0	1	0	1	0	1	0	1	01	01	01	01
A	B	B	-	-	-	-	-	-	00	--	--	--
B	-	-	D	E	-	-	-	-	--	00	--	--
D	-	-	-	-	H	I	-	-	--	--	00	--
E	-	-	-	-	I	I	-	-	--	--	00	--
H	-	-	-	-	-	-	A	A	--	--	--	00
I	-	-	-	-	-	-	A	A	--	--	--	01

Figure 9.26. *Table d'état réduite de la machine à temps explicite.*

En ré-appliquant toujours la première règle de minimisation, on peut maintenant trouver d'autres équivalences en affectant certains états indéterminés. Ainsi on peut arriver à deux classes d'équivalence telles que par exemple (A,D,H) et (B,E,I). En conservant A et B et en éliminant D, H, E et I, la table devient celle présentée sur la Figure 9.27.

Etats	Etats Suivants								Sortie			
	t1=1		t2=1		t3=1		t4=1		t1	t2	t3	t4
	0	1	0	1	0	1	0	1	01	01	01	01
A	B	B	-	-	A	B	A	A	00	--	00	00
B	-	-	A	B	B	B	A	A	--	00	00	01

Figure 9.27. *Table d'état minimale de la machine à temps explicite.*

Une seule bascule est donc nécessaire pour réaliser cette machine. En supposant une bascule D et en codant l'état a 0 et l'état b 1 sur la sortie Q de cette bascule on obtient la table 9.28.

Etats	Etats Suivants								Sortie			
	t1=1		t2=1		t3=1		t4=1		t1	t2	t3	t4
	0	1	0	1	0	1	0	1	01	01	01	01
0	1	1	-	-	0	1	0	0	00	--	00	00
1	-	-	0	1	1	1	0	0	--	00	00	01

Figure 9.28. *Table d'état codée de la machine à temps explicite.*

Ce qui donne : $S = t4.e.Q$

$D = t1 + t2.e + t3.(e + Q)$

Bilan : Si l'on fait le bilan global du nombre de bascules, on s'aperçoit que globalement le nombre de bascules nécessaires à la réalisation de ce dispositif est identique à celui déterminé précédemment (3 bascules). En effet, il faut ajouter à la bascule de la machine ainsi réalisée, les deux bascules nécessaires à la réalisation du compteur. Concevoir une telle machine peut toutefois être intéressant dans les deux cas suivants:

1. Le dispositif (compteur) donnant les informations sur le bit présent en entrée de la machine existe déjà dans le circuit,
2. Plusieurs machines manipulant les mêmes données doivent être conçues sur le même circuit (auquel cas, le compteur n'est réalisé qu'une seule fois). Dans ce dernier cas, si n est le nombre de machines et m le nombre de bascules du compteur, le gain par rapport à une approche classique est de (n-1)*m.

9.5. Implantations partitionnées

Le premier réflexe à avoir, face à un problème, un tant soit peu complexe à résoudre, est de le scinder le problème en deux. La démarche précédente peut être répétée, pour chaque demi-problème, jusqu'à obtenir des sous-ensembles dont la réalisation tient en quelques circuits élémentaires, en quelques lignes de code source dans un langage ou dans un diagramme de transitions qui ne dépasse pas une dizaine d'états différents. L'architecture résultante sera ainsi constituée de plusieurs machines communiquant entre elles comme illustré sur la figure 9.29.

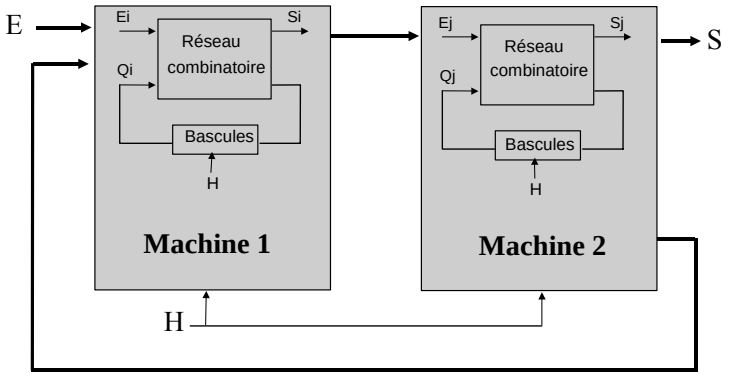


Figure 9.29. *Machine d'état partitionnée*

Exemple : On désire concevoir un séquenceur produisant la séquence suivante : 0, 1, 2, 3, …, n-1, n, n-1, n-2, …, 3, 2, 1, 0, 1, 2, 3, …n. Un tel système peut être synthétisé à partir d'un graphe comportant 2*n états. Mais la synthèse peut également être envisagée en partitionnant la machine en 2 machines d'état dont un compteur / décompteur commandé par un signal Up/Dn (Figure 9.30).

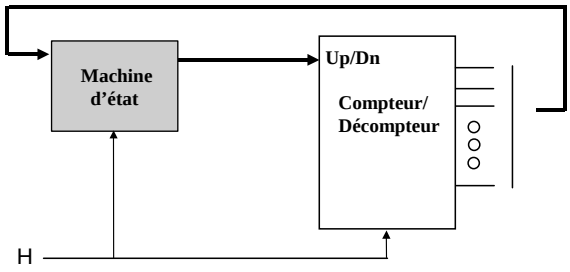


Figure 9.30. *Machine d'état partitionnée*

La synthèse se limite alors à la synthèse de la machine d'état commandant l'entrée Up/Dn du compteur / décompteur. Le graphe représentant le fonctionnement de cette machine ainsi que la structure de la machine obtenue après synthèse de ce graphe est donné sur la figure 9.31.

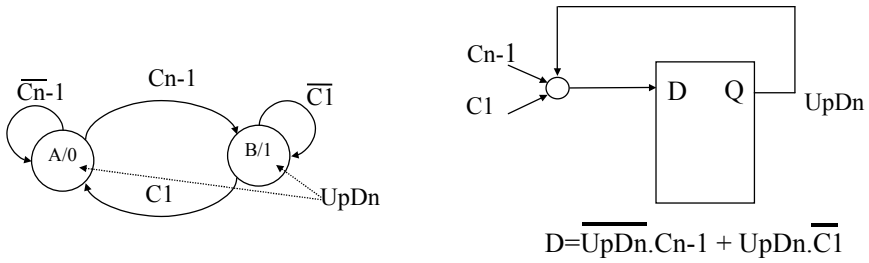


Figure 9.31. *Machine de contrôle du signal Up/Dn*

Remarque : Outre l'aspect complexité, de telle implantations peuvent avoir un intérêt certain lorsqu'on cherche à minimiser les chemins critiques pour augmenter la vitesse de fonctionnement globale du système. Cet aspect ne sera pas développé ici.

