



Electronique Numérique

Jean-Christophe Le Lann
lelannje@ensta-bretagne.fr

Table des matières

1	Le transistor	7
1.1	Introduction	7
1.2	Silicium et dopage	7
1.3	Jonction PN	9
1.4	Transistor MOS	9
1.5	MOS complémentaires	11
1.5.1	Principe	11
1.6	Niveaux d'abstraction	12
1.7	Conclusions	12
2	Représentations numériques	15
2.1	Notion de bases ou radix	16
2.1.1	Décomposition en puissance de la base	16
2.1.2	Conversion entre bases	16
2.1.3	Exemples	17
2.1.4	Conversions binaire, hexadécimal, octal	18
2.1.5	Bits, bytes, nibbles, words	18
2.2	Représentation des entiers négatifs.	19
2.2.1	Complément à 2 d'un nombre	19
2.2.2	Astuces pour complémenter à 2	20
2.2.3	A propos du bit de signe	20
2.2.4	Conversion d'un nombre signé vers un nombre non signé	21
2.3	Représentations alternatives, obsolètes ou rares	21
2.3.1	Représentation en Magnitude signée	21
2.3.2	Représentation en complément à 1	21
2.3.3	Code BCD et code de Gray pour les entiers non-signés	21
2.4	Calcul d'addition et soustraction	23
2.4.1	Addition	23
2.4.2	Soustraction	23
2.4.3	Petit complément sur la soustraction	24
2.5	Données composées ou symboliques	25
2.5.1	Données composées	25
2.5.2	Données symboliques ou énumérées.	25
2.5.3	Code alphanumérique ASCII	26
2.5.4	Code alphanumérique UTF-8	26
2.6	Calcul en virgule fixe et en virgule flottante	27
2.6.1	Les nombres en virgule fixe	27
2.6.2	Les nombres en virgule flottante. Norme IEEE 754	28
2.7	Manipulations numériques	28
2.7.1	Récupération d'un champ	28

TABLE DES MATIÈRES

2.7.2	Conversion par programmation	28
2.7.3	Autres manipulations au niveau bit	29
2.8	Vers la détection d'erreur : bit de parité	29
2.9	Conclusion	29
3	Logique booléenne	31
3.1	Introduction	31
3.2	Définitions	31
3.2.1	Axiomes de l'Algèbre de Boole	32
3.2.2	Principaux théorèmes de l'Algèbre de Boole	32
3.2.3	Théorèmes de De Morgan	33
3.3	Représentation des fonctions booléennes	33
3.3.1	Monôme	33
3.3.2	Fonctions booléennes	33
3.3.3	Fonction booléenne incomplète ou ϕ -booléenne	34
3.3.4	Minterm et maxterms	34
3.3.5	Décomposition de Shannon et Arbres de décision binaires	34
3.4	Simplification des fonctions booléennes	34
3.4.1	Par calcul algébrique	34
3.4.2	Par tableau de Karnaugh	35
3.5	Conclusion	36
4	Circuits combinatoires	37
4.1	Définition	37
4.2	Portes logiques de base	38
4.3	Fonctions logiques "complexes"	39
4.4	Mapping technologique	40
4.5	Chemin critique et fréquence de fonctionnement	40
4.6	Arithmétique de base	41
4.6.1	Additionneur	41
4.6.2	Soustracteur	43
4.6.3	Additionneur-soustracteur	43
4.6.4	Multiplieur	44
4.6.5	Diviseur	45
4.7	Shifter	45
4.8	Multiplexeur	46
4.9	Comparateur	46
4.10	Codeurs et Décodeurs	47
4.11	Conclusion	48
5	Circuits séquentiels	49
5.1	Introduction	49
5.2	Discrétiser le temps	50
5.3	Bascule D	51
5.3.1	Fonction d'échantillonnage de la bascule D. Set-up et hold. Metastabilité.	51
5.3.2	Décalage temporel : la raison d'être de la bascule D	52
5.4	Quelques <i>patterns</i> de conception synchrone	52
5.4.1	Registre à décalage	53
5.4.2	LFSR : linear feedback shift register	53
5.4.3	Bascule D et multiplexeur : mémorisation	54
5.4.4	Compteurs	54
5.5	Initialisation d'une bascule D	54

TABLE DES MATIÈRES

5.6	Conclusion	56
6	Machines d'états finis	57
6.1	Introduction	57
6.2	Machines d'états finis	57
6.3	Diagramme états-transitions	59
6.4	Consistance ou <i>causalité</i> d'une machine d'états finis	59
6.5	Encodage des états	60
6.6	Méthode générale de conception d'un automate	61
6.7	Exemple complet	62
6.8	Gérer la complexité	64
6.9	Méthode particulière : cas de l'encodage <i>one-hot</i>	64
6.10	Conclusion	65
7	VHDL	67
7.1	Introduction : les langages de description matérielle	67
7.2	La structure globale d'un programme VHDL	68
7.2.1	Notion d'entité	68
7.2.2	Notion d'architecture	68
7.3	Les éléments clés du langage	68
7.3.1	Assignations concurrentes	68
7.3.2	Processus	68
7.3.3	Instanciation de composants	68
7.3.4	Décrire des machines d'états finis	68
7.4	Simulation en VHDL	68
7.4.1	Flot de conception général	68
7.4.2	Bancs de tests ou <i>Testbench</i>	68
7.4.3	Algorithme de simulation	68
7.5	Synthèse matérielle	68
7.5.1	Exemple du flot Xilinx	68
7.6	Conclusion	68
8	Thèmes avancés	69
8.1	Synthèse comportementale ou HLS	69
8.2	Modèles de calculs	69
8.3	Langages synchrones	69

TABLE DES MATIÈRES

Chapitre 1

Le transistor CMOS

La naissance des 0 et 1

1.1 Introduction

Le transistor reste le moteur numéro 1 de l'innovation au XX et XXI siècle. Bien que notre cours d'électronique numérique ait pu s'en passer –moyennant quelques acrobaties–, il aurait été dommage de le passer totalement sous silence. Nous ne dévoilons ici qu'un coin du voile. Sachez qu'un grand nombre d'ingénieurs et chercheurs –y compris français– continuent de travailler à son amélioration, ainsi qu'à trouver un successeur digne de lui...

1.2 Silicium et dopage

Le silicium est l'élément chimique dénoté *Si* dans la table périodique des éléments, de numéro atomique 14. Sa forme cristallographique présente un double réseau (identique à celui du diamant), décalés l'un de l'autre du quart de la diagonale principale. Sur chacun des réseaux les atomes de silicium se situent au sommet d'un cube, ainsi que sur chacune des faces de ce cube (il est dit "cubique face centrée").

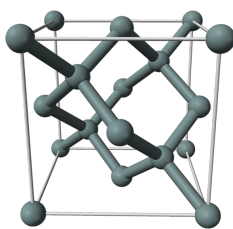


FIGURE 1.1: La maille élémentaire du cristal de silicium

Ses propriétés physico-chimiques en font un semi-conducteur : sa conductivité électrique est notamment bien inférieure à celle d'un métal. Dans le double réseau, chaque atome de silicium peut être considéré comme au centre d'un tétraèdre, chacun des atomes auquel il est lié se trouvant sur un des quatre sommets du tétraèdre. Les liaisons covalentes sont très solides et permettent la formation d'un cristal parfait. Tous les électrons étant utilisés dans les liaisons, aucun n'est disponible pour permettre le passage d'un courant électrique, du moins aux températures très basses ; le cristal présente une résistivité assez élevée. Lorsque la température s'élève, sous l'effet

de l'agitation thermique, des électrons réussissent à s'échapper et participent à la conduction. Ce sont les électrons situés sur la couche la plus éloignée du noyau qui s'impliquent dans les liaisons covalentes. Dans le cristal, ces électrons se situent sur des niveaux d'énergie appelée bande de valence. Les électrons qui peuvent participer à la conduction possèdent des niveaux d'énergie appartenant à la bande de conduction. Entre la bande de valence et la bande de conduction peut se situer une bande interdite. Pour franchir cette bande interdite l'électron doit acquérir de l'énergie (thermique, photon...). Pour les isolants la bande interdite est quasi infranchissable, pour les conducteurs elle est inexistante. Les semi-conducteurs ont une bande interdite assez étroite.

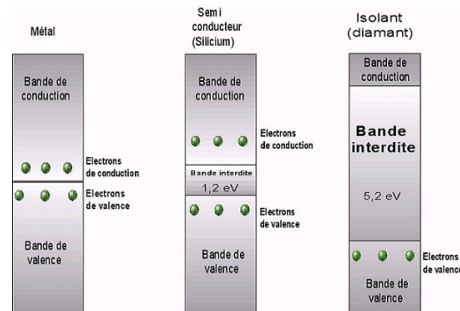


FIGURE 1.2: Bande de valence et de conduction de différents types de matériaux

L'atome qui a perdu un électron devient un ion positif et le trou ainsi formé peut participer à la formation d'un courant électrique en se déplaçant. Dans un cristal pur, à température ordinaire, les électrons libres sont malgré tout extrêmement rares - de l'ordre de 3 pour 10¹³ atomes (10 puissance 13). Si l'électron libre est capté par un atome, il y a recombinaison. Pour une température donnée ionisation et recombinaison s'équilibrent ; la résistivité diminue quand la température augmente. Un semi-conducteur dont la conductivité ne doit rien à des impuretés est dit *intrinsèque*.

Lors de la formation du cristal de silicium il suffit d'introduire une infime quantité d'impuretés sous la forme d'atomes d'aluminium (possédant seulement 3 électrons sur leur couche externe) pour que le nombre de trous dans le cristal augmente considérablement. Le cristal est dit dopé et comme les porteurs de charges majoritaires sont des trous, positifs, le cristal est dit dopé P. Les électrons libres qui correspondent à la conductivité intrinsèque sont appelés porteurs minoritaires. Si un électron est arraché d'un atome voisin et vient combler le trou, tout se passe comme si c'était le trou qui s'était déplacé. On peut également doper le cristal avec des impuretés pentavalentes (atomes possédant 5 électrons sur leur couche externe), comme l'arsenic ou l'antimoine. On se retrouve alors avec un électron supplémentaire, donc libre. Les porteurs de charges majoritaires sont alors de polarité négative, le cristal est dit dopé N. Les porteurs de charge minoritaires sont dans ce cas les trous (positifs) de la conductivité intrinsèque.

Un atome pentavalent comme l'arsenic possède 5 électrons sur sa couche externe. En tant qu'impureté dans un cristal de silicium (tétravalent) il fournit un électron au cristal. Il est dit atome donneur. Si l'impureté est un atome trivalent (3 électrons sur sa couche externe, comme le bore ou l'indium) il est dit atome accepteur car il va capter un électron et générer un trou. Les porteurs majoritaires sont beaucoup plus nombreux que les porteurs minoritaires (10⁶ à 10¹² fois plus nombreux).

Le fait d'introduire en très faible quantité des impuretés (opération appelée dopage) dans un cristal de semi-conducteur améliore fortement la conductivité du cristal. Si un cristal de germanium ou de silicium a reçu des impuretés pentavalentes (arsenic, phosphore, antimoine) il

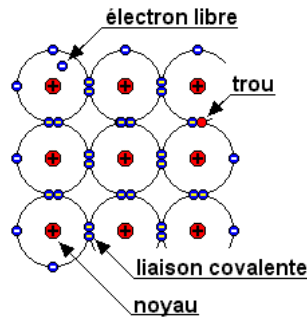


FIGURE 1.3: Présence des différents porteurs de charge

devient un semi-conducteur à conductivité N (ex : silicium N). Un cristal de germanium dopé par des impuretés trivalentes (indium, gallium, bore) devient un semi-conducteur P.

1.3 Jonction PN

Le fait d'introduire en très faible quantité des impuretés (opération appelée dopage) dans un cristal de semi-conducteur améliore donc fortement la conductivité du cristal. Si un cristal de germanium ou de silicium a reçu des impuretés pentavalentes (arsenic, phosphore, antimoine) il devient un semi-conducteur à conductivité N (ex : silicium N). Un cristal de germanium dopé par des impuretés trivalentes (indium, gallium, bore) devient un semi-conducteur P.

Une jonction entre le type p et de type n de silicium est appelé une **diode**.



FIGURE 1.4: Jonction PN ou diode

Lorsque la tension sur le semi-conducteur de type p , appelée anode, est supérieure à celle la tension sur la partie dope n (cathode), la diode est polarisée en direct et le courant circule.

Lorsque la tension d'anode est inférieure ou égale à la tension de cathode, la diode est polarisée en inverse et très peu de courant circule. Ce dipôle est aussi appelé diode de redressement car il est utilisé pour réaliser les redresseurs qui permettent de transformer le courant alternatif en courant continu.

1.4 Transistor MOS

Le transistor est un composant électronique qui a deux utilités majeures : il permet d'**amplifier un courant ou une tension**, ou il peut être utilisé comme un **interrupteur commandable en courant ou en tension**. Dans la suite, nous n'utiliserons que cette seconde capacité. Il existe deux catégories de transistor :

- les transistors bipolaires, constitués par la succession de trois zones dopées PNP ou NPN.
- les transistors MOS, qui représentent désormais plus de 85% du marché.

La figure suivante présente la structure d'un tel transistor, ici de type n .

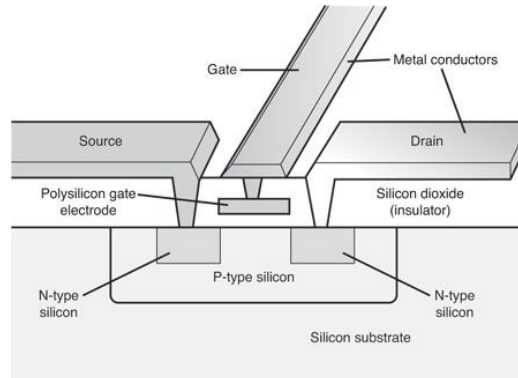


FIGURE 1.5: Transistor $n - MOS$

Un tel transistor présente une structure en sandwich :

- une grille (gate) conductrice (généralement du métal ou du silicium polycristallin)
- une couche isolante (oxyde de silicium SiO_2)
- le substrat de silicium, dont la tension est généralement à la masse.
- deux zones dopées de manière identique (ici dopées n), appelées source et drain.

Ces structures sont fabriquées en utilisant une série d'étapes de traitements chimiques impliquant l'oxydation du silicium, introduction sélective de dopants (impuretés), le dépôt et la gravure de fils métalliques et des contacts.

Comme on peut l'observer, un transistor $n - MOS$ est construit à partir d'un substrat p et de deux régions dopées de type n (c'est l'inverse pour un transistor de type p). La grille permet de contrôler le courant qui circule entre la source et le drain. Dans le cas d'un nMOS, le substrat est à la masse : par conséquent les jonctions p-n entre les source-substrat et drain-substrat sont polarisées en inverse. Si la grille est également à la masse, aucun courant ne circule entre les deux jonctions : le transistor est non-passant, ou fermé, ou "OFF".

Si on élève la tension de la grille, un champ électrique se crée, qui attire les électrons libres sous la couche isolante. Si cette tension est suffisamment élevée le nombre d'électrons dépasse le nombre de trous : une région, appelée "canal" se crée, qui se comporte comme un semiconducteur de type n , alors que le substrat initial est p ! Un chemin s'établit pour les électrons entre la source et le drain : le transistor devient passant. Il est ouvert ou "ON". Pour les pMOS, la situation est exactement opposée.

La tension "suffisante" est appelée VDD et représente la valeur logique 1 dans les circuits numériques. Dans les années 70 et 80, VDD valait traditionnellement 5 volts. Plus récemment, les transistors ne pouvant supporter de telles tensions, VDD a été réduit à 3.3V, puis 1.8V etc... La tension appelée VSS (ou ground) représente quant à elle la valeur logique 0. Elle vaut normalement 0 volts.

En simplifiant le mécanisme à l'extrême, les transistors MOS peuvent être vus comme des interrupteurs ON-OFF. Lorsque la grille d'un transistor nMOS est 1, le transistor est OFF et il existe un chemin de conduction entre source et drain. Lorsque la grille est à 0, le transistor est OFF et pratiquement aucun courant ne circule entre la source et le drain. Le transistor pMOS fonctionne à l'opposé, étant ON lorsque la grille est à 0 et OFF lorsque la tension de grille est 1.

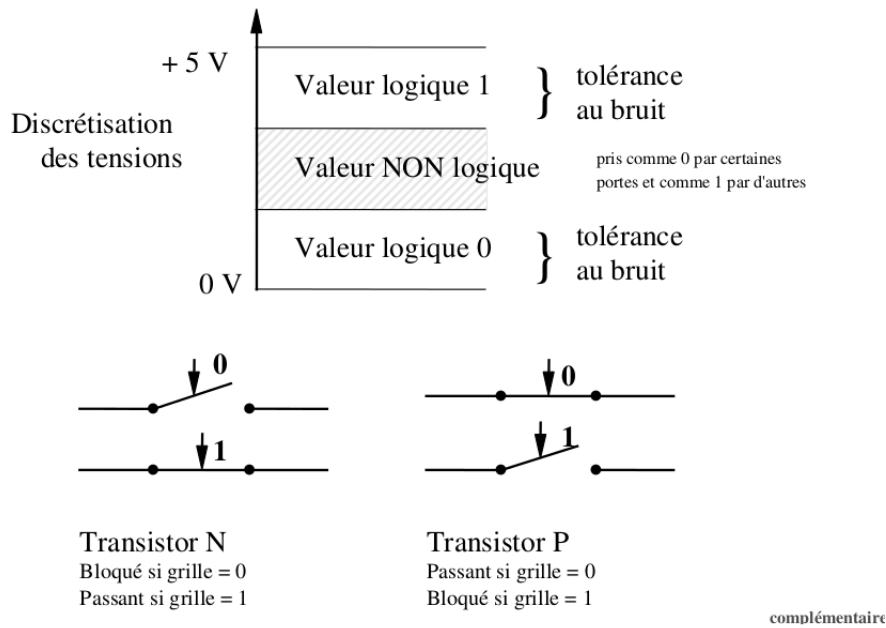
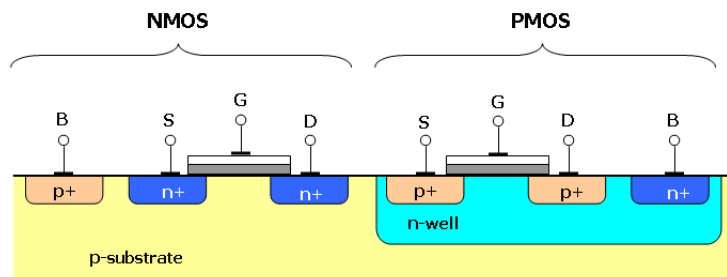


FIGURE 1.6: CMOS

1.5 MOS complémentaires

1.5.1 Principe

Nous sommes désormais dotés de deux types de transistors. Technologiquement, il a longtemps été question de ne réaliser les circuits numériques à l'aide d'un seul type de transistor (N ou P). Mais la technique qui sort largement vainqueur est la logique complémentaire CMOS. Elle consiste à utiliser de manière duale les transistors P et N, dans le but de réaliser une fonction logique. Les transistors P sont utilisés pour tirer à 1 et les transistors N pour tirer à 0. Il n'y a pas de perte de seuil. Pour associer ces deux types de transistors, les techniques de dépôts microélectroniques deviennent plus complexes : il faut être à même de présenter un substrat P et N ! En fait, on réalise une zone N sur un substrat P...

FIGURE 1.7: Transistor $n - MOS$

A titre de premier exemple, considérons le cas d'un *inverseur* CMOS.

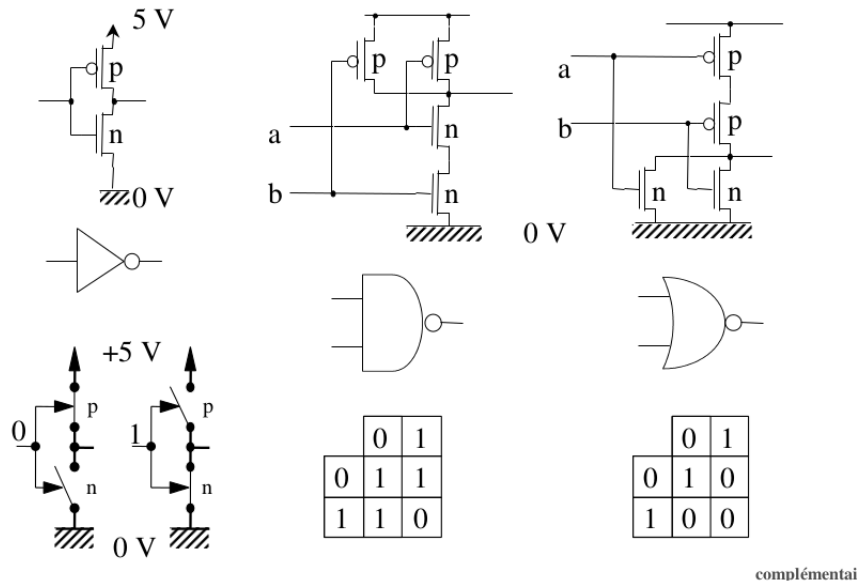


FIGURE 1.8: CMOS

1.6 Niveaux d'abstraction

1.7 Conclusions

Ce chapitre se veut court, car l'ENSTA-Bretagne n'entend pas former des spécialistes de la microélectronique, ni de la nano-électronique. Je vous invite toutefois à ne pas perdre de vue, à travers vos lectures, les avancées probables dans le domaine : les progrès de la physique théorique et appliquée nous apporteront très certainement des révolutions à la hauteur de celles que nous avons vécu dans les années 70. On peut même inciter à garder un oeil critique sur les techniques numériques : le tableau suivant compare les avantages et inconvénients de l'électronique analogique et numérique : comme on peut le voir le numérique se révèle bourré de défauts (exemple : il est plus compliqué de réaliser des opérations arithmétiques, il est très bruyant,...).

