Chapitre 1

Modèles de conception

On décrit dans ce chapitre les modèles théoriques élémentaires sur lesquels repose le travail de conception de circuits numériques. Le lecteur familier avec l'algèbre booléenne pourra sauter la section 1.1.

1.1 Variables et fonctions booléennes

Une **variable booléenne** est une variable prenant ses valeurs dans un ensemble $\mathcal{B}$ à deux éléments : $\{0, 1\}$, $\{\mathsf{true}, \mathsf{false}\}$, *etc.*.

L'ensemble $\mathcal{B}$ est classiquement muni de trois opérations :

— l'addition ou *ou logique* : $x + y = \begin{cases} 0 & \text{si } x = 0 \text{ et } y = 0 \\ 1 & \text{sinon} \end{cases}$

— la multiplication ou *et logique* : $x.y = \begin{cases} 1 & \text{si } x = 1 \text{ et } y = 1 \\ 0 & \text{sinon} \end{cases}$

— la complémentation. $\overline{x} = \begin{cases} 1 & \text{si } x = 0 \\ 0 & \text{si } x = 1 \end{cases}$

La structure $(\mathcal{B}, +, .,)$ définit ce que l'on appelle classiquement une **algèbre de Boole**[1]. Les principales propriétés de cette algèbre sont rappelées dans l'encadré 1.1.

Une **fonction booléenne** – on dira aussi **fonction logique** – est une fonction opérant sur des valeurs booléennes. Formellement :

$$f : \mathcal{B}^m \to \mathcal{B}^n$$

Les fonctions logiques peuvent être représentées par une **table de vérité**. Une telle table liste l'ensemble des combinaisons possibles pour les entrées de la fonction et, pour chaque combinaison, donne la valeur correspondante des sorties. La

1. Du nom de G. Boole, mathématicien anglais du XVII^{e} siècle qui a formalisé la théorie et l'a utilisée dans le cadre de la logique mathématique.

figure 1.1 par exemple donne la table de vérité de la fonction $f(a, b, c) = \overline{a}.b + c$. Les huit combinaisons possibles des trois variables d'entrée a, b et c sont ici listées dans l'ordre dit du *binaire naturel* (cf. ci-dessous). La troisième ligne de la table, par exemple, indique que $f(0, 0, 1) = 1$.

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

FIGURE 1.1 – Table de vérité de la fonction $f(a, b, c) = \overline{a}.b + c$

Les fonctions logiques permettent de représenter des fonctions *numériques* dès lors que l'on dispose d'une représentation des nombres sous la forme de variables booléennes. La représentation la plus simple est celle dite du **binaire naturel**. La représentation d'un entier positif est donnée dans ce cas par l'écriture de cet entier en base 2. Par exemple, le nombre `13` s'écrira `1101` avec cette représentation parce que

$$13 = 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0$$

Avec cette représentation, un ensemble de n variables booléennes – on parle plutôt, dans ce contexte de *bit* – permet de coder tous les entiers compris entre 0 et $2^n - 1$. Avec 8 bits par exemple, on peut représenter tous les entiers entre 0 (= 00000000) et 255 (= 11111111). Dans ce contexte, par exemple, la fonction f décrite par la table de vérité de la figure 1.1 peut s'interpréter comme la fonction indiquant si son argument, compris entre 0 et 7, et codé en binaire sur trois bits, est un nombre premier [2].

L'encadré 1.2 décrit comment obtenir la représentation binaire d'un entier donné.

Des variantes – que l'on détaillera pas ici, en renvoyant le lecteur aux ouvrages cités en bibliographie –, permettent par ailleurs de représenter

— des entiers relatifs (en utilisant la représentation dite en *complément à 2*),
— des décimaux avec virgule fixe (on affecte alors une partie des bits de la représentation à la partie entière et l'autre à la partie décimale),

2. En effet, on a ici $f(1) = f(2) = f(3) = f(5) = f(7) = 1$ et $f(0) = f(4) = f(6) = 0$.

— des décimaux avec virgule flottante (la norme IEEE-754 par exemple définit le codage de tels nombres sous la forme d'une *mantisse*, d'un *exposant* et d'un *signe*).

