Ensta Bretagne
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00

Polycopié

UV 1.5

14 décembre 2017

Electronique Numérique

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

Avant propos

Ce fascicule propose une introduction à l'Electronique Numérique aux étudiants de première année de l'ENSTA Bretagne. Dans un temps très contraint (moins de 10 séances), nous abordons des notions aussi fondamentale que la représentation des données numériques (les nombres, mais pas seulement...), la logique Booléenne, les portes logiques élémentaires, les circuits séquentiels synchrones, etc. Nous avons conservé un premier chapitre consacré à des rappels de physique, qui aboutissent à l'idée du transistor en commutation : il est là moins pour donner des bases de Micro-électronique, que pour simplement *évoquer* l'idée d'un continuum dans l'Histoire de l'Electronique et l'avènement des techniques. Il permet également d'insister sur la montée en abstraction en Electronique : elle est illustrée à travers un schéma emprunté à Alain Guyot de l'INPG.

A l'inverse, nous apportons un soin particulier lors de la découverte des machines d'états finis dont on connaît l'importance dans le domaine des systèmes embarqués, y compris lorsque ceux-ci sont à "logiciels prépondérants" : nous sommes d'emblée dans la capacité de modéliser des systèmes numériques et d'en dériver mécaniquement les équations constitutives. Au passage, nous prenons le temps de définir la notion de causalité dans de telles machines : il s'agit là d'une première intrusion de techniques formelles, dans un univers informatique qui en parait – pour l'étudiant pythoniste– le plus souvent dépourvu.

Nous prenons enfin le parti de profiter de ce cours pour introduire le langage VHDL, support incontournable de la conception de circuits numériques : simulation et synthèse sur FPGA sont abordées. Le simulateur GHDL retenu est open-source : il s'utilise naturellement en ligne de commande et permet selon nous de banaliser de telles simulations. Enfin, lors d'une ultime séance de travaux pratiques, nous synthétisons un petit circuit sur FPGA, à l'aide de scripts TCL. Là encore, notre souhait est à la fois de faire acquérir aux élèves des notions de base, mais également de lever un coin du voile sur un domaine en plein essor. Toutes ces notions seront bien entendus approfondies en deuxième année.

Table des matières

1 Le transistor **9**
1.1 Introduction . . . 9
1.2 Silicium et dopage . . . 9
1.3 Jonction PN . . . 11
1.4 Transistor MOS . . . 11
1.5 MOS complémentaires . . . 13
1.5.1 Principe . . . 13
1.6 Niveaux d'abstraction . . . 13
1.7 Conclusions . . . 13

2 Représentations numériques **17**
2.1 Notion de bases ou radix . . . 18
2.1.1 Décomposition en puissance de la base . . . 18
2.1.2 Conversion entre bases . . . 18
2.1.3 Exemples . . . 19
2.1.4 Conversions binaire, hexadécimal, octal . . . 20
2.1.5 Bits, bytes, nibbles, words . . . 20
2.2 Représentation des entiers négatifs. . . . 21
2.2.1 Complément à 2 d'un nombre . . . 21
2.2.2 Astuces pour complémenter à 2 . . . 22
2.2.3 A propos du bit de signe . . . 22
2.2.4 Conversion d'un nombre signé vers un nombre non signé . . . 23
2.3 Représentations alternatives, obsolètes ou rares . . . 23
2.3.1 Représentation en Magnitude signée . . . 23
2.3.2 Représentation en complément à 1 . . . 23
2.3.3 Code BCD et code de Gray pour les entiers non-signés . . . 23
2.4 Calcul d'addition et soustraction . . . 25
2.4.1 Addition . . . 25
2.4.2 Soustraction . . . 25
2.4.3 Petit complément sur la soustraction . . . 26
2.5 Données composées ou symboliques . . . 27
2.5.1 Données composées . . . 27
2.5.2 Données symboliques ou énumérées. . . . 27
2.5.3 Code alphanumérique ASCII . . . 28
2.5.4 Code alphanumérique UTF-8 . . . 28
2.6 Calcul en virgule fixe et en virgule flottante . . . 29
2.6.1 Les nombres en virgule fixe . . . 29
2.6.2 Les nombres en virgule flottante. Norme IEEE 754 . . . 30
2.7 Manipulations numériques . . . 30
2.7.1 Récupération d'un champ . . . 30

2.7.2 Conversion par programmation . . . 30
2.7.3 Autres manipulations au niveau bit . . . 31
2.8 Vers la détection d'erreur : bit de parité . . . 31
2.9 Conclusion . . . 31

3 Logique booléenne **33**
3.1 Introduction . . . 33
3.2 Définitions . . . 33
3.2.1 Axiomes de l'Algèbre de Boole . . . 34
3.2.2 Principaux théorèmes de l'Algèbre de Boole . . . 34
3.2.3 Théorèmes de De Morgan . . . 35
3.3 Représentation des fonctions booléennes . . . 35
3.3.1 Monôme . . . 35
3.3.2 Fonctions booléennes . . . 35
3.3.3 Fonction booléenne incomplète ou ϕ-booléenne . . . 36
3.3.4 Minterm et maxterms . . . 36
3.3.5 Décomposition de Shannon et Arbres de décision binaires . . . 36
3.4 Simplification des fonctions booléennes . . . 36
3.4.1 Par calcul algébrique . . . 36
3.4.2 Par tableau de Karnaugh . . . 37
3.5 Conclusion . . . 38

4 Circuits combinatoires **39**
4.1 Définition . . . 39
4.2 Portes logiques de base . . . 40
4.3 Fonctions logiques "complexes" . . . 41
4.4 Mapping technologique . . . 42
4.5 Chemin critique et fréquence de fonctionnement . . . 42
4.6 Arithmétique de base . . . 43
4.6.1 Additionneur . . . 43
4.6.2 Soustracteur . . . 45
4.6.3 Additionneur-soustracteur . . . 45
4.6.4 Multiplieur . . . 46
4.6.5 Diviseur . . . 47
4.7 Shifter . . . 47
4.8 Multiplexeur . . . 48
4.9 Comparateur . . . 48
4.10 Codeurs et Décodeurs . . . 49
4.11 Conclusion . . . 50

5 Circuits séquentiels **51**
5.1 Introduction . . . 51
5.2 Discrétiser le temps . . . 52
5.3 Bascule D . . . 53
5.3.1 Fonction d'échantillonnage de la bascule D. Set-up et hold. Metastabilité. 53
5.3.2 Décalage temporel : la raison d'être de la bascule D . . . 54
5.4 Quelques *patterns* de conception synchrone . . . 54
5.4.1 Registre à décalage . . . 55
5.4.2 LFSR : linear feedback shift register . . . 55
5.4.3 Bascule D et multiplexeur : mémorisation . . . 56
5.4.4 Compteurs et timers . . . 56
5.5 Initialisation d'une bascule D . . . 57

5.6 Conclusion 58

6 Machines d'états finis **59**
6.1 Introduction 59
6.2 Machines d'états finis 59
6.3 Diagramme états-transitions 61
6.4 Consistance ou *causalité* d'une machine d'états finis 61
6.5 Encodage des états 62
6.6 Méthode générale de conception d'un automate 63
6.7 Exemple complet 64
6.8 Gérer la complexité 66
6.9 Méthode particulière : cas de l'encodage *one-hot* 66
6.10 Conclusion 67

7 VHDL **69**
7.1 Avant propos 69
7.2 Introduction : les langages de description matérielle 70
7.3 La structure globale d'un programme VHDL 71
7.3.1 Déclaration de librairies et packages 71
7.3.2 Notion d'entité 72
7.3.3 Notion d'architecture 72
7.4 Les éléments clés de l'architecture 73
7.4.1 Assignations concurrentes des signaux 73
7.4.2 Processus 73
7.4.3 Instanciation de composants et d'entités 75
7.5 Décrire des machines d'états finis 77
7.5.1 FSM au niveau logique 77
7.5.2 FSMs au niveau RTL 77
7.6 Simulation en VHDL 79
7.6.1 Flot de conception général 79
7.6.2 Bancs de tests ou *Testbench* pour la vérification 80
7.6.3 Modèles de référence 80
7.7 Fonctionnement interne d'un simulateur VHDL 81
7.8 Utilisation du simulateur GHDL 82
7.8.1 Introduction 82
7.8.2 Commandes essentielles 82
7.9 Conclusion 83

8 Synthèse sur FPGA **85**
8.1 FPGA : un circuit reconfigurable 85
8.2 Flot de synthèse 87
8.3 Expérience pratique 87

Chapitre 1

Le transistor CMOS
La naissance des 0 et 1

1.1 Introduction

Ce chapitre vise à présenter le transistor CMOS de manière très succinte. Plus précisément, nous cherchons à illustrer les étapes successives qui permettent de passer du Silicium, l'élément chimique de la table périodique des éléments, à de véritables interrupteurs miniaturisés. Nous passerons sous silence les capacités d'amplification de ces transistors, ainsi que les équations fondamentales qui en régissent le fonctionnement. On pourra se reporter à différents ouvrages comme [8] pour plus de détails, y compris concernant les *process* de fabrication, également survolés ici.