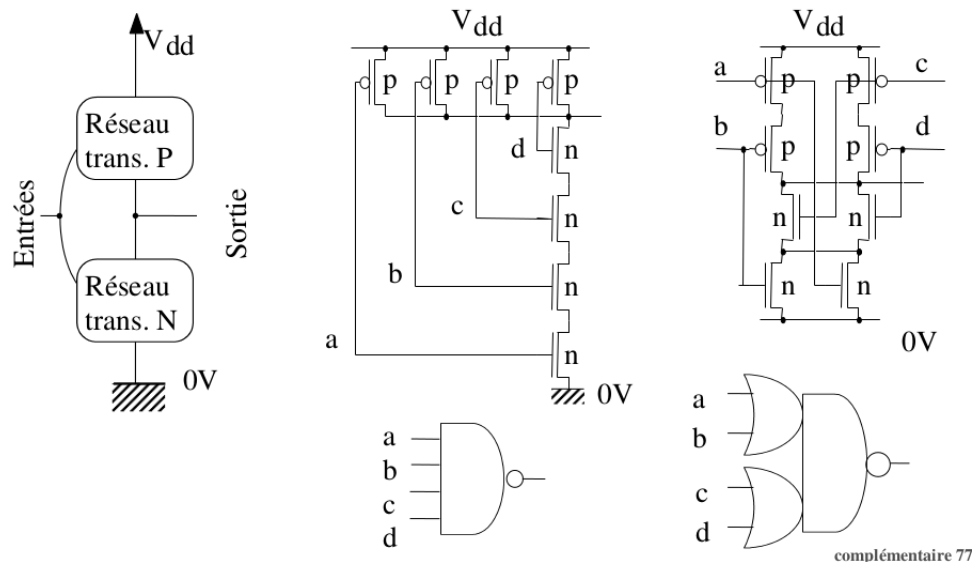


FIGURE 1.9: CMOS

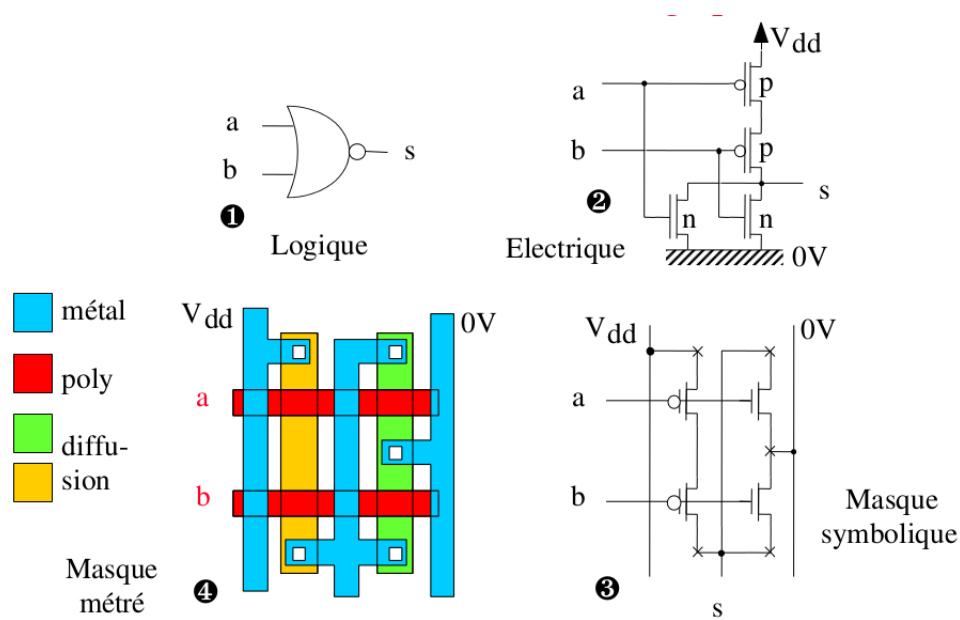


FIGURE 1.10: CMOS

Analogique	Logique
Précision limitée (techno)	Précision arbitraire (# bits)
Valeur approchée ($\pm 5\%$)	Valeur exacte
Logique infidèle	fidélité absolue (pas de dérive)
Compensations nécessaires	pas de compensation
Valeurs continues	valeurs discrètes (bruit de quantification)
Temps continu	Temps discret (bruit d'échantillonnage)
Silencieuse et sensible	Bruyante et insensible
Exemple: multiplieur de Gilbert (Mos en faible inversion) 14t	Exemple: multiplieur 5x5 bits 550 transistors MOS bloqués/saturés

FIGURE 1.11: CMOS

Chapitre 2

Représentations numériques

Sommaire

2.1	Notion de bases ou radix	16
2.1.1	Décomposition en puissance de la base	16
2.1.2	Conversion entre bases	16
2.1.3	Exemples	17
2.1.4	Conversions binaire, hexadécimal, octal	18
2.1.5	Bits, bytes, nibbles, words	18
2.2	Représentation des entiers négatifs.	19
2.2.1	Complément à 2 d'un nombre	19
2.2.2	Astuces pour complémenter à 2	20
2.2.3	A propos du bit de signe	20
2.2.4	Conversion d'un nombre signé vers un nombre non signé	21
2.3	Représentations alternatives, obsolètes ou rares	21
2.3.1	Représentation en Magnitude signée	21
2.3.2	Représentation en complément à 1	21
2.3.3	Code BCD et code de Gray pour les entiers non-signés	21
2.4	Calcul d'addition et soustraction	23
2.4.1	Addition	23
2.4.2	Soustraction	23
2.4.3	Petit complément sur la soustraction	24
2.5	Données composées ou symboliques	25
2.5.1	Données composées	25
2.5.2	Données symboliques ou énumérées.	25
2.5.3	Code alphanumérique ASCII	26
2.5.4	Code alphanumérique UTF-8	26
2.6	Calcul en virgule fixe et en virgule flottante	27
2.6.1	Les nombres en virgule fixe	27
2.6.2	Les nombres en virgule flottante. Norme IEEE 754	28
2.7	Manipulations numériques	28
2.7.1	Récupération d'un champ	28
2.7.2	Conversion par programmation	28
2.7.3	Autres manipulations au niveau bit	29
2.8	Vers la détection d'erreur : bit de parité	29
2.9	Conclusion	29

2.1 Notion de bases ou radix

2.1.1 Décomposition en puissance de la base

Les nombres que nous manipulons tous les jours s'expriment en base (ou *radix* 10. On parle de base 10 car on utilise 10 symboles (chiffres) allant de 0 à 9 inclus. Par la suite, on s'autorisera à parler de *digits*. Dans un nombre, ces digits sont en fait des *coefficients* du système décimal : pour former un nombre, on multiplie ces coefficients par les puissances successives de 10. Par exemple le nombre 142 s'exprime ainsi :

$$142 = 1 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

Il en est de même des nombres fractionnaires comme :

$$142.34 = 1 \times 10^2 + 4 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2}$$

La formule générale pour la base 10 est bien entendu :

$$x = \sum_{i=-\infty}^{i=\infty} c_i \cdot 10^i, \text{ avec } c_i \in \{0 \dots 9\}$$

et se généralise pour toute base r :

$$x = \sum_{i=-\infty}^{i=\infty} c_i \cdot r^i, \text{ avec } c_i \in \{0 \dots (r-1)\}$$

Les ordinateurs n'utilisent pas la base 10, mais la base 2, qui ne possède que deux valeurs de coefficients possibles : 0 et 1.

Afin d'éviter toute confusion par la suite, nous plaçons en indice du nombre l'indication de la base utilisée, ce qui évitera de nous perdre lors des changements de base. Par exemple, on peut noter :

$$10110_{10} = 1001110111110_2$$

En pratique, il existe deux autres bases indispensables aux informaticiens et électroniciens ¹ : la base octale (base 8) et la base hexadécimale (base 16). La première n'utilise que les digits allant de 0 à 7. En hexadécimal, on doit introduire de nouveaux symboles au delà du 9 : on utilise –dans l'ordre– les lettres (majuscules ou minuscules) A, B, C, D, E et F, correspondant respectivement à 10, 11, 12, 13, 14 et 15. La table suivante montre les 18 premiers nombres dans le système décimal, binaire, octal et hexadécimal, ainsi que deux autres représentations sur lesquelles nous reviendrons un peu plus tard).

2.1.2 Conversion entre bases

Grâce aux formules précédentes, nous savons transformer un nombre exprimé dans une base r en un nombre décimal. Il est à présent important de savoir faire l'inverse, à savoir : convertir un nombre décimal x_{10} vers tout autre base r . Ceci peut se faire de manière mécanique : il suffit de diviser le nombre x (dividende) par r (diviseur), puis de répéter l'opération sur le quotient jusqu'à ce que ce quotient soit nul. Le premier reste obtenu sera le digit le plus à droite ², et le dernier reste le digit le plus à gauche ³.

Dans le cas de nombres fractionnaires, il faut procéder de la manière suivante : multiplier le nombre par r jusqu'à obtenir un nombre sans partie fractionnaire. Ceci n'est évidemment pas toujours possible.

¹Personnellement, je n'ai jamais eu à utiliser la première...

²En base binaire, on parle de LSB : least significant bit

³MSB : most significant bit

base (ou nom du code)					
10	2	8	16	BCD	Gray
0	0	0	0	0000	0
1	1	1	1	0001	1
2	10	2	2	0010	11
3	11	3	3	0011	10
4	100	4	4	0100	110
5	101	5	5	0101	111
6	110	6	6	0110	101
7	111	7	7	0111	100
8	1000	10	8	1000	1100
9	1001	11	9	1001	1101
10	1010	12	a	00010000	1111
11	1011	13	b	00010001	1110
12	1100	14	c	00010010	1010
13	1101	15	d	00010011	1011
14	1110	16	e	00010100	1001
15	1111	17	f	00010101	1000
16	10000	20	10	00010110	11000
17	10001	21	11	00010111	11001

TABLE 2.1: Table des correspondances entre bases 10, 2, 8, 16, puis code BCD et code de Gray

2.1.3 Exemples

- **Convertir le nombre 12_{10} en binaire**

Solution :

$$12/2 = 6 + 0$$

$$6/2 = 3 + 0$$

$$3/2 = 1 + 1$$

$$1/2 = 0 + 1$$

Par conséquent : $(12)_{10} = (1100)_2$

- **Convertir $(0.75)_{10}$ en binaire**

Solution :

$$0.75 \times 2 = 1.5 = 1 + 0.50$$

$$0.50 \times 2 = 1.0 = 1 + 0$$

Ainsi :

$$(0.75)_{10} = (0.11)_2$$

- **Convertir $(0.39654)_{10}$ en binaire**

Solution :

$$0.39654 \times 2 = 0.79308 = 0 + 0.79308$$

$$0.79308 \times 2 = 1.58616 = 1 + 0.58616$$

$$0.58616 \times 2 = 1.17232 = 1 + 0.17232$$

$$0.17232 \times 2 = 0.34464 = 0 + 0.34464$$

etc...Cet conversion n'a pas de fin. La partie fractionnaire ne peut s'exprimer sous la forme d'une somme exacte de puissance de 2. On notera la solution :

$$(0.39654)_{10} = (0.0110...)_{2}$$

Il est important que vous sachiez rapidement effectuer ces conversions avec votre ordinateur ou calculatrice préférée ⁴.

2.1.4 Conversions binaire, hexadécimal, octal

Comme $2^4 = 16$, il s'en suit que la représentation binaire de chaque digit de la base 16 nécessite 4 bits. Pour la base octale, il en suffit de 3. De ce fait, les conversions de nombres binaires vers l'hexadécimal ou l'octal se réalisent simplement, par groupement de bits. En partant de la droite, il suffit par exemple de regrouper 4 bits pour obtenir un digit hexa. Par exemple :

$$\text{le nombre } 101101_2 = 10_1101_2 = 0010_1101_2 = 2D_{16} = \mathbf{0x2D}$$

Dans le cas de nombre fractionnaires, le sens de lecture s'inverse pour la partie fractionnaire. Ainsi :

$$\text{le nombre } 1011.01_2 = 1011.0100_2 = B.4_{16}$$

On procède de même pour la traduction octale.

$$(10110001101011.1111)_2 = (101\ 110\ 001\ 101\ 011\ .\ 111\ 100)_2 = (26153.74)_8$$

2.1.5 Bits, bytes, nibbles, words

La plus petite unité informatique est le bit, qui est l'abréviation de *binary digit*. Toutefois rares sont les ordinateurs qui ne traitent qu'un seul bit à la fois : la plus petite la grandeur manipulable par la quasi-totalité des ordinateurs reste l'**octet** : c'est l'aggrégation de 8 bits. Sa traduction anglaise est le *byte*, qui se prononce *bait* et qui ne doit pas être confondu avec le bit. Le regroupement de 4 bits est moins connu, mais indispensable : il s'agit du **quartet** ou *nibble* en anglais. Nous verrons plus loin en quoi il est important. Enfin, on parle de **mot** ou *word* pour le regroupement de 16 bits, sans toutefois que cette définition soit parfaitement admise. On parlera d'ailleurs de processeurs qui opèrent sur des mots de 16 bits, 32 bits, 64 bits etc...

Les bits des quartets, octets et mots sont généralement manipulés simultanément, lorsque la machine possède les circuits qui le permettent. Nous y reviendrons par la suite.

⁴Pour votre information, il est possible d'utiliser Google pour effectuer cela : tapez "0x123 in octal" dans l'invite de Google...Vous obtenez 0o443

Première addition binaire Considérons le cas de l'addition binaire. Seuls 4 combinaisons sont à étudier :

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \text{ et une retenue de } 1 \end{aligned}$$

Nous verrons, au chapitre suivant, comment forcer le silicium à réaliser électriquement ce calcul !

2.2 Représentation des entiers négatifs.

Jusqu'ici nous n'avons pas parlé de nombres négatifs. Comment représenter des nombres négatifs ?

La manière la plus intuitive serait de considérer un nombre négatif comme la juxtaposition de deux informations représentées isolément : le signe et la valeur absolue ("magnitude"). Le signe peut être encodé sur 1 seul bit : 0 ou 1 représentant par exemple le + et le - respectivement. On appelle cette représentation "magnitude signée". Cette représentation "positionnelle" est tout à fait envisageable, mais elle se révèle **inefficace matériellement**. Elle oblige notamment à revoir le calcul de l'addition. Elle a donc rapidement été abandonnée au cours de l'histoire de l'Informatique. A l'inverse, certaines opérations sont grandement simplifiées par le recours à la **représentation complémentée** d'un nombre. Aujourd'hui, les ordinateurs stockent les valeurs négative en binaire, en complément à 2. Nous allons donc commencer par cette "bonne" représentation, puis nous reviendrons sur ces autres représentations obsolètes ou rares.

2.2.1 Complément à 2 d'un nombre

Soit un nombre W de bits et soit $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$ de bits. Nous avons vu précédemment (sans la nommer) que la simple fonction :

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

définit la valeur non-signée ("unsigned") de \vec{x} , dans \mathbb{N} . Les valeurs limites de cette fonction sont :

$$\begin{cases} UMax_w &= \sum_{i=0}^{w-1} 2^i = 2^w - 1 \\ UMin_w &= \sum_{i=0}^{w-1} 0 = 0 \end{cases}$$

On a donc :

$$B2U_w : \{0, 1\}^w \rightarrow \{0, \dots, 2^w - 1\}$$

. La fonction inverse est notée $B2U_w^{-1}$. Elle permet de passer à un code binaire sur w bits à partir d'un nombre compris entre 0 et $2^w - 1$. Cette fonction possède la très bonne propriété de fournir un encodage unique de ce nombre entier positif.

De manière similaire, on définit désormais la fonction qui permet de calculer la valeur du vecteur \vec{x} dans le cas d'une valeur signée, c'est-à-dire négative ou positive, **en complément à 2**.

$$B2T_w(\vec{x}) \doteq -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

signé	bit vector	non-signé
-4	100	4
-3	101	5
-2	110	6
-1	111	7
0	000	0
1	001	1
2	010	2
3	011	3

TABLE 2.2: Entiers relatifs de -4 à 3 , traduits en complément à 2 sur 3 bits et conversion en entier non-signé)

Désormais, le bit de poids fort, le plus à gauche, contribue de manière négative à la somme totale. Précisons également que le 'T' vient de "two's complement". Calculons les valeurs min et max de cette fonction. Pour $\vec{x} = [10 \dots 0]$ et $\vec{x} = [01 \dots 1]$ respectivement, on trouve ⁵ :

$$\begin{cases} TMin_w &= -2^{w-1} \\ TMax_w &= \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1 \end{cases}$$

On a donc :

$$B2U_w : \{0, 1\}^w \rightarrow \{-2^{w-1}, \dots, 2^{w-1} - 1\}$$

On pourra également dénoter cette fonction $C_{2,w}(\vec{x})$.

Testons ces formules, dans un cas simple : $C_{2,3}$, c'est-à-dire le complément à 2 sur 3 bits. Dans ce cas, on peut compter de -4 à 3 . On constate par exemple que $C_{2,3}(-3_{10}) = 101_2 = 5_{10}$

2.2.2 Astuces pour complément à 2

Le complément à 1 d'un nombre binaire consiste à simplement inverser tous ses bits. Le complément à 2 d'un nombre binaire consiste à :

- laisser inchangé le 1 le plus à droite, ainsi que tous les 0 les plus à droite.
- modifier tous les autres digits.

Il existe toute fois une autre méthode, qui se retient plus facilement (peut-être). Elle consiste en une formule simple :

$$-A = /A + 1$$

exemple : pour trouver le complément à deux de 1101100_2 . On inverse tous les bits et l'on additionne 1.

$$0010011 + 1 = 0010100$$

2.2.3 A propos du bit de signe

Cherchons à faire croire le nombre de bits dans la représentation d'un nombre négatif (disons -5). Un petit calcul mène aux résultats consignés dans la table 2.3. On s'aperçoit que le bit le plus à

⁵On rappelle que la formule d'une suite géométrique de raison q :

$$S_n = a \sum_{k=0}^{n-1} q^k = a \frac{1 - q^n}{1 - q}$$

$C_{2,4}(-5)$	1011
$C_{2,5}(-5)$	11011
$C_{2,6}(-5)$	111011
$C_{2,7}(-5)$	1111011

TABLE 2.3: Position du bit de signe pour un nombre de bits croissant : le bit se "propage" à gauche)

gauche (MSB) se propage vers la gauche, jusqu'au MSB. Notons ainsi qu'il n'est pas exact de dire que le bit de signe se situe *seulement* en MSB : nos exemples montrent bien que le "caractère négatif" d'un nombre peut apparaître, selon la valeur w , bien avant le MSB. Toutefois, on doit retenir que c'est le bit MSB qui permet, d'un seul coup d'oeil, de discriminer les valeurs positives des valeurs négatives. Dans cette représentation, le bit en position $n - 1$ est le bit dit "de signe" : il doit être vu que comme un indicateur sur le signe du nombre représenté (1 signifie '-' et 0 signifie '+'), mais les autres bits ne représentent pas directement $|n|...$

2.2.4 Conversion d'un nombre signé vers un nombre non signé

Parmi les allers-retours entre représentations des nombres, il est fréquent de chercher à convertir un nombre signé en nombre non-signé. Dans ce cas, on a : $\forall x \in \mathbb{Z}_w / TMin_w \leq x \leq TMax_w$:

$$TU_w(x) = \begin{cases} x + 2^w & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$$

2.3 Représentations alternatives, obsolètes ou rares

2.3.1 Représentation en Magnitude signée

La représentation en magnitude signée est simplement :

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \sum_{i=0}^{w-2} x_i 2^i$$

2.3.2 Représentation en complément à 1

Il existe également une représentation des nombres en complément à 1, très légèrement différente du complément à 2.

$$B2T_w(\vec{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$$

Ces deux représentations ont la curieuse (et facheuse) propriété de proposer (chacune) deux représentations du 0 : soit en tant que +0, soit en tant que -0. On imagine aisément que, bien que ces représentations soient différentes, il faudrait, au final, que les calculateurs numériques les traitent comme bien évidemment égales. Ces "cas" à distinguer sont à l'origine de l'abandon de ces représentations, dans les systèmes numériques.