Les variables booléennes jouent un rôle fondamental dans le domaine des circuits numériques car elles peuvent être représentées physiquement par des *signaux électriques* via une correspondance très simple. Typiquement, pour un tel signal, une tension haute (proche de celle de l'alimentation), représentera un « 1 », et une tension basse (proche de la masse), représentera un « 0 » [3]. Une fonction logique peut alors être réalisée (« implémentée ») sous la forme d'un circuit dont les entrées (resp. sorties) encodent, sous la forme de tensions, les valeurs des arguments (resp. résultats) de la fonction. La manière de relier le *temps* physique, associé à la réalisation, à la définition de la fonction booléenne amène alors à distinguer deux grandes classes de fonctions : les fonctions combinatoires et les fonctions séquentielles.

Encadré 1.1 : Algèbre booléenne

On rappelle ici très brièvement les principales règles et théorèmes de l'algèbre booléenne.

— commutativité : $\forall\, x, y \in \mathcal{B}, \quad x + y = y + x \quad \text{et} \quad x.y = y.x$
— associativité : $\forall\, x, y, z \in \mathcal{B}, \quad x + (y + z) = (x + y) + z \quad \text{et} \quad x.(y.z) = (x.y).z$
— éléments neutres : $\forall\, x, \quad x + 0 = x \quad \text{et} \quad x.1 = x$
— distributivité : $\forall\, x, y, z, \quad x.(y + z) = x.y + x.z$
— éléments absorbants : $\forall\, x, \quad x + 1 = 1 \quad \text{et} \quad x.0 = 0$
— $\forall\, x, \quad x + \overline{x} = 1 \quad \text{et} \quad x.\overline{x} = 0$
— $\forall\, x, \quad x + x = x \quad \text{et} \quad x.x = x$
— $\forall\, x, \quad \overline{\overline{x}} = x$
— $\forall\, x, y, \quad \overline{x + y} = \overline{x}.\overline{y}$
— $\forall\, x, y, \quad \overline{x.y} = \overline{x} + \overline{y}$

On appelle par ailleurs *ou exclusif* l'opération $\oplus$ définie ainsi : $x \oplus y = x.\overline{y} + \overline{x}.y$.

Cette opération vérifie $x \oplus y = \begin{cases} 1 & \text{si } x = 0 \text{ et } y = 1 \text{ ou } x = 1 \text{ et } y = 0 \\ 0 & \text{si } x = 0 \text{ et } y = 0 \text{ ou } x = 1 \text{ et } y = 1 \end{cases}$

Encadré 1.2 : Représentation binaire d'un entier

Pour obtenir la représentation binaire d'un nombre entier (positif ou nul ici), il faut décomposer le nombre en puissances successives de 2. Pour de « petits » nombres, cela peut être fait immédiatement. Par exemple $13 = 12 + 1 = 8 + 4 + 1$, d'où $13_{10} = 1101_2$. Sinon, il faut procéder par divisions successives, en suivant l'algorithme suivant :

3. Les notions de tension « haute » et « basse » sont définies précisément en fonction de *seuils*, qui dépendent de la technologie utilisée.

Entrée : Un entier $n \geq 0$
Sortie : La représentation binaire $b_{m-1} \dots b_0$ de cet entier sur m bits
$i \leftarrow 0$;
while $i < m$ **do**
  $b_i \leftarrow n \bmod 2$;
  $n \leftarrow n/2$;
  $i \leftarrow i + 1$;
end

1.2 Fonctions combinatoires

Une fonction logique est dite *combinatoire* si ses sorties à l'instant t ne dépendent que de ses entrées à cet instant (figure 1.2) :

$$S(t) = F(E(t)) \qquad \forall t \geq 0$$

En pratique, toutefois, la *réalisation* d'une fonction combinatoire fera qu'il existera toujours un délai entre la modification de la valeur des entrées [4] et la mise à jour des sorties correspondantes. Ce délai, appelé *temps de de propagation* est lié aux phénomènes électroniques mis en jeu pour réaliser la fonction (effets capacitifs, vitesse finie du signal électrique sur les fils, ...). On le modélisera ici [5] sous la forme d'une durée τ, de telle sorte que la caractérisation réelle d'une fonction combinatoire se fera sous la forme

$$S(t) = F(E(t - \tau))$$

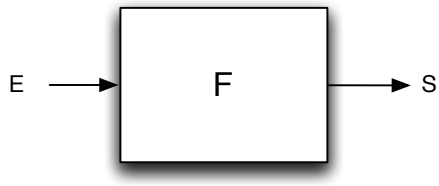


FIGURE 1.2 – Fonction combinatoire

Les fonctions combinatoires sont les plus simples à spécifier et à réaliser. Elles jouent néanmoins un rôle crucial dans la réalisation de systèmes plus complexes [6].

