Université Libre de Bruxelles

Synthèse

Circuits logiques et numériques
ELEC-H-305

Auteur :
Cédric Hannotier

Professeur :
Dragomir Milojevic

Année 2015 - 2016

Appel à contribution

Synthèse OpenSource

Ce document est grandement inspiré de l'excellent cours donné par Dragomir Milojevic à l'EPB (École Polytechnique de Bruxelles), faculté de l'ULB (Université Libre de Bruxelles). Il est écrit par les auteurs susnommés avec l'aide de tous les autres étudiants et votre aide est la bienvenue ! En effet, il y a toujours moyen de l'améliorer surtout que si le cours change, la synthèse doit être changée en conséquence. On peut retrouver le code source à l'adresse suivante

`https://github.com/nenglebert/Syntheses`

Pour contribuer à cette synthèse, il vous suffira de créer un compte sur *Github.com*. De légères modifications (petites coquilles, orthographe, ...) peuvent directement être faites sur le site ! Vous avez vu une petite faute ? Si oui, la corriger de cette façon ne prendra que quelques secondes, une bonne raison de le faire !

Pour de plus longues modifications, il est intéressant de disposer des fichiers : il vous faudra pour cela installer LaTeX, mais aussi *git*. Si cela pose problème, nous sommes évidemment ouverts à des contributeurs envoyant leur changement par mail ou n'importe quel autre moyen.

Le lien donné ci-dessus contient aussi le `README` contient de plus amples informations, vous êtes invités à le lire si vous voulez faire avancer ce projet !

Licence Creative Commons

Le contenu de ce document est sous la licence Creative Commons : *Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)*. Celle-ci vous autorise à l'exploiter pleinement, compte- tenu de trois choses :

1. *Attribution* ; si vous utilisez/modifiez ce document vous devez signaler le(s) nom(s) de(s) auteur(s).
2. *Non Commercial* ; interdiction de tirer un profit commercial de l'œuvre sans autorisation de l'auteur
3. *Share alike* ; partage de l'œuvre, avec obligation de rediffuser selon la même licence ou une licence similaire

Si vous voulez en savoir plus sur cette licence :

`http://creativecommons.org/licenses/by-nc-sa/4.0/`

Merci !

Table des matières

1 Systèmes de numérotation **1**
- 1.1 Représentation des nombres . . . 1
- 1.2 Conversions . . . 1
- 1.3 Opération arithmétiques . . . 3
- 1.4 Nombres négatifs . . . 3
  - 1.4.1 Signe et Valeur Absolue (SVA) . . . 3
  - 1.4.2 Complément à la base . . . 3
  - 1.4.3 Overflow en C_2 . . . 4
- 1.5 Virgule flottante : Forme généralisée . . . 4
  - 1.5.1 Standard IEEE 754 . . . 5

2 Codes correcteurs d'erreurs et Algèbre de Boole **6**
- 2.1 Code . . . 6
  - 2.1.1 Codes pondérés . . . 6
  - 2.1.2 Codes non-pondérés . . . 7
  - 2.1.3 Codes correcteurs . . . 8
- 2.2 Algèbre de Boole . . . 9
  - 2.2.1 Définition . . . 9
  - 2.2.2 Algèbre de Boole à 2 valeurs . . . 10
- 2.3 Fonctions logiques . . . 12
  - 2.3.1 Représentation des fonctions logiques . . . 12
- 2.4 Réalisation matérielle . . . 14

3 Fonctions Booléennes et circuits logiques **15**
- 3.1 Fonctions Booléennes . . . 15
  - 3.1.1 Fonction logique . . . 15
- 3.2 Modes de représentation . . . 15
  - 3.2.1 Tables de Vérité . . . 15
  - 3.2.2 Expression algébriques . . . 16
- 3.3 Simplification . . . 17
- 3.4 Logique à 2 et à plusieurs niveaux . . . 18
  - 3.4.1 Expansion de Boole (Shannon) . . . 19

4 Simplification des fonctions logiques : K-Maps **20**
- 4.1 Méthodes de simplification . . . 20
- 4.2 Tables de Karnaugh . . . 20
  - 4.2.1 Notion de sous-cube dans un n-cube . . . 21
- 4.3 Simplification des fonctions logiques avec les K-Maps . . . 23
  - 4.3.1 Méthode de simplification (algorithme) . . . 24

4.3.2 Remarque 24
4.4 Fonctions non-complètement spécifiées 25

5 Synthèse de circuits et Quine Mc.Cluskey **26**
5.1 Synthèse des circuits à partir de spécification verbales 26
5.2 Additionneurs re-visités 26
5.2.1 Addition bit à bit 26
5.2.2 Additionneur *Carry Look Ahead* (CLA) 27
5.3 Encodeurs de priorité 28
5.4 Simplification des fonctions : Méthode Quine-Mc.Cluskey 28
5.4.1 Étape 1 : Analyse 28

6 Quine Mc.Cluskey et Circuits séquentiels **31**

7 Synthèse des systèmes séquentiels synchrones **32**

Chapitre 1

Systèmes de numérotation

1.1 Représentation des nombres

Soit un nombre N en base r, sa forme généralisé s'écrit :

$$N = \underbrace{\sum_{i=0}^{i=n} a_i r^i}_{\text{Partie entière}} + \underbrace{\sum_{j=1}^{i=m} b_j r^{-j}}_{\text{Partie fractionnaire}} \tag{1.1}$$

Les indexes i, j s'appellent les poids. On distingue 2 parties :

- Partie entière : $n+1$ chiffres $(a_0, \ldots, a_n)$
  - $i = n$: bit de poids plus fort (Most Significant Bit (MSB))
  - $i = 0$: bit de poids le plus faible (Least Significant Bit (LSB), donné par $\frac{N-a_0}{r}$)
- Partie fractionnaire : m chiffres $(b_1, \ldots, b_m)$

Parmi l'infinité de bases r possibles, on en distingue 4 :

1.2 Conversions

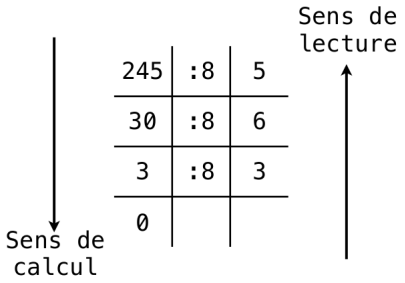
De manière générale, pour passer d'une base p à une base q, on fera $(N)_p \to (N)_{10} \to (N)_q$.

Pour passer de la base p à la base 10, il suffit de réécrire le nombre sous sa forme générale et de calculer le résultat.
Pour passer de la base 10 à la base q, il faut séparer la partie entière de la fractionnaire :

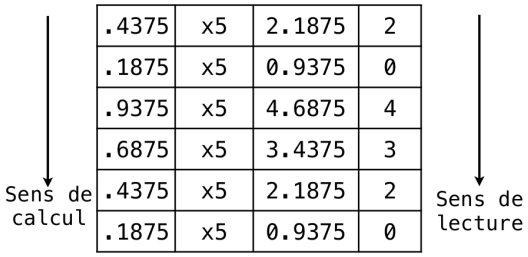
- Partie entière : diviser par la base q, noter le reste de la division et répéter l'opération jusqu'à ce que le nombre soit plus petit que q. Une fois le résultat obtenu, le chiffre résultant se lit dans le sens **contraire** que celui du calcul. Exemple :

Type	Base
Décimale	{0,1,2,3,4,5,6,7,8,9}
Binaire	{0,1}
Octal	{0,1,2,3,4,5,6,7}
Hexadécimal	{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

Tableau 1.1 – Bases utiles



— Partie fractionnaire : écrire la partie fractionnaire sous la forme $0.xxx...$, la multiplier par q, noter le chiffre avant le . du résultat, réinitialiser ce chiffre à 0 et répéter l'opération. Le nombre résultant se lit dans le **même** sens que celui du calcul. Exemple :



⚠Faire gaffe avec l'hexadécimal, ne pas oublier de remplacer les chiffres > 9 par les lettres correspondantes

Néanmoins, dans le cas des bases utiles (Tableau 1.1), on préfèrera passer par la base 2. En effet, il suffit de regrouper les chiffres par groupe de x où x vaut l'exposant entre 2 bases (exemple : $x = 4$ pour les bases $2 \leftrightarrow 16$). On regroupera les chiffres en partant de part et d'autre de la virgule.