2.3.3 Code BCD et code de Gray pour les entiers non-signés

Nous avons parlé des représentations des entiers dans différentes bases. Lorsque le nombre total d'entiers à encoder est connu, on peut également chercher à représenter ces entiers par d'autres encodages ou *système de numération*. Cela est très fréquent dans les ordinateurs, car les architectures sont par nature finie en dimension. Hors de question de compter à l'infini !

Il existe de nombreux encodages, du plus simple ou plus compliqué. Par exemple, certains codages prédisposent à la simplification de la détection et même la correction d'erreur lors d'une communication numérique (portable, TV numérique,...)! Nous ne pouvons nous attarder sur tous ces codages, mais sachez que c'est en fait une discipline à part entière, avec ses problématiques et enjeux propres.

Code BCD

Le codage BCD signifie "binary coded decimal". Chaque digit décimal 0 à 9 est alors traduit directement par 4 bits. Ceci est évidemment assez naturel.

Chiffre	Quartet	Chiffre	Quartet
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Pour coder un nombre tel 147 il suffit de coder chacun des chiffres 1, 4 et 7 ce qui donne 0001, 0100, 0111. Cette représentation a un intérêt pratique certain : lorsqu'on réalise un appareil qui affiche des valeurs entières de plusieurs digits, le code BCD permet d'isoler l'électronique de chaque digit, de manière triviale.

Code de Gray

Souvenons nous par exemple que le passage de 7 à 8 en binaire classique fait commuter 4 bits : tous les bits changent de valeurs! Le code de Gray (aussi appelé *codage binaire réfléchi*) vise à éviter de telles commutations simultanées : il est constitué d'une manière telle que *seul 1 digit binaire change entre un entier n et son successeur $n + 1$* . En pratique, cela évite de commuter un grand nombre de dispositifs en même temps, lors d'un comptage régulier. Le codage de gray fait partie d'un ensemble de codes appelés à "distance minimale".

décimal	binaire classique	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Il existe un algorithme pour passer d'un nombre x à l'autre $x + 1$:

- on calcule le nombre de 1 dans x . On inverse le dernier bit de x quand ce nombre de 1 est pair.

- si le nombre de 1 est impair, on inverse le bit à gauche du 1 qui est le plus à droite.

Le terme “binaire réfléchi” provient d’une seconde méthode, graphique. A partir des deux codes

$$\begin{aligned} 0 &\rightarrow 0000 \\ 1 &\rightarrow 0001 \end{aligned}$$

on réalise une réflexion dans un miroir, et un ajout de 1 en tête, afin d’obtenir valeurs suivantes :

$$\begin{aligned} 0 &\rightarrow 0000 \\ 1 &\rightarrow 0001 \\ - &- - - - \\ 2 &\rightarrow 0011 \\ 3 &\rightarrow 0010 \end{aligned}$$

En langage informatique comme Python, Ruby, Java ou C, il est remarquablement facile de passer d’un nombre à sa représentation Gray :

```
g = b ^ (b >> 1)
```

L’inverse est plus compliqué ⁶.

2.4 Calcul d’addition et soustraction

2.4.1 Addition

L’addition se pose exactement de la même manière que dans la base 10.

2.4.2 Soustraction

La soustraction de deux nombres binaires ⁷ se fait en utilisant le complément à 2 : soient deux nombres x et y . La soustraction $x - y$ se réalise en utilisant le fait que $x - y = x + (-y)$:

1. on additionne x et le complément à 2 de y
2. si le résultat présente une retenue finale, on l’oublie !
3. s’il n’y a pas de retenue finale, on prend le complément à 2 du résultat et on place un signe “-” devant le nombre.

exemple : $x = 1101100_2$ et $y = 1011011_2$

$$1101100 - 0100101 = 1\ 0010001$$

Le résultat est donc $x - y = 0010001_2$

On retiendra que, lors d’une soustraction, la méthode générale consiste à considérer les valeurs absolues des opérandes mis en jeu $\{|x|, |y|\}$. En passant en signé, le nombre de bit est

$$n = 1 + \max(\lceil \log_2(|x|) \rceil, \lceil \log_2(|y|) \rceil)$$

⁶Je suis preneur d’une solution élégante...

⁷ceci se généralise dans tout autre base.

2.4.3 Petit complément sur la soustraction

On cherche en ici à illustrer cette soustraction dans deux cas :

1. celui où une retenue apparaît lors de l'addition. Cette retenue peut apparaître au delà (à gauche) du bit de signe. La question est de savoir ce que l'on doit en faire...
2. celui où aucune retenue n'apparaît.

Cas où une retenue apparaît lors de l'addition

Prenons l'exemple de $63 - 37$. Procédons de manière mécanique :

- Pour représenter 63, il faut $\lceil \log_2(63) \rceil = 6$ bits.
- Idem pour +37 : il faut également 6 bits. Donc il est nécessaire d'avoir **7 bits** pour -37 .
- On va représenter 63 et -37 sur 7 bits.
- $63_{10} = 0111111_2$
- On représente -37 en complément à 2 : on part de la représentation binaire de $+37_{10} = 0100101_2$. On inverse tous les bits (complément à 1) et on ajoute 1. On trouve : 1011011
- On fait la somme : on trouve un *résultat sur 8 bits* : 10011010 . Le bit le plus à gauche est *une retenue que l'on peut oublier dans le résultat*. Nous allons en reparler dans un instant...
- Le résultat est donc positif (bit de signe à '0') et vaut 0011010 . On peut vérifier $0011010_2 = 26_{10}$, qui est bien le résultat attendu.

On peut se poser des questions quant à l'avant dernière étape, consistant à “oublier” la retenue : pourquoi cette oubli ? Avant d'expliquer le phénomène, recommençons le calcul précédent, mais avec un grand nombre de bits : au lieu de 7 bits, passons à 10 bits par exemple. En refaisant exactement le même calcul, on voit apparaître cette retenue en position 11. Tous les autres bits restent inchangés. En fait, le complément à 2 d'un entier n , sur k bits, est la quantité $2^k - n$. Donc en faisant l'opération $m - n$, on passe à $m + (2^k - n) = 2^k + (m - n)$. Il est donc normal de voir 2^k apparaître systématiquement.

Cas où aucune retenue n'apparaît

Plaçons nous désormais dans le cas où aucune retenue n'apparaît. Pour cela, effectuons par exemple : $12 - 14$.

- Pour représenter 12, il faut $\lceil \log_2(12) \rceil = 4$ bits.
- Idem pour +14 : il faut également 4 bits. Donc il est nécessaire d'avoir **5 bits** pour -14 .
- On va représenter 12 et -14 sur 5 bits.
- $12_{10} = 01100_2$
- On représente -14 en complément à 2 : on part de la représentation binaire de $+14_{10} = 01110_2$. On inverse tous les bits (complément à 1) et on ajoute 1. On trouve : 10010
- On fait la somme : on trouve un *résultat sur 5 bits* : 11110 . Le bit le plus à gauche n'est cette fois-ci pas une retenue.

- Le résultat est donc *négatif* (bit de signe à '1') et vaut 11110. Si on veut retrouver sa représentation “-y”, on doit donc, à nouveau, passer par le complément à 2. Invertissons tous les bits et ajoutons 1 : on trouve $00010 = 2_{10}$. Conclusion : le résultat vaut $-y = -2_{10}$. C'est bien le résultat attendu.

On peut maintenant se poser la question : qu'est-il advenu de la quantité 2^k évoquée dans l'autre cas ? On a effectué ici $12 - 14$ en passant par le complément à 2, soit :

$$12 + (2^5 - 14) = 2^5 + (12 - 14) = 32 + 12 - 14 = 44 - 14 = 30$$

C'est bien ce nombre que l'on voit apparaître au final : $30 = 11110_2$. On peut d'ailleurs dire que 30 est le complément à 2 de 2, sur 5 bits.

2.5 Données composées ou symboliques

2.5.1 Données composées

Les représentations simples (bits, octets etc) ne sont généralement pas suffisantes. En effet, un problème informatique particulier nécessite une représentation particulière des données manipulées. Par exemple, si on souhaite manipuler une coordonnées (x,y), on devra agglomérer deux nombres, qui peuvent posséder des plages de valeurs éventuellement différentes. Cette notion se retrouve dans les langages de programmation classiques : la notion de struct en langage C, de record en VHDL, et d'attributs en Java, etc. Toutefois, il est toujours possible de créer un nombre particulier, qui rassemble plusieurs informations. Dans le cas de coordonnées x et y représentées respectivement par un mot de 16 bits et un octet, on peut créer un nombre *pos* par le calcul suivant :

$$pos = (x \ll 8) + y$$

Ce calcul consiste à décaler x de huit positions à gauche, afin de laisser de la place à y : lors de l'addition, y aura exactement 8 emplacements pour que ses 8 bits soient positionnés.

2.5.2 Données symboliques ou énumérées.

Il existe d'autres cas où la représentation sous forme de nombre n'est pas imposée par la nature de l'algorithme. Par exemple, c'est le cas si l'on parle d'un ensemble de 5 couleurs (bleu, blanc, rouge, vert, noir) que peut prendre une variable *c*. Dans ce cas, il faudra passer par un *encodage* des valeurs possibles de la variable : on choisira par exemple 0=bleu, 1=blanc, 2=rouge, 3=vert, 4=noir. Le calculateur numérique devra bien entendu connaître cette règle d'encodage. Généralement, les langages informatiques réalisent cet encodage pour vous. Dans le cas des 5 couleurs, on devra utiliser 3 bits afin de compter jusqu'à 5 :

bleu	→ 0	→ 000
blanc	→ 1	→ 001
rouge	→ 2	→ 010
vert	→ 3	→ 011
noir	→ 4	→ 100

La formule générale stipule que pour n valeurs énumérées, il faut $\lceil \log_2(n) \rceil$ bits de représentation, où le symbole \lceil est une fonction signifiant “entier supérieur ou égal le plus proche de”.

Certaines combinaisons binaires peuvent ne pas avoir de sens (101 par exemple, dans notre exemple). On peut mesurer l'efficacité de la représentation en calculant le ratio entre les bits utiles et ceux inutiles.

<div> <div> <div>b₇</div> <div>b₆</div> <div>b₅</div> </div> <div> <div>b₄</div> <div>b₃</div> <div>b₂</div> <div>b₁</div> </div> </div>					0	0	0	0	1	1	1	1	1
					0	0	0	1	1	0	0	1	1
Bits					0	1	2	3	4	5	6	7	
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p	
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q	
0	0	1	0	2	STX	DC2	"	2	B	R	b	r	
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s	
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t	
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u	
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v	
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w	
1	0	0	0	8	BS	CAN	(8	H	X	h	x	
1	0	0	1	9	HT	EM)	9	I	Y	i	y	
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z	
1	0	1	1	11	VT	ESC	+	;	K	[k	{	
1	1	0	0	12	FF	FC	,	<	L	\	l		
1	1	0	1	13	CR	GS	-	=	M]	m	}	
1	1	1	0	14	SO	RS	.	>	N	^	n	~	
1	1	1	1	15	SI	US	/	?	O	_	o	DEL	

FIGURE 2.1: Encodage ASCII

2.5.3 Code alphanumérique ASCII

Le code ASCII est à part : il s'agit d'un *encodage des caractères*, normalisé au niveau international dans les années 60. Il était important de s'assurer que les futurs échanges numériques recevraient un sens directement interprétable en terme de caractères : lettres, chiffres, caractères spéciaux, etc... Ils sont numérotés de 0 à 127 : seuls 7 bits suffisent à les représenter. Plusieurs caractères ne sont pas affichables. Le code 10₁₀ représente l'échappement (retour à la ligne). Le chiffre 0 correspond au code ASCII 48₁₀ et le *a* minuscule au code 97₁₀. Au niveau binaire, on représente ainsi la lettre *j* :

b7	b6	b5	b4	b3	b2	b1
1	1	0	1	0	1	0

2.5.4 Code alphanumérique UTF-8

Le code UTF-8 peut être vu comme une évolution majeure du code ASCII ; il vise à fournir un "répertoire universel de caractères codés", tout en conservant une retro-compatibilité avec l'ASCII. Les symboles représentables en UTF8 englobent par exemple des caractères issus de langues étrangères (grec, hébreux, hiraganas japonais, etc), voire des musicaux (clé de sol, etc). L'UTF8 est, en 2017, utilisé par près de 90% des sites web. Techniquement, un code UTF8 se présente comme une suite de 1 à 4 octets. Le nombre précis d'octets constitutifs d'un code est lui-même encodé dans les bits de poids fort du code. Par exemple, lorsque le MSB vaut '0', cela signifie qu'un seul octet est nécessaire : cela correspond au code ASCII (il reste donc 7 bits pour coder les 128 caractères représentables en ASCII). Pour les codes au delà de l'ASCII, c'est le nombre de '1' en tête ("leading ones") du premier octet qui indique le nombre d'octets qui suivront : par exemple un code UTF8 qui commence par "11110xxx" indique que le symbole codé nécessite 4 octets. Chaque octet suivant commence par "10".

Définition du nombre d'octets utilisés

Représentation binaire UTF-8	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
1110xxxx 10xxxxxx 10xxxxxx	3 octets codant 12 à 16 bits
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 octets codant 17 à 21 bits

Ce principe pourrait être étendu jusqu'à huit octets pour un seul point de code, mais UTF-8 pose la limite à quatre¹.

Character	Binary code point	Binary UTF-8	Hexadecimal UTF-8
U+0024	0100100	00100100	24
U+00A2	000 10100010	11000010 10100010	C2 A2
U+20AC	00100000 10101100	11100010 10000010 10101100	E2 82 AC
U+24B62	00010 01001011 01100010	11110000 10100100 10101101 10100010	F0 A4 AD A2

2.6 Calcul en virgule fixe et en virgule flottante

2.6.1 Les nombres en virgule fixe

Nous avons déjà vu que les nombres qui possèdent une partie fractionnaire c'est-à-dire qui présentent une virgule après la partie entière peuvent également se représenter en binaire ou dans tout autre base. On utilise simplement les puissances négatives de la base en question.

$$x = \sum_{i=-\infty}^{i=\infty} c_i \cdot r^i, \text{ avec } c_i \in \{0 \dots (r-1)\}$$

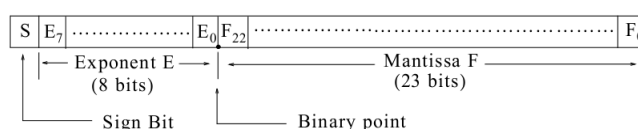
En binaire, ces extensions s'appellent les "décimales binaires" : la première décimale binaire est $\frac{1}{2}$, la seconde est $\frac{1}{4}$, la troisième est $\frac{1}{8}$ et ainsi de suite.

Dans certains processeurs de traitement du signal, on utilise une représentation des nombres dite *en virgule fixe* : on restreint cette extension à une puissance donnée, afin que les opérations se fassent avec une position établie de la virgule fixée, une bonne fois pour toute, ce qui simplifie l'architecture physique de l'unité de calcul. La position de la virgule est alors connue et admise, au point où on peut l'oublier : cela revient à manipuler un simple entier. Par exemple, on peut associer le nombre 125 à une mesure de 1.250 m. Pour une virgule placée au rang k , la différence entre deux nombres consécutifs (appelée résolution) est égale 2^{-k} et sa dynamique (différence entre le nombre le plus grand et le plus petit) est réduite à 2^{n-k} , n étant le nombre de bits du nombre.

Le calcul en virgule fixe est idéal dans le cas de l'addition : la place de la virgule n'est pas appelée à changer dans le résultat. Ce n'est pas le cas de la multiplication : pour deux nombres ayant 1 chiffre après la virgule, on obtient un résultat qui possède 2 chiffres après la virgule. Il en est de même de 2 nombres de 2 chiffres chacun : le résultat est sur 4 chiffres, ce qui peut provoquer des débordements. Enfin, deux nombres de 2 digits présentant une virgule à l'extrême gauche donneront un résultat de multiplication à 4 chiffres après la virgule : il faudra tronquer le résultat... Bref : il est peut-être souhaitable de conserver une liberté dans la gestion des virgules, quand cela est possible.

2.6.2 Les nombres en virgule flottante. Norme IEEE 754

Le calcul scientifique nous amène à des formules qui mélangent différents ordres de grandeur, peu compatibles avec la représentation en virgule fixe. Les ordinateurs possèdent pour la plupart une unité en virgule flottante ; c'est désormais le cas de 100% des ordinateurs de bureau. Seuls certains processeurs embarqués n'en possèdent pas, car l'unité de calcul flottante est consommatrice de surface. La représentation en virgule flottante est normalisée par l'organisme IEEE, responsable de la standardisation de bon nombre de techniques, langages etc. Il s'agit ici de s'assurer que les manipulations flottantes d'un programme ne dépendront pas de l'architecture de la machine : Intel, AMD et. doivent fournir la même représentation. Il existe en réalité deux représentations normalisées IEEE 854 : simple et double précision. On ne s'attarde ici que sur la première représentation.



La représentation des flottants selon la norme IEEE simple précision se fait à l'aide de mots de 32 bits, constitués :

- un bit de signe, dénoté S .
- 8 bits d'exposants (7 à 0), dénoté E
- 23 bits pour la partie fractionnaire, dénotés de 22 à 0. E et F sont toujours représentés en base 2, non complémentés à 2.

2.7 Manipulations numériques

2.7.1 Récupération d'un champ

Revenons à la représentation de la position (x,y) . Bien entendu, lorsque l'on utilise le nombre particulier Pos précédent, il est nécessaire de connaître le mécanisme qui l'a créé, afin de récupérer une coordonnée x ou y :

Listing 2.1: utilisation d'un masque hexadécimal en C

```
y = (0x0000FF) && Coord
```

Ce calcul permet d'extraire y du nombre Coord. C'est un ET binaire, bit à bit, entre Coord et une constante exprimée en hexadécimal. $0x0000FF$ a été créé de manière à annuler la contribution des bits de x dans Coord. Pour cela les 4 digits hexadécimaux sont effectivement à 0 ; ils représentent 16 bits (4 fois 4 quartets) à 0. Quel que soit la valeur x , le ET bit à bit retournera des bits à 0. A l'inverse, les 2 derniers quartets de la constante sont à F et représentent donc 8 bits à 1. En conséquence, sur les 2 derniers quartets résultats, on retrouvera bien forcément la valeur de y , et elle seule, dans le résultat.

2.7.2 Conversion par programmation

Nous avons établi que $C_{2,3}(-2_{10}) = 6_{10} = 110_2$. Comment trouver ces représentations, à l'aide d'un langage de programmation s'exécutant sur votre ordinateur ? On peut écrire :

Listing 2.2: conversion signé vers non signé sur un nombre de bits restreint en langage Ruby

```
x = -2          % valeur a convertir
u = -2 & (2**3-1) % obtention de la valeur non signee. u=6
s = u.to_s(2)   % conversion en chaine de caracteres. s="110"
```

Pour comprendre cette manipulation, il faut savoir que les entiers signés sont codés sur 32 bits. Par conséquent, on peut établir que la valeur -2 est représentée comme : $11\dots110_2$.

La seconde ligne consiste à **masquer** tous les bits au delà du troisième bit. Ces bits sont à 1, car ils représentent le bit de signe.

2.7.3 Autres manipulations au niveau bit

La section précédente nous a permis de récupérer un champ particulier d’une donnée composée, grâce à une manipulation de bits. Il existe en fait de nombreuses manipulations qu’il est utile de connaître, notamment dans le domaine de la programmation pour des systèmes embarqués : les micro-contrôleurs actionnent et agissent sur leur environnement en écrivant et lisant des données précises stockées dans des circuits spéciaux appelés *registres*. Ces registres sont à l’image de l’exemple de la position (x, y) , agglomérant plusieurs informations : par exemple, une alarme peut se manifester dans un bit de ce registre, à une position donnée (ex : le bit 7 du registre “alarmes” peut signifier “porte non-verrouillée”). Il est donc important d’accéder à chacun des bits ou champs entiers de ces registres.