Chapitre 10

Synthèse des systèmes séquentiels asynchrones

Dans beaucoup de situation pratiques, les entrées du système à réaliser ne sont pas gérées par une horloge et sont donc sujettes à modification à des instants quelconques. Dans ce cas il n'est pas possible de concevoir un système synchrone. Dans ce chapitre nous nous intéresserons donc à la synthèse des systèmes séquentiels asynchrones.

10.1. Structure des systèmes séquentiels asynchrones

Dans le mode synchrone, les éléments de mémorisation sont des bascules. Les modifications d'état du système ne peuvent donc intervenir qu'à des instants très précis déterminés par des signaux d'horloge. Par contre, dans le mode asynchrone, la fonction de mémorisation est réalisée par de simples boucles de rétroaction. L'évolution des états ne dépend donc que des modifications intervenant sur les entrées primaires (E_i) de la machine. La représentation la plus générale d'un système asynchrone est donné sur la Figure 10.1.

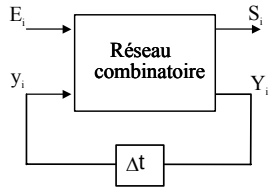


Figure 10.1. *Modèle général d'un systèmes séquentiel asynchrone*

Remarque : L'élément retard Δt est un élément virtuel permettant de modéliser les retards des signaux dans la logique combinatoire.

Le circuit est dit dans un état stable si et seulement si $y_i=Y_i$ pour tout i= 1, 2, ..., k. En réponse à un changement d'entrée, la logique combinatoire produit un nouvel ensemble de valeurs sur Y_i. Si ces valeurs sont différentes de celles de y_i ce circuit entre dans un état instable. On obtiendra un nouvel état stable lorsqu'à nouveau on aura $y_i=Y_i$. Si aucun un état stable n'est atteint le système oscille.

Une variation d'entrée est obligatoire pour que le système évolue d'un état stable vers un autre état stable. D'autre part, vu les délais inhérents aux structures, il est impossible de garantir le changement de deux variables simultanément. La synthèse d'un système asynchrone doit interdire de telles situations. Cette restriction signifie en effet qu'une seule entrée peut changer à un instant donné, et que le temps écoulé entre deux variations est plus grand que le délai interne de la structure. Lorsqu'un changement apparaît sur une entrée, nous ferons donc l'hypothèse qu'aucune variation sur toutes entrées ne peut intervenir avant que le circuit soit dans un état stable.

Un tel mode de fonctionnement est appelé mode fondamental. Notons qu'un mode de fonctionnement différent appelé mode pulsé est aussi possible mais ne sera pas étudié ici.

10.2. Méthode de synthèse

La méthode de synthèse de systèmes séquentiels asynchrones proposée a pour but de minimiser le nombre de variables secondaires (Y_i). Ceci a pour effet direct de minimiser le nombre de fonctions du réseau combinatoire et par conséquent le nombre de portes du circuit. Cette méthode se décompose en plusieurs étapes. Ces étapes sont les suivantes:

1 : Modélisation du cahier des charges

- Graphe d'état
- Table d'état (ou des phases) primitive

2: Minimisation du nombre d'états

- Elimination des états équivalents
- Fusionnement d'états

3: Codage des états

- Graphe d'adjacence
- Assignation des états

4: Synthèse

- Synthèse des variables secondaires
- Synthèse des sorties

10.2.1. Modélisation du cahier des charges

Tout comme pour les systèmes séquentiels synchrones, le cahier des charges d'un système est généralement donné en langage courant. Pour faire la synthèse d'un tel cahier des charges, la première étape est de l'exprimer dans un modèle mathématique. Le modèle généralement utilisé pour représenter le cahier des charges d'un système asynchrone est également un graphe appelé graphe d'état ou graphe de fluence.

10.2.1.a. Graphe d'état

Tout comme pour les systèmes séquentiels synchrones, les nœuds du graphe d'état représentent les états du système et les arcs orientés, les possibilités de passage d'un état à un autre. Par contre, le fait qu'en asynchrone, il n'y ait plus d'horloge introduit une différence notable dans l'interprétation du graphe. En effet, le passage d'un état à un autre ne se fait non plus sur un coup d'horloge mais directement lorsqu'une entrée varie. Cette différence de fonctionnement peut se traduire par une différence de structure du graphe d'état. En effet, chaque état étant stable tant qu'il n'y a pas de variation d'entrée, les entrées peuvent être portées sur les cercles représentant les états. Quant aux sorties, elles peuvent également être portées sur les cercles représentant les états puisqu'elles sont fonction des variables secondaires caractérisant ces états et des entrées qui sont maintenant portées sur les états (Figure 10.2.).

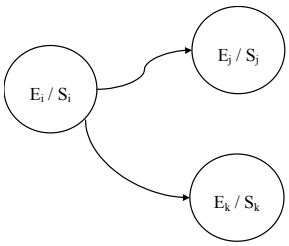


Figure 10.2. *Structure générale du graphe d'état d'un système séquentiel asynchrone*

Remarque : Dans le cas des systèmes séquentiels asynchrones, la notion de machine de Moore ou Mealey n'existe plus. En effet, les états dépendant directement des entrées (de manière combinatoire), les sorties dépendent donc toujours des entrées, que ce soit directement ou indirectement au travers de variables secondaires.

Exemple : Le système considéré a deux entrées e_1 et e_2 et une sortie S. La sortie S doit passer à 1 chaque fois que la séquence e_1e_2 = 10, 11, 01, 00 intervient sur les entrées. Quand S est à 1, S doit repasser à 0 dès la première variation d'une des deux entrées.

Le graphe d'état représentant ce cahier des charges est représenté sur la figure 10.3.