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 réseau les atomes de silicium se situent au sommet d'un cube, ainsi que sur chacune des face de ce cube (il est dit "cubique face centrée").

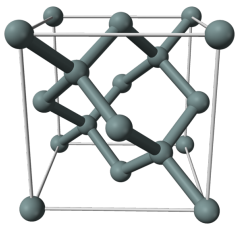


FIGURE 1.1: La maille élémentaire du cristal de silicium

Ses propriétés physico-chimiques en fond 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.

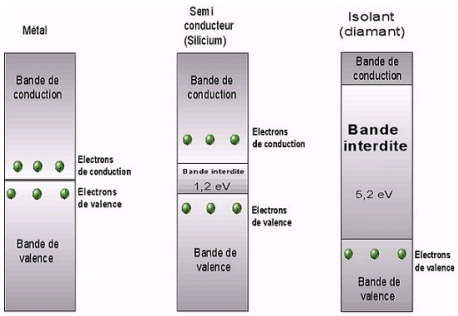


FIGURE 1.2: Bande de valence et de conduction de différents types de matériaux

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 à1012 fois plus nombreux).

Le fait d'introduire en très faible quantité des impuretés (opération appelée dopage) dans

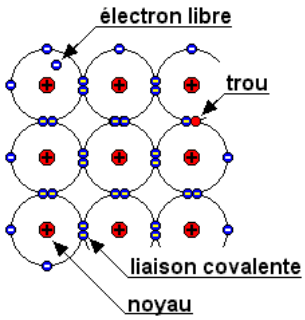


FIGURE 1.3: Présence des différents porteurs de charge

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 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**.

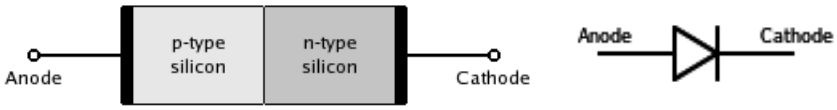


FIGURE 1.4: Jonction PN ou diode

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.

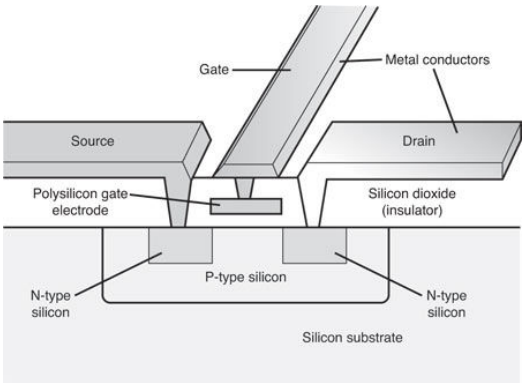


FIGURE 1.5: Transistor $n - MOS$

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 SiO2)
- 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 circuit 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.

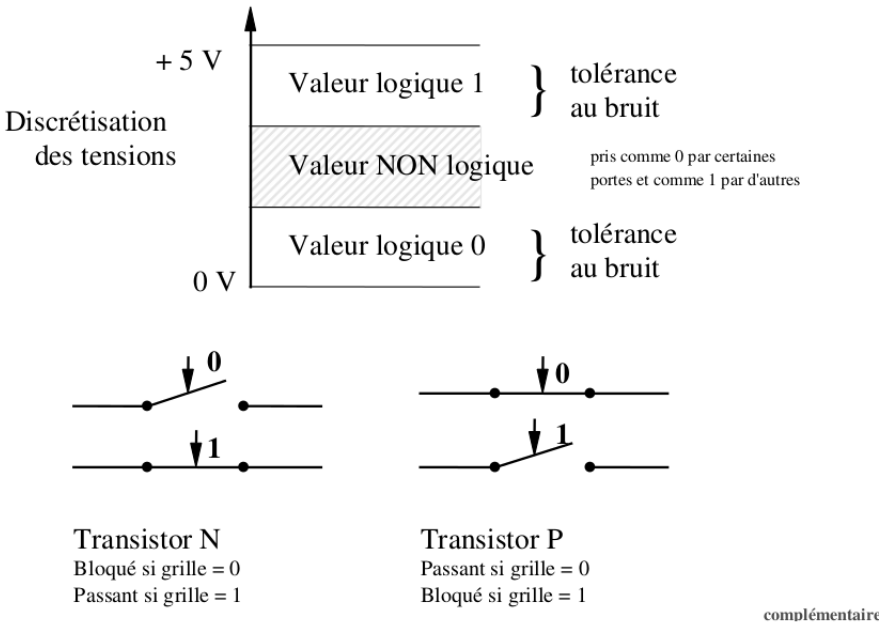


FIGURE 1.6: CMOS

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 micro-électroniques 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...

A titre de premier exemple, considérons le cas d'un *inverseur* 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 micro-électronique, 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 apporterons très certainement des révolutions à la hauteur de celles que nous

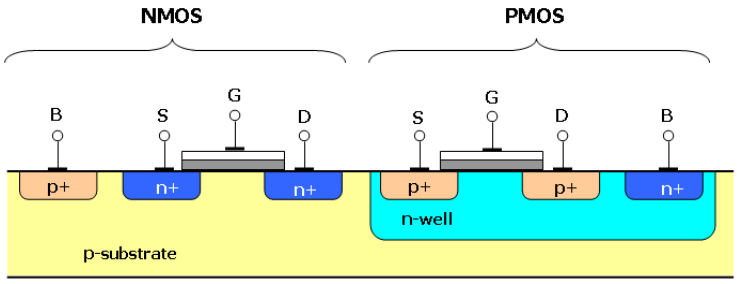


FIGURE 1.7: Transistor $n - MOS$

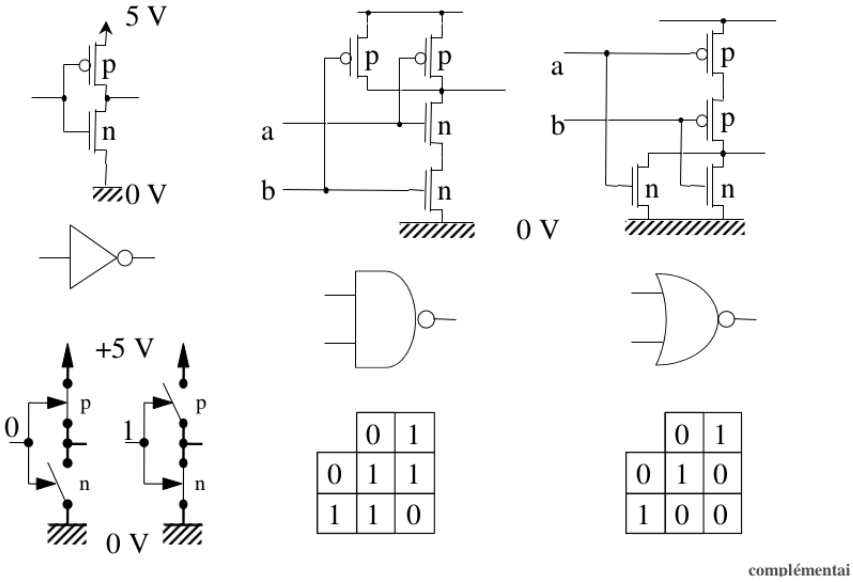


FIGURE 1.8: CMOS

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,...).