Listing 2.3: Accès aux bits par programme informatique

```
a = a | 0x4; /* mise a 1 du bit 2 de la variable a */
a |= 0x4; /* idem */
b &= ~(0x4) /* mise a 0 du bit 2 */
b &= ~(1 << 2) /* idem, mais plus explicite */
c ^= ~(1 << 5) /* inversion du bit 5 */
e >>= 2 /* division de e par 4 */
```

2.8 Vers la détection d’erreur : bit de parité

Lors de la transmission d’une donnée binaire codée sur plusieurs bits, il est fréquent que l’on lui adjoigne un bit supplémentaire, appelé *bit de parité*. La valeur 0 ou 1 dépend des autres bits, de manière à ce que la somme totale des 1 du code binaire soit paire. Ceci permet, à moindre frais, de s’assurer qu’une erreur (simple) ne s’est pas produite lors de la transmission. En effet, le receptrice peut alors vérifier la parité de la donnée reçue. L’hypothèse sous-jacente stipule qu’une seule erreur binaire est tolérée. Au delà, il faudra rajouter des bits plus sophistiqués...

2.9 Conclusion

Ce chapitre vous a permis de rentrer de plain-pied dans les représentations numériques, notamment avec la prise de connaissance de différents codes. Il existe d’autres codes que nous n’avons pas abordé, comme les codes en excès de 3 où le code de Aiken. Nous nous appuyerons sur ces premières connaissances afin de construire –bientôt– nos premières applications numériques.

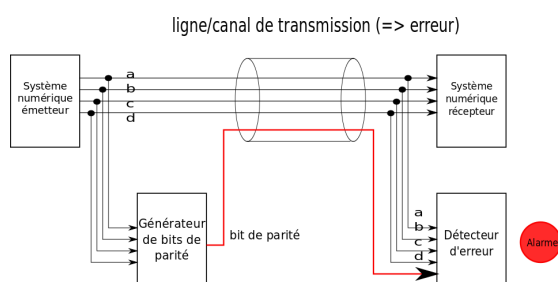


FIGURE 2.2: first figure

Entrées				Sortie
D	C	B	A	P
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0
mot				bit de parité
code				

FIGURE 2.3: second figure

Chapitre 3

Logique booléenne

3.1 Introduction

Nous nous intéressons ici à une algèbre sur les propositions logiques : nous serons à même d'effectuer des calculs sur la logique à 2 valeurs : vrai ou faux. C'est George Boole qui l'introduisit au milieu du XIX^e siècle. Connu pour ses travaux sur les équations différentielles, il l'est encore plus lorsqu'il fait paraître un traité qui fit date, intitulé *"An Investigation Into the Laws of Thought"*, au titre très annonciateur. Son intention sous-jacente était de traduire des idées et concepts en équations, puis d'effectuer des calculs sur la véracité de ces idées : là aussi, on peut rester admiratif devant tant d'intuition ! Pourtant, il semble que ses travaux sont longtemps restés cantonnés à des jeux de salons mondains. A titre d'exemple, voici une énigme qui peut être résolue grâce à l'algèbre de Boole : on dispose de 3 boîtes A, B et C qui contiennent chacune un jeton. Le jeton peut être bleu, blanc ou rouge. Le jeu consiste à déterminer quelle est, parmi les suivantes, l'unique proposition vraie :



1. la boîte A contient le jeton rouge.
2. la boîte B ne contient pas le jeton rouge
3. la boîte C ne contient pas le jeton bleu.

C'est seulement au XX^e siècle que Claude Shannon redécouvrit le pouvoir de modélisation de l'algèbre de Boole et son applicabilité directe au domaine de l'Electronique.

3.2 Définitions

Soit $\mathbb{B} = \{0, 1\}$ l'espace de Boole. Une variable booléenne simple est définie sur \mathbb{B} . Il est également possible de travailler sur plusieurs variables : une variable booléenne **générale** est un n-uplet $X = (x_1, x_2, \dots, x_n) \in \mathbb{B}$ et peut prendre 2^n valeurs. La valeur de X est appelée un point. Un littéral est une variable booléenne simple ou son complément. Par exemple, a et \bar{a} sont deux littéraux distincts correspondant à la même variable.

Les opérations de l'algèbre de Boole sont les suivantes :

- La **Conjonction** (ou produit logique) de deux variables (bits) : on parlera plus volontiers du **ET** logique, dénoté par un signe croix (\times).

- La **Disjonction** (ou somme logique) de deux variables (bits) : on parlera plus volontiers du **OU** logique, dénoté par un signe plus (+).
- La **Négation** (ou complementation ou inversion) d'un seul bit : on parlera plus volontiers du **NON**, dénoté par une barre au dessus de la variable (\bar{a}).

Ces opérations peuvent être représentées par des **tables de vérité**, qui énumèrent explicitement les correspondances entre les entrées et les sorties :

a	b	$a \times b$	a	b	$a + b$	a	\bar{a}
0	0	0	0	0	0	0	1
0	1	0	0	1	1	0	1
1	0	0	1	0	1	1	0
1	1	1	1	1	1	1	0

$x \times y$ vaut 1 si x vaut 1 **et** y vaut 1, 0 sinon.

$x + y$ vaut 1 si x vaut 1 **ou** y vaut 1, 0 sinon.

\bar{x} vaut 1 si x vaut 0, 0 sinon.

Il est à noter que le OU logique n'a pas la même signification ici et dans le langage de la rue : lorsqu'on utilise le OU dans la vie de tous les jours, il s'agit d'une exclusion (c'est l'un e ou l'autre des variables qui est sensée être vraie). On parlera alors de OU *exclusif*, alors que le OU (+) est un OU *inclusif*. Retenez toutefois que le OU *exclusif* est é galement utilisable et utilisé en électronique. Son symbole est généralement le \oplus .

3.2.1 Axiomes de l'Algèbre de Boole

Soit x, y, z des variables booléennes. Les axiomes de l'Algèbre de Boole sont les suivants :

1. **+** est **associatif** : $x + (y + z) = (x + y) + z$
2. **\times** est **associatif** : $x \times (y \times z) = (x \times y) \times z$
3. **+** est **commutatif** : $x + y = y + x$
4. **\times** est **commutatif** : $x \times y = y \times x$
5. **existence d'un élément neutre pour +** : $\exists 0 / x + 0 = x$
6. **existence d'un élément neutre pour \times** : $\exists 1 / x \times 1 = x$
7. **\times est distributif sur +** : $x.(y + z) = x.y + x.z$
8. **+** est **distributif sur \times** : $x + (y.z) = (x + y).(x + z)$. Ce résultat est plus surprenant !
9. **existence du complément**

3.2.2 Principaux théorèmes de l'Algèbre de Boole

1. **+** est **idempotent** : $x + x = x$
2. **\times** est **idempotent** : $x \times x = x$
3. **1 est absorbant pour +** : $x + 1 = 1$
4. **0 est absorbant pour \times** : $x \times 0 = 0$
5. **unicité du complément**

6. **complément du complément** : $\overline{(\overline{x})} = x$

7. $x + x.y = x$

8. $x \times (x + y) = x$

9. $x + \bar{x} \times y = x + y$

3.2.3 Théorèmes de De Morgan

Parmi les théorèmes de l'algèbre de Boole, les théorèmes de De Morgan sont particulièrement intéressants et utiles pour la suite. Observons le tableau suivant, qui expose le calcul de deux fonctions $f_1(a, b) = a + b$ et $f_2(a, b) = \bar{a}.\bar{b}$.

a	b	$a + b$	$f_1 = a + b$	\bar{a}	\bar{b}	$f_2 = \bar{a}.\bar{b}$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Ce calcul démontre que les deux fonctions sont en fait égales :

$$\boxed{a + b = \bar{a}.\bar{b}}$$

On peut également démontrer que :

$$\boxed{\bar{a}.\bar{b} = \bar{a + b}}$$

Ce résultat peut bien entendu se généraliser pour plus de 2 variables. La manière de retenir se résultat consiste à s'apercevoir que "la descente de la barre sur les opérandes" s'accompagne du changement de l'opération "+" en "*" (et vice versa)

3.3 Représentation des fonctions booléennes

3.3.1 Monôme

Un monôme m est un produit de p variables simples distinctes sous formes normales (x) ou complémentées (\bar{x}). Un monôme canonique est un monôme de degré n dans \mathbb{B}^n .

3.3.2 Fonctions booléennes

Une fonction booléenne est une fonction de \mathbb{B}^n vers \mathbb{B}^m .

- $n = 1, m = 1$: la fonction est dite fonction simple d'une variable simple.
- $n > 1, m = 1$: la fonction est dite fonction simple d'une variable générale.
- $n > 1, m > 1$: la fonction est dite fonction générale d'une variable générale.
- $n = 1, m > 1$: la fonction est dite fonction générale d'une variable simple.

Exemple : la fonction $F(a, b) = a + b$ est une fonction booléenne simple d'une variable générale (a, b)

3.3.3 Fonction booléenne incomplète ou ϕ -booléenne

Si F n'est pas définie en $X \in \mathbb{B}^n$, on a l'habitude de poser $F(X) = \phi$, où ϕ représente à la fois 0 et 1 superposés. ϕ représente la valeur indéfinie (en anglais “don't care”), qui signifie que la fonction peut prendre indistinctement la valeur 0 ou 1 en ce point. On dit que F couvre un point X si $F(X) = 1$.

3.3.4 Minterm et maxterms

Nous avons établi qu'une variable booléenne peut apparaître sous sa forme normale ou complémentée. Par ailleurs, le produit de deux variables A et B conduit à $2^2 = 4$ combinaisons possibles : $A.B$, $\bar{A}.B$, $A.\bar{B}$ et $\bar{A}.\bar{B}$. Ces 4 s'illustrent sur un diagramme de Venn. Ces 4 produits sont des *minterms* ou *produits standards*. En d'autres mots, un minterm est un produit composé de deux variables (ou plus) ou de leur compléments.

3.3.5 Décomposition de Shannon et Arbres de décision binaires

La décomposition de Shannon ne sera pas utilisée dans ce cours, mais mérite toutefois d'être évoquée. En effet, elle est à la base des représentations manipulables par les ordinateurs : il faut se projeter dans des systèmes numériques composés de millions d'équations logiques, pour lesquels il est important de manipuler efficacement ces notions ! Cette décomposition s'écrit de la manière suivante :

$$F(x_1, x_2, \dots, x_n) = x_1.F(1, x_2, \dots, x_n) + \bar{x}_1.F(0, x_2, \dots, x_n)$$

Son intérêt est qu'il est possible de l'appliquer récursivement. Cette décomposition s'illustre de la manière suivante, appelée diagrammes (ou arbres) de décisions binaires (BDD : binary decision diagrams) :

Cette décomposition a permis l'essor d'outils de preuves formelles sur les circuits –mais pas seulement– : les “solvers” de satisfiabilité par exemple permettent de trouver s'il existe une variable totale qui rende une formule booléenne vraie (problème SAT). C'est un problème très difficile ¹ !

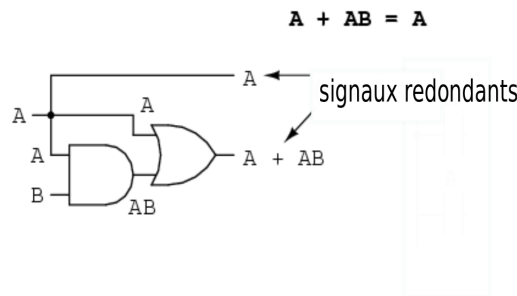
En réalité, le terme de BDD est généralement associé à ROBDD : Reduced Ordered Binary Decision Diagram. L'intérêt des ROBDD est qu'il présente une forme canonique (unique) pour une fonction donnée et quel que soit l'ordre d'évaluation des variables. Cette propriété le rend très utile dans la recherche d'équivalence entre 2 circuits : implémentent-ils intrinsèquement la même formule, bien qu'ils se présentent sous une forme électronique différente ?

3.4 Simplification des fonctions booléennes

3.4.1 Par calcul algébrique

La simplification d'expressions booléennes est un exercice difficile –y compris pour les ordinateurs qui les effectuent–, mais qui est nécessaire pour optimiser le matériel. L'exemple suivant montre un exemple minuscule. Les deux signaux de sortie sont totalement identiques du point de vue logique : en fait, on n'avait besoin d'aucune porte logique !

¹SAT est le premier problème connu dit NP-complet...



Cet exemple suggère également comment simplifier le circuit, par application des règles de composition algébrique vues précédemment. Voici le cheminement que l'on peut utiliser :

1. $A + A \times B$. Factorisons par A
2. $A(1 + B)$. Appliquons l'identité $1 + B = 1$
3. $A \times 1$. Il s'en suit :
4. A

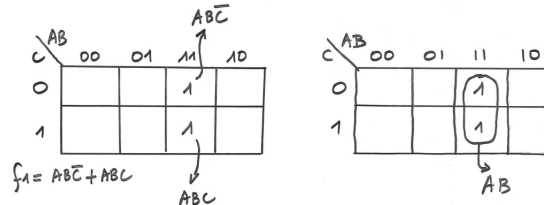
Toutefois, la bonne application successive de ces règles est difficile pour des formules complexes. Cette application dépend de votre capacité à "sentir" la simplification... Il existe heureusement une autre méthode, plus mécanique.

3.4.2 Par tableau de Karnaugh

Les tableaux de Karnaugh permettent d'appliquer de manière mécanique des simplifications qui marchent à coup sûr. Ces tableaux sont en fait un ensemble de cases. En général, une expression booléenne à n variables peut être représentée par un tableau de Karnaugh à 2^n cases, où chaque case représente une ligne d'une table de vérité équivalente. Dans la méthode des tableaux de Karnaugh, il est nécessaire de disposer les entrées du tableau d'une manière particulière : sans cette disposition spéciale, la méthode ne peut s'appliquer. Cette manière particulière consiste à s'arranger afin que des cases contigües ne diffèrent que d'une seule variable. On retrouve ici le code de Gray. Lorsque les entrées sont correctement énumérées, on peut disposer les monômes de la fonction à simplifier dans ce tableau. Il est recommandé d'écrire au préalable la fonction logique sous la forme de sa table de vérité : il ne faudra pas confondre cette table de vérité initiale et le tableau de Karnaugh associé ! Placer les monômes dans le tableau consiste à détecter les '1' de la fonction, dans sa table de vérité, et disposer ces '1' dans le tableau de Karnaugh. Dès lors que ces monômes sont correctement reportés dans le tableau, on cherche des regroupements de '1' : on cherche précisément les regroupements de '1' qui présentent un nombre d'éléments qui soit une puissance de 2 : 2, 4, 8, 16 etc... Ces regroupements doivent être soigneusement entourés. Du fait de l'adjacence des entrées du tableau, ces regroupements induisent une simplification. Plus le nombre d'éléments regroupés est élevé, plus les simplifications seront importantes. Si aucun regroupement n'est possible, cela signifie qu'aucune simplification ne peut s'appliquer.

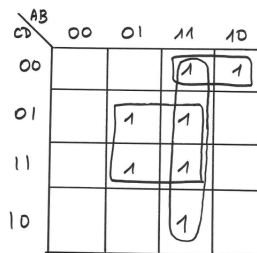
Avant de nous lancer, prenons soin de remarquer que l'adjacence des cases fonctionne également sur les bords des tableaux : le haut et le bas sont adjacents ! Ainsi que la gauche et la droite ! Mieux encore, les 4 angles sont également adjacents. Cela signifie qu'un ensemble de quatre '1' disposés aux quatre angles se prêtent à une simplification booléenne.

Observons un premier cas : simplifions la fonction $f_1(a, b, c) = a.b.\bar{c} + a.b.c$. La simplification algébrique est ici triviale, mais passons par un tableau de Karnaugh. Les deux monômes, disposés dans le tableau, permettent un regroupement de 2 cases. La variable 'c' est ici simplifiable, *du fait de cette adjacence* (le retour à la méthode algébrique le confirme). La fonction vaut donc au final $f_1(a, b, c) = a.b$.



On peut s'autoriser désormais à utiliser cette méthode pour des simplifications moins triviales. Ainsi, la fonction $f(a, b, c, d) = a.b + \bar{a}.b.\bar{c}.d + \bar{a}.b.c.d + a.\bar{b}.\bar{c}.\bar{d}$ vaut, simplifiée $f(a, b, c, d) = a.b + b.d + a.\bar{c}.\bar{d}$

$$F = AB + \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}D + A\bar{B}\bar{C}\bar{D}$$



La bonne nouvelle est que ces simplifications ont été automatisées dans des logiciels. Au cours de l'Histoire, plusieurs méthodes ont vu le jour : notamment l'algorithme de Quine Mac Cluskey, puis des méthodes symboliques, à base de BDD, encore plus puissantes. Toutefois, les algorithmes soulèvent des problèmes de complexité informatique insoupçonnés : il n'existe pas de méthodes qui puisse garantir le caractère optimal du résultat, pour tous les cas !

3.5 Conclusion

Ce chapitre nous a permis de nous familiariser avec l'algèbre de Boole. Il s'agit là de bases essentielles à la mise en équation de problèmes d'électronique numérique. Ce socle solide va nous permettre d'aborder l'ensemble de la conception en Electronique et Informatique (embarquée ou non) de manière sereine. Dans le chapitre suivant, nous allons directement chercher à représenter ces équations issues de l'algèbre de Boole par de véritables circuits électroniques.

Chapitre 4

Circuits combinatoires

Sommaire

4.1	Définition	37
4.2	Portes logiques de base	38
4.3	Fonctions logiques "complexes"	39
4.4	Mapping technologique	40
4.5	Chemin critique et fréquence de fonctionnement	40
4.6	Arithmétique de base	41
4.6.1	Additionneur	41
4.6.2	Soustracteur	43
4.6.3	Additionneur-soustracteur	43
4.6.4	Multiplieur	44
4.6.5	Diviseur	45
4.7	Shifter	45
4.8	Multiplexeur	46
4.9	Comparateur	46
4.10	Codeurs et Décodeurs	47
4.11	Conclusion	48

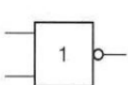
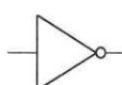
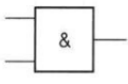
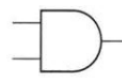
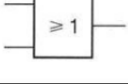
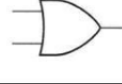
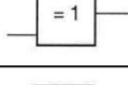





4.1 Définition

Un système numérique est dit combinatoire si toutes ses sorties, à tout instant t , ne dépendent que de la combinaison de ses valeurs d'entrées à un instant $t - \delta$. La définition proposée ici pour les systèmes combinatoires cherche en fait à dire : les *systèmes combinatoires n'ont pas de mémoire d'un état interne passé*, alors que c'est le cas pour les systèmes séquentiels, caractérisé par cet état. Notons que, dans le détail de la physique sous-jacente, les choses sont un peu plus compliquées : on peut en effet noter que, même dans un circuit combinatoire, il existe un effet mémoire lié aux différentes mini-capacités (condensateurs) qui sont susceptibles d'emmagasiner de l'énergie (sur les connexions etc). Toutefois, on sait que si l'on attend suffisamment longtemps, cet effet mémoire s'évanouit...

Il faut donc retenir l'intuition des systèmes combinatoires : après avoir positionné des entrées, et lorsqu'on attend un temps suffisant (le δ de la définition), on verra un résultat en sortie, qui ne dépend que des entrées...

4.2 Portes logiques de base

Le tableau suivant résume les principales portes logiques dont nous ferons usage. Ces portes se présentent sous plusieurs formes équivalentes : graphiques, équationnelles ou tabulées. Il faudra être à même de basculer d'une de ces représentations à une autre. Dans le tableau figurent deux types de symboles : à gauche on trouve les symboles IEC, et à droite les symboles ANSI (dits "américains"). Les symboles IEC ne sont quasiment pas utilisés désormais, et nous conseillons ici vivement de leur préférer les symboles américains. L'IEC reste utilisé dans le domaine du contrôle industriel, qui n'est qu'une infime partie de l'Electronique traitée ici, dans sa généralité.