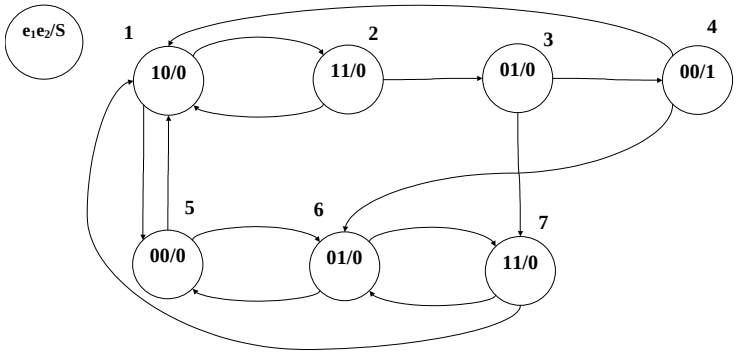


Figure 10.3. *Graphe de machine détectant les séquences 10/11/01/00*

10.2.1.b. Table d'état primitive

Le cahier des charges d'un système peut également être modélisé sous une forme tabulaire qui est plus facile à manipuler qu'une représentation sous forme de graphe. Cette représentation tabulaire, appelée table d'état ou table des phases primitive, est directement déductible du graphe d'état. Elle représente les différentes possibilités d'états suivants de chacun des états du système et ceci en fonction des entrées. De manière à repérer plus facilement les états stables et instables, les états stables seront surlignés (grisés). Outre le repérage des états stables, la seule différence avec les systèmes séquentiels synchrones est qu'il existe des états non-atteignables du fait de l'impossibilité de variation simultanée des entrées. Les sorties associées à chaque état sont également représentées sur cette table.

La table d'état correspondant au graphe d'état précédent est représentée sur la figure 10.4.

Etats	Etats Suivants 00	01	11	10	Sortie
1	5	-	2	1	0
2	-	3	2	1	0
3	4	3	7	-	0
4	4	6	-	1	1
5	5	6	-	1	0
6	5	6	7	-	0
7	-	6	7	1	0

Figure 10.4. *Table d'état de la machine détectant les séquences 10/11/01/00*

10.2.2. Minimisation du nombre d'états

10.2.2.a. Elimination des états équivalents

Le nombre d'états de la machine influe directement sur le nombre de variables secondaires et donc sur le nombre de portes nécessaires à la réalisation du circuit. Or, le nombre d'états utilisés pour représenter le cahier des charges, que ce soit sur le graphe des phases primitives ou sur la table d'état, n'est pas nécessairement minimum. Deux règles permettent de déterminer les états équivalents et par conséquent de minimiser le nombre d'états nécessaires.

- Règle R1 : Deux états sont équivalents s'il sont stables pour les mêmes valeurs d'entrée (état stable dans la même colonne), et si pour chaque combinaison d'entrée, ils ont même sorties et même états suivants.
- Règle R2 : Les états sont regroupés en différentes classes selon la position de l'état stable et les valeurs de sorties associées. Ainsi, deux états étant stables pour les mêmes valeurs d'entrée (état stable dans la même colonne) et ayant même sorties (pour chaque combinaison d'entrée) sont dans la même classe. Les états appartenant à une même classe sont équivalents s'il ne peuvent être séparés. Or les états appartenant à une même classe doivent être séparés si les états suivants associés à chacun d'eux sont dans des classes différentes.

Lorsque plusieurs états sont équivalents, il suffit de garder qu'un seul représentant par classe d'équivalence et de renommer les états suivants en conséquence.

Exemple : Les règles de minimisation appliquées à la table des phases primitive de la Figure 10.4 donnent:

R1: Aucun état n'est stable pour les mêmes valeurs d'entrée, n'a la même sortie et les mêmes états suivants.

R2: les états peuvent être regroupés en cinq classes.

(1)	(2)	(3)	(4)	(5)	Classes
(1)	(4)	(5)	(3 , 6)	(2 , 7)	Etats
			437-/567-	-321/-671	Etats suivants
			245-/345-	-451/-451	Classes des états suiv.

Les états 3 et 6 doivent être dissociés et le processus réitéré.

(1)	(2)	(3)	(4)	(5)	(6)	Classes
(1)	(4)	(5)	(3)	(6)	(2 ,7)	Etats
					-321/-671	Etats suivants
					-461/-561	Classes des états suiv.

Les états 2 et 7 doivent être séparés. Il y a maintenant qu'un seul état par classe. Il n'y a donc pas d'états équivalents. Pour réaliser cette machine, le nombre minimum d'états est de 7.

Remarque : Il est beaucoup plus rare de trouver des états équivalents en asynchrone qu'en synchrone. Cela provient essentiellement du fait que la structure même du graphe (entrées sur les états) conduit à repérer beaucoup plus facilement les états équivalents et donc à éviter les duplications.

10.2.2.b. Fusionnement d'états

A priori, le nombre d'états obtenu après minimisation devrait nous donner le nombre de variables secondaires nécessaires à la réalisation de la machine. En fait, chaque état étant caractérisé par les valeurs d'entrées pour lequel il est stable, des états différents, stables pour des valeurs d'entrées différentes, peuvent très bien être codés de manières identiques. L'étape suivante est donc essayer de fusionner certains états pour, en leur attribuant le même code, diminuer le nombre de variables secondaires nécessaires à la réalisation de la machine.

Règle de fusionnement : Deux états sont fusionnables uniquement s'il ont mêmes état suivants (pas d'incompatibilité sur les états suivants compte tenu des états indéterminés).

Afin de déterminer les fusionnements qui conduisent à un nombre d'états minimum, on réalise un graphe appelé graphe de fusionnement. Les nœuds de ce graphe sont les états et il y a présence d'une arête entre deux nœuds si et seulement si les deux états correspondants sont fusionnables.

Le graphe de fusionnement correspondant à la table d'état de la Figure 10.4 et donné sur la Figure 10.5.

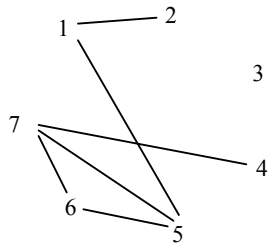


Figure 10.5. *Graphe de fusionnement*

Ce graphe donne les possibilités de regroupement d'états. Tous les états appartenant à un même groupement doivent être fusionnables entre eux. La recherche du nombre de groupement minimum est un problème classique (recherche de cliques dans un graphe) que nous ne détaillerons pas ici. Dans l'exemple précédent le nombre de groupements minimum est 4 mais comme le montre la Figure 10.6 il peut y avoir plusieurs solutions.