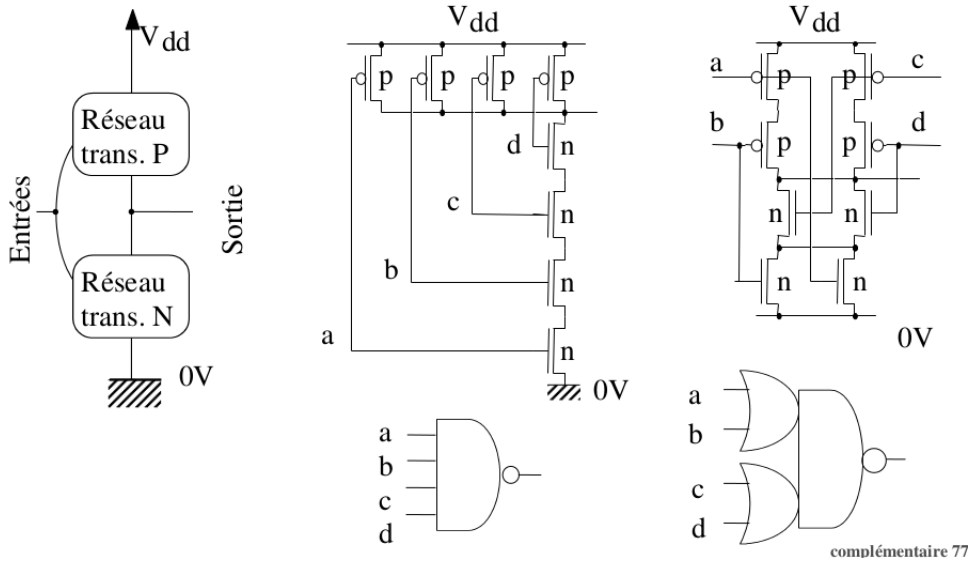


FIGURE 1.9: CMOS

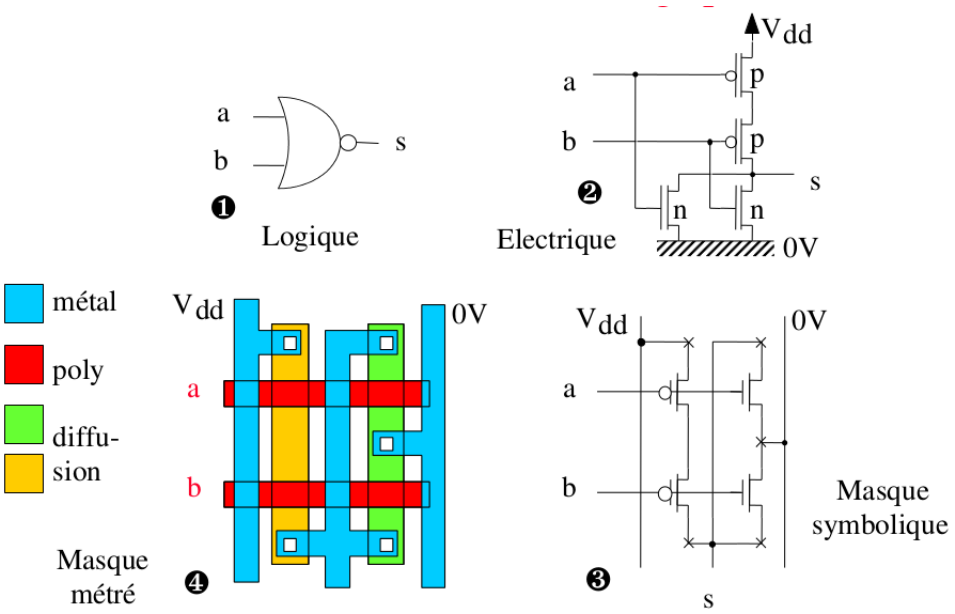


FIGURE 1.10: CMOS

Analogique	**Logique**
Précision limitée (techno)	Précision arbitraire (# bits)
Valeur approchée (±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 . . . 18
2.1.1 Décomposition en puissance de la base . . . 18
2.1.2 Conversion entre bases . . . 18
2.1.3 Exemples . . . 19
2.1.4 Conversions binaire, hexadécimal, octal . . . 20
2.1.5 Bits, bytes, nibbles, words . . . 20
2.2 Représentation des entiers négatifs. . . . 21
2.2.1 Complément à 2 d'un nombre . . . 21
2.2.2 Astuces pour complémenter à 2 . . . 22
2.2.3 A propos du bit de signe . . . 22
2.2.4 Conversion d'un nombre signé vers un nombre non signé . . . 23
2.3 Représentations alternatives, obsolètes ou rares . . . 23
2.3.1 Représentation en Magnitude signée . . . 23
2.3.2 Représentation en complément à 1 . . . 23
2.3.3 Code BCD et code de Gray pour les entiers non-signés . . . 23
2.4 Calcul d'addition et soustraction . . . 25
2.4.1 Addition . . . 25
2.4.2 Soustraction . . . 25
2.4.3 Petit complément sur la soustraction . . . 26
2.5 Données composées ou symboliques . . . 27
2.5.1 Données composées . . . 27
2.5.2 Données symboliques ou énumérées. . . . 27
2.5.3 Code alphanumérique ASCII . . . 28
2.5.4 Code alphanumérique UTF-8 . . . 28
2.6 Calcul en virgule fixe et en virgule flottante . . . 29
2.6.1 Les nombres en virgule fixe . . . 29
2.6.2 Les nombres en virgule flottante. Norme IEEE 754 . . . 30
2.7 Manipulations numériques . . . 30
2.7.1 Récupération d'un champ . . . 30
2.7.2 Conversion par programmation . . . 30
2.7.3 Autres manipulations au niveau bit . . . 31
2.8 Vers la détection d'erreur : bit de parité . . . 31
2.9 Conclusion . . . 31

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 :

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

et se généralise pour toute base r :

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

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 :

$$10110_{10} = 10011101111110_2$$

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 en 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.

[1] Personnellement, je n'ai jamais eu à utiliser la première...
[2] En base binaire, on parle de LSB : least significant bit
[3] MSB : 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 :

$$\begin{aligned} 12/2 &= 6 + 0 \\ 6/2 &= 3 + 0 \\ 3/2 &= 1 + 1 \\ 1/2 &= 0 + 1 \end{aligned}$$

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

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

$$\begin{aligned} 0.75 \times 2 &= 1.5 = 1 + 0.50 \\ 0.50 \times 2 &= 1.0 = 1 + 0 \end{aligned}$$

  Ainsi :

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

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

$$
\begin{aligned}
0.39654 \times 2 &= 0.79308 = 0 + 0.79308 \\
0.79308 \times 2 &= 1.58616 = 1 + 0.58616 \\
0.58616 \times 2 &= 1.17232 = 1 + 0.17232 \\
0.17232 \times 2 &= 0.34464 = 0 + 0.34464
\end{aligned}
$$

  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.

[4] Pour 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 $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ 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 :

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

On a donc :

$$B2U_w : \{0,1\}^w \to \{0, \ldots, 2^w - 1\}$$

. 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 $\vec{x} = [10\ldots0]$ et $\vec{x} = [01\ldots1]$ 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 :

$$B2U_w : \{0,1\}^w \to \left\{-2^{w-1}, \ldots, 2^{w-1} - 1\right\}$$

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 complémenter à 2

Le complément à 1 d'un nombre binaire consite 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

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

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

$C_{2,4}(-5)$	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 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 : $\forall x \in \mathbb{Z}_w / TMin_w \leq x \leq TMax_w$:

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

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 singifie "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

$$0 \rightarrow 0000$$
$$1 \rightarrow 0001$$

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

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

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 résulat 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

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

[6] Je suis preneur d’une solution élégante...
[7] ceci se généralise dans tout autre base.

2.4.3 Petit complément sur la soustraction

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

1. celui où une retenue apparaît lors de l'addition. Cette retenue peut apparaître au dela (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 $\rceil log_2(63) = 6$ 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 $\rceil log_2(12) = 4$ 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. Inversons 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 :

$$pos = (x << 8) + y$$

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 :

$$\begin{array}{rll} \text{bleu} & \rightarrow 0 & \rightarrow 000 \\ \text{blanc} & \rightarrow 1 & \rightarrow 001 \\ \text{rouge} & \rightarrow 2 & \rightarrow 010 \\ \text{vert} & \rightarrow 3 & \rightarrow 011 \\ \text{noir} & \rightarrow 4 & \rightarrow 100 \end{array}$$

La formule générale stipule que pour n valeurs énumérées, il faut $\rceil log_2(n)$ 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.

b_7 →					0	0	0	0	1	1	1	1
b_6 →					0	0	1	1	0	0	1	1
b_5 →					0	1	0	1	0	1	0	1
Bits	b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → / Row ↓	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 10_{10} représente l'échappement (retour à la ligne). Le chiffre 0 correspond au code ASCII 48_{10} et le *a* minuscule au code 97_{10}. 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 quatre[1].