Base	Nombre		
10	245		
2	11110101		
Par 3	11	110	101
8	3	6	5
Par 4		1111	0101
16		F	5

Base	Nombre		
16	1F2		
2	0001|1111|0010		
Par 3	111	110	010
8	7	6	2

FIGURE 1.1 – Conversion entre bases utiles

⚠Surtout ne pas oublier les chiffres manquants pour la partie fractionnaire sinon un $(110)_2$ se transformera en $((0)11)_2$

Astuce En binaire, lorsqu'un nombre est composé majoritairement de 1 et qu'on veut sa valeur, comme le nombre 111110010, comptez le nombre de chiffre $\Rightarrow x$ et faire $2^x - 1$ et

soustraire 2^i pour chaque 0. Dans notre exemple :

$$(111110010)_2 = (?)_{10} \tag{1.2}$$
$$x = 9 \tag{1.3}$$
$$2^9 - 1 = (511)_{10} \tag{1.4}$$
$$511 - 2^0 - 2^2 - 2^3 = (498)_{10} \tag{1.5}$$
$$(111110010)_2 = (498)_{10} \tag{1.6}$$

1.3 Opération arithmétiques

Les additions, soustractions, multiplications et divisions se déroulent comme en base 10, si ce n'est qu'au lieu de reporter quand on arrive au-dessus de 9, on reporte quand on arrive au dessus de la base−1 (grosso modo). Voir TP 1

Remarque dans le cas de codage binaire sur 8 bits (chiffre maximum 255), si l'on fait $236 + 170$, nous obtenons un chiffre au-dessus de la limite pour 8 bits, nous aurons donc un $9^{\text{ème}}$ bit. Le résultat sera donc tronqué car codé sur 8 bits.Ce problème de débordement s'appelle l'*overflow*

1.4 Nombres négatifs

En binaire, il existe 3 modes de représentation pour les nombres négatifs :

1. Signe et Valeur Absolue (SVA)
2. Complément à la base (C_1)
3. Complément à 2 (C_2)

1.4.1 Signe et Valeur Absolue (SVA)

Par convention :
- 1 bit réservé pour le signe tel que :
  - 0 = positif
  - 1 = négatif
- le reste réservé pour la valeur absolue

Ainsi, sur un mode à 8 bits, on peut représenter des chiffres $\in [-127, 127]$
Pour faire des opérations arithmétiques avec cette notation, il faut :

1. Comparer les signes pour déterminer le signe des résultats
2. Comparer la magnitude des nombres pour déterminer le sens ($A < B \rightarrow B - A$, $A > B \rightarrow A - B$)

Niveau matériel, c'est galère...

1.4.2 Complément à la base

Complément à 1 (C_1)

Cette notation est valeur $\forall$ base. Le principe est de faire la soustraction comme une addition. Soit 2 nombre A et B, en base r, codés sur m chiffres

$$A - B = A + (-B) \tag{1.7}$$
$$= A + (r^m - B) = A + B' \tag{1.8}$$

$$\text{où} \quad \begin{array}{l} B' = \textit{complément à la base } r \\ B + B' = r^m \end{array}$$

Mais pour arriver à ça, il faut quand même faire la soustraction $r^m - B$. Réorganisons

$$B' = (r^m - B) = ((r^m - 1) - B) + 1 \tag{1.9}$$

$(r^m - 1) - B$ est un complément de chaque chiffre de B

$$(r^m - 1) - B = (\underbrace{(r-1)\ (r-1)}_{\text{⚠Concaténation}} \dots\ (r-1) - (b_{m-1}\ b_{m-2} \dots\ b_0) \tag{1.10}$$

$$= ((r-1) - b_{m-1}((r-1)b_{m-2}) \dots ((r-1) - b_0) \tag{1.11}$$

$$= b'_{m-1}\ b'_{m-2} \dots\ b'_0 \tag{1.12}$$

Cette méthode est très pratique en **binaire**, chaque chiffre est simplement inversé ! Seul problème, nous avons 2 manière de représenter le 0 (sur 8 bits : 0000000 et 11111111). Les opérations arithmétique en C_1 sont faisable mais comporte parfois quelques cas particuliers.

Complément à 2 (C_2)

Au lieu d'utiliser le complément à 1 $B' = (r^m - 1) - B$, on utilise le complément à 2 $B' = (r^m - 1) - B + 1$. Grosso modo, on ajoute 1 au complément à 1. Contrairement à C_1 et SAV, on peut représenter les nombre $\in [-128, 127]$ et nous n'avons qu'une notation pour 0. La conséquence de tout ça est qu'il sera beaucoup plus facile de faire des opération arithmétique (soustraction $\Rightarrow$ sommation). Seul problème, l'overflow...

Méthode de conversion pour complément

Nous utiliserons la plupart du temps le complément à 2. Pour convertir un nombre en complément à 1 ou 2 rien de plus simple :

1. Prendre la valeur absolue du nombre en **base 2** et compléter avec des 0 pour avoir les m bits demandés
2. Si le nombre est négatif :
   (a) Inverser chaque bit $\Rightarrow C_1$
   (b) $C_1 + 1 \Rightarrow C_2$

1.4.3 Overflow en C_2

une règle très simple permet de savoir si nous somme en overflow ou si le bit débordant peut être oublié sans risque

Si les 2 derniers bits du résultats sont **différent** (01 ou 10)$\Rightarrow$ **OK**
Si Si les 2 derniers bits du résultats sont **les mêmes** (11 ou 00) $\Rightarrow$ **Overflow**

1.5 Virgule flottante : Forme généralisée

Les nombres en virgule fixe à 32 ou 64 bits limitent fortement les calculs et augmenter le nombre de bit n'est pas une solution $\rightarrow$ Nombre en virgule flottante :

$$N = mantisse \times (base)^{exposant} \tag{1.13}$$

1.5.1 Standard IEEE 754

On distingue 2 type de précision :

— Simple précision : 32 bits

Nombre de bits	1	8	23
Type	Signe de la mantisse	Exposant (0 à 255)	Fraction normalisée
Biais		127	

Tableau 1.2 – IEEE 754 - Simple Précision

— Double précision : 64 bits

Nombre de bits	1	11	52
Type	Signe de la mantisse	Exposant (0 à 2047)	Fraction normalisée
Biais		1023	

Tableau 1.3 – IEEE 754 - Double Précision

Pour convertir un chiffre en virgule flottante il faut :

1. Convertir le nombre en binaire
2. Mettre sous forme $1.abcd... \ (\times x)$
3. Déterminer le signe (0 ou 1, voir sous-section 1.4.1)
4. Calculer l'exposant : $E = x + biais$
5. Écrire la mantisse ($abcd...$)

⚠Le 1 de $1.abcd...$ n'est pas a écrire dans la mantisse

Chapitre 2

Codes correcteurs d'erreurs et Algèbre de Boole

2.1 Code

Pour pouvoir, en binaire, représenter les chiffres de 0 à 9, il nous faut au minimum 4 bits ($\log_2 10$). Donc, nous perdons 6 codes du à l'arrondissement à 4 bits (1010, 1011, 1100, 1101, 1101 et 1111). En octal et hexadécimal, c'est plus pratique car il n'y a pas de perte du à l'arrondissement ($\log_2 8 = 3$ et $\log_2 16 = 4$).Il y a donc, d'autres façons de coder en fonction de l'utilisation qu'on en fait. On en distingue 3 classes :

1. Codes pondérés (*weighted codes*)
   - 8421 *Binary coded Decimal* (BCD) où chaque chiffre est codé séparément
   - Codes auto-complémentaires
     - 2421 Code
     - Code excédent 3 (Excess 3)

2. Codes non-pondérés
   - Code Gray (code cyclique)
   - *American Standard Code for Information Interchage* (code ASCII)

3. Code détecteurs d'erreur

2.1.1 Codes pondérés

Voici une table pour bien comprendre comment fonctionnent les codes auto-complémentaires (le $-$ de -3 correspond au chiffre négatif)

Décimal	8421 (BCD)	2421	642-3
0	0000	0000	0000
1	0001	0001	0101
2	0010	0010	0010
3	0011	0011	1001
4	0100	0100	0100
5	0101	1011	1011
6	0110	1100	0110
7	0111	1101	1101
8	1000	1110	1010
9	1001	1111	1111

Tableau 2.1 – Code auto-complémentaire

Addition en BCD

L'addition en BCD est simple, on additionne par 4 bits (chaque chiffre composant la base 10). Si le chiffre est ≥ 10, on ajoute $+6\,(0110)_2$ car cela correspond à un report de 10 en binaire. En effet, regroupé par 4 correspond à de l'hexadécimal, du coup, pour passer de 10 à 0, il faut ajouter 6 ($A \overset{+1}{\to} B \overset{+1}{\to} \ldots \overset{+1}{\to} F \overset{+1}{\to} 0$). Exemple :

```
 532       0101  0011  0010
+268      +0010  0110  1000
              1     1
_____________________________
 800      1000   1010  1010
                 0110  0110
_____________________________
         (1000   0000  0000)2
          ↓      ↓     ↓
_____________________________
         (   8      0     0)10
```

2.1.2 Codes non-pondérés

Code Gray

Le code Gray est un code suivant le principe de *Look-up Table*, c'est-à-dire que la conversion ne suit pas une règle mais une table de correspondance associant des valeurs. Le principe du Gray est que 2 codes voisins ne diffèrent que par la valeur d'un bit.