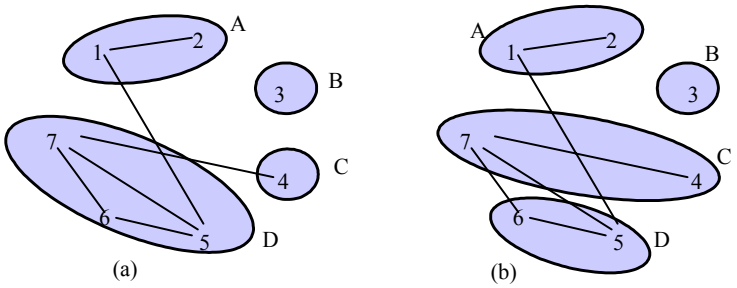


Figure 10.6. *Fusionnement d'états*

Remarque : Certaines solutions peuvent être "meilleures" que d'autres. Le choix d'une solution plutôt qu'une autre pourra s'appuyer sur des critères dont certains apparaîtront clairement par la suite. Notons dors et déjà que la solution (a) est à priori meilleure que la solution (b).

Une fois, les groupements déterminés, on peut réaliser le fusionnement des états composant un même groupement. Ceci conduit à une nouvelle table d'état appelée table des phases réduite. Lors de la fusion d'états, plusieurs situation peuvent se présenter. On peut avoir à fusionner des états stables, des états instables et des états indéterminés. La règle est la suivante:

- Un état stable et un état instable donnent un état stable.
- Un état stable et un état indéterminé donnent un état stable.
- Un état instable et un état indéterminé donnent un état instable

Exemple : En reprenant les groupements de la Figure 10.6.a on obtient la table. des phases réduite représentée sur la Figure 10.7.

Etats	Etats Suivants			
	00	01	11	10
A (1,2)	5	3	2	1
B (3)	4	3	7	-
C (4)	4	6	-	1
D (5,6,7)	5	6	7	1

Figure 10.7. *Table des phases réduite.*

10.2.3. Codage des états

Précédemment, nous avons noté qu'il n'est pas possible de prévoir un changement simultané des entrées. Pour les variables secondaires il en est de même. Si deux ou plusieurs variables secondaires nécessitent des changements de valeurs simultanés, il est impossible de garantir le fonctionnement de la machine. En conséquence, l'assignation des variables secondaires aux états d'une machine asynchrone réduite doit tenir compte de cette contrainte.

10.2.3.a. Courses et cycles

Ces notions seront abordées à partir de l'exemple de la table réduite de la Figure 10.10.

Sur cet exemple, lorsque l'état est y_1y_2=00 et que les entrées sont e_1e_2=00, l'état suivant est y_1y_2=11. La transition entre l'état 00 et l'état 11 introduit un changement des deux variables secondaires. La pratique impose qu'une seule variable y_1 ou y_2 commute à la fois. Selon le cas le système passera au préalable par l'état 01 ou 10. Ces états comportent dans l'exemple le même état instable 11 que l'état stable d'arrivée désiré.

Etats y_1y_2	Etats Suivants			
	00	01	11	10
00	11	00	10	01
01	11	00	11	01
11	11	00	10	11
10	11	10	10	11

Figure 10.8. *Table des phases réduite codée*

Définitions : Une situation nécessitant la variation de plus d'une variable secondaire est appelée une ***course***. Si l'état final que le système atteint ne dépend pas de l'ordre dans lequel les variables changent, alors la course est ***non critique***, dans le cas contraire la course est ***critique***.

Supposons maintenant que le système soit dans l'état y_1y_2=11 et que les entrées passent de 00 à 01. On désire par conséquent aller vers l'état stable 00. Si y_1 change plus vite que y_2, le circuit ira dans l'état 01 et atteindra finalement l'état stable 00. Par contre, si y_2 change plus vite que y_1, le système ira vers l'état stable 10 et s'y maintiendra. Nous avons une course critique et le fonctionnement du système sera incorrect.

Les courses critiques peuvent selon les cas être évitées, en dirigeant le système vers des états instables intermédiaires. Par exemple, lorsque le système est dans l'état y_1y_2=01 et e_1e_2=11, l'état d'arrivée voulu est 10. Mais puisque cette transition nécessite le changement simultané des deux variables secondaires y_1 et y_2, l'état instable est codé 11, dirigeant ainsi le système vers l'état 11, puis vers l'état 10.

Définition : La situation, pour laquelle le système passe par une séquence d'états instables est appelée ***cycle***. Lors de l'assignation des états, il est important de vérifier que chaque cycle se termine sur un état stable. Tout cycle ne comportant pas d'état stable appelé ***cycle instable*** doit absolument être évité.

10.2.3.b. Elimination des courses critiques

Sur l'exemple précédent afin d'éliminer la course critique de la colonne 01, on peut sélectionner un autre codage des variables secondaires (Figure 10.9).

Etats	Etats Suivants			
y_1y_2	00	01	11	10
00	10	00	10	01
01	10	00	11	01
10	10	00	11	10
11	10	11	11	10

Figure 10.9. *Assignation valide*

Définition : Une assignation qui ne contient ni course critique ni cycle instable est appelé une ***assignation valide***.

Remarque : Lorsqu'une colonne d'une table des transitions ne comporte qu'un seul état stable, il ne peut exister de course critique sur cet état. Les adjacences ne doivent pas être considérées sur cet état, toutefois il est dangereux de laisser libre les cases non spécifiées de la colonne, car au moment de l'écriture des fonctions logiques ces cases pourront être affectées à des valeurs conduisant à des cycles instables.

10.2.3.c. Méthode d'assignation

Dans de nombreuses situations, une assignation valide ne peut être obtenue par simple permutation des codes des états. Pour déterminer une assignation valide (pas de courses critiques ni de cycles instables) on doit étudier tous les passages d'états instables à états stables. Le passage d'un état instable à l'état stable correspondant peut se faire soit directement soit par l'intermédiaire d'états instables identiques ou d'états indéterminés.

La détermination d'une assignation valide peut être facilitée par la réalisation d'un graphe ou chaque état représente un sommet du graphe et chaque arc représente le fait que les deux états reliés doivent avoir un code adjacent. Ce graphe est appelé graphe d'adjacence. Afin de construire ce graphe, il est nécessaire d'introduire la notion de liaison essentielle et de liaison libre.

Définition : Lorsque pour aller d'un état instable à l'état stable correspondant, il n'y a pas d'autre solution que d'y aller directement, nous dirons qu'il y a une ***liaison essentielle*** entre les deux états. Lorsque pour aller d'un état instable à l'état stable correspondant, il y a plusieurs solutions nous dirons qu'il y a une ***liaison libre*** entre les deux états.

Toute liaison essentielle impose un arc sur le graphe d'adjacence. Toute liaison libre conduit à un ou plusieurs arcs mais le fait que la liaison soit libre laisse une certaine liberté quant à l'affectation de ces arcs. Une assignation valide, peut être déterminée en suivant la procédure suivante.