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
𤭢	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 déja 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.

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

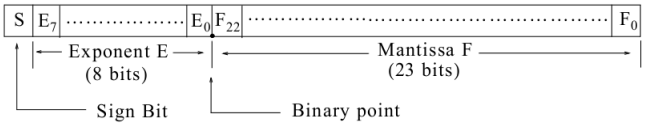
En binaire, ces extensions s'appellent les "décimales binaires" : la première décimale binaire est $\frac{1}{2}$, la seconde est $\frac{1}{4}$, la troisième est $\frac{1}{8}$ et ainsi de suite.

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 mulitplication : 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 headecimal. $0x0000FF$ a été créé de manière à annuler la contribution des bits de x dans Coord. Pour cela les 4 digits hexadéciamaux 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 commme : $11\ldots110_2$.

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 recepteur peut alors verifier 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 permi de rentrer de plain-pied dans les représentations numériques, notamment avec la prise de connaissance de différents codes. Ce domaine est vaste !

Il existe d'autres codes que nous n'avons pas aborder, comme les codes en excès de 3 où le code de Aïken. Nous nous appuyerons sur ces premières connaissances afin de construire –bientôt– nos premières applications numériques. Nous recommandons ici, en complément, la lecture de Bryant

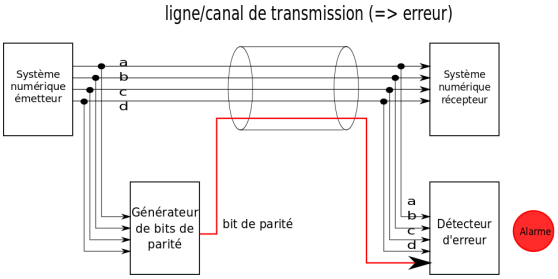


FIGURE 2.2: Emetteur, récepteur et canal bruité

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 1	0
mot	bit de parité

code

FIGURE 2.3: Table de vérité du bit de parité

et O'Hallaron [1] pour plus de détails sur la partie arithmétique. Il est également intéressant de regarder d'autres présentations orientées sur la Théorie de l'Information comme Cover & Thomas [5].

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 XIXe 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 Though"*, 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 grace 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 XXe 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 $X = (x_1, x_2, \ldots, x_n) \in \mathbb{B}$ et peut prendre 2^n valeurs. L a valeur de X est appelée un point. Un litéral est une variable booléenne simple ou son complément. Par exemple, a et $\overline{a}$ 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 complémentation 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	$a \times b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a + b$
0	0	0
0	1	1
1	0	1
1	1	1

a	$\bar{x}$
0	1
1	0

$x \times y$ vaut 1 si x vaut 1 **et** x vaut 1, 0 sinon.
$x + y$ vaut 1 si x vaut 1 **ou** x 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** : $x + (x + z) = (x + y) + z$
2. **× est associatif** : $x \times (x \times z) = (x \times y) \times z$
3. **+ est commutatif** : $x + z = y + x$
4. **× 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 ×** : $\exists 1 / x \times 1 = x$
7. **× est distributif sur +** : $x.(y + z) = x.y + x.z$
8. **+ est distributif sur ×** : $x + (y.z) = (x + y).(x + z)$. 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. **× est idempotent** : $x \times x = x$
3. 1 **est absorbant pour +** : $x + 1 = 1$
4. 0 **est absorbant pour ×** : $x \times 0 = 0$
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 $f_1(a, b) = \overline{a + b}$ et $f_2(a, b) = \overline{a}.\overline{b}$.

a	b	$a + b$	$f_1 = \overline{a + b}$	$\overline{a}$	$\overline{b}$	$f_2 = \overline{a}.\overline{b}$
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 :

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

On peut également démontrer que :

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

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 $(\overline{x})$. 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 : $A.B, \overline{A}.B, A.\overline{B}$ et $\overline{A}.\overline{B}$. 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 complém ents.

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 f aut 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 :

$$F(x_1, x_2, \ldots, x_n) = x_1.F(1, x_2, \ldots, x_n) + \overline{x_1}.F(0, x_2, \ldots, x_n)$$

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 de cision diagrams) :

Cette décomposition a permi 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 un e variable totale qui rende une formule booléenne vraie (problème SAT). C'est un problème très difficile [1] !

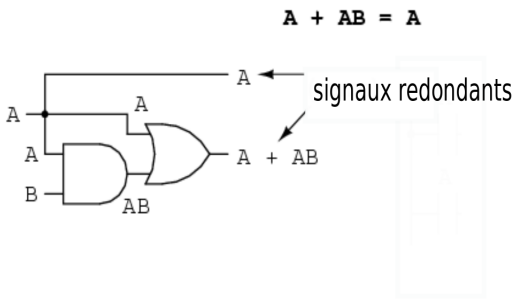
En réalité, le terme de BDD est généralement associé à ROBDD : Reduced Ordered Binary Decision Diagram. L'intérêt des ROBDD est qu'il présente une forme canonique (unique) pour une fonction donnée et quel que soit l'ordre d'évaluation des variables. Cette propriété le rend très utilse dans la recheche d'équivalence entre 2 circuits : implémentent-ils intrinsèquement la même for mule, 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 !

[1] SAT 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

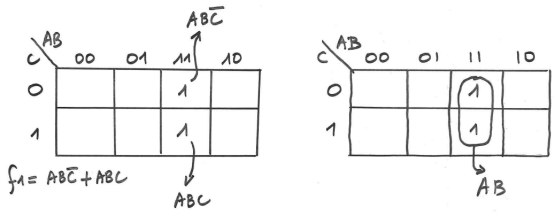
Toutefois, la bonne application successive de ces règles est difficile pour des formules complexes. Cette application dépend de votre capacité à "sentir" la simplification...Il existe heur eusement une autre méthode, plus mécanique.

3.4.2 Par tableau de Karnaugh

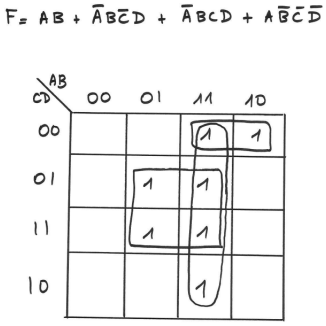
Les tableaux de Karnaugh permettent d'appliquer de manière mécanique des simplifications qui marchent à coup sûr. Ces tableaux sont en fait un ensemble de cases. En général, une expression booléenne à n variables peut être représentées par un tableau de Karnaugh à $2n$ cases, où chaque case représente une ligne d'une table de vérité équivalente. Dans la méthode des tableaux de Karnaugh, il est nécessaire de disposer les entrées du tableaux 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 contigues 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 monomes 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 de 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 simplifications booléenne.

Observons un premier cas : simplifions la fonction $f_1(a,b,c) = a.b.\overline{c} + a.b.c$. La simplifications 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 $f(a,b,c,d) = a.b + \overline{a}.b.\overline{c}.d + \overline{a}.b.c.d + a.\overline{b}.\overline{c}.\overline{d}$ vaut, simplifiée $f(a,b,c,d) = a.b + b.d + a.\overline{c}.\overline{d}$



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 **39**
4.2 Portes logiques de base **40**
4.3 Fonctions logiques "complexes" **41**
4.4 Mapping technologique **42**
4.5 Chemin critique et fréquence de fonctionnement **42**
4.6 Arithmétique de base **43**
4.6.1 Additionneur 43
4.6.2 Soustracteur 45
4.6.3 Additionneur-soustracteur 45
4.6.4 Multiplieur 46
4.6.5 Diviseur 47
4.7 Shifter **47**
4.8 Multiplexeur **48**
4.9 Comparateur **48**
4.10 Codeurs et Décodeurs **49**
4.11 Conclusion **50**

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 et aucunement des sorties précédemment calculées. 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'emmagaziner de l'énergie (sur les connexions etc). Ainsi les sorties dépendent des valeurs des entrées à un instant $t - \delta$ 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			
	IEC	ANSI	Table	Equation
Porte NON (NO)	1		entrée sortie 0 1 1 0	$f=\bar{a}$
Porte ET (AND)	&		entrées sortie 0 0 0 0 1 0 1 0 0 1 1 1	$f=a.b$
Porte OU (OR)	≥ 1		entrées sortie 0 0 0 0 1 1 1 0 1 1 1 1	$f=a+b$
Porte OU exclusif (XOR)	= 1		entrées sortie 0 0 0 0 1 1 1 0 1 1 1 0	$f=a \oplus b$
Porte NON-ET (NAND)	&		entrées sortie 0 0 1 0 1 1 1 0 1 1 1 0	$f=\overline{a.b}$
Porte NON-OU (NOR)	≥ 1		entrées sortie 0 0 0 0 1 1 1 0 1 1 1 1	$f=\overline{a+b}$