Décimal	Gray
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	101

Tableau 2.2 – Code Gray

Code ASCII

Le code ASCII est utilisé pour coder les caractères dans des systèmes de traitement numérique, comme les majuscules, les signes de ponctuation et cetera. Le code est basé sur 8 bits, donc 256 possibilités, mais en réalité, on en utilise que 128.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	@	96	60	`
1	1	*[START OF HEADING]*	33	21	!	65	41	A	97	61	a
2	2	*[START OF TEXT]*	34	22	"	66	42	B	98	62	b
3	3	*[END OF TEXT]*	35	23	#	67	43	C	99	63	c
4	4	*[END OF TRANSMISSION]*	36	24	$	68	44	D	100	64	d
5	5	*[ENQUIRY]*	37	25	%	69	45	E	101	65	e
6	6	*[ACKNOWLEDGE]*	38	26	&	70	46	F	102	66	f
7	7	*[BELL]*	39	27	'	71	47	G	103	67	g
8	8	*[BACKSPACE]*	40	28	(	72	48	H	104	68	h
9	9	*[HORIZONTAL TAB]*	41	29	)	73	49	I	105	69	i
10	A	*[LINE FEED]*	42	2A	*	74	4A	J	106	6A	j
11	B	*[VERTICAL TAB]*	43	2B	+	75	4B	K	107	6B	k
12	C	*[FORM FEED]*	44	2C	,	76	4C	L	108	6C	l
13	D	*[CARRIAGE RETURN]*	45	2D	-	77	4D	M	109	6D	m
14	E	*[SHIFT OUT]*	46	2E	.	78	4E	N	110	6E	n
15	F	*[SHIFT IN]*	47	2F	/	79	4F	O	111	6F	o
16	10	*[DATA LINK ESCAPE]*	48	30	0	80	50	P	112	70	p
17	11	*[DEVICE CONTROL 1]*	49	31	1	81	51	Q	113	71	q
18	12	*[DEVICE CONTROL 2]*	50	32	2	82	52	R	114	72	r
19	13	*[DEVICE CONTROL 3]*	51	33	3	83	53	S	115	73	s
20	14	*[DEVICE CONTROL 4]*	52	34	4	84	54	T	116	74	t
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	5	85	55	U	117	75	u
22	16	*[SYNCHRONOUS IDLE]*	54	36	6	86	56	V	118	76	v
23	17	*[ENG OF TRANS. BLOCK]*	55	37	7	87	57	W	119	77	w
24	18	*[CANCEL]*	56	38	8	88	58	X	120	78	x
25	19	*[END OF MEDIUM]*	57	39	9	89	59	Y	121	79	y
26	1A	*[SUBSTITUTE]*	58	3A	:	90	5A	Z	122	7A	z
27	1B	*[ESCAPE]*	59	3B	;	91	5B	[	123	7B	{
28	1C	*[FILE SEPARATOR]*	60	3C	<	92	5C	\	124	7C	|
29	1D	*[GROUP SEPARATOR]*	61	3D	=	93	5D	]	125	7D	}
30	1E	*[RECORD SEPARATOR]*	62	3E	>	94	5E	^	126	7E	~
31	1F	*[UNIT SEPARATOR]*	63	3F	?	95	5F	_	127	7F	*[DEL]*

FIGURE 2.1 – Table ASCII

2.1.3 Codes correcteurs

Distance de Hamming quantifie la distance (la différence)entre 2 séquences de symboles (ex : $d(0111, 1010) = 3$).

n-cube cube à n dimensions (2^n sommets) où chaque n-bit string est représenté par un des sommets. **Les sommets adjacents ont une distance de Hamming de 1**.

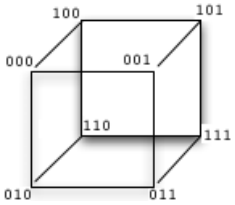


FIGURE 2.2 – n-cube pour $n = 3$

Principe général

Sur n-bits, on peut coder 2^n mots différents, si on rajoute 1 bit, nous avons n mots supplémentaires. Nous pouvons donc grâce à cela, définir n mots qualifié d'erronés, ce qui sera utile pour détecter les erreurs (comme des erreurs de transmission).

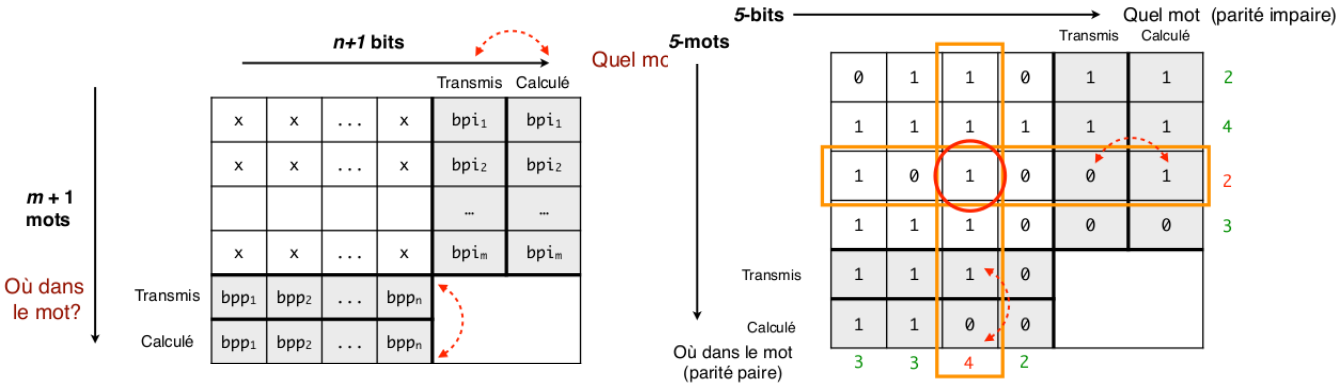
Codes de parité paire ou impaire (*even (odd) parity codes*)

On code les mots de n-bits sur $(n+1)$ bits, ainsi nous avons autant de mots corrects que d'erronés. Le critère de correction est défini par le $(n+1)^{\text{ème}}$ bit.

Il faut compter le nombre de **1** et suivre la convention définie. Les 2 conventions sont :
- Bit de parité paire : 1 si c'est un nombre **impair**
- Bit de parité impair : 1 si c'est un nombre **pair**

Lors du calcul de parité, on peut ou non tenir compte du bit de parité (le $(n+1)^{\text{ème}}$). Ici, on en tiendra pas compte.

Ce qui est intéressant, c'est de rajouter un bit de parité pour chaque poids (pour m-mits de n-bits, nous aurions donc des mots de $n+1$ pour le bit de parité et $m+1$ mot pour le bit de parité de poids). Ainsi, nous pourrions déterminer exactement le bit qui est erroné.



2.2 Algèbre de Boole

2.2.1 Définition

Algèbre de Boole est un quadruplet $\{B, ', \cdot, +\}$ où
- B est un ensemble de 2 valeurs
- $'$ est l'opérateur de complément (parfois symbole $^-$)
- $\cdot$ est l'opérateur **et**
- $+$ est l'opérateur **ou**

En fonction des définitions des opérateurs, on peut définir plusieurs algèbre de Boole. On ne s'intéressera ici que de celle-ci pour 2 valeurs.

2.2.2 Algèbre de Boole à 2 valeurs

Définition

L'algèbre de Boole à 2 valeurs est défini par (introduite par SHANNON) :

$B = \{0, 1\}$
$0 =$ faux
$1 =$ vrai
$+$ ou inclusif (or)
$\cdot$ et (and)

Les opérateurs $+$ et $\cdot$ peuvent être définis par **Tables de Vérités** (TdV) suivantes :

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

Le nombre de variables ($= a$) détermine le nombre de lignes de la TdV (2^a). On remarque entre autre une petite propriété des TdV sur la valeur de chaque variable. Le LSD (ici y), se définit comme : $0, 1, 0, 1, \ldots$ ensuite (ici x) : $00, 11, 00, 11, \ldots$ et encore après $0000, 1111, 0000, \ldots$ Donc chaque variable répète sa valeur 2^n fois où $n =$ poids de chaque variable.