	Symboles		Table	Equation										
	IEC	ANSI												
Porte NON (NO)			<table><tr><td>entrée</td><td>sortie</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	entrée	sortie	0	1	1	0	$f=\bar{a}$				
entrée	sortie													
0	1													
1	0													
Porte ET (AND)			<table><tr><td>entrées</td><td>sortie</td></tr><tr><td>0 0</td><td>0</td></tr><tr><td>0 1</td><td>0</td></tr><tr><td>1 0</td><td>0</td></tr><tr><td>1 1</td><td>1</td></tr></table>	entrées	sortie	0 0	0	0 1	0	1 0	0	1 1	1	$f=a.b$
entrées	sortie													
0 0	0													
0 1	0													
1 0	0													
1 1	1													
Porte OU (OR)			<table><tr><td>entrées</td><td>sortie</td></tr><tr><td>0 0</td><td>0</td></tr><tr><td>0 1</td><td>1</td></tr><tr><td>1 0</td><td>1</td></tr><tr><td>1 1</td><td>1</td></tr></table>	entrées	sortie	0 0	0	0 1	1	1 0	1	1 1	1	$f=a+b$
entrées	sortie													
0 0	0													
0 1	1													
1 0	1													
1 1	1													
Porte OU exclusif (XOR)			<table><tr><td>entrées</td><td>sortie</td></tr><tr><td>0 0</td><td>0</td></tr><tr><td>0 1</td><td>1</td></tr><tr><td>1 0</td><td>1</td></tr><tr><td>1 1</td><td>0</td></tr></table>	entrées	sortie	0 0	0	0 1	1	1 0	1	1 1	0	$f=a\oplus b$
entrées	sortie													
0 0	0													
0 1	1													
1 0	1													
1 1	0													
Porte NON-ET (NAND)			<table><tr><td>entrées</td><td>sortie</td></tr><tr><td>0 0</td><td>1</td></tr><tr><td>0 1</td><td>1</td></tr><tr><td>1 0</td><td>1</td></tr><tr><td>1 1</td><td>0</td></tr></table>	entrées	sortie	0 0	1	0 1	1	1 0	1	1 1	0	$f=\overline{a.b}$
entrées	sortie													
0 0	1													
0 1	1													
1 0	1													
1 1	0													
Porte NON-OU (NOR)			<table><tr><td>entrées</td><td>sortie</td></tr><tr><td>0 0</td><td>1</td></tr><tr><td>0 1</td><td>0</td></tr><tr><td>1 0</td><td>0</td></tr><tr><td>1 1</td><td>0</td></tr></table>	entrées	sortie	0 0	1	0 1	0	1 0	0	1 1	0	$f=\overline{a+b}$
entrées	sortie													
0 0	1													
0 1	0													
1 0	0													
1 1	0													

Première approche : et, ou, non Les portes logiques de base reflètent directement les opérations élémentaires de l'algèbre de Boole : conjonction, disjonction et négation. Ces portes permettent de construire directement un circuit électronique à partir d'une (ou plusieurs) expression(s) booléenne(s) donnée(s). L'assemblage de telles portes logiques (auxquelles on ajoutera bientôt un élément séquentiel) forme une *netlist* (ou, en français, *liste d'interconnexions*). Dans des cas réels, cette netlist peut contenir plusieurs millions de tels éléments de base. A l'inverse, à partir d'un schéma de netlist, il est possible de revenir à une ou plusieurs équations booléennes. Tout système numérique peut être envisagé comme tel : un vaste ensemble (graphe) de portes logiques.

Deuxième approche : autres portes logiques Dans la pratique, il est utile de pouvoir également disposer de portes logiques comme le XOR (ou exclusif), Nand, Nor à deux entrées, mais également à plusieurs entrées (AND3, AND6 etc). Technologiquement, ces portes complexes sont construites à façon par un *fondeur*. Parmi ces fondeurs, on trouve des sociétés comme STMicroelectronics, Samsung, Texas Instrument, IBM, TSMC, Intel etc. Certaines d'entre elles ne produisent de circuits que pour leur propre fabrication (Intel) tandis que d'autres (TSMC) permettent à une société tierce d'accéder à ces fonderies.

Troisième approche : non-et (nand) Une seconde manière de parler des portes logiques de base est de se pencher sur les propriétés de l'algèbre de Boole. On peut se rendre à l'évidence que les trois fonctions AND, OR et NOT peuvent se calculer à l'aide d'une autre porte logique *unique*, qui devient ainsi la brique élémentaire de tout circuit numérique. Il s'agit de la porte NAND (non-et). Le calcul est également possible avec le NOR, mais il est de tradition de considérer uniquement le NAND. Pour simplifier l'écriture, posons $\overline{a.b} = a \odot b$. On a donc $a \odot 1 = \text{not}(a)$. Prenons la formule $y = a + b$ et exprimons là à l'aide de \odot . Posons $\alpha = a$ et $\beta = b$. On a donc :

$$y = a + b \quad (4.1)$$

$$y = \overline{\alpha.\beta} \quad (4.2)$$

$$y = \alpha \odot \beta \quad (4.3)$$

$$y = (a \odot 1) \odot (b \odot 1) \quad (4.4)$$

De même, on peut démontrer que :

$$y = a.b \quad (4.5)$$

$$y = \overline{\alpha.\beta} \quad (4.6)$$

$$(4.7)$$

d'où :

$$\overline{y} = \overline{\overline{\alpha.\beta}} \quad (4.8)$$

$$= a \odot b \quad (4.9)$$

et :

$$y = (a \odot b) \odot 1$$

Il est donc possible d'exprimer toute la logique booléenne à l'aide du NAND, et ainsi construire n'importe quel circuit à l'aide de la porte logique NAND ¹. Au delà de cette curiosité mathématique, il se trouve qu'en terme de transistors, il est plus facile de réaliser les portes NAND et NOR que des portes non complémentées.

4.3 Fonctions logiques "complexes"

Assemblage de portes logiques En s'en tenant au 3 portes logiques précédentes, on peut connecter les sorties des unes aux entrées des autres de manière à réaliser des fonctions de plus plus complexes, à l'image de la fonction présentée en début de ce chapitre. Cet assemblage ne nécessite pas de précautions particulières, alors cela était nécessaire en électronique analogique (adapation d'impédance etc). Il faudra toutefois veiller à ne pas "chaîner" un "trop" (cela reste à définir...) grand nombre de portes logiques les unes derrière les autres : bien que fonctionnellement correct, un tel assemblage a tout lieu d'affecter la performance finale du circuit : nous allons en parler dans un paragraphe à suivre.

Notion de "Nuage combinatoire" Cet assemblage de portes logiques représente une fonction mathématique qui associe, à plusieurs entrées logiques, plusieurs sorties logiques. Nous allons rapidement créer des fonctions logiques de plus en plus complexes : une fois qu'elles seront établies, *une bonne fois pour toutes*, il n'y aura plus lieu d'en étudier la structure. Très rapidement, nous chercherons donc à *abstraire ces fonctions* : les langages de description matériel comme VHDL nous y aideront notamment, car de simples constructions du langage permettront, en réalité, de

¹Un livre d'introduction aux circuits profite de cette curiosité et s'intitule "Nand to Tertis"

"parler" de ces fonctions. Ainsi la simple addition numérique, constituée d'un grand nombre de porte logiques, sera simplement abstraite par le symbole "+", ce qui est fort pratique ! De même, au lieu de décrire des multiplexeurs, on sera bientôt tenté de les substituer par un "if..then else" du langage. Mais sans en référer à VHDL, il est parfois pratique de réaliser *graphiquement* ces fonctions : il n'est donc pas rare de remplacer la fonction numérique par un simple "nuage". Il s'agit d'un nuage "combinatoire". L'assemblage d'"ordre supérieur" de tels nuages nous permettra de passer une autre échelle et concevoir des circuits encore plus complexes, en appliquant des "patterns" ou "motifs de conception".

4.4 Mapping technologique

Pendant longtemps, les ingénieurs disposaient de circuits *discrets*, se présentant sous la forme d'un boîtier bien caractéristique, avec un nombre de "pattes" (entrées et sorties) très limité (une dizaine). Pour concevoir un système, il s'agissait de sélectionner les bons boîtiers et de réaliser les soudures appropriées pour réaliser la netlist "théorique". A l'heure actuelle (2017), cette sélection se fait par des algorithmes, au sein de programmes complexes (synthétiseurs logiques). Le nombre d'équations logiques, lui-même, dépasse par ailleurs les capacités d'un être humain à les gérer. De plus, ces composants discrets ont quasiment disparu, au profit de composants programmables, sur lesquels nous reviendront. Précisons que, sur ces composants programmables, le mapping technologique garde tout son sens, puisqu'il s'agit encore de "mapper" une netlist théorique sur un ensemble de composants.

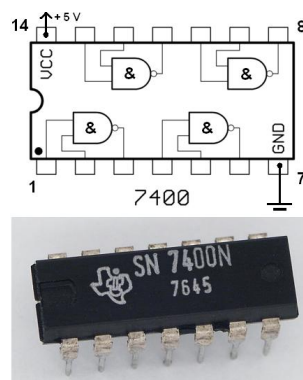


FIGURE 4.1: Exemple de circuit de la (mythique!) série 74

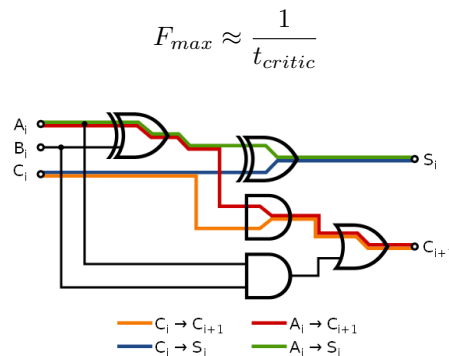
4.5 Chemin critique et fréquence de fonctionnement

La notion de chemin critique est d'une importance capitale. Il permet de calculer la fréquence maximale à laquelle on pourra soumettre un nouveau jeu d'entrées au circuit, sans perturber le calcul précédent. Cette fréquence détermine *in fine* les performances finales de votre ordinateur.

Souvenons nous que, physiquement, nous avons affaire une interconnexion de composants, qui possèdent leurs propres résistance et capacité (et une inductance non significative). Un simple transistor possède un tel modèle équivalent... Un signal électrique se présentant sur une entrée mettra un certain temps à agir sur les sorties du circuit : ce délai est directement relié à la capacité. Si on considère non plus une seule entrée, mais un ensemble d'entrées, la question que l'on peut se poser est la suivante : comment déterminer, parmi ces signaux, celui qui mettra le plus de temps à se propager vers les sorties ? Pour cela, il suffit de parcourir la netlist et détecter le *plus long chemin entre les entrées et les sorties*.

Ce chemin est précisément le chemin combinatoire critique. On fait généralement un abus de langage (toléré!) en associant ce chemin critique avec le *temps de propagation calculé sur ce chemin* : il faut connaître le temps caractéristique de traversée de chaque porte, ainsi que le temps de propagation sur chacun des fils qui les relient. En pratique, des algorithmes calculent ce temps. Dès lors que l'on connaît ce temps, on sait que c'est le laps de temps minimum pendant lequel, le circuit, *dans son intégralité*, ne pourra être utilisé pour un nouveau calcul. Si l'on possédait un oscilloscope suffisamment précis, on pourrait constater, au cours de ce calcul, qu'une très grande agitation règne sur chacun des fils, et que cette agitation semble se propager des entrées vers les sorties, de manière anarchique. Lorsque cette "vague" est passée, les fils de sortie se sont enfin stabilisés : on peut lire la sortie combinatoire. A l'inverse, si au cours de ce temps, on s'ingénie à vouloir utiliser la sortie, elle sera immanquablement fautive et instable.

Pour calculer la fréquence finale, on peut, *en première approximation* prendre l'inverse de ce temps ².



Cette "vague" complexe ne semble guère propice à l'élaboration de systèmes complexes...Fort heureusement, nous serons rapidement en mesure de nous permettre d'"oublier" totalement ces phénomènes microscopiques, pour nous concentrer sur l'essence des services que devra délivrer notre système. Cette "libération" passe par la notion de circuits séquentiels

4.6 Arithmétique de base

Nous abordons ici l'utilisation de portes logiques combinatoire afin d'élaborer des opérateurs de calculs "classiques" : additions d'entiers, multiplications etc. On suppose d'emblée que l'on dispose d'*entrées* et de *sorties* numériques : il s'agit d'un ensemble de fils présentant (chacun) une tension 0 ou 5 Volts (selon la technologie, ces tensions peuvent varier). Pour véhiculer deux opérandes A et B, il faut connaître la dynamique des valeurs que l'on veut traiter pour A et B : à partir de cette dynamique, on dispose d'un ensemble de fils pour A et d'un ensemble de fils pour B, etc.

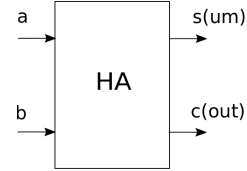
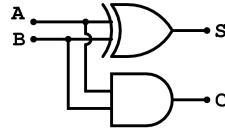
4.6.1 Additionneur

Il existe un grand nombre de manière d'assembler les portes logiques pour réaliser un calcul comme un additionneur. La plus traditionnelle consiste à d'abord étudier le *demi-additionneur* (1 bit), puis l'additionneur 1 bit *complet* (qui s'appuiera sur le demi-additionneur), puis chercher à étendre cet addition à des opérandes à plusieurs bits. Au coeur du procédé, on procède en utilisant le même algorithme que celui enseigné dans notre enfance : on effectue une addition sur les digits les plus à droite, et on continue vers la gauche, en "marquant" d'éventuelles retenues.

²Nous verrons plus loin qu'il faut lui adjoindre deux autres composantes temporelles, liées aux bascules D

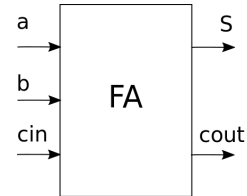
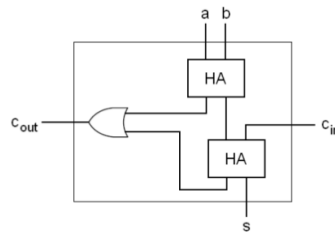
Demi-additionneur 1 bit On parle de demi-additionneur afin de signifier qu'on ne s'intéresse pas, dans un premier temps, à la retenue entrante. Le table de vérité et le circuits logique correspondant sont représentés ici. Les variables s et c dénotent la somme et la retenue (sortante). Ce circuit s'appelle donc un demi-additionneur et est représenté par le symbole "HA" (half adder)

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Additionneur 1 bit On introduit désormais la retenue entrante, afin de réaliser une addition sur n'importe quelle "colonne" de digits. Un calcul permet de nous rendre compte que le demi-additionneur précédent peut être réutilisé. Il s'agit ici de notre premier *assemblage hiérarchique* de circuits.

a	b	c_{in}	S	C_{out}
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



Le calcul évoqué est exposé ici. Au préalable on établit l'égalité suivante où \oplus représente le ou exclusif :

$$\overline{a \oplus b} = \overline{a \cdot \bar{b} + \bar{a} \cdot b} = (\bar{a} + b) \cdot (a + \bar{b}) = \bar{a} \cdot \bar{b} + a \cdot b$$

On a désormais :

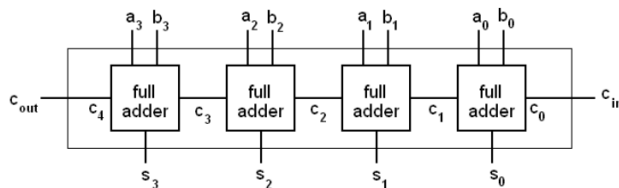
$$S = a \cdot \bar{b} \cdot c_i + a \cdot b \cdot c_i + a \cdot \bar{b} \cdot c_i + a \cdot b \cdot c_i \quad (4.10)$$

$$= a \cdot (a \oplus c_i) + a \cdot (b \oplus c_i) \quad (4.11)$$

$$= a \oplus b \oplus c_i \quad (4.12)$$

De même, on trouve : $c_o = a \cdot b + (a \oplus b) \cdot c_i$. Le demi additionneur peut donc effectivement être ré-utilisé judicieusement, comme dessiné sur la figure précédente.

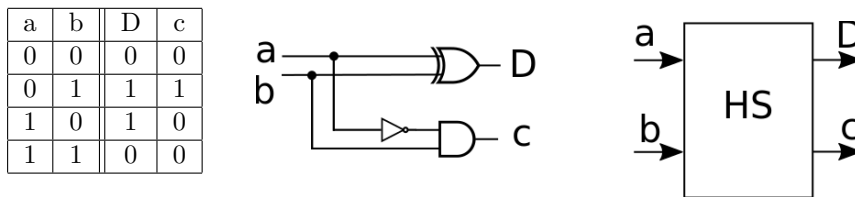
Additionneur n bits La manière la plus naturelle (mais pas la plus efficace!) consiste à chaîner les retenues sortantes et entrante : on parle d'*additionneur à propagation de retenue* ou *Ripple-carry Adder*. Nous présentons ici un additionneur 4 bits avec retenue entrante. En général cette retenue est fixée à 0, mais nous verrons un peu plus loin qu'elle peut également être très utile. En exercice, on peut tenter de déterminer le chemin critique de ce circuit (l'exercice est moins trivial qu'il n'y paraît...).



4.6.2 Soustracteur

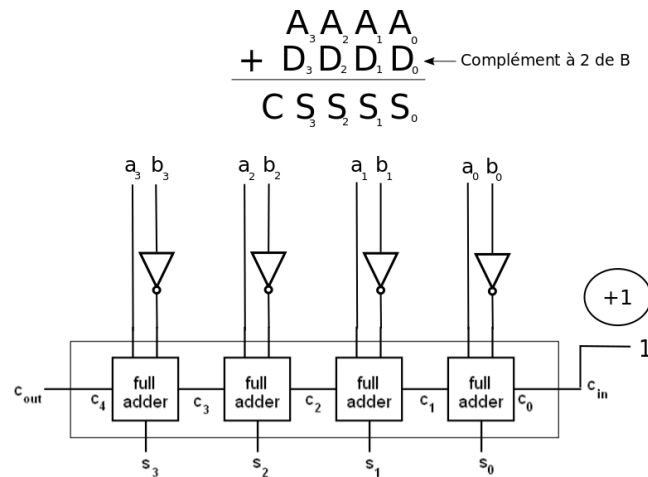
Cherchons désormais les équations du soustracteur logique.

Utilisation d'une table de vérité On procède de même que précédemment, en écrivant la table de vérité du demi-soustracteur (HS), puis aux équations logiques. Sans surprise, ces équations ressemblent de très près à celle du demi-additionneur : en numérique aussi, l'addition et la soustraction sont des opérations similaires. On remarque l'adjonction d'un unique inverseur dans le calcul de la retenue sortante.



On pourrait procéder exactement de la même manière que nous l'avons fait pour établir la structure du soustracteur complet. Toutefois, nous n'allons pas le faire ici, mais passer plutôt à l'utilisation du complément à 2.

Utilisation du complément à 2 On sait en effet qu'algébriquement on a : $a - b = a + (-b)$. On peut donc voir la soustraction comme une addition de deux nombres dont le second est l'*inverse de b*. Attention ! Nous parlons d'"inverse" dans l'algèbre des nombres entiers, mais nous savons désormais (voire le premier chapitre) que $-b$ doit se représenter en complément à 2 en logique booléenne. Il faut donc procéder en inversant les entrées b_i et en ajoutant '1' : cet ajout se fait par la retenue entrante, dont on saisit désormais l'intérêt.



4.6.3 Additionneur-soustracteur

Désormais munis d'additionneurs et de soustracteurs, nous pouvons nous pencher sur d'autres circuits intéressants. Avant d'aborder la multiplication, étudions l'additionneur-soustracteur. Comme son nom l'indique, il s'agit d'un circuit capable de réaliser, à volonté, soit l'addition, soit la soustraction. Il s'agit donc par là de notre premier circuit *programmable*, qui nous rapproche au passage d'un ordinateur classique : selon le programme, considéré comme extérieur au processeur (il est stocké dans une mémoire séparée) qui s'exécute, c'est tantôt tel opérateur numérique qui opère,

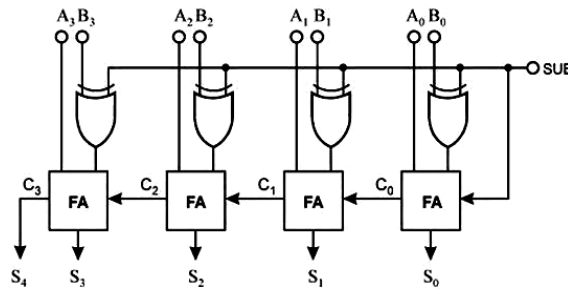
tantôt tel autre. Dans le cas de l'additionneur-soustracteur, on procède par la même observation que la soustraction précédemment étudiée : quelles entrées d_i dois-je fournir à l'additionneur ? Dans le cas où je souhaite réaliser l'addition, on doit avoir $d_i = b_i$ (les a_i restent bien évidemment inchangés), alors que pour la soustraction $d_i = \overline{b_i}$, sans oublier l'introduction d'un '1' sur la retenue entrante. Pour rendre commandable l'opération, on doit disposer d'une entrée supplémentaire, ici appelée "sub" : quand $sub = 1$, le circuit effectuera la soustraction, et l'addition dans l'autre cas. Etablissons les équations logiques des d_i :

b_i	sub	d_i
0	0	0
0	1	1
1	0	1
1	1	0

On voit que l'on a affaire à un magnifique "ou exclusif" !

$$d_i = b_i \oplus sub$$

. Le schéma se réduit donc au suivant. Notons que l'introduction du '1' en retenue entrante correspond à la commande "sub".