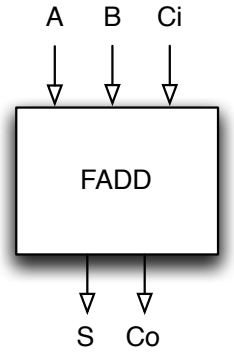
4. Sur la figure 1.2, E (resp. S) désigne en *ensemble* d'entrées (resp. de sorties).
5. Une analyse plus fine sera faite au chapitre 2.
6. Ne serait-ce que parce que, comme on le verra plus loin, tout système *séquentiel* peut se décomposer sous la forme d'un élément de mémorisation et de deux fonctions combinatoires.

La spécification d'une fonction combinatoire, autrement dit la donnée de la relation qu'elle établit entre ses entrées et ses sorties, se fait typiquement soit sous la forme d'**équations logiques**, soit sous la forme d'une **table de vérité**. On peut d'ailleurs passer de la première forme à la seconde par simple tabulation (en listant l'ensemble des combinaisons d'entrées possibles et en calculant les sorties pour chacune des ces combinaisons), et de la seconde à la première par les techniques classiques de synthèse et de simplification d'équations logiques (formes canoniques, tableaux de Karnaugh, *etc.*).

Ce document n'ayant pas pour objectif de faire un exposé théorique et pratique de ces techniques de synthèses et de simplification [7], on se contente ici de présenter la démarche de réalisation sur un exemple.

1.2.1 Exemple

On considère la fonction combinatoire décrite sur la figure 1.3. Cette fonction est un additionneur binaire complet (en anglais, *full adder*). Etant donné deux bits A et B et une retenue entrante Ci, elle produit la somme S et la retenue sortante Co.



A	B	Ci	S	Co
0	0	0	0	0
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 1.3 – Délimitation et table de vérité de la fonction FADD

Il est facile – en utilisant par exemple la méthode rappelée dans l'encadré 1.3 – de tirer de la table de la vérité donnée sur la figure 1.3 les équations de la fonction FADD :

$$\begin{aligned} S &= A \oplus B \oplus Ci \\ Co &= A.B + A.Ci + B.Ci \end{aligned}$$

7. Exposé dont l'importance a fortement diminué dans la mesure où ces opérations sont désormais largement prises en charges par les outils de synthèse logiques des compilateurs VHDL.

Encadré 1.3 : Synthèse d'équations logiques à partir d'une table de vérité

On rappelle ici comment procéder sur l'exemple de la fonction `FADD`.

Etape 1. On écrit les équations des sorties sous *forme canonique* à partir de la table de vérité. Pour cela, on repère les combinaisons des variables d'entrées pour lesquelles chaque sortie est à 1 et on somme les *termes produits*[a] correspondants. Dans chaque *p-terme*, une variable apparaît en facteur si elle vaut 1 dans la combinaison correspondante ou sous forme complémentée si elle vaut 0 dans cette combinaison. Cela donne ici :

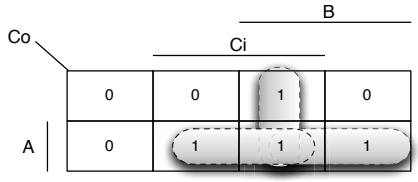
— $S = \overline{A}.\overline{B}.Ci + \overline{A}.B.\overline{Ci} + A.\overline{B}.\overline{Ci} + A.B.Ci$

— $Co = \overline{A}.B.Ci + A.\overline{B}.Ci + A.B.\overline{Ci} + A.B.Ci$

Etape 2. On simplifie ces équations. Pour S, cela peut se faire algébriquement :

$$\begin{aligned} S &= \overline{A}.\overline{B}.Ci + \overline{A}.B.\overline{Ci} + A.\overline{B}.\overline{Ci} + A.B.Ci \\ &= \overline{A}.(\overline{B}.Ci + B.\overline{Ci}) + A.(\overline{B}.\overline{Ci} + B.Ci) \\ &= \overline{A}.(B \oplus Ci) + A.(\overline{B \oplus Ci}) \\ &= A \oplus (B \oplus Ci) \\ &= A \oplus B \oplus Ci \end{aligned}$$

Pour Co, on peut utiliser un tableau de Karnaugh :



$Co = A.B. + A.Ci + B.Ci$

a. Dans le jargon, *p-termes*.