Première approche : et, ou, non Les portes logiques de base reflètent directement les opérations élémentaires de l'algèbre de Boole : conjunction, 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 telles portes logiques (auquelles on ajoutera bientôt un élement séquentiel) forme une *netlist* (ou, en français, *liste d'interconnexions*) . Dans des cas réels, cette netlist peut contenir plusieurs millions de tels élements 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 produise 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 $a \odot 1 = not(a)$. Prenons la formule $y = a + b$ et exprimons là à l'aide de $\odot$. Posons $\alpha = a$ et $\beta = b$. On a donc :

$$y = a + b \tag{4.1}$$
$$y = \overline{\alpha.\beta} \tag{4.2}$$
$$y = \alpha \odot \beta \tag{4.3}$$
$$y = (a \odot 1) \odot (b \odot 1) \tag{4.4}$$

De même, on peut démontrer que :

$$y = a.b \tag{4.5}$$
$$y = \overline{\alpha}.\overline{\beta} \tag{4.6}$$
$$\tag{4.7}$$

d'où :

$$\overline{y} = \overline{\overline{\alpha}.\overline{\beta}} \tag{4.8}$$
$$= a \odot b \tag{4.9}$$

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 dela 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 commme VHDL nous y aideront notamment, car de simples constructions du langage permettront, en réalité, de

[1] Un 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 boitier 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 apropriées 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.

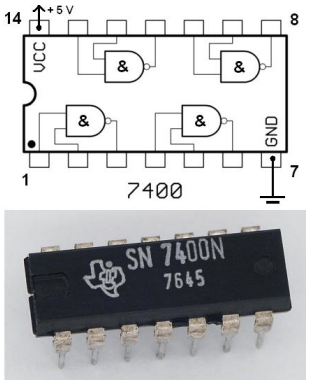


FIGURE 4.1: Exemple de circuit de la (mythique !) série 74

4.5 Chemin critique et fréquence de fonctionnement

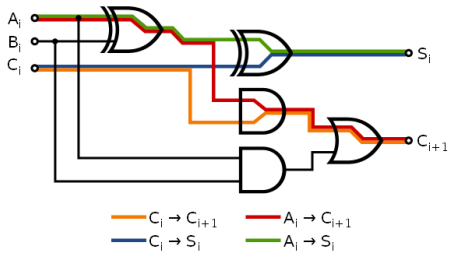
La notion de chemin critique est d'une importance capitale. Il permet de calculer la fréquence maximale à laquelle on pourra soumettre un nouveau jeu d'entrées au circuit, sans perturber le calcul précédent. Cette fréquence détermine *in fine* les performances finales de votre ordinateur.

Souvenons nous que, physiquement, nous avons affaire une interconnexion de composants, qui possèdent leurs propres resistance 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 connait 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 fausse et instable.

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

$$F_{max} \approx \frac{1}{t_{critic}}$$



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 déliver 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.

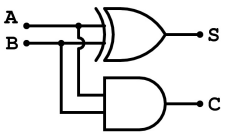
4.6.1 Additionneur

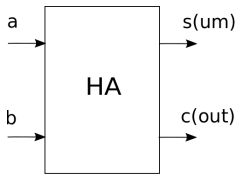
Il existe un grand nombre de manière d'assembler les portes logiques pour réaliser un calcul comme un additionneur. La plus traditionnelle consiste à d'abord étudier le *demi-additionneur* (1 bit), puis l' additionneur 1 bit *complet* (qui s'appuiera sur le demi-additionneur), puis chercher à étendre 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.

[2] Nosu 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)

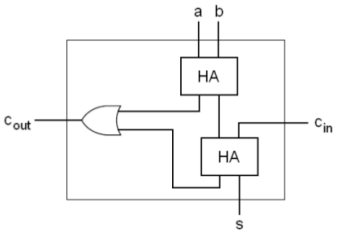
a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

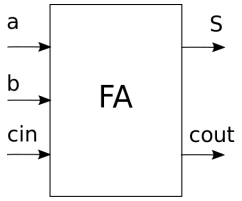




Additionneur 1 bit On introduit désormais la retenue entrante, afin de réaliser une addition sur n'importe quelle "colonne" de digits. Un calcul permet de nous rendre compte que le demi-additionneur précédent peut être réutilisé. Il s'agit ici de notre premier *assemblage hierarchique* de circuits.

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* :

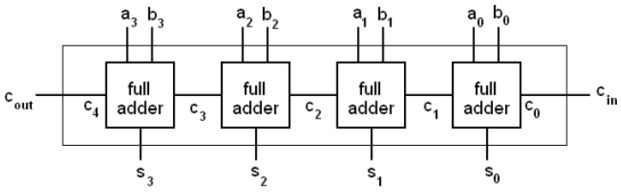
$$\overline{a \oplus b} = \overline{a.\overline{b} + \overline{a}.b} = (\overline{a} + b).(a + \overline{b}) = \overline{a}.\overline{b} + a.b$$

On a désormais :

$$S = a.\overline{b}.c_i + a.b.c_i + a.\overline{b}.c_i + a.b.c \tag{4.10}$$
$$= a.(a \oplus c_i) + a(b \oplus c_i) \tag{4.11}$$
$$= a \oplus b \oplus c_i \tag{4.12}$$

De même, on trouve : $c_o = a.b + (a \oplus b).c_i$. 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 parait...).

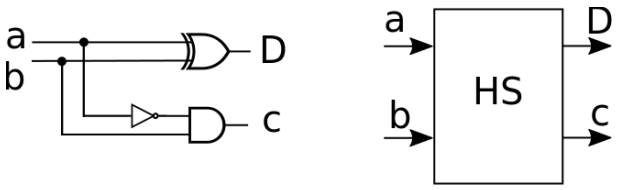


4.6.2 Soustracteur

Cherchons désormais les équations du soustracteur logique.

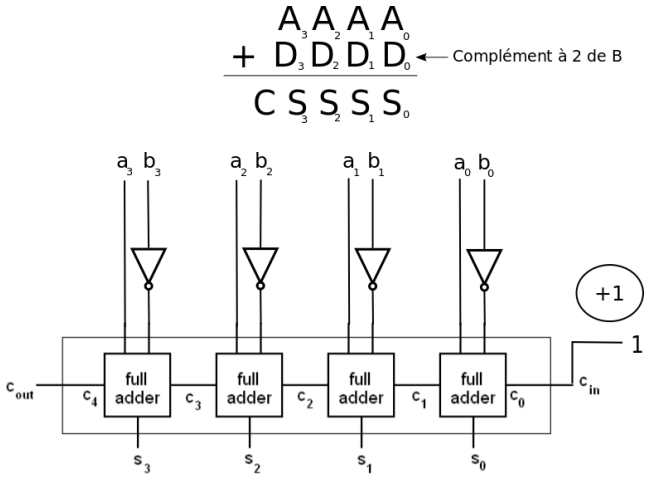
Utilisation d'une table de vérité On procède de même que précédemment, en écrivant la table de vérité du demi-soustracteur (HS), puis aux équations logiques. Sans surprise, ces équations ressemblent de très près à celle du demi-additionneur : en numérique aussi, l'addition et la soustraction sont des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

a	b	D	c
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



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 mulitplication, 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,

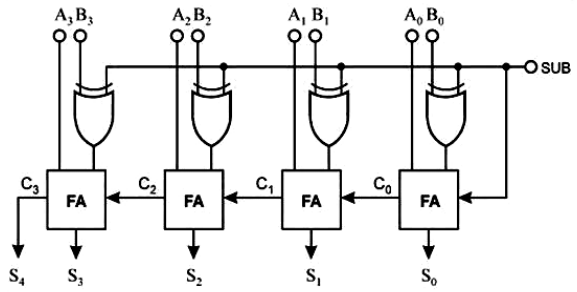
tantôt tel autre. Dans le cas de l'additionneur-soustracteur, on procède par la même observation que la soustraction précédemment étudiée : quelles entrées d_i dois-je fournir à l'additionneur ? Dans le cas où je souhaite réaliser l'addition, on doit avoir $d_i = b_i$ (les a_i restent bien évidemment 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