4.6.4 Multiplieur

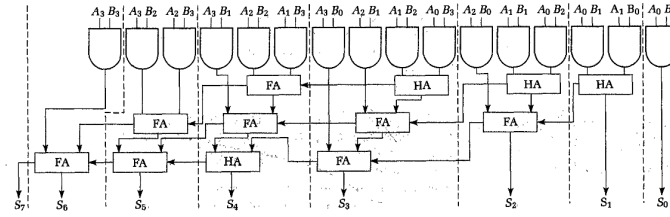
Méthode naturelle Pour implémenter un multiplieur en logique combinatoire, il faut simplement comprendre ce qui se passe lorsqu'on "pose" une multiplication traditionnelle et reporter le principe en respectant la logique booléenne. Le produit de 13 par 11 est donné ici en exemple.

$$\begin{array}{r}
 \text{multiplicand} \quad 1101 \quad (13) \\
 \text{multiplier} \quad * 1011 \quad (11) \\
 \hline
 1101 \\
 0000 \\
 1101 \\
 1101 \\
 \hline
 10001111 \quad (143) \\
 128 + 8 + 4 + 2 + 1 = 143
 \end{array}$$

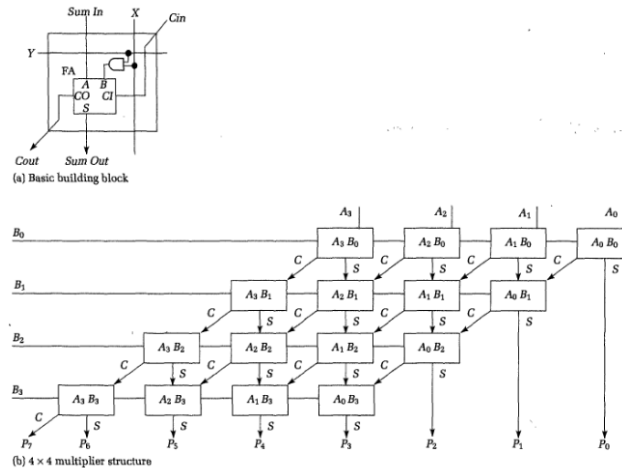
On se convaincra facilement que la multiplication se réduit, ligne par ligne, à une conjonction booléenne (et logique). Ligne à ligne, il faut ensuite additionner soit deux opérandes (demi-additionneur), soit trois lorsqu'une retenue est possible (full adder 1 bit).

$$\begin{array}{r}
 \begin{array}{cccc}
 A_3 & A_2 & A_1 & A_0 \\
 B_3 & B_2 & B_1 & B_0
 \end{array} \\
 \hline
 \begin{array}{cccc}
 A_3 \cdot B_0 & A_2 \cdot B_0 & A_1 \cdot B_0 & A_0 \cdot B_0 \\
 A_3 \cdot B_1 & A_2 \cdot B_1 & A_1 \cdot B_1 & A_0 \cdot B_1 \\
 A_3 \cdot B_2 & A_2 \cdot B_2 & A_1 \cdot B_2 & A_0 \cdot B_2 \\
 A_3 \cdot B_3 & A_2 \cdot B_3 & A_1 \cdot B_3 & A_0 \cdot B_3
 \end{array} \\
 \hline
 \begin{array}{ccccccc}
 S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0
 \end{array}
 \end{array}$$

Etonnamment, le schéma ³ se révèle très irrégulier.



Multiplieur avec topologie régulière On peut chercher à réaliser un multiplieur, en cherchant un circuit élémentaire "brique de base", que l'on aboute savamment, mais de manière parfaitement régulière, en 2 dimensions. Les microélectroniciens apprécient particulièrement ce type de circuits, qui se placent facilement sur silicium. Une tel circuit est donné ici à titre de curiosité.



4.6.5 Diviseur

Le diviseur est un peu plus complexe. On repart là aussi de la division classique. Lors du calcul de A/B , on soustrait le diviseur B de manière répétitive aux bits du dividende A , après l'avoir multiplié par '1' ou '0'. Ce bit de multiplication '0' ou '1' est sélectionné pour chaque étape de soustraction de telle manière que le résultat de la soustraction n'est jamais négatif. Le résultat est composé des bits de multiplication successifs, alors que le reste est le résultat de la dernière soustraction. Cette algorithme combinatoire peut-être implémenté comme une série de soustraction. Chaque soustracteur calcule la différence entre 2 nombres en entrée, mais si le résultat est négatif, l'opération est annulée et remplacée par une soustraction de 0.

4.7 Shifter

Shifter à position fixe *Shift* signifie "décaler" en anglais. L'opération f consiste à décaler, à droite ou à gauche, les bits x_i de l'unique entrée x présentée en entrée. Le décalage d'un nombre fixe Δ de positions est tel que $y_i = f(x_i) = x_{i-\Delta}$. On décale à gauche pour $\Delta > 0$ et à droite sinon. Cette opération sur les seuls indices a la particularité de ne nécessiter aucune porte logique :

³dû à Randu Katz, dont on pourra consulter le livre

il s'agit simplement de prendre un fil d'entrée et de le connecter sur le bon fils de sortie. C'est une économie en silicium conséquente.

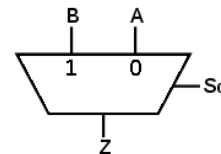
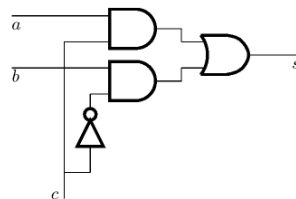
Le shifter à position fixe a notamment la particularité d'être d'une très grande utilité lorsque Δ est une puissance de 2. En effet, décaler à gauche (resp. à droite) d'une position Δ revient à multiplier (resp. diviser) par $\log_2(\Delta)$. Par exemple, pour multiplier un nombre par 2 (resp. par 4, 8, 16 etc), il suffit, en binaire, de le décaler à gauche d'une position $\Delta = 1$ (resp. $\Delta = 2, 3, 4$ etc). Alors que le multiplieur coûte cher en portes logiques, la cas de la multiplication par de telles puissances de 2 ne coûte rien.⁴

Barrel shifter Lorsque le nombre Δ n'est pas fixé, mais devient une entrée dur circuit, il faut être à même de décaler "à volonté". Cette opération plus délicate nécessite un réseau de multiplexeurs.

4.8 Multiplexeur

Le multiplexeur a comme fonction de *router* une de ses entrées e_i vers l'unique sortie f , en fonction d'une commande c . La commande c indique quelle e_i doit effectivement passer. Mathématiquement cela peut s'écrire : $f(\{e_i, i \in 0..n-1\}, c) = e_c$. Il faut donc $\lceil \log_2 n \rceil$ bits pour constituer le mot de contrôle c . Mais calculons simplement sa table de vérité dans le cas de 2 entrées ($a = e_0$ et $b = e_1$) et d'une commande c (sur un seul bit) :

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



Le multiplexeur est un des éléments clés de l'émulation d'une certaine intelligence par du hardware : il permet de filtrer et d'acheminer des données vers un calcul (ou un élément de stockage), selon le calcul de certains critères transformés en commandes de ces multiplexeurs. Ils seront logiquement un élément clé de la notion de *datapaths* ou "chemins de données", constitutifs de la partie calculatoire d'un processeur, l'autre partie étant précisément la partie contrôle, et l'ensemble pouvant être interdépendant (un contrôle pouvant dépendre d'un calcul etc).

Crossbar

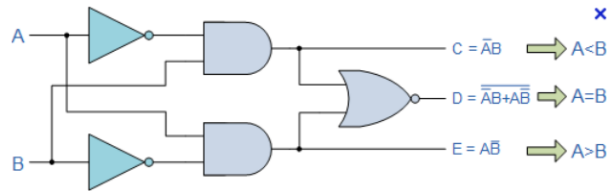
4.9 Comparateur

Le comparateur participe souvent aux décisions évoquées à l'instant.

⁴Dans le jeu d'instruction d'un processeur, cela a-t-il un intérêt ? La réponse est clairement oui, car il faut parfois plusieurs cycles d'horloges pour attendre le résultat d'un multiplieur, afin de laisser au multiplieur (éventuellement totalement combinatoire) le temps d'effectuer son calcul. A l'inverse, le shift est quasi immédiat et ne nécessite donc qu'un seul cycle machine. Nous y reviendrons...

Comparateur 1 bit Etudions tout d'abord le comparateur entre 2 entrées a et b codées sur 1 bit. Calculons les 3 fonctions $<$, $=$ et $>$.

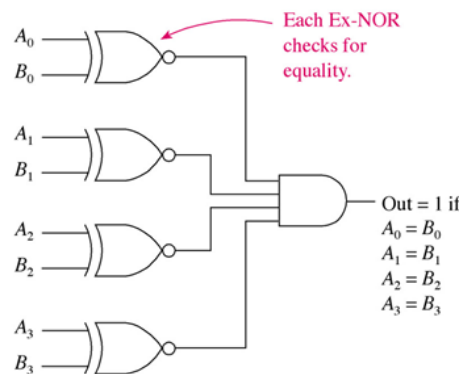
a	b	<	=	>
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0



Comparateur d'égalité de 2 entiers Pour passer à une comparaison d'égalité de 2 entiers a et b codés sur n bits, il suffit de les comparer bit à bit et s'assurer que chacune des comparaisons vaut bien '0'. Cela peut s'écrire :

$$Eq(a, b) = \prod_{i=0}^{i=n-1} a_i \oplus b_i$$

Le circuit dans le cas de $n = 4$ est donné sur la figure suivante.



4.10 Codeurs et Décodeurs

La fonction d'un codeur est la suivante : il transmet en sortie le code d'un symbole placé en entrée. Généralement ce code est optimisé et plus simple à transmettre que le symbole initial. Par exemple, à partir de 3 symboles "bleu", "blanc", "rouge", il est possible de ne transmettre que 2 bits établis au préalable par la fonction de codage : par exemple "00", "01", "10". A l'inverse, le décodeur permettra par exemple de n'allumer qu'une des 3 couleurs "bleu", "blanc" et "rouge" à la réception de 2 bits.

couleur	f_1	f_0
bleu	0	0
blanc	0	1
rouge	1	0

f_1	f_0	bleu	blanc	rouge
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
0	0	0	0	0

Décodeur 7-segments Parmi les décodeurs fréquemment utilisés, on trouve le décodeur 7-segments. Un afficheur 7 segments se présente comme un ensemble de 7 leds a,b,c,d,e,f,g que l'on peut alumer ou éteindre individuellement. Nous calculerons en TD les 7 équations logiques associées pour les nombres hexadécimaux allant de 0 à 15.



4.11 Conclusion

Ce chapitre a permis de nous familiariser avec un certain nombre de fonctions combinatoires usuelles. Leur caractère combinatoire s'est manifesté par le fait qu'à aucun moment, il n'a été nécessaire, au cours des calculs menés, de se référer au passé des signaux en présence. Le calcul s'est toujours réalisé des entrées vers les sorties, sans retour possible. Parmi les fonctions combinatoires intéressantes, l'arithmétique figure en bonne place. Souvenons nous que c'est bien par ces procédés que nous sommes en mesure d'utiliser un matériau (le silicium ici) pour calculer ! Il faut savoir que le domaine de l'arithmétique sur silicium est un domaine toujours actif. Il est notamment toujours à l'honneur dans le domaine de la cryptographie et des télécommunications en général.

Chapitre 5

Circuits séquentiels

Sommaire

5.1	Introduction	49
5.2	Discrétiser le temps	50
5.3	Bascule D	51
5.3.1	Fonction d'échantillonnage de la bascule D. Set-up et hold. Metastabilité.	51
5.3.2	Décalage temporel : la raison d'être de la bascule D	52
5.4	Quelques <i>patterns</i> de conception synchrone	52
5.4.1	Registre à décalage	53
5.4.2	LFSR : linear feedback shift register	53
5.4.3	Bascule D et multiplexeur : mémorisation	54
5.4.4	Compteurs	54
5.5	Initialisation d'une bascule D	54
5.6	Conclusion	56

5.1 Introduction

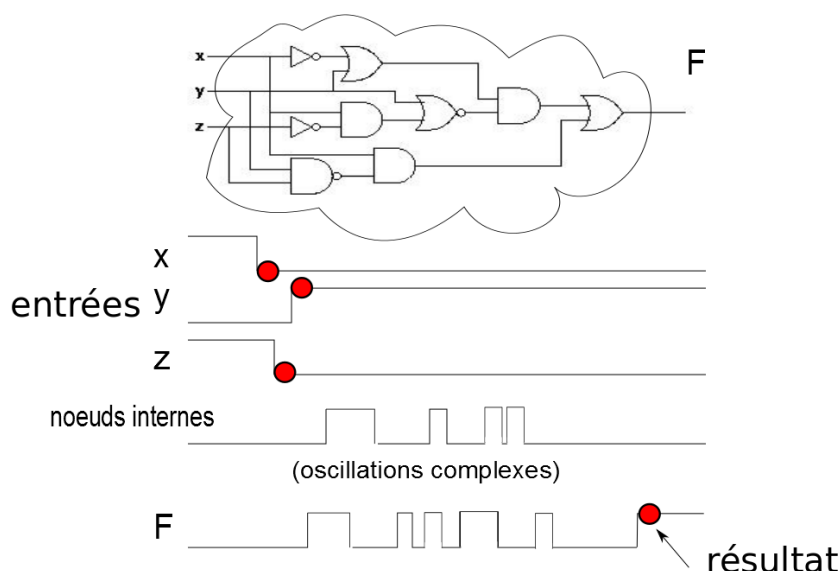
Nous sommes désormais armés pour représenter et manipuler les nombres sous forme binaire (chapitre 2), puis les traiter par une électronique numérique (chapitre précédent). Toutefois, les opérateurs étudiés jusqu'ici se bornaient à des traitements combinatoires. En aucune manière, nous ne nous sommes autorisés jusqu'ici à prendre en compte des valeurs du passé. En particulier, **il nous est interdit de reboucler un signal sur le nuage combinatoire dont il est issu**. Par exemple, il nous est jusqu'ici impossible de réaliser directement un circuit qui réalise une suite numérique comme :

$$u_n = u_{n-1} + 42, n \in \mathbb{N}^+$$

Les circuits que nous allons considérer désormais sont appelés circuits *séquentiels*. Nous nous intéressons ici, précisément à l'indice des suites précédentes : on considère que les indices successifs représentent des instants logiques discrets. Pour parler de la valeur précédente du signal u_n , nous devons nous doter d'un moyen électronique de franchir un nombre de "pas temporels". Par la suite, nous parlerons de "cycles" (ou "coup d'horloge") plutôt que de "pas temporels". Ainsi, dans la suite précédente, il existe un cycle entre u_n et u_{n-1} . C'est donc une seconde discrétisation –celle du temps– qui nous préoccupe ici, après celle qui consistait à discrétiser les valeurs d'un signal. Ces circuits, qui sont à même de se référer à des valeurs du passé, sont des circuits *séquentiels*.

5.2 Discrétiser le temps

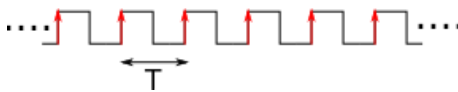
Eviter la "cacophonie" Les circuits combinatoires précédents se présentent sous la forme d'un nuage : ses comportements sont, d'un point de vue microscopique, très complexes à analyser. Ainsi, si l'on disposait d'un oscilloscope suffisamment précis, on constaterait un très grand nombre d'événements (ou "oscillations" ou "changement de valeurs") *internes* fluctuants dans le temps et donc d'une mesure à l'autre.



De même, aux bords du circuit, sur les entrées-sorties, ces événements apparaissent au grè des stimulations extérieures (générateurs de signaux, capteurs, etc), parfaitement *asynchrones* : par ce terme "asynchrone", on indique qu'il n'existe pas a priori de référentiel temporel qui puisse donner un "tempo" commun à tous. Nous cherchons donc ici à trouver un moyen d'éviter cette cacophonie qui semble se profiler, et toutes les tracasseries d'ingénieurs qui lui seraient liées. On observe d'ores-et-déjà qu'il semble suffire d'attendre un temps "suffisamment long" pour que le signaux se stabilisent et que les sorties délivrent leur résultat attendu.

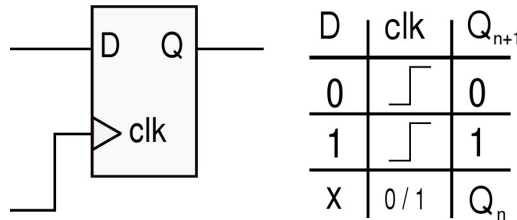
Notion d'horloge périodique Le moyen technique de créer de l'ordre dans la succession des événements d'un circuit est de recourir à une horloge commune. Cette horloge est ici un signal carré, parfaitement périodique, de période T . Là encore, l'analogie musicale s'impose : à la manière d'un chef d'orchestre, l'horloge vise à donner un rythme unique et commun à tous les éléments en présence. Il serait possible de préciser le ratio entre le temps du signal à '1' et à '0', mais comme nous allons le voir ceci n'est pas d'un grand intérêt : en effet, au sein de ce signal carré, nous allons uniquement nous intéresser au *front montant* de cette horloge. Idéalement, ce front montant peut être vu comme instantané. Par analogie avec les mathématiques du traitement du signal, l'ensemble des fronts montants peut être vu comme un *peigne de Dirac* parfait, ou "shah" de Dirac ¹.

¹Toutefois, même si cette référence mathématique est intéressante, elle a ses limites et n'est guère référencée par les Electroniciens. La raison en est simple : en général les traiteurs de signaux font une hypothèse algorithmique forte qui consiste à supposer que le signal est *disponible*, dans son intégralité, au moment du calcul. L'Electronicien quant à lui s'inquiète plus de la manière de capter et traiter ce flux, souvent à la volée (*streaming*)

$$\text{III}(t) = \sum_{n=-\infty}^{n=+\infty} \delta(t - nT_e)$$


5.3 Bascule D

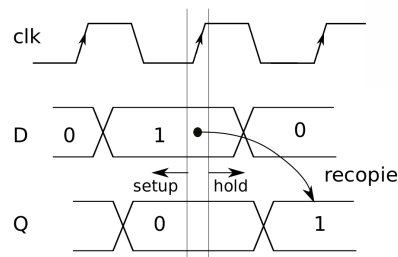
La bascule D est l'élément clé de notre discrétisation du temps. C'est le seul élément électronique sensible au front montant de l'horloge. La bascule D n'est donc pas une "nouvelle porte logique", mais a un status particulier et un fonctionnement particulier. Néanmoins, il est d'usage de présenter sa table de vérité, en analogie avec les portes logiques. Alors que les entrées d'une table de vérité classique font apparaître des valeurs logiques, ici la table fait intervenir le front montant.



La fonction basique de cette bascule D est de réaliser une copie de son entrée 'D' sur sa sortie 'Q'. Cette copie a lieu sur le front montant de l'horloge. Il faut physiquement un très court temps de propagation entre le signal d'horloge et la sortie, appelé "clock-to-Q". Ce temps physique est à prendre en compte à un moment ou un autre, mais n'oublions pas notre but initial de nous "dépolluer" de toute préoccupation d'ordre "physique", pour nous concentrer sur la réalisation mathématique de notre suite numérique (discrète). La fonction de copie semble donc bien modeste. Elle assure pourtant la distinction entre une valeur "avant" et une valeur "après" le front montant d'horloge. En dehors de ce bref instant d'échantillonnage, la bascule D conserve la valeur précédemment échantillonnée.

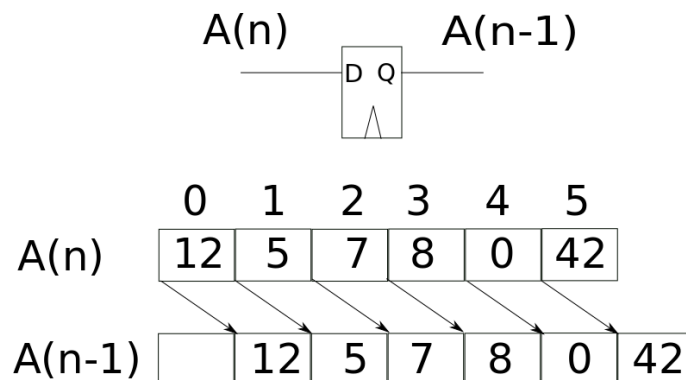
5.3.1 Fonction d'échantillonnage de la bascule D. Set-up et hold. Méta-stabilité.

Une première utilisation intéressante de la bascule D est qu'elle permet d'échantillonner un signal d'entrée qui varie de manière plus rapide que l'horloge. Pour que cet échantillonnage se passe correctement, la bascule doit être utilisée dans des conditions particulières : notamment, il est nécessaire que la signal (numérique) à échantillonner soit *stable* un peu avant et un peu après le front montant. Ces deux temps à respecter s'appelle respectivement le temps de *set-up* et de *hold*. Si par mégarde ou malchance, le signal d'entrée fluctue précisément à cet instant, la bascule entre dans un état particulier dit *métastable*. Cet état ne permet pas de garantir avec certitude que le bon '0' ou le bon '1' apparaîtront en sortie de la bascule. Pire, cette sortie peut même se mettre à fluctuer sans sembler pouvoir se stabiliser.