1.3 Fonctions définies structurellement

En pratique, l'approche décrite ci-dessus n'est pas toujours applicable directement. Considérons par exemple la fonction spécifiée sur la figure 1.4. Il s'agit d'un additionneur BCD à quatre chiffres (le principe du codage BCD est rappelé dans l'encadré 1.4). Il prend en entrée deux nombres codés en BCD de quatre chiffres et produit les cinq chiffres codant, en BCD toujours, la somme de ces deux nombres. Chaque chiffre BCD est codé sur quatre bits.

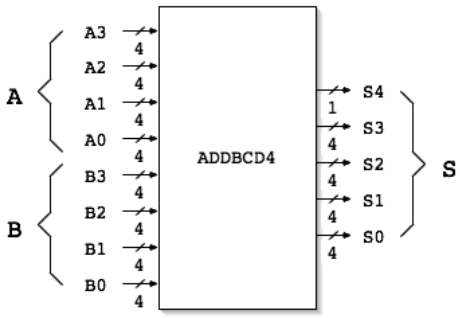
Par exemple, si

— $A_3 = 0010$, $A_2 = 0011$, $A_1 = 1000$, $A_0 = 0010$ (soit $A = 2382$ en décimal), et

— $B_3 = 1001$, $B_2 = 0100$, $B_1 = 0110$, $B_0 = 0011$ (soit $B = 9463$ en décimal), alors

— $S_4 = 1$, $S_3 = 0001$, $S_2 = 1000$, $S_1 = 0100$, $S_0 = 0101$ (soit $S = 11845$ en décimal)

On notera au passage que l'addition ainsi définie n'est *pas* une addition binaire sur 16 bits.




```
                2 3 8 2 (BCD)
              + 9 4 6 3 (BCD)
              ----------
               1 1 8 4 5 (BCD)

    0010 0011 1000 0010 (BCD)
+   1001 0100 0110 0011 (BCD)
-----------------------
 1 0001 1000 0100 0101 (BCD)
 0 1011 0111 1110 0101 (BIN)
```

FIGURE 1.4 – Additionneur BCD 4 chiffres

En théorie, la fonction ADDBCD4 étant purement combinatoire, on peut la décrire complètement avec une table de vérité. En pratique, ce n'est évidemment pas envisageable ici : une telle table compterait en effet $2^{32} = 65536$ lignes (car il y $2 \times 4 \times 4 = 32$ entrées binaires) !

La solution passe par une *décomposition* de cette fonction en fonctions plus simples[8]. Ici, il est aisé de voir (en se remémorant la technique opératoire apprise à l'école primaire) que la fonction ADDBCD4 peut se réaliser en « cascadant » quatre fonctions opérant chacune sur un chiffre, comme indiqué sur la figure 1.5.

La fonction ADDBCD1 est délimitée et spécifiée sur les figures 1.6. L'entrée CI correspond à la retenue issue de l'addition des chiffres de rang immédiatement inférieur (égale à 0 pour les chiffres de poids faible, comme indiqué sur la figure 1.5). La somme, sous la forme d'un chiffre BCD codé sur quatre bits, est disponible sur la sortie S. La sortie CO donne la retenue sortante, pour l'addition des chiffres de rang immédiatement supérieur (la dernière retenue donnant directement S4, comme indiqué sur la figure 1.5).