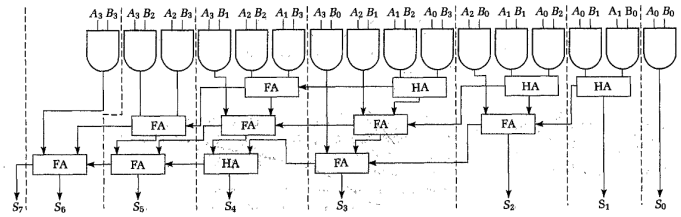
Méthode naturelle Pour impléménter un multiplieur en logique combinatoire, il faut simplement comprendre ce qui se passe lorsqu'on "pose" une multiplication traditionnelle et reporter le principe en respectant la logique booléenne. Le produit de 13 par 11 est donné ici en exemple.

```
multiplicand    1101     (13)
multiplier    * 1011     (11)
                1101
               1101
              0000
             1101
             10001111    (143)
128 + 8 + 4 + 2 + 1 = 143
```

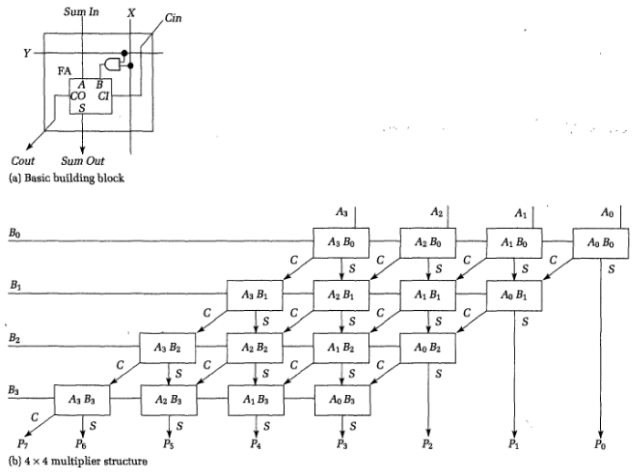
On se convainct facilement que la multiplication se réduit, ligne par ligne, à un conjonction booléenne (et logique). Ligne à ligne, il faut ensuite additionner soit deux opérandes (demi-additionneur), soit trois lorsqu'une retenue est possible (full adder 1 bit).

			A_3	A_2	A_1	A_0
			B_3	B_2	B_1	B_0
			$A_3 \bullet B_0$	$A_2 \bullet B_0$	$A_1 \bullet B_0$	$A_0 \bullet B_0$
		$A_3 \bullet B_1$	$A_2 \bullet B_1$	$A_1 \bullet B_1$	$A_0 \bullet B_1$	
	$A_3 \bullet B_2$	$A_2 \bullet B_2$	$A_1 \bullet B_2$	$A_0 \bullet B_2$		
$A_3 \bullet B_3$	$A_2 \bullet B_3$	$A_1 \bullet B_3$	$A_0 \bullet B_3$			
S_6	S_5	S_4	S_3	S_2	S_1	S_0

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 savemment, mais de manière parfaitement régulière, en 2 dimensions. Les microélectroniciens aprécient 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 dernirèe 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 :

[3] dû à Randu Katz, dont on pourra consulter le livre [6]

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.

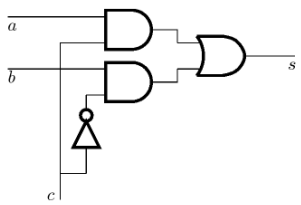
Le shifter à position fixe a notamment la particularité d'être d'une très grande utilité lorsque Δ est une puissance de 2. En effet, décaler à gauche (resp. à droite) d'une position Δ revient à multiplier (resp. diviser) par $log_2(\Delta)$. Par exemple, pour multiplier un nombre par 2 (resp. par 4,8,16 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]

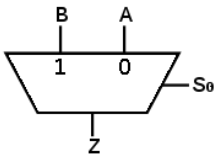
Barrel shifter Lorsque le nombre Δ n'est pas fixé, mais devient une entrée dur circuit, il faut être à même de décaler "à volonté". Cette opération plus délicate nécessite un réseau de multiplexeurs.

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 $\lceil(\log_2 n)$ 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 élement 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

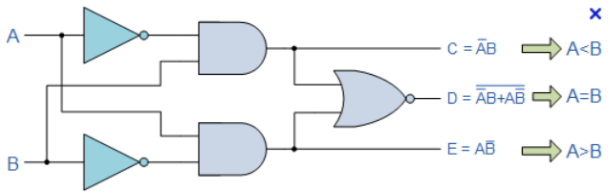
4.9 Comparateur

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

[4] Dans 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 $>$.

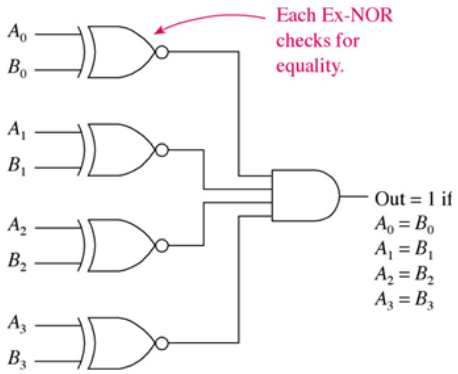
a	b	$<$	$=$	$>$
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0



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 figrue 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. On complètera la lecture de ce chapitre par la consulation d'autres ouvrages comme [7],[4].

Chapitre 5

Circuits séquentiels

Sommaire

5.1 Introduction **51**
5.2 Discrétiser le temps **52**
5.3 Bascule D **53**
5.3.1 Fonction d'échantillonnage de la bascule D. Set-up et hold. Metastabilité. 53
5.3.2 Décalage temporel : la raison d'être de la bascule D 54
5.4 Quelques *patterns* de conception synchrone **54**
5.4.1 Registre à décalage 55
5.4.2 LFSR : linear feedback shift register 55
5.4.3 Bascule D et multiplexeur : mémorisation 56
5.4.4 Compteurs et timers 56
5.5 Initialisation d'une bascule D **57**
5.6 Conclusion **58**

5.1 Introduction

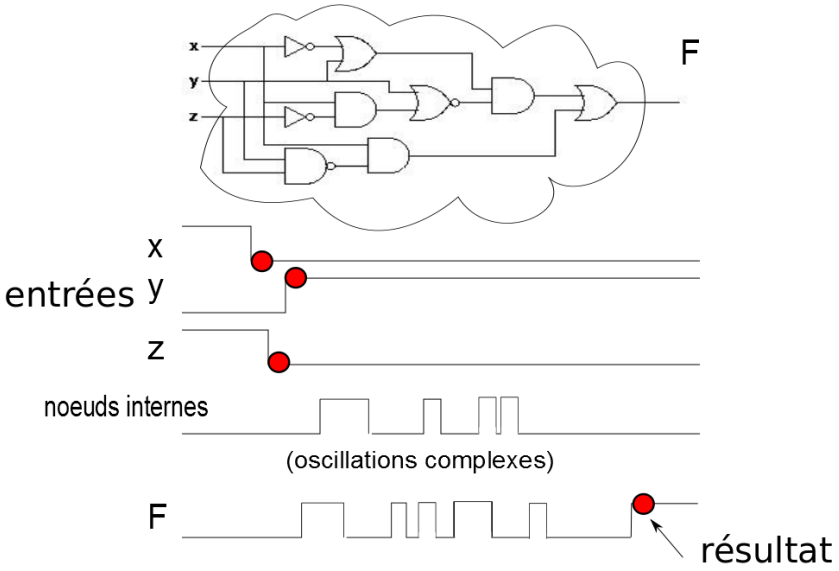
Nous sommes désormais armés pour représenter et manipuler les nombres sous forme binaire (chapitre 2), puis les traiter par une électronique numérique (chapitre précédent). Toutefois, les opérateurs étudiés jusqu'ici se bornaient à des traitements combinatoires. En aucune manière, nous ne nous sommes 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 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.

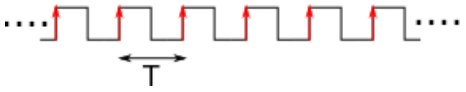


De même, aux bords du circuit, sur les entrées-sorties, ces événements apparaissent au gré des stimulations extérieures (générateurs de signaux, capteurs, etc), parfaitement *asynchrones* : par ce terme "asynchrone", on indique qu'il n'existe pas a priori de référentiel temporel qui puisse donner un "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 évènements 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].