Axiomes

E. V. HUNTIGTON posa 6 axiomes pour le cas de l'algèbre de Boole à 2 valeurs $B = \{0, 1\}$. Ces axiomes sont vérifiables par TdV. Les axiomes sont :

– Axiome 1. B est **fermé** pour $+$ et pour $\cdot$

x	x'
0	1
1	0

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

– Axiome 2. B a un **neutre** pour $+$ (noté 0) et pour $\cdot$ (noté 1)

$$\begin{array}{l} 0+0=0 \\ 1+0=1 \end{array} \quad \text{et} \quad \begin{array}{l} 0\cdot 1=0 \\ 1\cdot 1=1 \end{array} \tag{2.1}$$

– Axiome 3. B est **commutatif** par rapport à $+$ et $\cdot$

$$x + y = y + x \quad \text{et} \quad x \cdot y = y \cdot x \tag{2.2}$$

– Axiome 4. $\cdot$ **distribue** $+$ et $+$ distribue $\cdot$

$$x \cdot (y + z) = x \cdot y + x \cdot z \quad \text{et} \quad x + (y \cdot z) = (x + y) \cdot (x + z) \tag{2.3}$$

Preuve par TdV

– Axiome 5. $\exists$ **complément** de x (noté x', $\bar{x}$ ou not(x))
– Axiome 6. Il y a au moins 2 éléments x, y du B tels que $x \neq y$

Théorèmes

À partir de ces axiomes, nous pouvons définir les théorèmes suivant (à droite, les théorèmes issus du Principe de dualité) :

Normaux

– Théorème 1. Indempotence pour + et ·

$$x + x = x \quad \text{et} \quad x \cdot x = x \tag{2.4}$$

– Théorème 2.

$$x + 1 = 1 \quad \text{et} \quad x \cdot 0 = 0 \tag{2.5}$$

– Théorème 3. Absorption

$$x \cdot (x + y) = x \tag{2.6}$$

– Théorème 4. Involution

$$(x')' = x \tag{2.7}$$

– Théorème 5. Associativité

$$\begin{gathered}(x + y) + z = x + (y + z) \\ \text{et} \\ (x \cdot y) \cdot z = x \cdot (y \cdot z)\end{gathered} \tag{2.8}$$

– Théorème 6. Lois de De Morgan

$$\begin{gathered}(x + y)' = x' \cdot y' \\ \text{et} \\ (x \cdot y)' = x' + y'\end{gathered} \tag{2.9}$$

– Théorème 7. Consensus

$$x \cdot y + x' \cdot z + y \cdot z = x \cdot y + x' \cdot z \tag{2.10}$$

Duaux

– Théorème 1. Indempotence pour + et ·

$$x \cdot x = x \quad \text{et} \quad x + x = x \tag{2.4}$$

– Théorème 2.

$$x \cdot 0 = 0 \quad \text{et} \quad x + 1 = 1 \tag{2.5}$$

– Théorème 3. Absorption

$$x + (x \cdot y) = x \tag{2.6}$$

– Théorème 4. Involution

$$(x')' = x \tag{2.7}$$

– Théorème 5. Associativité

$$\begin{gathered}(x \cdot y) \cdot z = x \cdot (y \cdot z) \\ \text{et} \\ (x + y) + z = x + (y + z)\end{gathered} \tag{2.8}$$

– Théorème 6. Lois de De Morgan

$$\begin{gathered}(x \cdot y)' = x' + y' \\ \text{et} \\ (x + y)' = x' \cdot y'\end{gathered} \tag{2.9}$$

– Théorème 7. Consensus

$$(x + y) \cdot (x' + z) \cdot (y + z) = (x + y) \cdot (x' + z) \tag{2.10}$$

Principe de dualité

Dans l'algèbre de Boole, tout résultat peut se présenter sous 2 formes dites duales.
Soit S un résultat, son dual $S*$ est obtenu en **permutant** :

— les opérateurs + et ·
— les symboles 0 et 1 de B

Si un résultat S est vrai dans l'algèbre de Boole,
il en est de même pour son dual

Nous pouvons bien évidement généraliser ce principe pour une expression Booléenne à n-variables

2.3 Fonctions logiques

Entrées arguments d'une fonction logique

Sorties évaluation de(s) la(es) fonction(s)

Chaque sortie aura une fonction logique qui lui est propre

2.3.1 Représentation des fonctions logiques

Il y a différentes formes pour représenter des fonctions logiques :

- **Fonctions logiques** : représentation compacte pour un faible nombre de variables. Pratique pour une manipulation manuelle ou «crayon et papier».
- **Tables de Vérité** : représentation tabulaire, proche de la représentation de type mémoire numérique (style *Look-up-Table*).
- **Schématique** : représentation graphique ; proche du monde de la réalisation physique ; facile à comprendre (mais peut être très compliqué pour un grand nombre d'arguments).
- **Diagrammes de Venn** : représentation graphique.

Peut importe la forme initiale, la fonction peut être transformée d'un forme à l'autre.

Exemple de fonction logique :

$$F = xz + xy'z + x'yz' \tag{2.11}$$
$$= x \cdot z + x \cdot y' \cdot z' + x' \cdot y \cdot z' \tag{2.12}$$

Une fonction logique peut s'écrire sous forme de Somme de Produit (SdP) ou sous forme de Produit de Somme (PdS)

Table de Vérité vers SdP

x	y	z	F	
0	0	0	1	+
0	0	1	1	+
0	1	0	0	
0	1	1	1	+
1	0	0	0	
1	0	1	0	
1	1	0	1	+
1	1	1	1	+

F vaut 1 lorsque (x ET y ET z) valent quelque chose:

0 ET 0 ET 0

ou

0 ET 0 ET 1

ou

0 ET 1 ET 1

...

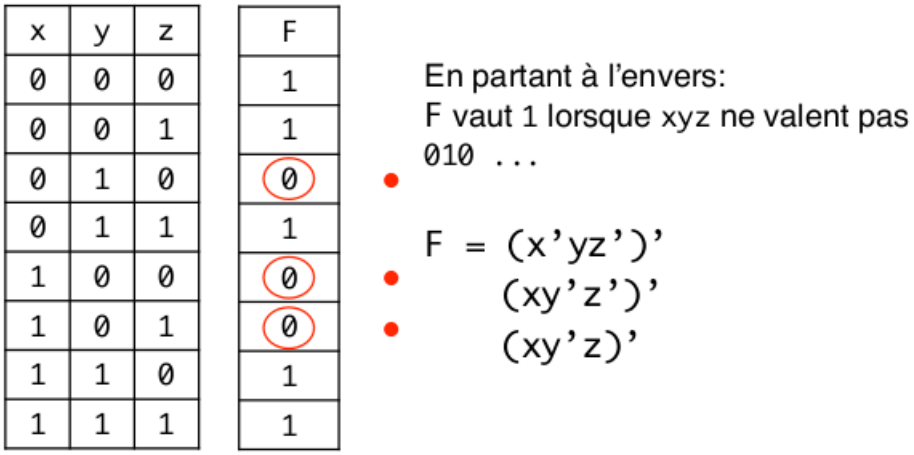
Pour écrire sous forme SdP, il suffit de prendre toutes les lignes où $F = 1$ de réunir les variables d'une même ligne par $\cdot$ et joindre chaque colonne par $+$ en transformant les variables de cette manière :

- a si $a = 1$
- a' si $a = 0$

Ainsi, la SdP est :

$$F = x' \cdot y' \cdot z' + x \cdot' y' \cdot z + x' \cdot y \cdot z + x \cdot y \cdot z' + x \cdot y \cdot z \tag{2.13}$$

Table de Vérité vers PdS



x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Pour écrire sous forme PdS, il suffit de prendre toutes les lignes où $F = 0$ de réunir les variables d'une même ligne par $+$ et joindre chaque colonne par $\cdot$ en transformant les variables de cette manière :

— a si $a = 0$
— a' si $a = 1$

Ainsi, la PdS est :

$$F = (x + y' + z) \cdot (x' + y + z) \cdot (x' + y + z') \tag{2.14}$$

OU BIEN

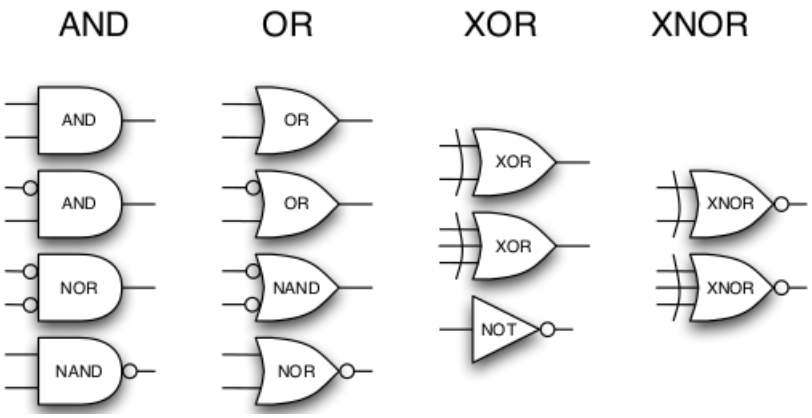
On peut aussi faire la SdP pour $F = 0$ (en gardant bien les conventions pour SdP) et noté F' au lieu de F et ensuite faire De Morgan ($(F')' = F$)

$$F' = x' \cdot y \cdot z' + x \cdot y' \cdot z' + x \cdot y \cdot z \tag{2.15}$$