1. Analyser la table des phases réduite par colonnes et reporter sur le graphe d'adjacence les liaisons essentielles.
2. S'il n'existe pas de codage respectant les adjacences nécessaires, aller en 4.
3. Reporter sur le graphe d'adjacence les liaisons libres en choisissant une solution qui permette de coder les états en respectant les adjacences indiquées. Si une solution existe déterminer un codage respectant les adjacences indiquées par le graphe. Sinon aller en 4.

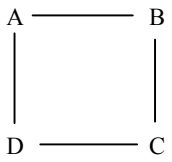
4. Rajouter une variable secondaire est recommencer en 3 (il n'existe plus de liaisons essentielles).

Remarque : S'il existe un état tel que tous les états suivants soientt indéterminés ou si le nombre d'états de la table des phases réduite n'est pas une puissance de 2, il n'existe pas de liaison essentielle.

Exemple : Reprenons la table des phases réduite de la Figure 10.7. Il existe des liaisons essentielles dans les colonnes e_1e_2=00 et e_1e_2=01 (Figure 10.10.a). Ces liaisons essentielles conduisent au graphe d'adjacence de la Figure 10.10.b .

Etats	Etats Suivants			
	00	01	11	10
A (1,2)	5	3	2	1
B (3)	4	3	7	-
C (4)	4	6	-	1
D (5,6,7)	5	6	7	1

(a)



(b)

Figure 10.10. *Liaisons essentielles*

On remarquera que sur ce graphe, si on doit rajouter un arc, il n'est plus possible de trouver un code respectant les adjacences indiquées avec deux variables. Il faut donc prendre en compte les liaisons libres en évitant de rajouter un arc. Ceci peut être fait en incluant les cycles présentés sur la Figure 10.11.a. Le graphe d'adjacence correspondant nous conduit à définir un codage adapté (Figure 10.11.b).

Etats	Etats Suivants			
	00	01	11	10
A (1,2)	5	3	2	1
B (3)	4	3	7	-
C (4)	4	6	-	1
D (5,6,7)	5	6	7	1

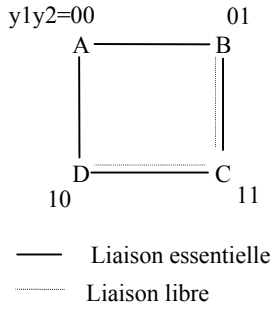


Figure 10.11. Liaisons essentielles et liaisons libres

Le codage de la table des phases réduite peut maintenant être réalisé (Figure 10.12). On remarquera le codage correspondant aux états (y_1y_2=01, e_1e_2=11) (y_1y_2=11, e_1e_2=11) (y_1y_2=11, e_1e_2=10). Ce codage est impératif pour que l'assignation soit valide.

Etats	Etats Suivants			
y_1y_2	00	01	11	10
00	10	01	00	00
01	11	01	11	--
11	11	10	10	10
10	10	10	10	00

Figure 10.12. *Assignation valide*

Remarque : Dans la quatrième colonne, il n'y a pas de possibilité d'avoir de course critique. L'affectation de la valeur 10 à l'état (y_1y_2=11, e_1e_2=10) n'est à priori pas obligatoire. Toutefois l'affectation de la valeur 00 à ce même état aurait entraîné l'affectation à 00 de l'état (y_1y_2=01, e_1e_2=10) afin d'éviter toute possibilité d'obtention d'un cycle instable.

10.2.3.d. Augmentation du nombre de variables secondaires

Il est souvent nécessaire d'augmenter le nombre de variables secondaires afin de résoudre le problème des courses critiques. Ce problème va être illustré sur l'exemple de la Figure 10.13.

Etats	Etats Suivants			
	00	01	11	10
A (2,6)	1	2	4	6
B (4,7)	1	3	4	7
C (5,8)	1	2	5	8
D (1,3)	1	3	5	6

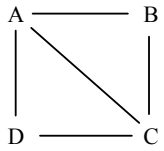


Figure 10.13. *Impossibilité d'assignation avec 2 variables*

Nous observons que la ligne a doit être adjacente aux trois autres lignes b, c et d. Bien évidemment, il est impossible d'effectuer des assignations adjacentes pour tous les états avec seulement deux variables d'états. Par conséquent, une troisième variable doit être ajoutée. L'ajout de cette variable à pour effet de créer des états supplémentaire (E,F,G,H) qui permettrons d'assurer des passages par code adjacent (Figure 10.14). Dans le cas général, pour une assignation comportant p variables d'états secondaires, chaque état peut être adjacent à p autres états au maximum

Etats	Etats Suivants			
	00	01	11	10
A (2,6)	1	2	4	6
B (4,7)	1	3	4	7
C (5,8)	1	2	5	8
D (1,3)	1	3	5	6
E	-	-	-	-
F	-	-	-	-
G	-	-	-	-
H	-	-	-	-

Figure 10.14. *Introduction d'une variable supplémentaire*

Les huit combinaisons de trois variables d'états sont représentées par les cases de la Figure 10.15. Pour déterminer une assignation valide, nous commençons par placer un état stable (A) dans la case $y_1y_2y_3$=000 pour indiquer que la ligne A sera assignée à l'état secondaire 000. De façon similaire, nous plaçons les états (B), (C) et (D) dans les trois cases adjacentes à la case (A). Cela implique toutefois que chacune des transitions des lignes B vers D et D vers C nécessite le changement de deux variables secondaires. Ces changements multiples peuvent être accomplis en amenant le système vers les destinations finales par l'intermédiaire d'états instables (Figure 10.15).

y_3 \ y_1y_2	00	01	11	10
0	A	C	F	D
1	B			E

Figure 10.15. *Assignation d'états*

Ce problème d'assignation est un problème classique d'algorithmique qui ne sera pas détaillé ici.

10.2.4. Synthèse

Cette dernière étape de la synthèse des machines asynchrones consiste à établir les fonctions logiques des variables secondaires ainsi que celles liées aux sorties de la machine.

10.2.4.a. Synthèse des variables secondaires

Une fois les assignations d'états réalisées sur la table des phases réduite nous avons toutes les information pour faire la synthèse des variables secondaires.