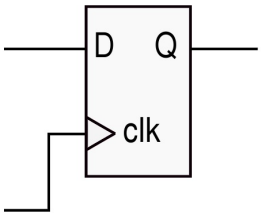
[1] Toutefois, même si cette référence mathématique est intéressante, elle a ses limites et n'est guère référencée par les Electroniciens. La raison en est simple : en général les traiteurs de signaux font une hypothèse algorithmique forte qui consiste à supposer que le signal est *disponible*, dans 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*)

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



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.

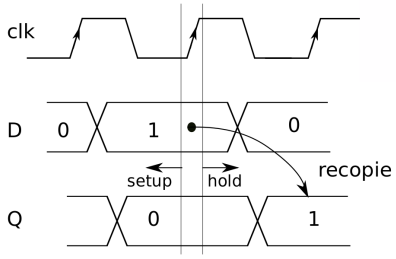


D	clk	Q_{n+1}
0	↑	0
1	↑	1
X	0 / 1	Q_n

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.

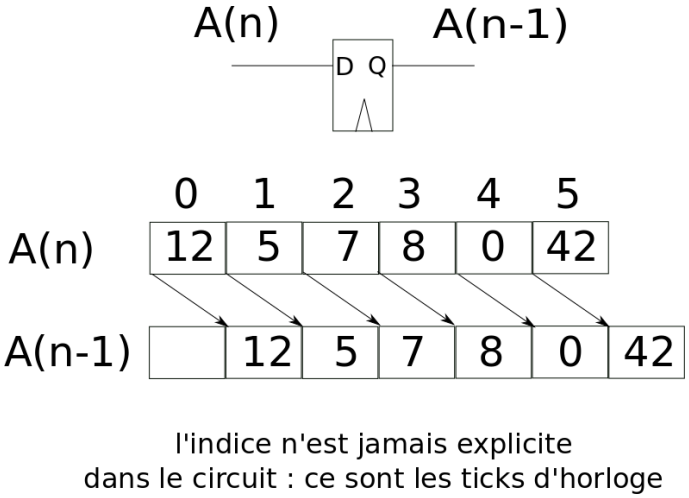
5.3.1 Fonction d'échantillonnage de la bascule D. Set-up et hold. Metastabilité.

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.



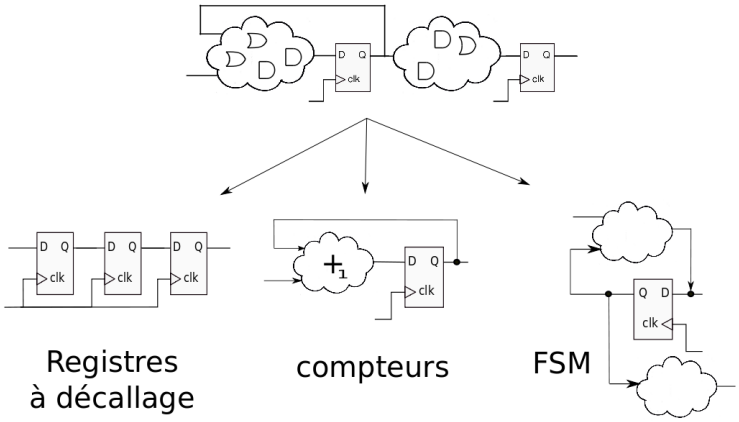
5.3.2 Décalage temporel : la raison d'être de la bascule D

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.



5.4 Quelques *patterns* de conception synchrone

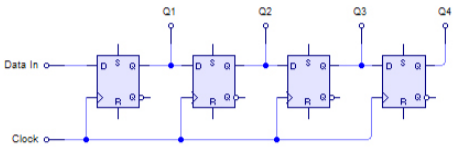
Tout système numérique, simple ou complexe, est une combinaison de portes logiques et de bascules D. Un tel assemblage générique est présenté sur la figure suivante : on observe un enchaînement de traitements combinatoires et de bascule D. Des rebouclages peuvent exister, mais il dooti forcément exister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

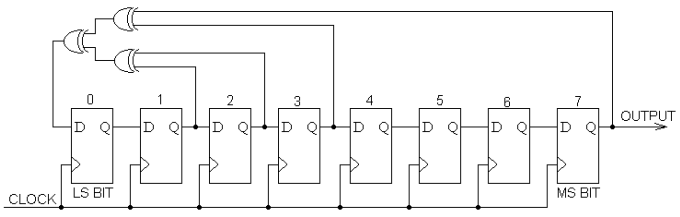
Les registres à décalage sont des assemblages linéaire de bascule D, chaînées les unes à la suite des autres. Lorsqu'une donnée se présente, la précédente est déjà dans le premier étage, et l'avant-dernière dans le deuxième étage etc. A chaque front montant d'horloge, de manière synchrone, toutes les 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 na 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

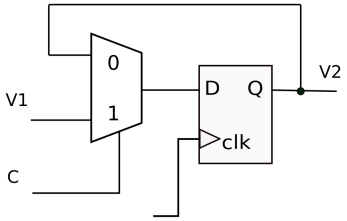
Les LFSR ont l'intérêt de créer de manière simple des suites en apparence aléatoires. En réalité, les suites de valeurs produites sont pseudo-aléatoires et parfaitement récurrentes à partir d'un certain rang. La simplicité des LFSR les rendent malgré tout très attrayants dans de nombreuses situations. 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.

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



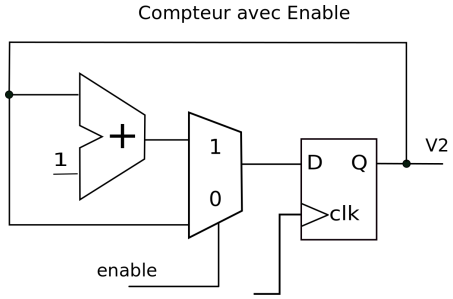
5.4.3 Bascule D et multiplexeur : mémorisation

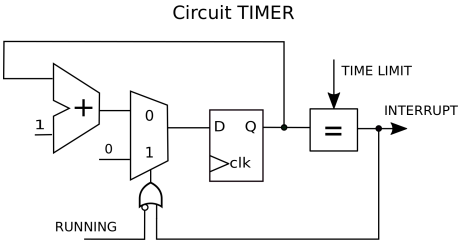
La seule fonction de décalage semble très éloignée des fonctions que l'on peut attendre d'un processeur numérique. De fait, la bascule D es tle plus souvent utilisée conjointement à un multiplexeur. Ce multiplexeur (combinatoire) permet de sélectionner à volonté l'action d'*échantillonner ou non* : le 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 et timers

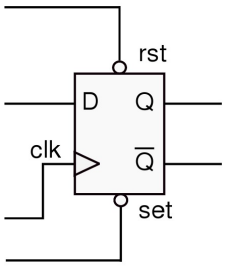
La fonction d'un compteur est d'incrémenter un registre. Cet incrément peut se faire sur tous les 'top' d'horloge, ou de manière commandée en utilisant un signal d'autorisation. Les compteurs peuvent notamment servir à réaliser des *timers* : ces dispositifs très importants en embarqué permettent d'interrompre un circuit au bout d'un certain temps. Ce temps correspond à la valeur limite d'un compteur. L'intérêt de ces timers est par exemple de laisser un processeur réaliser une fonction principale, et de le détourner périodiquement vers une tâche annexe : c'est la notion d'*interruptions*.





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 utilisé, 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 :

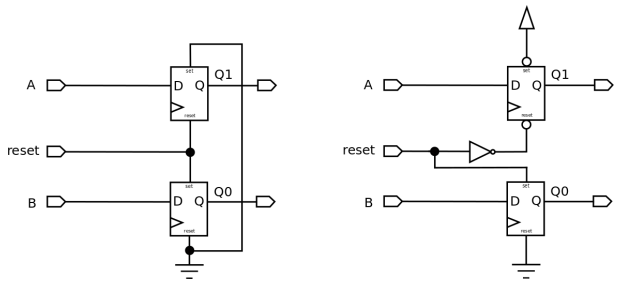
- $set = POL_s \implies Q = 1$
- $reset = POL_r \implies Q = 0$

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 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 reste 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 $\Longrightarrow$ 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 **59**
6.2 Machines d'états finis **59**
6.3 Diagramme états-transitions **61**
6.4 Consistance ou *causalité* d'une machine d'états finis **61**
6.5 Encodage des états **62**
6.6 Méthode générale de conception d'un automate **63**
6.7 Exemple complet **64**
6.8 Gérer la complexité **66**
6.9 Méthode particulière : cas de l'encodage *one-hot* **66**
6.10 Conclusion **67**

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 séquencement 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 $a_1 a_2 \ldots a_n$, $n \geq 0$ est $\lambda(q_0), \lambda(q_1) \ldots \lambda(q_n)$ où $q_0, \ldots, q_n$ 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

$_l ambda(q, a)$ 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 $a_1, \ldots a_n$ est $\lambda(q_0, a_1)\lambda(q_1, a_2) \ldots \lambda(q_{n-1}, a_n)$ où $q_0, q_1, \ldots, q_n$ est la séquence des états tels que $\lambda(q_{i-1}, a_i) = q_i$ pour $1 \leq i \leq n$.