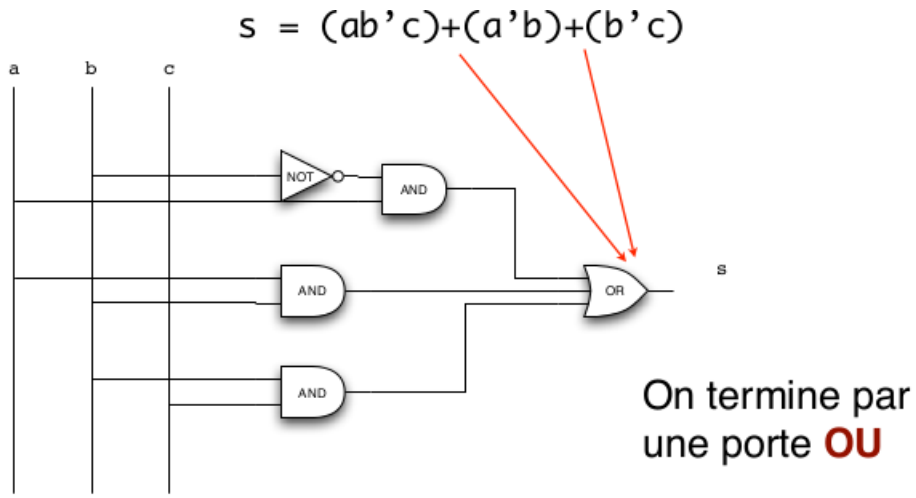
$$F = (F')' = (x' \cdot y \cdot z')' \cdot (x \cdot y' \cdot z')' \cdot (x \cdot y \cdot z)' \tag{2.16}$$
$$= (x + y' + z) \cdot (x' + y + z) \cdot (x' + y + z') \tag{2.17}$$

Schématique (logigrammes)

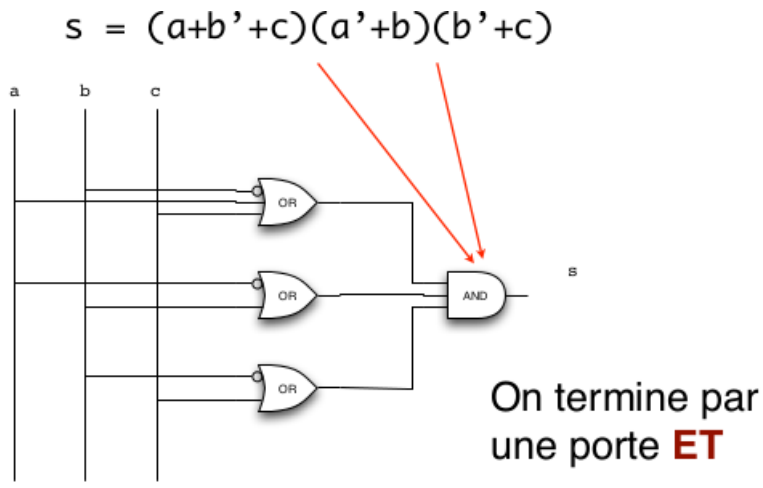
Voici les différents symboles pour les opérateurs :



Dans un circuit en forme de SdP, on finit par une porte OU



Dans un circuit en forme de PdS, on finit par une porte ET



2.4 Réalisation matérielle

Plus tard...

Chapitre 3

Fonctions Booléennes et circuits logiques

3.1 Fonctions Booléennes

3.1.1 Fonction logique

Comme précédemment introduit, on peut définir :

$$F = f(a_0, a_1, \ldots, a_{n-1}) \tag{3.1}$$

$$\text{où} \begin{cases} f = \text{fonction logique (Booléenne)} \\ a_i = \text{arguments de la fonction } (i = 0, \ldots, n-1) \\ F = \text{résultat de l'évaluation} \end{cases}$$

On peut faire l'analogie avec un système composé de circuits logiques, où les a_i seraient les entrées et F_i les différentes sorties calculées par des fonctions logiques (une fonction logique par sortie, indépendante du reste). De ceci sort la notion de *concurrence*, c'est-à-dire que plusieurs évaluations se font en parallèle.

3.2 Modes de représentation

Rappelons quelques représentations pour une fonction logique :

- Tables de Vérité
- Expression logique (plusieurs options possibles, certaines plus «belle» que d'autres)
- Logigramme

3.2.1 Tables de Vérité

Représente la fonction logique de façon unique. Toutes combinaisons des valeurs d'entrée on une valeur de sortie. S'il y a plusieurs sorties, chacune aura sa propre TdV.

Ainsi, pour un nombre n d'arguments, la TdV énumérera toutes les combinaisons possibles d'entrées et de sorties. Donc :

- TdV constitué de 2^n lignes
- 2^{2^n} fonctions logiques différentes (0 ou 1 pour chaque ligne)

À titre informatif :

	xy			
	00	01	10	11
Valeurs de la fonction F	0	0	0	0
	0	0	0	1
	0	0	1	0
	0	0	1	1
	0	1	0	0
	0	1	0	1
	0	1	1	0
	0	1	1	1
	1	0	0	0
	1	0	0	1
	1	0	1	0
	1	0	1	1
	1	1	0	0
	1	1	0	1
	1	1	1	0
	1	1	1	1

Expression	Nom
$F_0=0$	Zéro
$F_1=xy$	ET
$F_2=xy'$	Inhibition
$F_3=x$	Transfert
$F_4=x'y$	Inhibition
$F_5=y$	Transfert
$F_6=xy'+x'y$	XOR
$F_7=x+y$	OU
$F_8=(x+y)'$	NOR
$F_9=xy+x'y'$	XNOR (=)
$F_{10}=y'$	Complément
$F_{11}=x+y'$	Implication
$F_{12}=x'$	Complément
$F_{13}=x'+y$	Implication
$F_{14}=(xy)'$	NET
$F_{15}=1$	Un

3.2.2 Expression algébriques

À partir de la TdV, on peut définir des expressions algébriques sous 2 formes :

1. Forme canonique standard
2. Forme canonique non-standard

2 expressions différentes (une standard, l'autre non-standard) d'une même TdV sont équivalentes.

Forme canonique standard

En procédant comme dans le paragraphe Table de Vérité vers SdP, nous obtenons une somme de produits dont chaque terme est composé de toutes les variables. Ces termes sont appelés **Mintermes**. Pour alléger l'écriture de la somme des Mintermes, nous convertissons en décimal chaque combinaison en définissant le *MSB* et le *LSB*.

LSB

$(\,)_{10}$	x	y	z	F
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

Minterme

$F = x'y'z'$
$+ x'y'z$
$+ x'yz'$
$+ x'yz$

$= \sum(0,1,2,3)$

En procédant comme dans le paragraphe Table de Vérité vers PdS, nous obtenons un produit de sommes dont chaque terme est composé de toutes les variables. Ces termes sont appelés **Maxtermes**. Même méthode pour la notation abrégée (⚠ pas $\sum$ mais $\prod$)

LSB

	x	y	z	F
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

Maxterme

La combinaison des variables à l'entrée, telle que la fonction logique vaut 0.

F = (x'+y+z)(x'+y+z')
(x'+y'+z)(x'+y'+z')
$= \prod(4, 5, 6, 7)$

Ainsi, la forme canonique standard s'exprime sous 2 formes :

1. Somme des Mintermes - Forme disjonctive normale

   Somme logique (OU) des termes produits (ET) pour lesquels la fonction logique a pour valeur «1»

2. Produit des Maxtermes - Forme conjonctive normale

   Produit logique (ET) des termes sommées (OU) pour lesquels la fonction logique a pour valeur «0»

Forme canonique non-standard

Le but est de simplifier l'expression de la fonction logique. Par exemple, la fonction logique précédente pouvait se résumer à $F = x'$.
On peut réduire :

- le nombre de termes dans la somme ou le produit
- chaque terme de la somme ou le produit

Après simplification, la fonction logique se compose de termes appelés **monômes** (ne possédant pas toutes les variables).

Pour repasser sous forme canonique, rien de très compliqué

- Minterme

$$xy' = xy' \cdot 1 = xy' \cdot (z + z') = xy'z + xy'z' \tag{3.2}$$

- Maxterme

$$x + y' = x + y' + 0 = x + y' + zz' = (x + y' + z) \cdot (x + y' + z') \tag{3.3}$$

3.3 Simplification

La simplification des fonctions logiques comportent certains intérêt comme :

- ↘ du nombre de portes logiques dans le circuit
- ↗ de la vitesse de commutation (moins de transistors, moins de portes en séries)

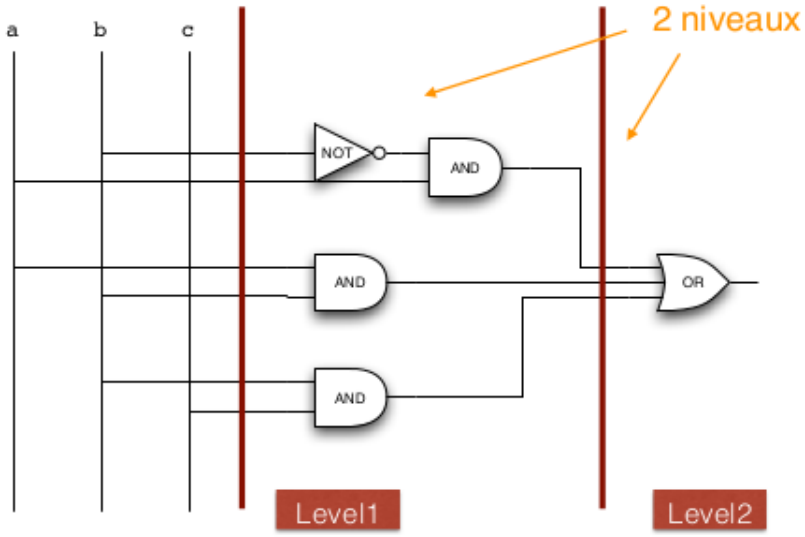
— *searrow* du prix

Il y a néanmoins des problèmes avec la simplification via les axiomes et les théorèmes, nous n'avons aucune certitude que