Exemple : En reprenant l'exemple précédent, à partir de la table de la Figure 10.12 on peut réaliser les tables de Karnaugh qui nous permettrons d'obtenir les relations de Y_1 et Y_2 (Figure 10.16).

y_1y_2 \ e_1e_2	00	01	11	10
00	1	0	0	0
01	1	0	1	-
11	1	1	1	1
10	1	1	1	0

$Y_1 = e_1'.e_2' + y_1.e_2 + y_2.e_1$

y_1y_2 \ e_1e_2	00	01	11	10
00	0	1	0	0
01	1	1	1	-
11	1	0	0	0
10	0	0	0	0

$Y_2 = y_1'.y_2 + y_2.e_1'.e_2' + y_1'.e_1'.e_2$

Figure 10.16. *Synthèse des variables secondaires*

Remarque : Rappelons que plusieurs outils de minimisation de fonctions logiques booléennes peuvent être utilisés: tableaux de Karnaugh, McCluskey, consensus ...

10.2.4.b. Synthèse des sorties

Pour l'instant, seuls les états stables possèdent des sorties spécifiées. Pour retrouver ces sorties, il suffit de se ramener à la table des phases réduite et à la table des phases primitive. Ces sorties peuvent également être présentées sous forme de table.

Exemple : La table de sortie présentée sur la Figure 10.17 est celle déduite de la table d'état réduite (Figure 10.7) et de la table d'état primitive (Figure 10.4).

Etats y_1y_2	Sorties 00	01	11	10
00	-	-	0	0
01	-	0	-	-
11	1	-	-	-
10	0	0	0	-

Figure 10.17. *Synthèse des sorties*

Notre but ici est de considérer l'assignation des sorties pour les états instables de la machine réduite. Cette assignation dépend des changements des sorties déjà spécifiées ainsi que des objectifs de conception (machine lente ou rapide).

Supposons que la valeur des sorties pour les états indéterminés nous soit indifférente. Dans ce cas, la valeur des sorties peut rester non spécifiée afin d'optimiser postérieurement la fonction logique de sortie. Mais dans ce cas, on s'expose à des commutations transitoires des sorties. En effet, supposons que le système doive aller d'un état stable vers un autre état stable pour lequel la sortie est identique. Si la sortie affectée à un des états instables intermédiaires est différente, la sortie présentera une commutation transitoire lors du passage entre ces deux états.

Supposons maintenant que le système doive aller d'un état stable vers un autre état stable pour lequel la sortie est différente. La valeur de sortie attribuée à l'état transitoire peut alors correspondre soit à l'état de départ soit à l'état d'arrivée. Ce choix doit être fait selon l'objectif désiré, à savoir si l'on désire un système commutant au plus tôt ou au plus tard. On parlera alors de machine lente et de machine rapide.

Exemple : En reprenant l'exemple précédant, les tables de sorties en machine rapide (a) et en machine lente sont données sur la Figure 10.18. Ces tables sont établies à partir de la connaissance des passages d'états instables à états stables (Figure 10.11).

Etats y_1y_2	Sorties 00	01	11	10
00	0	0	0	0
01	1	0	0	-
11	1	0	0	0
10	0	0	0	0

(a) Srapide = $y_2.e_1'.e_2'$

Etats y_1y_2	Sorties 00	01	11	10
00	0	0	0	0
01	0	0	0	-
11	1	1	0	1
10	0	0	0	0

(b) Slente = $y_1.y_2.(e_1' + e_2')$

Figure 10.18. *Machine rapide et machine lente*

10.3. Synthèse de dispositifs synchrones élémentaires

Un dispositif synchrone élémentaire est un composant synchronisé par une horloge. Une bascule est par exemple un dispositif synchrone élémentaire. Pour concevoir un tel système plusieurs démarches peuvent être envisagées. Il en est une qui est de considérer l'horloge H comme une entrée banalisée est ainsi de considérer le système global comme étant un système asynchrone. Dans ce chapitre nous réaliserons la synthèse des 3 bascules suivantes :

- La bascule D-Latch,
- La bascule D,

- La bascule D avec signaux de forçage « Clear » et « Preset »

10.3.a. Synthèse d'une D-Latch

Une bascule D-Latch est un dispositif comportant 2 entrées D et H et une sortie Q telles que :

- Si H = 1 (niveau 1) Q = D
- Sinon mémoire

Le graphe d'état d'une bascule D-Latch considéré comme un système asynchrone est donné sur la figure 10.19.

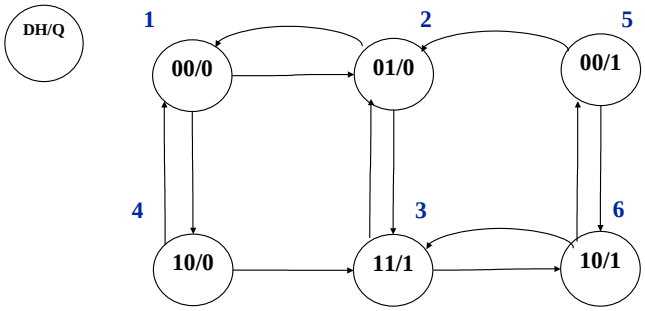


Figure 10.19. *Graphe d'état d'une D-Latch*

La table d'état correspondante est représentée sur la figure 10.20.

Etats	Etats Suivants				Sortie
	00	01	11	10	
1	1	2	-	4	0
2	1	2	3	-	0
3	-	2	3	6	1
4	1	-	3	4	0
5	5	2	-	6	1
6	5	-	3	6	1

Figure 10.20. *Table d'état d'une D-Latch*

L'application des deux règles de minimisation du nombre d'état ne donnant rien, il n'y a pas, parmi ces états, d'états équivalents. Le graphe de fusionnement est représenté sur la figure 10.21.

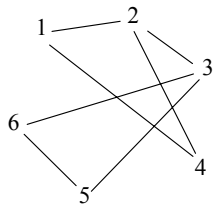


Figure 10.21. *Graphe de fusionnement*

En fusionnant les états 1,2,4 et 3,5,6, la table d'état devient (Figure 10.22)

Etats	Etats Suivants			
	00	01	11	10
A(1,2,4)	1	2	3	4
B(3,5,6)	5	2	3	6

Figure 10.22. *Table d'état réduite d'une D-Latch*

En codant 0 l'état A et 1 l'état B les tables d'état et de sortie sont les suivantes (Figure 10.23):

Etats y	Etats Suivants 00	01	11	10
0	0	0	1	0
1	1	0	1	1

Etats y	Sortie 00	01	11	10
0	0	0	1	0
1	1	0	1	1

Figure 10.23. *Table d'état codée et table de sortie d'une D-Latch*

La structure du système (D-Latch) ainsi obtenu est présenté sur la figure 10.24.

y = H'y + H.D

Q = y

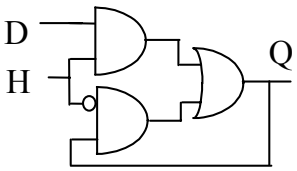


Figure 10.24. *Bascule D-Latch*

10.3.b. Synthèse d'une bascule D

Une bascule bascule D est un dispositif comportant 2 entrées D et H et une sortie Q telles que :

- Si H passe de 0 à 1 (front montant) Q = D
- Sinon mémoire

Le graphe d'état d'une bascule D considéré comme un système asynchrone est donné sur la figure 10.25.