5.3.2 Décalage tempore : la raison d'être de la bascule D

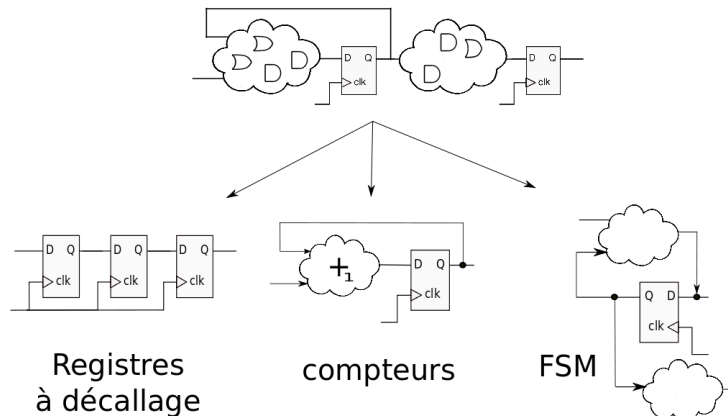
La véritable raison d'être de la bascule est qu'elle permet de décaler un signal d'entrée d'un cycle d'horloge : le signal Q ne pourra en effet être "vu" d'un calcul en aval qu'au cycle d'horloge qui suit.



l'indice n'est jamais explicite
dans le circuit : ce sont les ticks d'horloge

5.4 Quelques *patterns* de conception synchrone

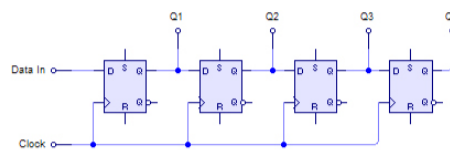
Tout système numérique, simple ou complexe, est une combinaison de portes logiques et de bascules D. Un tel assemblage générique est présenté sur la figure suivante : on observe un enchaînement de traitements combinatoires et de bascule D. Des rebouclages peuvent exister, mais il doit forcément exister une bascule D sur les chemins concernés : aucun chemin bouclé sur lui-même ne peut être purement combinatoire. Ce schéma générique peut être décliné dans une très grande variété de patterns de conception. Le pattern le plus étudié est probablement les automates d'états finis ("FSM" : finite state machine), auxquels nous consacrons le chapitre suivant.



Pour l'heure, on peut citer quelques autres circuits numériques connus, qui font usage de la bascule D ².

5.4.1 Registre à décalage

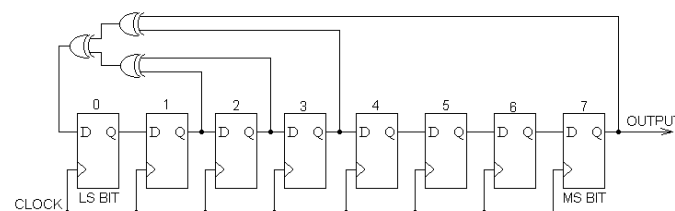
Les registres à décalage sont des assemblages linéaires de bascule D, chaînées les unes à la suite des autres. Lorsqu'une donnée se présente, la précédente est déjà dans le premier étage, et l'avant-dernière dans le deuxième étage etc. A chaque front montant d'horloge, de manière synchrone, toutes les bascules prennent leur entrée et modifient leur sortie Q, *en même temps*. C'est une forme primitive de traitement parallèle. A titre de curiosité, la simulation logicielle d'un tel dispositif n'est pas immédiate ; il faut doubler le nombre de variables et calculer dans un premier temps les *états futurs* de toutes les bascules, puis, dans un second temps, réaliser la mise à jour de toutes les variables. Le décalage synchronisé de ces registres est fascinant et confine à l'harmonie. Toutefois, on ne peut manquer de faire la comparaison avec le travail à la chaîne : le taylorisme, né dans l'industrie de l'automobile s'en est grandement inspiré (voir "les temps modernes" de Charlie Chaplin) !



5.4.2 LFSR : linear feedback shift register

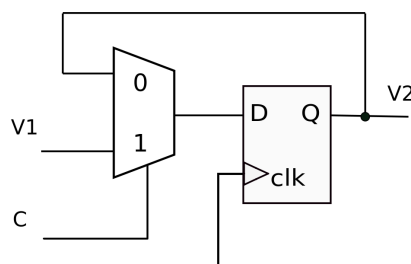
Les LFSR ont l'intérêt de créer de manière simple des suites en apparence aléatoires. En réalité, les suites de valeurs produites sont pseudo-aléatoires et parfaitement récurrentes à partir d'un certain rang. La simplicité des LFSR les rendent malgré tout très attrayants dans de nombreuses situations. Noter la présence d'un ou plusieurs OU exclusifs, qui sont les véritables "créateurs de désordre" au sein de la suite. La place de ces Xor peut être définie à l'aide de polynômes générateurs.

²En Informatique théorique, l'importance des automates est telle que ces circuits électroniques sont vus comme de automates particuliers. L'Electronicien ne fera pas cet amalgame.



5.4.3 Bascule D et multiplexeur : mémorisation

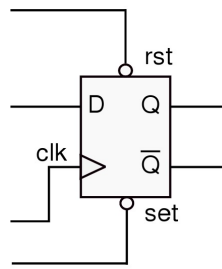
La seule fonction de décalage semble très éloignée des fonctions que l'on peut attendre d'un processeur numérique. De fait, la bascule D est le plus souvent utilisée conjointement à un multiplexeur. Ce multiplexeur (combinatoire) permet de sélectionner à volonté l'action d'*échantillonner* ou *non* : le signal de contrôle du multiplexeur permet effectivement soit de router le signal vers l'entrée D de la bascule, soit de refaire circuler la donnée précédemment stockée dans la bascule, au coup d'horloge précédent. On parle effectivement de *recirculation* de la donnée Q, qui tourne en boucle de la sortie, vers l'entrée etc : la donnée est ainsi piégée. Ce piégeage est une *mémorisation*. Très souvent, ce mécanisme est présenté comme intégré au sein de certaines bascules, tant il est commun : le signal de contrôle du multiplexeur s'appelle alors un *enable*. Lorsque l'enable est à '1', la donnée présentée en entrée est effectivement échantillonnée, sinon, c'est la valeur précédente qui reste "occupe" la bascule.



5.4.4 Compteurs

5.5 Initialisation d'une bascule D

Les symboles précédents des bascules D sont incomplets ; deux signaux importants n'y figurent pas, à savoir : le signal "set" et le signal "reset". Ces signaux interviennent au tout démarrage du circuit, à sa mise sous tension : ce temps particulier s'appelle la phase de *reset du circuit*. A cet instant particulier (qui dure quelques millisecondes), il est nécessaire de placer toutes les bascules D dans un état connu ('0' ou '1'), bien maîtrisé. Cette mise sous tension provient d'une action extérieure (appui sur le bouton de la télé, etc), qui prendra fin pour laisser place au régime permanent. Le signal "set" sert à initialiser la sortie Q de la bascule à '1', tandis que le signal "reset" place cette sortie Q à '0'. Cette prise en compte est *asynchrone* : les signaux "set" et "reset" sont prioritaires sur l'horloge. Après la phase de reset, c'est le régime permanent qui s'établit : l'horloge reprend alors "tous ses droits" et est la seule à commander la valeur de la sortie Q (recopie de D). Ce régime permanent est généralement considérablement plus long que le régime transitoire de reset. Si, pour une raison ou une autre, les signaux de set ou reset sont à nouveau utilisés, ils reprendront instantanément la "main" sur la sortie Q, et imposeront une nouvelle sortie Q. Mais encore une fois, le schéma classique de fonctionnement est : phase de reset suivi d'un très long régime permanent.



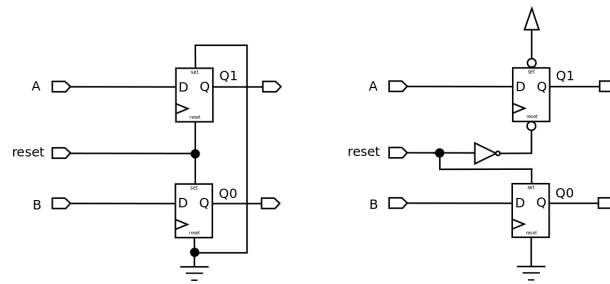
Pour bien mettre en oeuvre la phase de reset, on peut observer les équations d'implication suivantes, qui résument le mécanisme d'initialisation :

- $set = POL_s \implies Q = 1$
- $reset = POL_r \implies Q = 0$

Elles font intervenir un couple (POL_s, POL_r) . POL dénote la *polarité* sur set ou du reset, placé en indice. La polarité indique quelle est la valeur logique active : par exemple si $POL_s = 0$, cela signifie qu'il faudra produire un 0 sur le signal set pendant le démarrage, pour que l'on obtienne l'effet escompté sur Q , à savoir $Q = 1$. Notons que sauf exception, on a $POL_s = POL_r$. La polarité est également signalée sur les schémas par un petit cercle (présent ou absent) près du signal set ou reset de la bascule : la présence du cercle indique une polarité à '0' et '1' en cas d'absence. Lors du régime permanent, il faudra s'assurer que les signaux set et reset n'ont aucune influence sur Q et donc qu'ils sont bien commandés. S'ils sont commandés avec une valeur différente de leur polarité, ils n'auront aucun effet. Lors de la conception, il faut donc s'assurer que l'on relie correctement les boutons de reset à ces signaux, avec les bonnes polarités : par exemple, il faudra se poser la question concernant l'appui sur le bouton : cet événement produit-il une tension à '0' ou à '1' ? Et quelle est la polarité des signaux de reset de la bascule ?

Exemples Nous donnons ici deux exemples (voir schémas) impliquant 2 bascules D et un signal de reset arrivant de l'extérieur. Supposons que l'on cherche à initialiser les bascules dans l'état $Q_1 = 0$ et $Q_0 = 1$. On suppose que le signal de reset est actif haut : lorsqu'on appui sur le bouton de reset, le signal devient '1'. Lorsqu'on le relache, il repasse à '0'. Pour initialiser les bascules, il faut jouer sur les signaux set et reset des 2 bascules : cela fait 4 signaux à connecter de manière appropriée. Aucun signal ne doit rester non connecté (aucun signal "en l'air").

- Dans le cas du schéma de gauche, la solution consiste d'une part à connecter le signal reset à l'entrée reset de la bascule Q_1 et au set de la bascule Q_0 . Les autres entrées des bascules doivent être connectées, même si elles sont inactives : il suffit de les connecter à la masse (0).
- Dans le cas du schéma de droite, il faut adapter la polarité du signal de reset (actif haut) pour activer correctement le reset de la bascule (présence d'un cercle \implies polarité à 0).etc



Simplification grâce aux HDL Le mécanisme de reset peut être perçu comme relativement complexe. Qu'on se rassure : les langages de descriptions matérielle, dont VHDL (étudié plus loin), masque totalement ce travail de fourmi consistant à réaliser ces câblages minutieux.

5.6 Conclusion

Ce chapitre nous a permis de nous doter de l'élément clef des systèmes séquentiels, à savoir la bascule D. Nous avons pris le parti de ne pas "ouvrir le capot" de la bascule D, alors que c'est souvent ce que s'empresse de faire de nombreux manuels. Les raisons en sont multiples : d'une part les HDL reposent uniquement sur la synthèse de bascules D ; d'autre part, sous le capot de ces bascules, plusieurs types de structures peuvent apparaître, la plus connue étant la bascule RS asynchrone. C'est un montage tête-bêche de 2 portes logiques NOR (ou NAND) : la sortie de l'une est câblée sur l'autre et l'ensemble présente un cycle combinatoire que nous avons tenté de totalement exclure (règle de base : pas de boucles combinatoire en conception numérique!). A notre sens, ces deux seules raisons, associées à un temps d'apprentissage court (6 à 7 séances!) autorisent de tels sauts conceptuels.

Chapitre 6

Automates

Machines d'états finis

Sommaire

6.1	Introduction	57
6.2	Machines d'états finis	57
6.3	Diagramme états-transitions	59
6.4	Consistance ou <i>causalité</i> d'une machine d'états finis	59
6.5	Encodage des états	60
6.6	Méthode générale de conception d'un automate	61
6.7	Exemple complet	62
6.8	Gérer la complexité	64
6.9	Méthode particulière : cas de l'encodage <i>one-hot</i>	64
6.10	Conclusion	65

6.1 Introduction

Parmi les circuits séquentiels précédemment évoqués, les automates ont une place privilégiée. Ces automates, appelés également *machines d'états finis* ou encore *FSM (finite state machines)* permettent d'élaborer des circuits très complexes, de manière simple et systématique. Par ailleurs, les automates sont un objet d'étude à part entière pour les informaticiens théoriciens, qui les utilisent comme support à la modélisation de systèmes, y compris lorsque du logiciel prédomine dans ces systèmes : fondamentalement, tout système informatique peut être considéré comme un (très gros) automate. Cette base conceptuelle solide pour les Electroniciens et les Informaticiens l'est aussi – sans surprise – pour les Automaticiens. Les automates sont essentiels aux systèmes embarqués : ils assurent à la fois la notion de séquençement ainsi que les tâches de contrôle de dispositifs. Dans un premier temps, nous reviendrons sur des rappels concernant ces automates. Dans un second temps nous calculerons les équations logiques constitutives de ces FSM.

6.2 Machines d'états finis

Il y a lieu de distinguer deux types d'automates : Moore et Mealy. Très proches conceptuellement, il est toutefois nécessaires de bien comprendre leurs différence pour l'Ingénieur Numéricien. Nous verrons notamment que le couplage de tels automates peut-être source de tracasseries délicates.

Automate de Moore Une automate de Moore est un sextuplet $(Q, \Sigma, \Delta, \sigma, \lambda, q_0)$:

- Q est un ensemble fini d'états, q_0 est l'état initial
- Σ est l'alphabet d'entrée, Δ est l'alphabet de sortie
- δ est une application de $Q \times \Sigma$ dans Q
- λ est une application de Q dans Δ , donnant la sortie associée à chaque état

La sortie de l'automate de Moore en réponse à une entrée $a_1 a_2 \dots a_n$, $n \geq 0$ est $\lambda(q_0), \lambda(q_1) \dots \lambda(q_n)$ où q_0, \dots, q_n est la séquence d'états tels que $\lambda(q_{i-1}, a_i) = q_i$ pour $1 \leq i \leq n$. Remarque : Un automate de Moore retourne la sortie $\lambda(q_0)$ pour toute entrée.

Automate de Mealy Une automate de Mealy est un sextuplet $(Q, \Sigma, \Delta, \sigma, \lambda, q_0)$:

- Q est un ensemble fini d'états, q_0 est l'état initial
- Σ est l'alphabet d'entrée, Δ est l'alphabet de sortie
- δ est une application de $Q \times \Sigma$ dans Q
- λ est une application de $Q \times \Sigma$ dans Δ , donnant la sortie associée à chaque état

$\lambda(q, a)$ donne la sortie associée à une transition d'un état q sur l'entrée a . La sortie de l'automate de Mealy, en réponse à une séquence d'entrées a_1, \dots, a_n est $\lambda(q_0, a_1) \lambda(q_1, a_2) \dots \lambda(q_{n-1}, a_n)$ où q_0, q_1, \dots, q_n est la séquence des états tels que $\lambda(q_{i-1}, a_i) = q_i$ pour $1 \leq i \leq n$.

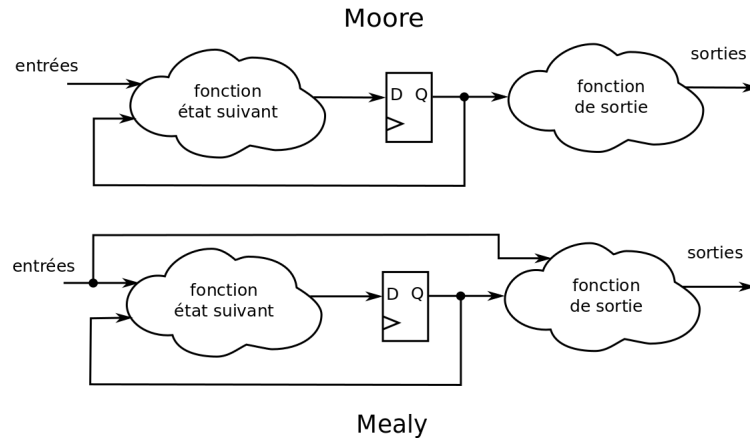


FIGURE 6.1: Automates de Moore et de Mealy

Comparaison Les deux définitions sont très proches l'une de l'autre. On doit seulement comprendre que dans le cas d'une machine de Mealy, les sorties dépendent des entrées et de l'état courant, alors que dans le cas de la machine de Moore, ces sorties dépendent uniquement de l'état courant. En règle générale, les machines de Mealy sont plus rapides : leur chemin critique est plus court. Par contre, elles sont souvent proscrites des bonnes règles de conception –en vigueur dans la plupart des sociétés– car elles peuvent induire des bugs difficiles à localiser. En effet, l'interconnexion de plusieurs automates de Mealy peut présenter des cycles combinatoires. On rappelle que ces cycles (ou boucles) combinatoires sont interdites en logique synchrone car elle ne permettent pas de déterminer la fréquence propre de l'horloge du circuit.

En règle générale, toutefois, on plus code naturellement avec des machines de Mealy. La seule chose à prendre en compte est de bien clore le chemin des sorties combinatoires par un registre adéquat. Cela fait partie des bonnes règles applicables par ailleurs : les entrées et les sorties d'un circuit un tant soit peu complexe doivent être "clockées", c'est-à-dire échantillonnées dans des registres.

Exemple Nous présentons ici un exemple de FSM décrite à l'aide de bascule (ici une seule) et de portes logiques. Les deux fonctions clés y sont représentées à savoir : la fonction d'état suivant et la fonction de sortie. Imaginer, concevoir, analyser et réaliser des automates à ce niveau de détail est sujet à erreur. Nous allons plutôt passer par une représentation plus adéquate, sur laquelle nous allons nous appuyer pour établir les équations. Cette représentation est graphique ; nous l'appellerons "diagramme à bulles" ou diagramme états-transitions.

6.3 Diagramme états-transitions

Dans cette représentation graphique, un état est dénoté par un cercle, alors que les transitions sont représentées par des flèches qui joignent les cercles. Il y a donc un état source et un état transition. Les transitions sont annotées par une expression booléenne qui indique la condition de changement d'état. Concernant les états, on indique généralement le nom de l'état à l'intérieur du cercle (ou toute indications permettant de distinguer les états entre eux). Il existe également un état de démarrage, dénoté par un double cercle (ou parfois cible d'une flèche sans source). L'automate évolue d'un état à l'autre, mais peut également réaliser des actions qui peuvent avoir lieu sur les transitions (Mealy) ou directement dans les états (Moore). Ces actions sont marquées ici après la double barre oblique.

Exemple : vending machine La figure 6.2 présente un automate appelé "vending machine". C'est un automate qui gère la délivrance d'un produit (soda, etc). Il se présente sous la forme d'un diagramme états-transitions à 5 états. L'environnement immédiat de cet automate est un ensemble de capteurs et d'actionneurs. Ces derniers peuvent également se présenter sous la forme d'automates, de telle sorte que c'est un ensemble complexe d'automates communicants qui constitue le système numérique complet. La FSM commande le mécanisme de délivrance par deux signaux "open" et "close". Le mécanisme de délivrance possède également un *timer* interne, non représenté ici qui décompte le temps à partir du moment où un ordre d'ouverture a été donné : lorsque le temps de délivrance (quelques secondes par exemple) s'est écoulé, le signal "top" est à '1'. La FSM centrale est donc également sensible à cette entrée du mécanisme, de sorte que l'ensemble apparait comme bouclé.