— l'expression est simplifiable
— l'expression d'arrivée est la plus simple

3.4 Logique à 2 et à plusieurs niveaux

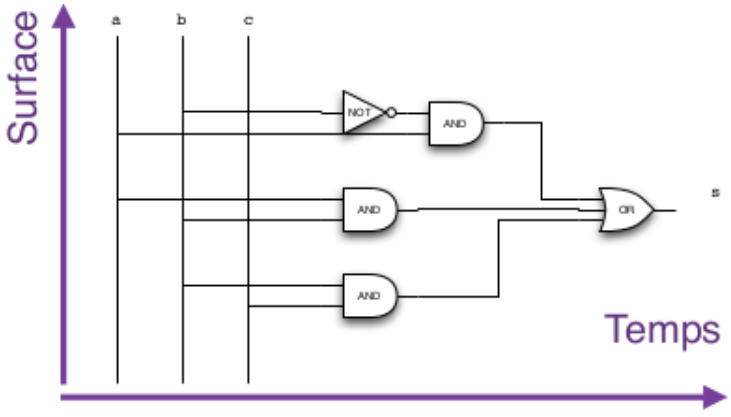
Toute fonction logique dans un logigramme est un plan de portes ET avec une porte OU au bout. De la peut sortir la notion de **niveau**. On définit le délai comme *le délai de 2 plans de portes, quelque soit la fonction*



Mais comment une SdP évolue avec la complexité croissante de la fonction logique (lorsque le nombre d'entrées augmente) ?

En théorie : uniquement la surface.
En pratique :

— chaque porte sera plus grande (capacitance ↗, impact sur le délai et la puissance)
— ↗ longueur des fils (délais)



Il faudra donc jouer sur ces 2 paramètres (surfaces/délais) en sachant que nous ne pourrons pas gagner sur tous les plans. Cette optimisation fera partie de l'*optimisation des circuits logiques* (chapitre important)

3.4.1 Expansion de Boole (Shannon)

Une fonction logique $F(x_i), \quad i = 0, \ldots, n$ peut être représentée comme :

$$F(x_i) = x_1 F_1(1, x_2, \ldots, x_n) + x_1' F_1(0, x_2, \ldots, x_n) \tag{3.4}$$
$$= (x_1 + F_1(x_j))(x_1' + F_1(x_j)) \tag{3.5}$$
$$= x_j F_1(x_j) + x_j' F_1(x_j) \qquad i = 0, \ldots, n \quad j = 1, \ldots, n \tag{3.6}$$

Ainsi, toute fonction logique peut être diviser en 2 fonctions logiques (une pour x, l'autre pour x'). Il y a **factorisation de la fonction logique**.

De cette manière, les 2 fonctions résultantes peuvent être évaluées séparément (en parallèle) et fusionner les résultats. Il en résulte des porte plus simples ($\searrow$ nombre d'arguments), donc un calcul qui explose en surface peut être décomposé en temps ($\searrow$ surface, $\nearrow$ niveaux/délais).

Chapitre 4

Simplification des fonctions logiques : K-Maps

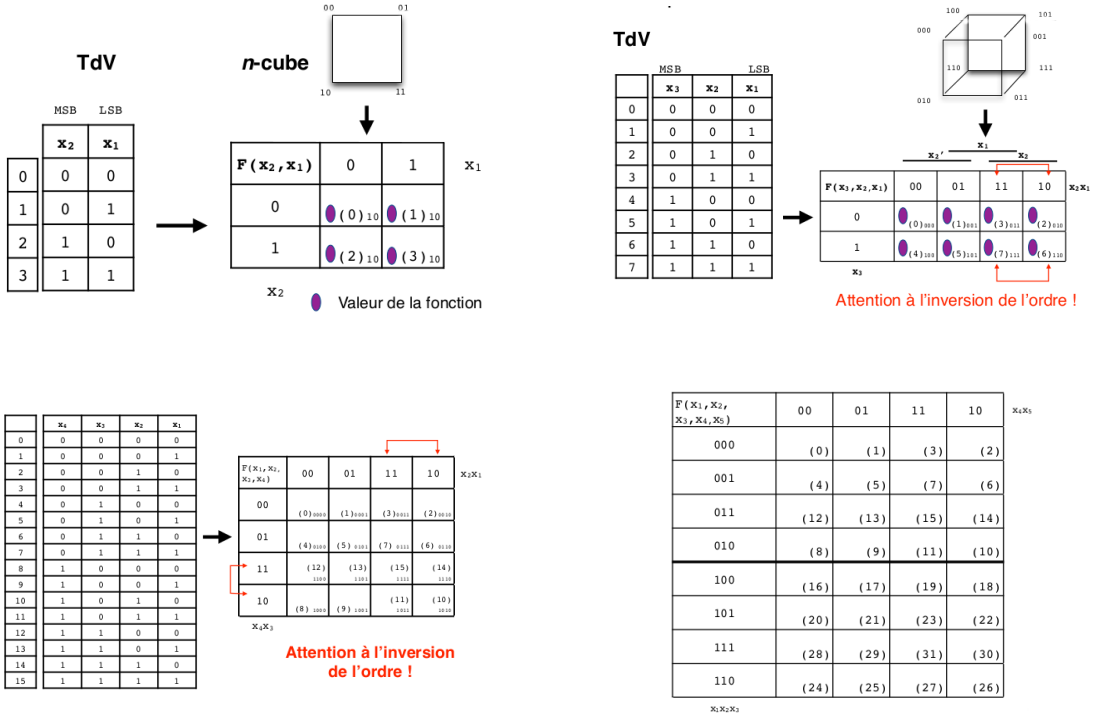
4.1 Méthodes de simplification

Il existe différentes méthodes de simplification dont :

- Via les axiomes et théorèmes déjà démontrées
  - La «qualité» de l'expression obtenue dépend fortement de la capacité de manipulation des axiomes et théorèmes.
- Tables de Karnaugh
  - Méthode graphique rendant la simplification plus aisée. Utilisée pour des problèmes à peu de variables (max. 5).
- Méthode de Quine-McCluskey
  - Méthode systématique permettant de trouver l'expression la plus simple. Facile à automatiser et peut résoudre des problèmes à beaucoup de variables

4.2 Tables de Karnaugh

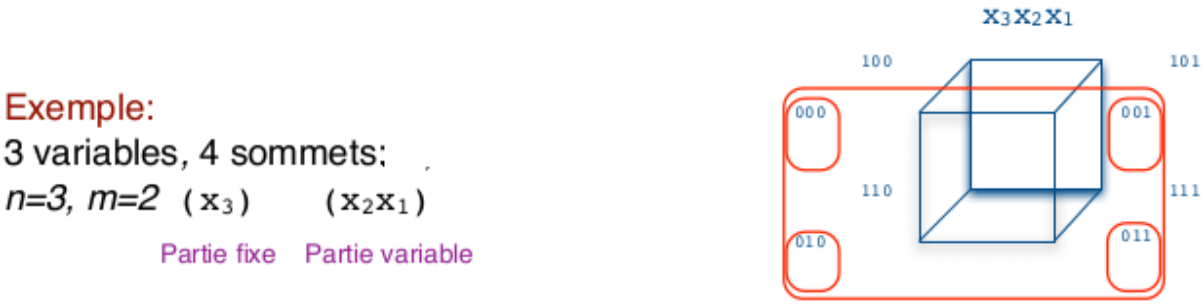
La K-Map est une représentation graphique en 2D des n-cubes.



La barre en gras pour 5 variables sert à représenter la profondeur. Il faut voir la partie du haut et la partie du bas comme superposée.
⚠ Ne pas oublier d'inverser 10 et 11 car sinon, on perd la notion de sommets adjacents (distances de Hamming seront > 1)

4.2.1 Notion de sous-cube dans un n-cube

Une sous-cube dans un n-cube est un ensemble de 2^m sommets ($\in$ une arrête, face,...) pour lesquels $n - m$ variables restantes ont la même valeur.