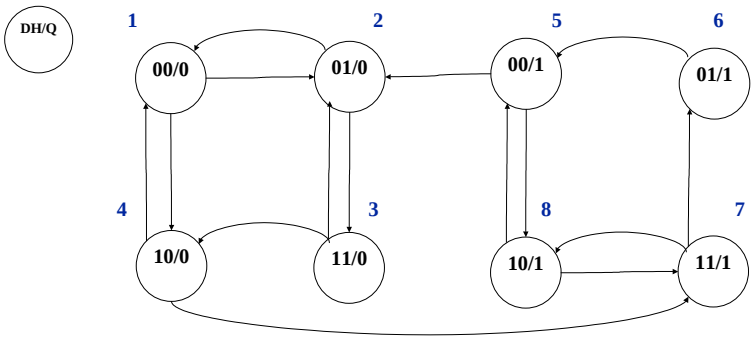


Figure 10.25. *Graphe d'état d'une bascule D*

La table d'état correspondante est représentée sur la figure 10.26.

Etats	Etats Suivants				Sortie
	00	01	11	10	
1	1	2	-	4	0
2	1	2	3	-	0
3	-	2	3	4	0
4	1	-	7	4	0
5	5	2	-	8	1
6	5	6	7	-	1
7	-	6	7	8	1
8	5	-	7	8	1

Figure 10.26. *Table d'état d'une bascule D*

L'application des deux règles de minimisation du nombre d'état ne donnant rien, il n'y a pas, parmi ces états, d'états équivalents. Le graphe de fusionnement est représenté sur la figure 10.27.

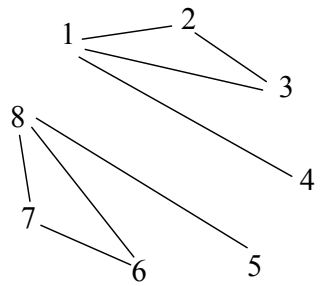


Figure 10.27. *Graphe de fusionnement*

En fusionnant les états 1,2,3 et 6,7,8, la table d'état devient (Figure 10.28)

Etats	Etats Suivants			
	00	01	11	10
A(1,2,3)	1	2	3	4
B(4)	1	-	7	4
C(5)	5	2	-	8
D(6,7,8)	5	6	7	8

Figure 10.28. *Table d'état réduite d'une bascule D*

Le graphe d'adjacence est présenté sur la figure 10.29.

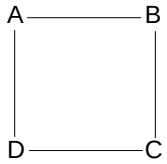


Figure 10.29. *Graphe d'adjacence*

En codant 00 l'état A, 01 l'état B, 11 l'état C et 10 l'état D, les tables d'état et de sortie sont les suivantes (Figure 10.30):

Etats	Etats Suivants			
y_1y_2	00	01	11	10
00	00	00	00	01
01	00	00	11	01
11	11	01	10	10
10	11	10	10	10

Etats	Sorties			
y_1y_2	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	1	0	1	1
10	1	1	1	1

Figure 10.30. *Table d'état codée et table de sortie d'une bascule D*

La structure d'une bascule D peut ainsi être obtenue à partir des équations suivantes :

$y_1 = y_1.y_2' + y_1.H' + y_2.D.H$

$y_2 = y_1.D'.H' + y_1.y_2.D' + y_1'.y_2.D + y_1'.D.H'$

$Q = y_1$

10.3.c. Synthèse d'une bascule D avec signaux de forçage « Clear » et « Preset »

Une bascule bascule D avec signaux de forçage est un dispositif comportant 2 entrées D et H, 2entrées de forçage Clear et Preset et une sortie Q telles que :

- Si Clear=Preset = 0
  - Si H passe de 0 à 1 (front montant) Q = D
  - Sinon mémoire
- Si Clear = 1 (Preset = 0), Q = 0
- Si Preset = 1 (Clear = 0), Q = 1

Le graphe d'état d'une bascule D considéré comme un système asynchrone est donné sur la figure 10.31.

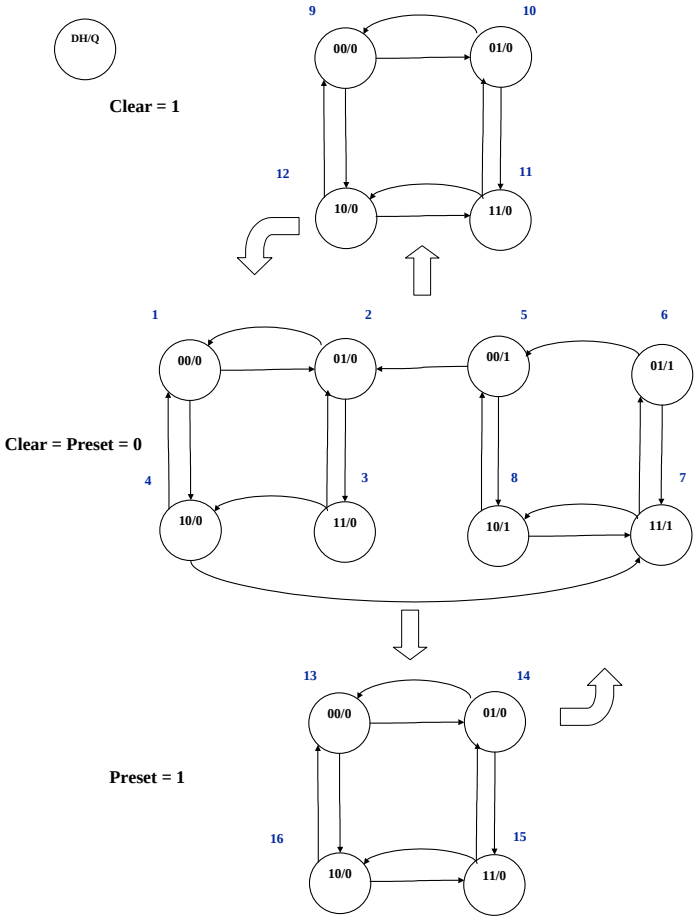


Figure 10.31. *Graphe d'état d'une bascule D avec Clear et Preset*

La table d'état correspondante est représentée sur la figure 10.32.