6.4 Consistance ou *causalité* d'une machine d'états finis

Pour que la FSM résultante soit considérée comme correcte ou "consistante", on doit examiner les transitions $c_i(s)$ partant de chaque état s :

- **Réactivité** Les conditions sur les transitions doivent être *collectivement exhaustives* : il existe forcément une transition à '1'.

$$\forall s \in S : \sum_i c_i(s) = 1$$

- **Déterminisme** Les conditions sur les transitions doivent être *mutuellement exclusives* : on ne peut pas avoir deux transitions à '1' en même temps.

$$\forall s \in S, \forall (i, j), i \neq j : c_i(s).c_j(s) = 0$$

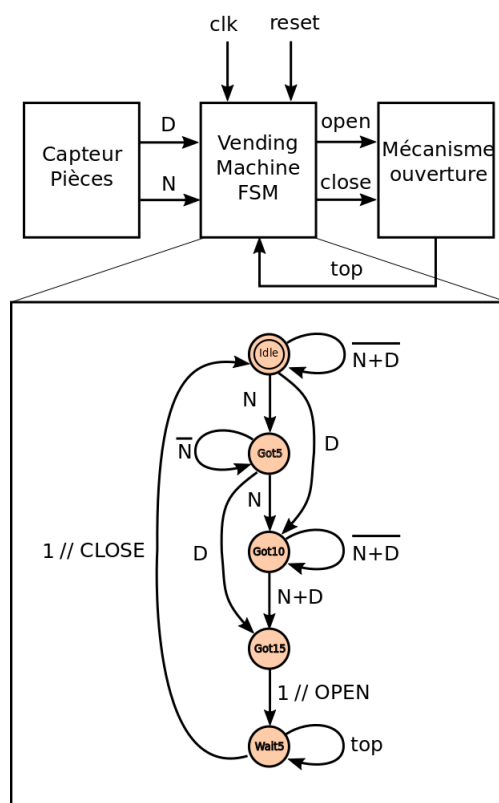


FIGURE 6.2: Vending machine et son environnement

Cela signifie simplement que dans un état, il existe un, et un seul, état suivant. Un tel automate, réactif et déterministe, est dit *causal*. Notons qu'un état peut avoir comme état suivant lui-même : dans ce cas, le système décrit ne change pas d'état, tout simplement.

6.5 Encodage des états

Le diagramme états-transitions est une abstraction graphique qui est d'une grande aide lors de la conception amont. Notamment, on a abstrait les états grâce à une simple bulle et un nom parlant ("symbolique"). Lors de la réalisation concrète en Electronique, il faudra substituer cet état symbolique par un certain nombre de bascules qui "stockent" les bits d'états. Il faut donc établir une correspondance (bijective) entre ces symboles et les codes binaires de ces états. Cette correspondance s'appelle l'encodage des états. Là encore, il existe plusieurs manières de réaliser cet encodage.

Encodage dense L'encodage le plus naturel consiste à numéroter chacune des états en présence, pris dans un ordre arbitraire. La valeur binaire du numéro d'un état est précisément le code de l'état. Cette manière de procéder conduit à une encodage dit "dense". Le terme "densité" utilisé s'explique par le fait que le nombre de bascules résultant est restreint : pour n états symboliques, on obtient $\lceil \log_2(n) \rceil$ bascules. Observons le cas de la FSM "vending machine", et un encodage dense choisi. Pour 5 états distincts, on a ici 3 bits d'états ; la densité est ici toute relative puisqu'avec 3 telles bascules, on pourrait encoder jusqu'à 8 états.

état	numéro	code binaire
Idle	0	000
Got5	1	001
Got10	2	010
Got15	3	011
Wait5	4	100

Encodage one-hot L'encodage *one-hot* ou, en français, "un bit par état", consiste à associer à chaque état d'une FSM de n états symbolique un code de n bits de longueur. Cet encodage est de ce fait bien moins intuitif que l'encodage dense précédent. Les bits d'un code "one-hot" sont particuliers : dans ce code, un seul bit est à '1' (tous les autres à '0'). Un exemple d'un tel encodage est donné dans le tableau suivant.

état	numéro	code binaire
Idle	1	00001
Got5	2	00010
Got10	4	00100
Got15	8	01000
Wait5	16	10000

Compromis temps-surface L'intérêt de l'encodage "one-hot" tient à la simplicité de conception des automates qui en découle. Nous y reviendrons bientôt. De plus les équations logiques des transitions, qui découleront, se révèlent plus simples (moins de portes logiques) et présentent des chemins critiques plus courts que dans le cas de l'encodage dense. Toutefois, étant donné le nombre plus important de bascules D engendrées dans le cas de l'encodage "one-hot", il faudra calculer soigneusement les gains éventuels lors de l'attribution définitive de l'encodage. En effet, on considère qu'une bascule D correspond à environ 10 portes logiques à 2 entrées (typiquement une porte AND). Ce nombre peut notamment se révéler prohibitif en comparaison du nombre de portes engendrées dans le cas d'un encodage dense. De même, signalons que ce seul encodage dense (et d'ailleurs tout encodage) est un choix, qui peut avoir des conséquences insoupçonnées en ce qui concerne la simplification des équations qui en découleront. C'est un problème très complexe, qui a mobilisé un grand nombre d'études théoriques jusque dans les années 1990. Ce calcul est désormais automatisé dans les outils de synthèse, sous la forme d'heuristiques.

6.6 Méthode générale de conception d'un automate

On cherche à établir les équations logiques de l'automate : les équations logiques de la fonction état suivant, ainsi que de la fonction de sortie. On doit établir une *table de vérité séquentielle* qui, à partir d'une présentation de l'ensemble des combinaisons de l'état courant et des entrées, recense toutes les sorties, mais également l'état futur du système. On suggère ici de procéder en 2 étapes.

Table de vérité symbolique La première consiste à conserver le nom *symbolique* des états tandis que la seconde fait apparaître l'*encodage* de ces états.

Table de vérité explicite Cette énumération conduit donc à un nouveau tableau, encore plus explicite, où figurent, à *gauche*, les sorties Q_i des bascules –c'est l'état courant du système– et, à *droite* les entrées D_i de ces mêmes bascules. Pour établir cette table de vérité séquentielle, on énumère toutes les valeurs possibles de la partie de gauche, et on calcule les sorties et les D_i .

état courant	entrée 1	...	entrée n	état suivant	sortie 1	...	sortie n
IDLE							
...							
GOT5							
...							

Q_n	...	Q_0	entrée 1	...	entrée n	D_1	...	D_0	sortie 1	...	sortie n

6.7 Exemple complet

Illustrons notre propos avec la FSM exposée sur la figure. Il s'agit d'un automate de de Moore car on constate que la sortie f a lieu dans les états, et non sur les transitions.

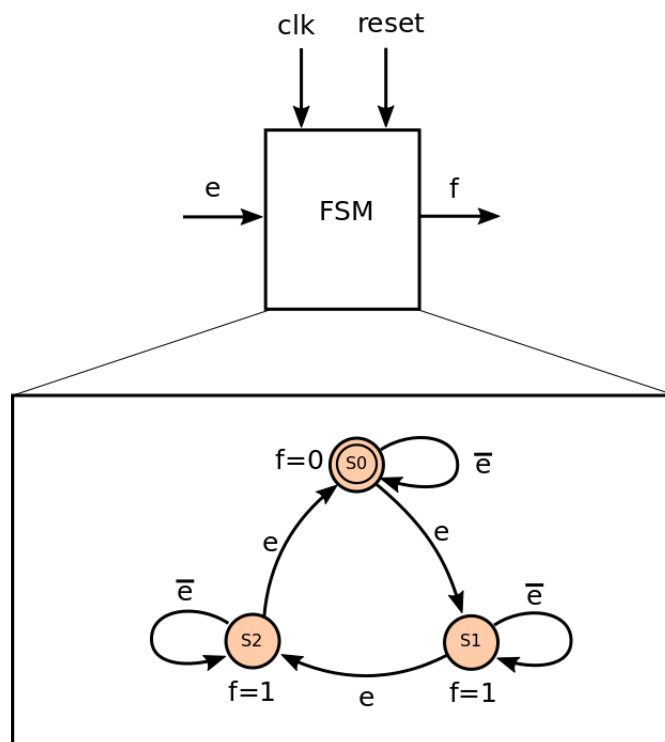


FIGURE 6.3: Tableaux de Karnaugh de l'exemple

Encodage La FSM présente 3 états. Choisissons un encodage dense :

état	numéro	code binaire
S0	0	00
S1	1	01
S2	2	10

Table de vérité symbolique Passons à la table de vérité. On reporte les observations du schéma précédents.

état courant	e	état suivant	f
S0	0	S0	0
S0	1	S1	0
S1	0	S1	1
S1	1	S2	1
S2	0	S2	1
S2	1	S0	1

TABLE 6.1: Table de vérité avec nom explicite des états.

Table de vérité avec encodage explicite Il en découle la table de vérité suivante.

Q_1	Q_0	e	D_1	D_0	f
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	0	1
1	0	1	0	0	1
1	1	0	x	x	x
1	1	1	x	x	x

TABLE 6.2: Table de vérité avec encodage explicite

Tableaux de Karnaugh et Equations logiques On établit les 3 tableaux de Karnaugh, relatif aux équations des 2 bascules et de la sortie. On effectue les regroupements simplificateurs, en profitant des valeurs *don't care* 'X', que l'on peut positionner à '0' ou '1' de manière à créer des regroupements plus intéressants.

D_0

$Q_1 Q_0$ e	00	01	11	10
0		1	X	
1	1		X	

D_1

$Q_1 Q_0$ e	00	01	11	10
0			X	1
1		1	X	

f

$Q_1 Q_0$ e	00	01	11	10
0		1	X	1
1		1	X	1

Les équations obtenues, sont :

$$D_0 = Q_0 \cdot \bar{e} + \bar{Q}_1 \cdot \bar{Q}_0 \cdot e \quad (6.1)$$

$$D_1 = Q_1 \cdot \bar{e} + Q_0 \cdot e \quad (6.2)$$

$$f = Q_1 + Q_0 \quad (6.3)$$

$$(6.4)$$

Circuit électronique Le circuit réalisant l'automate exemple est donné sur la figure 6.4. Notons qu'afin de ne pas surcharger le schéma, nous avons omis les fils relatifs à l'initialisation des bascules D, ainsi que l'horloge.

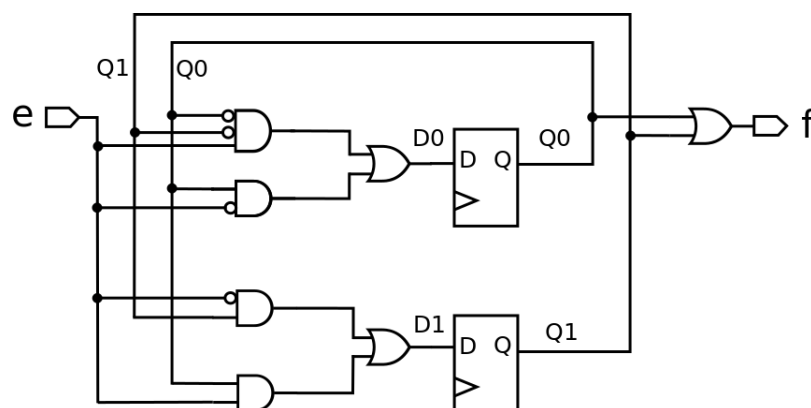


FIGURE 6.4: Implémentation matérielle de l'automate

6.8 Gérer la complexité

Position du problème On se rend compte que le nombre d'entrées dans cette table de vérité grandit très vite : on se souvient alors que pour n entrées, on devra avoir 2^n lignes d'énumérations des combinaisons dans cette table. Cela ne devient plus gérable par un humain. Il faut automatiser. Notons que le nombre précédent 2^n est indépendant du nombre des sorties ; la complexité des équations booléennes de ces sorties croît en fonction de n , ce qui n'est guère rassurant : on pourra avoir 2^n monômes dans ces équations. Bref : il faudra le plus souvent se résoudre à recourir à des synthétiseurs automatiques pour le calcul des équations ou recourir à des stratégies plus intelligentes que le calcul "brute force". Généralement, il est judicieux d'*observer* les relations qui existent entre les contraintes booléennes.

Contraintes mutuelles Ainsi, prenons le simple cas du dé électronique (dé à jouer, à 6 points maximum) : on verra en TD que ce dé peut être modélisé par une FSM. Pour simuler l'effet d'un dé qui est lancé, on appuie sur un bouton. Pendant que le bouton est enfoncé, une FSM allume et éteint les diodes correspondant aux points du dé, en comptant de 1 à 6. Avec une horloge dont la fréquence de fonctionnement est élevée, le joueur ne peut évidemment pas voir ce défilement. Lorsqu'il relâche le bouton, un chiffre est figé sur les points. Logiquement, on s'attend à ce qu'il y ait à établir 6 équations logiques : une équation par diode. On remarque toutefois que tous les points qui s'allument et s'éteignent sur une face du dé sont interdépendants et des "symétries" apparaissent dans les équations. Ce cas est montré sur la figure suivante. Ceci conduit à ne pas traiter 6 équations, mais seulement quelques unes d'entre elles, les autres en découlant facilement. Nous garderons donc en mémoire qu'il faut un peu d'observation et de perspicacité pour traiter de tels problèmes.

6.9 Méthode particulière : cas de l'encodage *one-hot*

Il existe un cas particulier d'encodage qui permet d'éviter totalement la méthode générale précédente : l'encodage *one-hot*. Posons nous la question de la signification de cet encodage : nous savons qu'il est appelé "un bit par état". Cela signifie qu'à un instant donné, une seule bascule aura une sortie Q_i à '1' ; nous connaissons donc instantanément, du fait de cette seule observation, dans quel état on se trouve. Nullement besoin d'observer les autres valeurs (à '0') des autres bascules. On dit que la bascule en question est "allumée" lorsque sa sortie vaut '1'. Si on observe désormais le fonctionnement de l'automate durant plusieurs coups d'horloges, on verra successivement s'allumer différentes bascules, *l'une après l'autre*. On peut par conséquent utiliser

les propriétés de cet encodage pour établir une correspondance *directe* entre le schéma initial et le circuit correspondant. Dit autrement : il suffit d'observer le diagramme états-transitions de l'automate pour construire directement le circuit ! Pour cela, il suffit d'établir des règles simples :

- On dessine une bascule D pour un état S
- Pour chacune des transitions sortantes d'un état S_i vers un état S_j , on dessine une porte AND connectée d'une part à la sortie Q de la bascule de l'état S_i , et d'autre part la condition booléenne c de la transition $S_i \rightarrow_c S_j$. Appelons $c_{i,j}$ la sortie de cette porte AND.
- Pour chacune des transitions entrantes d'un état S_j on dessine une porte OR connectée à l'entrée D_j et possédant autant d'entrées qu'il existe de transitions entrantes. A partir des états précédents S_i , on connecte le signal $c_{i,j}$ sur une des entrées de cette porte OR.

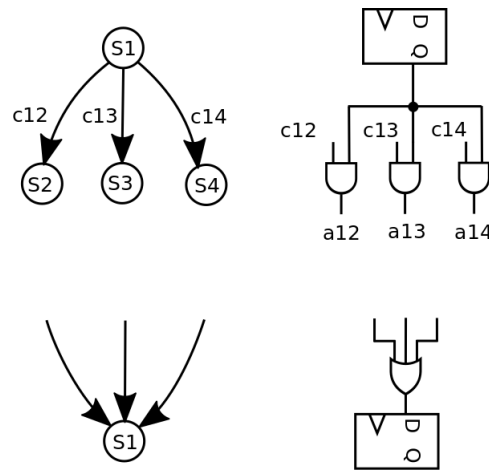


FIGURE 6.5: Procédé simplifiant la génération d'un circuit d'automate dans le cas d'un encodage *one-hot*.

Ce schéma de traduction est très simple à mettre en oeuvre. Il est résumé sur la figure ?? . On peut ne pas réfléchir en dessinant le circuit solution. A posteriori, on peut regarder le schéma et réécrire les équations logiques des entrées D_i .

6.10 Conclusion

Ce chapitre nous a permis de découvrir la notion d'automate matériel. L'implémentation matérielle de tels automates peut être réalisée de manière quasi-mécanique, en mobilisant nos connaissances en matière de logique combinatoire, tableaux de Karnaugh, mais également bascules D. Nous avons également abordé la notion de consistance de ces automates : un calcul formel permet de s'assurer que notre automate a du sens. Ce calcul se fait au préalable de la réalisation concrète du circuit. C'est là une propriété intéressante, liée à la logique booléenne, que l'on ne retrouve guère en matière logiciel. Là, le domaine du matériel semble en avance sur le logiciel : notons toutefois qu'une classe de langages informatiques, dédiés à la réalisation de logiciels embarqués, s'appuie sur les mêmes calculs et les mêmes raisonnements formels pour assurer la fiabilité des logiciels embarqués. Ces langages s'appellent *langages synchrones* : on comprend désormais bien cette appellation ! Je vous invite notamment à lire, voir ou écouter les conférences de Gérard Berry, à l'Académie des Sciences. Il illustre parfaitement, meixu que quiconque, ce propos.

Chapitre 7

Langages de description matérielle

L'exemple de VHDL

Sommaire

7.1	Introduction : les langages de description matérielle	67
7.2	La structure globale d'un programme VHDL	68
7.2.1	Notion d'entité	68
7.2.2	Notion d'architecture	68
7.3	Les éléments clés du langage	68
7.3.1	Assignations concurrentes	68
7.3.2	Processus	68
7.3.3	Instanciation de composants	68
7.3.4	Décrire des machines d'états finis	68
7.4	Simulation en VHDL	68
7.4.1	Flot de conception général	68
7.4.2	Bancs de tests ou <i>Testbench</i>	68
7.4.3	Algorithme de simulation	68
7.5	Synthèse matérielle	68
7.5.1	Exemple du flot Xilinx	68
7.6	Conclusion	68

7.1 Introduction : les langages de description matérielle

LA Silicon Valley est née de l'essor qui a suivi l'invention du transistor (1947). Diverses sociétés concevant des circuits à base de semi-conducteurs ont commencé à échanger des informations concernant la structure et le comportement temporel de circuits. On a rapidement compris que les simples dessins ne suffisaient pas : la description des circuits requière des langages spécifiques. Il s'agit des HDL ou "hardware description languages". Au niveau analogique, c'est SPICE qui s'est imposé : il permet de décrire de assemblages de composants tels que des diodes, résistances, capacités etc sous la forme d'une netlist textuelle. Très rapidement, la nécessité d'un format commun de *description* des circuits numériques s'est également imposée. Le DARPA (Département à la Défense américain) a initié en 1981 la création de VHDL. IBM, Intermetrics

et Texas Instrument ont pris le projet en main. La syntaxe de VHDL s'est fortement inspirée d'ADA, un langage informatique généraliste et très prometteur à l'époque. C'est finalement en 1987 que l'IEEE publie un premier standard mondial, amendé en 1993. Différentes évolutions ont eu lieu depuis, et des groupes de travail continuent de s'activer autour de l'évolution du langage. L'ambition du langage –initialement cantonné à la description de circuits– s'est étendue : il a été rapidement possible de simuler, puis de synthétiser des circuits décrits en VHDL. En parallèle, un autre langage aux buts similaires est né : Verilog, inventé pour la simulation logique par la société Gateway Design Automation, puis commercialisé par la société Cadence, qui reste un des leaders mondiaux de l'EDA (Electronic Design Automation). Verilog est également un standard IEEE désormais. A noter que plus tard, d'autres langages de description matérielle ont vu le jour : on pense notamment à SystemC, qui élève le niveau d'abstraction au dessus de ce qu'il est possible d'écrire en VHDL ou Verilog. Ce domaine de la description et de la modélisation de systèmes numériques reste très actif : ces outils de simulation et de synthèse sont la clé de voûte de toute l'Industrie du Semiconducteur.

7.2 Particularités de VHDL

VHDL est un langage très exigeant en terme de syntaxe, réputée verbeuse (mais explicite) mais également en terme de "sémantique" : outre le fait qu'il est fortement typé, il possède en réalité 2 sémantiques très différentes. D'une part, il sert à la simulation sur ordinateur : la signification d'un programme VHDL est donc imposée par ce simulateur, qui doit représenter le parallélisme final du circuit à l'aide d'un algorithme séquentiel. D'autre part, il sert à la synthèse de circuits qui existeront in fine, sur silicium. Le concepteur en VHDL doit donc en permanence "jongler" avec ce qui est limité à la simulation et ce qui devra subir le flot de synthèse. La prise en compte de cette dualité nécessite beaucoup d'expertise.

7.3 La structure globale d'un programme VHDL

7.3.1 Notion d'entité

7.3.2 Notion d'architecture

7.4 Les éléments clés du langage

7.4.1 Assignations concurrentes

7.4.2 Processus

7.4.3 Instanciation de composants

7.4.4 Décrire des machines d'états finis

7.5 Simulation en VHDL

7.5.1 Flot de conception général

7.5.2 Bancs de tests ou *Testbench*

7.5.3 Algorithme de simulation

7.6 Synthèse matérielle

7.6.1 Exemple du flot Xilinx

7.7 Conclusion

Chapitre 8

Thèmes avancés

Sommaire

8.1	Synthèse comportementale ou HLS	69
8.2	Modèles de calculs	69
8.3	Langages synchrones	69

8.1 Synthèse comportementale ou HLS

8.2 Modèles de calculs

8.3 Langages synchrones