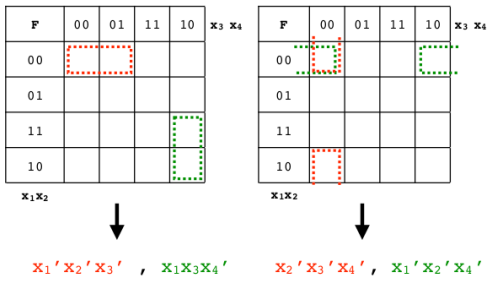


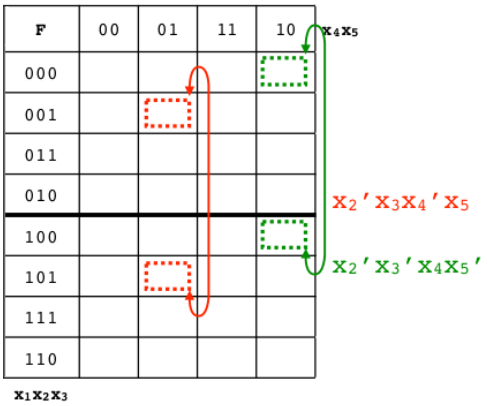
Utilité ?
Prenons un sous-cube comprenant les sommets 000 et 100 correspondant à $x_1'x_2'x_3'$ et $x_1x_2'x_3'$. Si la foction logique vaut 1 à ces sommets alors :

$$F = x_1'x_2'x_3' + x_1x_2'x_3' = (x_1' + x_1)x_2'x_3' = x_2'x_3' \tag{4.1}$$

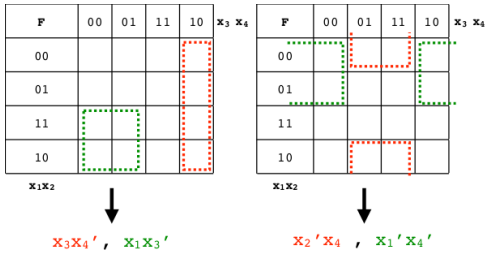
Comme définit plus haut, un sous-cube ne peut être que de taille 2^m.

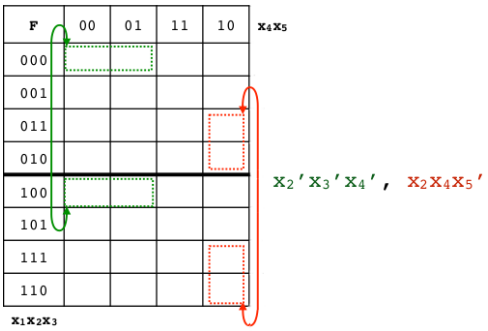
Sous-cube de taille 2



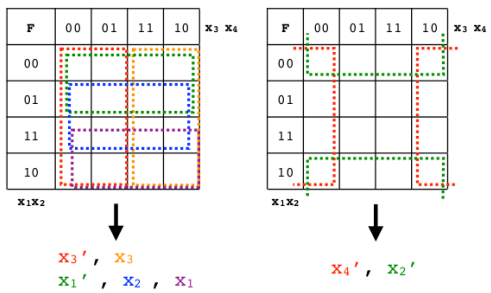


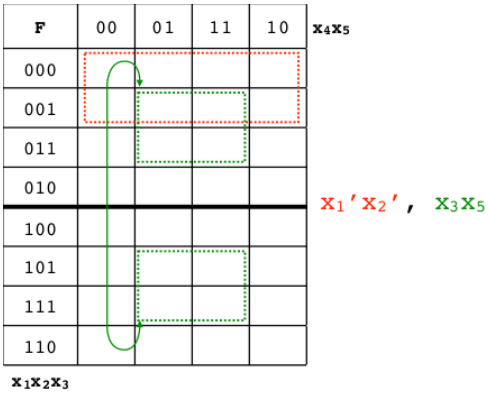
Sous-cube de taille 4



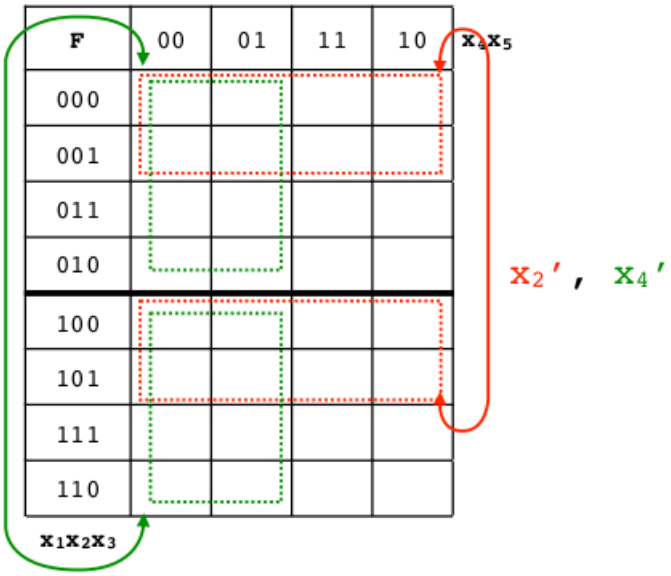


Sous-cube de taille 8





Sous-cube de taille 16



Ainsi, pour une K-Map à n variables :

1. n-cube de taille 1 : n variables (minterme)
2. n-cube de taille 2 : n-1 variables (on a pu simplifier 1 variable)
3. n-cube de taille 4 : n-2 variables (on a pu simplifier 2 variables)
4. n-cube de taille 8 : n-3 variables (on a pu simplifier 3 variables) ⋮
5. n-cube de taille 2^n : constante

4.3 Simplification des fonctions logiques avec les K-Maps

Consiste à trouver le plus petit nombre des plus grands sous-cubes permettant de couvrir tous les «1» dans une K-Map
On remarque 2 choses importantes :

— *[...] le plus petit nombre des sous-cube [...]* : Pour une SdP, influencer la taille de la porte OU et le nombre de portes ET
— *[...] les plus grands sous-cubes [...]* : influence la taille de chaque porte ET

Implicant premier est défini comme un sous-cube qui n'est défini au sein d'aucun autre sous-cube

Implicant premier essentiel est un implicant premier seul à couver au moins un «1» de la fonction logique F

Fonction logique simplifiée est représentée comme une somme :

— de tous les implicants premiers essentiels
— des certains implicants premiers de façon à couvrir tous les «1» d'une K-Map

4.3.1 Méthode de simplification (algorithme)

1. Rechercher tous les implicants premiers de la fonction logique écrite sous forme d'une K-Map
2. Parmi tous les implicants premiers de la fonction, isoler les implicants premiers essentiels
3. Couvrir le reste des «1» avec le moins d'implicants premiers possible

4.3.2 Remarque

Superposition des sous-cubes

$F(x_1,x_2,x_3)$	00	01	11	10	x_2x_3
0	0	1	1	0	
1	0	0	1	0	
x_1					

Doit-on superposer le vert et le rouge ou est-il préférable de faire un sous-cube de taille 1 ?
Nous pouvons $(x + x = x)$ et nous devons superposer les sous-cubes car :

$$\begin{array}{cc} \text{avec} & \text{sans} \\ x_1'x_3 + x_2x_3 & x_1'x_3 + x_1x_2x_3 \end{array} \tag{4.2}$$

Superposer simplifie donc l'expression

$F(x_1,x_2,x_3)$	00	01	11	10	x_2x_3
0	0	1	1	0	
1	0	0	1	1	
x_1					

Doit-on inclure le sous-cube bleu ?
Non car le bleu ne couvre aucun «1» qui ne serait pas encore couvert.

Remarque : Nous verrons qu'il existe des cas où il faut les inclure.

Fautes graves

Entraînera un 0 pour l'exercice :

- Ne pas respecter l'adjacence dans la création de la K-Map (lignes et colonnes ne peuvent différer que 1 ! bit)
- Faire des groupements des mintermes adjacents (ex : regrouper les colonnes 1 et 3...)
- Faire des groupements de mintermes par 3, 5, 6, 7,... (i.e. construire des sous-cubes qui ne sont pas 2^n)

4.4 Fonctions non-complètement spécifiées

Pour certaines combinaisons de valeur des variables, on ne se préoccupe pas du résultat de la fonction :

— Soit parce qu'on se fout de la sortie de ces combinaisons → **don't care**

— Soit parce que ces combinaisons n'arriveront jamais → **don't happen**

Que ce soit l'un ou l'autre, ils seront marqués par un - . On pourra choisir la valeur (0 ou 1) pour simplifier au mieux l'expression finale.

Il faut néanmoins souligner que contrairement au *don't happen* qui ne produira pas de sortie vu que ses combinaisons n'arriveront pas, le *don't care* produira une sortie.

F	00	01	11	10	x_2x_3
0	0	0	0	0	
1	0	1	1	0	
x_1					

$F = x_1x_3$

F	00	01	11	10	x_3x_4
00	1	0	0	-	
01	-	0	0	1	
11	0	0	0	1	
10	1	0	0	-	
x_1x_2					

Liste des implicants :

$x_2'x_4'$

x_3x_4'

$x_1'x_4'$

Fonction optimisée :

$F = x_2'x_4' + x_3x_4'$

Chapitre 5

Synthèse de circuits et Quine Mc.Cluskey

5.1 Synthèse des circuits à partir de spécification verbales

Spécification formelle Description complète, précise et non-ambiguë de toutes les propriétés d'un système (TdV, K-Map etc.).

Combinatoire Les sorties du système ne dépendent que des entrées.