Etats	Etats Suivants												Sortie
	Clear=Preset=0				Clear = 1				Preset = 1				Q
	00	01	11	10	00	01	11	10	00	01	11	10	
1	1	2	-	4	9	-	-	-	13	-	-	-	0
2	1	2	3	-	-	10	-	-	-	14	-	-	0
3	-	2	3	4	-	-	11	-	-	-	15	-	0
4	1	-	7	4	-	-	-	12	-	-	-	16	0
5	5	2	-	8	9	-	-	-	13	-	-	-	1
6	5	6	7	-	-	10	-	-	-	14	-	-	1
7	-	6	7	8	-	-	11	-	-	-	15	-	1
8	5	-	7	8	-	-	-	12	-	-	-	16	1
9	1	-	-	-	9	10	-	12	-	-	-	-	0
10	-	2	-	-	9	10	11	-	-	-	-	-	0
11	-	-	3	-	-	10	11	12	-	-	-	-	0
12	-	-	-	4	9	-	11	12	-	-	-	-	0
13	5	-	-	-	-	-	-	-	13	14	-	16	1
14	-	6	-	-	-	-	-	-	13	14	15	-	1
15	-	-	7	-	-	-	-	-	-	14	15	16	1
16	-	-	-	8	-	-	-	-	13	-	15	16	1

Figure 10.32. *Table d'état d'une bascule D avec Clear et Preset*

L'application des deux règles de minimisation du nombre d'état ne donnant rien, il n'y a pas, parmi ces états, d'états équivalents. Le graphe de fusionnement est représenté sur la figure 10.33.

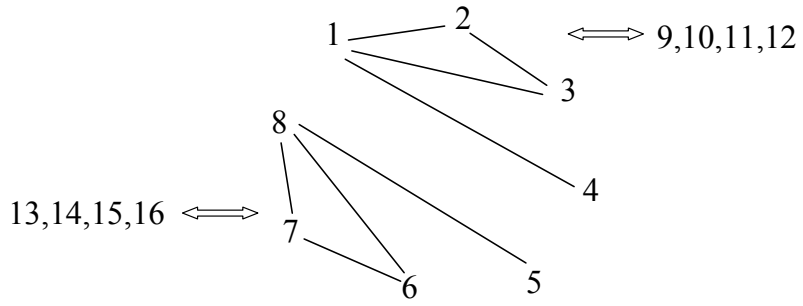


Figure 10.33. *Graphe de fusionnement*

En fusionnant les états 1,2,3,9,10,11,12 et 6,7,8,13,14,15,16, la table d'état devient (Figure 10.34)

Etats	Etats Suivants											
	Clear=Preset=0				Clear = 1				Preset = 1			
	00	01	11	10	00	01	11	10	00	01	11	10
A	1	2	3	4	9	10	11	12	13	14	15	-
B	1	-	7	4	-	-	-	12	-	-	-	16
C	5	2	-	8	9	-	-	-	13	-	-	-
D	5	6	7	8	-	10	11	12	13	14	15	16

Figure 10.34. *Table d'état réduite d'une bascule D avec Clear et Preset*

Le graphe d'adjacence est présenté sur la figure 10.35

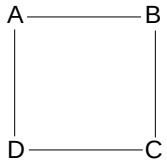


Figure 10.35. *Graphe d'adjacence*

En codant 00 l'état A, 01 l'état B, 11 l'état C et 10 l'état D, la table d'état codée (Figure 10.36) et la table de sortie (Figure 10.37) sont les suivantes :

Etats y_1y_2	Etats Suivants											
	Clear=Preset=0				Clear = 1				Preset = 1			
	00	01	11	10	00	01	11	10	00	01	11	10
00	00	00	00	01	00	00	00	00	10	10	10	-
01	00	00	11	01	00	-	-	00	-	-	-	11
11	11	01	10	10	01	-	-	-	10	-	-	10
10	11	10	10	10	-	00	00	00	10	10	10	10

Figure 10.36. *Table d'état codée d'une bascule D avec Clear et Preset*

Etats y_1y_2	Sortie											
	Clear=Preset=0				Clear = 1				Preset = 1			
	00	01	11	10	00	01	11	10	00	01	11	10
00	0	0	0	0	0	0	0	0	1	1	1	-
01	0	0	1	0	0	-	-	0	-	-	-	1
11	1	0	1	1	0	-	-	-	1	-	-	1
10	1	1	1	1	-	0	0	0	1	1	1	1

Figure 10.37. *Table d'état codée d'une bascule D avec Clear et Preset*

La structure d'une bascule D avec signaux de forçage Clear et Preset peut ainsi être obtenue à partir des équations suivantes :

$y_1 = Clear'.Preset'.[y_1.y_2' + y_1.H' + y_2.D.H] + Preset$

$y_2 = Clear'.Preset'.[y_1.D'.H' + y_1.y_2.D' + y_1'.y_2.D + y_1'.D.H'] + Clear.y_1.y_2 +$
$Preset.y_1'.y_2$

$Q = y_1$

Références

[McCluskey] « *Logic Design Principle* », E.J. Mc Cluskey,
Editions Prentice-Hall

[DeMicheli] « *Synthesis and optimization of digital circuits* », G. De Micheli,
Editions McGraw-Hill

[Devadas] « *Logic Synthesis* », S. Devadas, A. Ghosh, K. Keutzer, Editions
McGraw-Hill

[Lagasse_1] « *Logique Combinatoire* », J. Lagasse, M. Courvoisier, J.P. Richard
Editions Dunod Université

[Lagasse_2] « *Logique Combinatoire et séquentielle* », J. Lagasse, J Erceau
Editions Dunod Université

[Muller] « *Arithmétique des ordinateurs* », J.M. Muller.
Editions Masson

[Aumiaux] « Logique arithmétique et techniques synchrones », M. Aumiaux
Editions Masson

[Sasao_1] « *Representation of discrete functions* », T. Sasao, M Fujita
Kluwer Academic Publisher

[Sasao_2] « *Logic synthesis and optimization* », T. Sasao
Kluwer Academic Publisher

[Mange] « *Analyse et synthèse des systèmes logiques* », D. Mange
Traité d'électricité, Volume V, Presses Polytechniques Romandes, 1992.

[Letocha] « *Introduction aux circuits logiques* », J. Letocha,
McGraw-Hill, 1985.

[Toccir] « Circuits numériques - Théorie et applications », J. Toccir
Dunod, 1992

[Weber] « Circuits numériques et Synthèse logique – Un outil : VHDL », J. Weber
Masson, 1995