8. Selon l'adage bien connu : « diviser pour régner ».

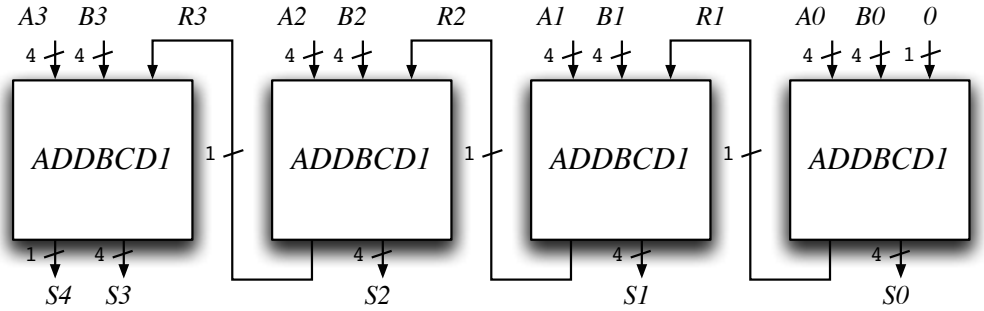
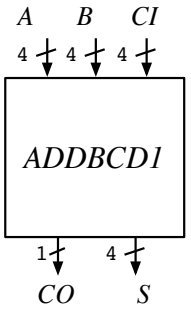


FIGURE 1.5 – Décomposition de l'additionneur BCD 4 chiffres




```
 A = 1000     (BCD)
 B = 0110     (BCD)
CI =    0
---------
 S = 0100     (BCD)
CO =    1
```

FIGURE 1.6 – Additionneur BCD 1 chiffre

La fonction ADDBCD1, bien que nettement plus simple, reste néanmoins difficile à spécifier sous la forme d'une table de vérité. En effet, avec 9 entrées binaires, cette table comprend encore $2^9 = 512$ lignes !

On pourrait être tenté de réitérer la technique de décomposition déjà utilisée, mais on se heurte ici au fait que la fonction ADDBCD1 n'étant *pas* une addition binaire, on ne peut la décrire comme une cascade d'additionneurs 1 bit.

La solution consiste à remarquer que

— si la somme (binaire) $T = A + B + CI$ est inférieure ou égale à 9, alors la sortie S de la fonction ADDBCD1 est T et la retenue CO est nulle,
— sinon, la sortie S vaut $T - 10$ et la sortie CO vaut 1.

Le schéma correspondant est donné sur la figure 1.7. L'additionneur calcule la somme binaire $T = A + B + CI$. Le comparateur compare cette somme à la valeur 9. La sortie de ce comparateur donne d'une part directement la sortie CO et sert d'autre part d'entrée de contrôle au multiplexeur, qui aiguille sur la sortie S soit la valeur $T - 10$ (si $CO = 1$), soit la valeur T (si $CO = 0$).

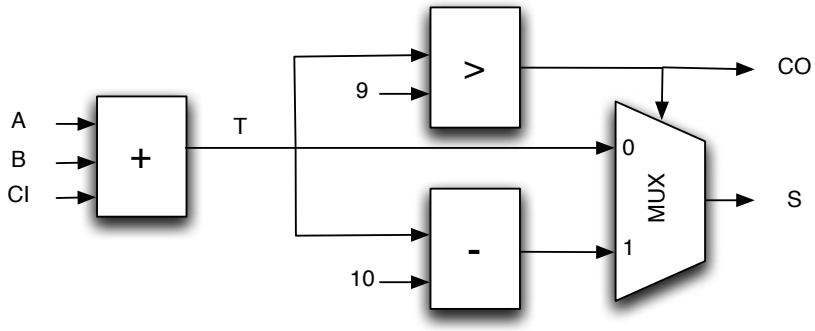


FIGURE 1.7 – Réalisation de la fonction ADDBCD1

Encadré 1.4 : Codage BCD

Le codage BCD (*Binary Coded Decimal*, en français : Décimal Codé Binaire) est un système de codage attribuant à chaque chiffre décimal de 0 à 9 un code sur quatre bits de la manière suivante :

chiffre décimal	code binaire
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Avec ce codage, le nombre 358, par exemple, est représenté par le groupe de 12 bits suivant : 0011 0101 1000. L'intérêt de cette représentation est la conversion immédiate de ou vers un nombre décimal (la conversion décimal $\leftrightarrow$ binaire l'est beaucoup moins). En contrepartie, la taille requise (en nombre de bits) est plus importante (le nombre 358 se code sur 9 bits en binaire pur).