Si le problème **est combinatoire**, alors on peut appliquer tout ce qui a été vu jusqu'à présent.
N.B. : commencer par déterminer le nombre d'entrées et de sorties

5.2 Additionneurs re-visités

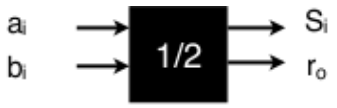
On souhaite réaliser un circuit additionneur de deux mots A et B codés sur 4 bits (peut généraliser pour n bits).
deux solutions s'offrent à nous :

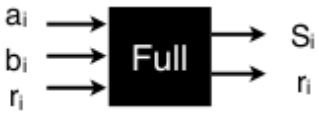
- Mimer le principe d'addition bit à bit (séquence de calcul dans le temps)
- Anticiper le report pour un certain nombre de bits (considérer le problème comme combinatoire)

5.2.1 Addition bit à bit

L'addition est faite grâce à :

- un 1/2 additionneur (deux opérantes d'un bit, deux bits d'entrées)
- $(n-1)\times$ additionneur complet (deux opérantes d'un bit + un bit de report d'une étage précédent, trois bits d'entrées)





S_i	0	1	b_i
0	0	1	
1	1	0	
a_i			

r_0	0	1	b_i
0	0	0	
1	0	1	
a_i			

S	00	01	11	10	a_ib_i
0	0	1	0	1	
1	1	0	1	0	
r_i					

r	00	01	11	10	a_ib_i
0	0	0	1	0	
1	0	1	1	1	
r_i					

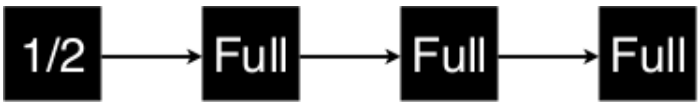
$$S_i = a_i'b_i + a_ib_i' = a_i \oplus b_i$$

$$r_0 = a_ib_i$$

$$S = a_i \oplus b_i \oplus r_i$$

$$r = a_ib_i + r_ib_i + r_ia_i$$

Ainsi, pour un additionneur complet sur 4 bits :



Le problème de ce système est la propagation du report entre les étages : le résultat final sera obtenu après la propagation du report à travers tous les étages de l'additionneur (n-délais).

5.2.2 Additionneur *Carry Look Ahead* (CLA)

Le principe est de faire de faire un additionneur complet en un seul circuit combinatoire, permettant d'éviter le problème de délais du à la propagation du report en l'anticipant(*Carry Look Ahead*).
Le problème étant combinatoire avec 4 entrées (4 bits) et 3 sorties ($(11)_2 + (11)_2 = (110)_2$), nous pouvons faire nos K-Maps :

c_2	00	01	11	10	b_1b_0
00	0	0	0	0	
01	0	0	1	0	
11	0	1	1	1	
10	0	0	1	1	
a_1a_0					

c_1	00	01	11	10	b_1b_0
00	0	0	1	1	
01	0	1	0	1	
11	1	0	1	0	
10	1	1	0	0	
a_1a_0					

c_0	00	01	11	10	b_1b_0
00	0	1	1	0	
01	1	0	0	1	
11	1	0	0	1	
10	0	1	1	0	
a_1a_0					

$$c_2 = a_1b_1 + a_1a_0b_0 + a_0b_1b_0$$

$$c_1 = a_1b_1'b_0' + a_1a_0'b_1' + a_1'a_0'b_1 + a_1'b_1b_0' + a_1'a_0b_1'b_0 + a_1a_0b_1b_0$$

$$c_0 = a_0b_0' + a_0'b_0$$

Pour un plus grand nombre de bits (32, 64 etc.), on préfère mettre en série car si $\nearrow$ entrées/-sorties $\rightarrow \nearrow$ taille du circuit (mémoire) $\Rightarrow$ explosion combinatoire.
Il y aura un délai de report du à la mise en série, mais moindre que le système vu en sous-section 5.2.1

5.3 Encodeurs de priorité

Un mot D est codé sur n bits. On veut connaître l'index du bit de poids le plus fort ayant un «1» et signaler la présence d'un 1 par un signal ANY.

Pour un mot à 4 bits :
- 4 entrées
- 3 sorties :
  - 2 bits pour coder l'index (index max = 4)
  - 1 bit pour le ANY

d_3	d_2	d_1	d_0	a_1	a_0	ANY
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	–	0	1	1
0	1	–	–	1	0	1
1	–	–	–	1	1	1

ANY	00	01	11	10	d_1d_0
00	0	1	1	1	
01	1	1	1	1	
11	1	1	1	1	
10	1	1	1	1	

d_3d_2

$a_1 = d_3 + d_2 + d_1 + d_0$

a_1	00	01	11	10	d_1d_0
00	0	0	0	0	
01	1	1	1	1	
11	1	1	1	1	
10	1	1	1	1	

d_3d_2

$a_1 = d_3 + d_2$

a_0	00	01	11	10	d_1d_0
00	0	0	1	1	
01	0	0	0	0	
11	1	1	1	1	
10	1	1	1	1	

d_3d_2

$a_0 = d_3 + d_2{}'d_1$

5.4 Simplification des fonctions : Méthode Quine-Mc.Cluskey

La méthode de Quine-Mc.Cluskey est :
- méthode systématique, analysant toutes les possibilités de regroupement.
- utilisée pour un grand nombre de variables
- garantit la meilleure solution
- facilement programmable

Elle se base sur 2 étapes :

1. Recherche des implicants premiers par la **méthode des tris successifs** (phase d'**analyse**)
2. Couverture de la fonction par un choix des implicants premiers (phase de **synthèse**)

5.4.1 Étape 1 : Analyse

Elle consiste en 9 étapes (oui, dans le cours c'est 11) :

1. Pour une fonction f de n variables, on construit m groupements de **tous les mintermes** de la fonction
2. Dans chaque groupement des mintermes (noté G_i)
   - i variables qui valent 1
   - $(n-i)$ qui valent 0

$f(a,b,c)=\sum(0,1,3,4,5,7)$

	a	b	c	f	
0	0	0	0	1	x
1	0	0	1	1	x
2	0	1	0	0	
3	0	1	1	1	x
4	1	0	0	1	x
5	1	0	1	1	x
6	1	1	0	0	
7	1	1	1	1	x

→

Groupements G_i

	a	b	c
G_0	0	0	0
G_1	0	0	1
	1	0	0
G_2	0	1	1
	1	0	1
G_3	1	1	1

3. On compare chaque minterme de G_i avec **TOUS** les mintermes appartenant à G_{i+1} en termes de la distance de Hamming ($\forall i$)

4. Si la distance de Hamming entre 2 mintermes = 1, on les marque par un x et on note **l'appariement** entre les 2 :

$$\left.\begin{array}{l}000\\010\end{array}\right\} \Rightarrow \text{0-0} \tag{5.1}$$

   impliquant la possibilité de faire un n-cube de plus grande taille (taille 2 à ce stade). Sinon, il s'agit d'un *impliquant premier* (n-cube de taille 1 à ce stade) qu'on notera IP_k dans l'ordre d'apparition

5. Tous les appariements de G_i et G_{i+1} sont écrits dans un nouveau groupement, noté G'_i

6. Après une première passe, il y a donc au moins $m-1$ groupements G'_i

7. L'ensemble G'_i constitue donc l'ensemble de tous les n-cube de taille 2

8. On répète l'opération avec les nouveaux groupements jusqu'à ce qu'il ne soit plus possible d'apparier

9. On dresse la liste de tous les implicants premiers (IPs) trouvés.

Remarques

1. On peut avoir $m = n$

2. Marquer **clairement** la séparation entre les G_i

3. Les *don't care* sont également pris en compte pour la distance de Hamming

-000	Distance de Hamming de 1	-000
-100	on **peut** apparier	--00
-000	Distance de Hamming de 2	
01-0	on **ne peut pas** apparier	

4. S'il y a présence d'indifférent (*don't care*, *don't happen*) au début du problème :
   — on fait l'hypothèse qu'ils valent 1 (et donc les rajoutes dans les groupement G_i sans aucune distinction)
   — Les implicants premiers non-utile (composés que d'indifférents) seront éliminés lors de la 2ème phase (couverture)

Exemple

	a	b	c
G_0	0	0	0
G_1	0	0	1
	1	0	0
G_2	0	1	1
	1	0	1
G_3	1	1	1

G_0, G_1 : 000, 001, 000, 100

G_2, G_3 : 011, 111, 101, 111

G_1, G_2 : 001, 011, 001, 101, ~~100~~, ~~011~~, 100, 101

Distance de Hamming =2

G_0' : 00- (0,1), -00 (0,4)

G_1' : 0-1 (1,3), -01 (1,5), 10- (4,5)

G_2' : -11 (3,7), 1-1 (5,7)

G_0' : 00- (0,1) X, -00 (0,4) X

G_1' : 0-1 (1,3) X, -01 (1,5) X, 10- (4,5) X

G_2' : -11 (3,7) X, 1-1 (5,7) X

→

G_0'' : -0- (0,1,4,5) IP1, -0- (0,4,1,5)

G_1'' : --1 (1,3,5,7) IP2, --1 (1,5,3,7)

Chapitre 6

Quine Mc.Cluskey et Circuits séquentiels

Chapitre 7

Synthèse des systèmes séquentiels synchrones