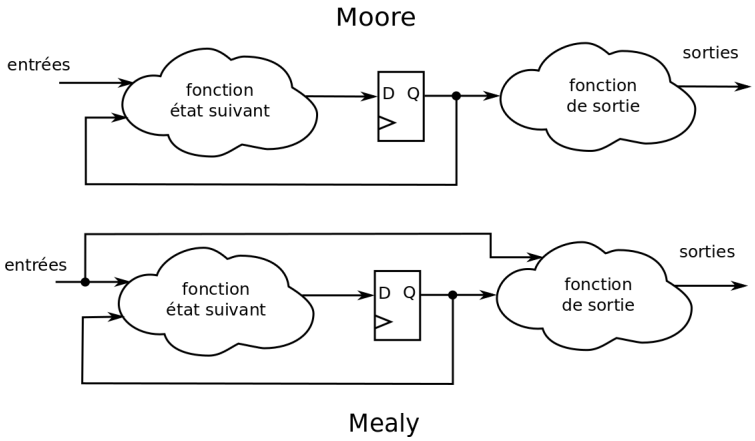


FIGURE 6.1: Automates de Moore et de Mealy

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 ıniveau 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'appelerons "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 flêches 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 lé 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 tem ps.

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

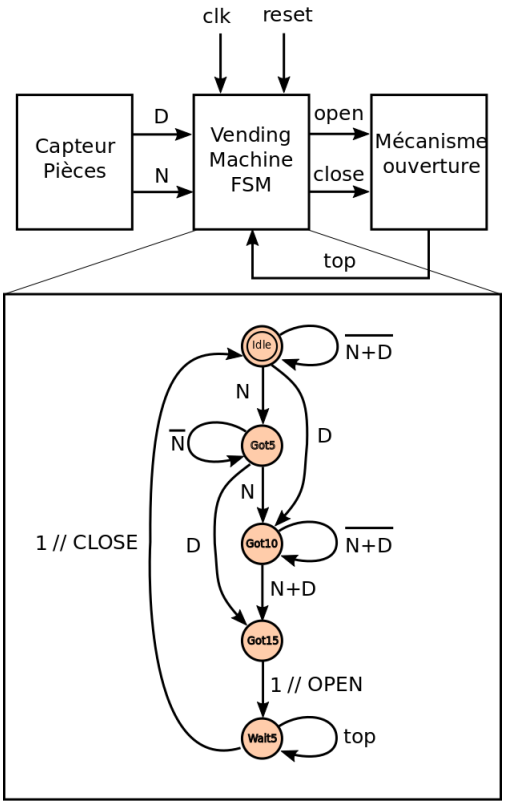


FIGURE 6.2: Vending machine et son environnement

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 révêlent 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 a pparaître 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 couran 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.

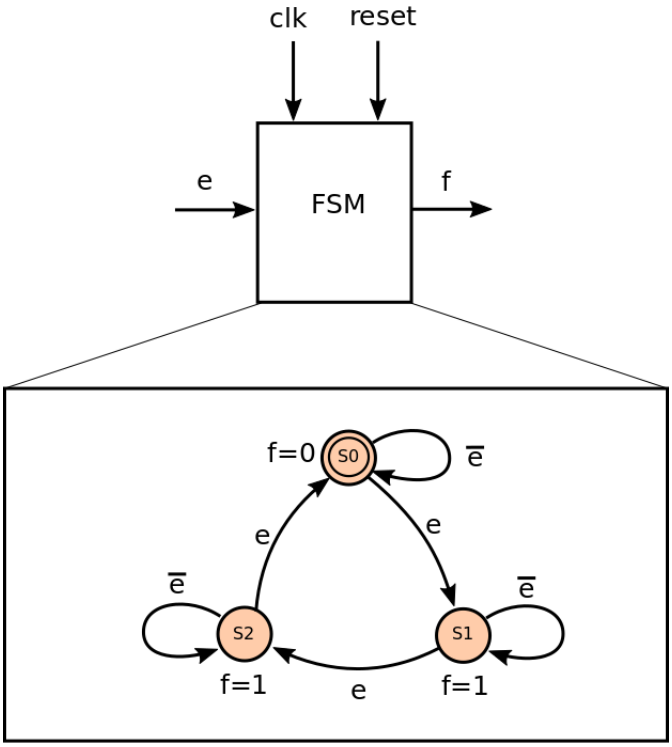


FIGURE 6.3: Tableaux de Karnaugh de l'exemple

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

e \ Q_1Q_0	00	01	11	10
0		1	X	
1	1		X	

D_1

e \ Q_1Q_0	00	01	11	10
0			X	1
1		1	X	

f

e \ Q_1Q_0	00	01	11	10
0		1	X	1
1		1	X	1

Les équations obtenues, sont :

$$D_0 = Q_0.\overline{e} + \overline{Q_1}.\overline{Q_0}.e \tag{6.1}$$
$$D_1 = Q_1.\overline{e} + Q_0.e \tag{6.2}$$
$$f = Q_1 + Q_0 \tag{6.3}$$
$$\tag{6.4}$$

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.

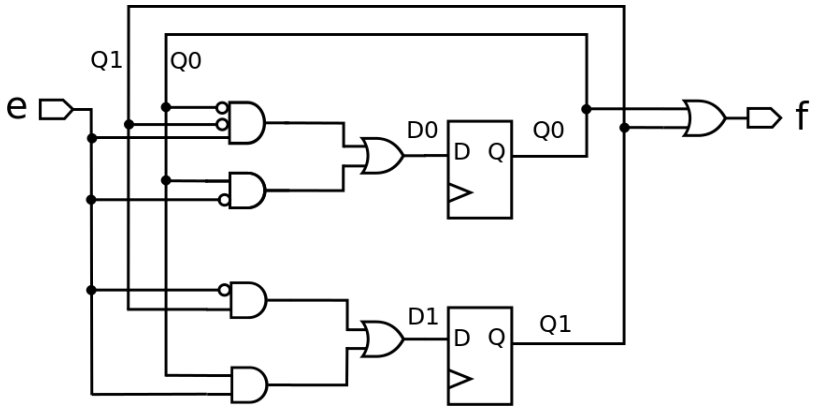


FIGURE 6.4: Implémentation matérielle de l'automate

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 statégies 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 élevé, 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 l'on 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 précedents S_i, on connecte le signal $c_{i,j}$ sur une des entrées de cette porte OR.

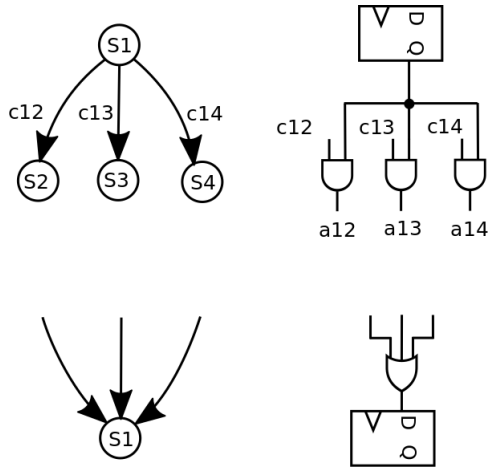


FIGURE 6.5: Procédé simplifiant la génération d'un circuit d'automate dans le cas d'un encodage *one-hot*.

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 découvir 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 languages 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 Avant propos **69**
7.2 Introduction : les langages de description matérielle **70**
7.3 La structure globale d'un programme VHDL **71**
7.3.1 Déclaration de librairies et packages 71
7.3.2 Notion d'entité 72
7.3.3 Notion d'architecture 72
7.4 Les éléments clés de l'architecture **73**
7.4.1 Assignations concurrentes des signaux 73
7.4.2 Processus 73
7.4.3 Instanciation de composants et d'entités 75
7.5 Décrire des machines d'états finis **77**
7.5.1 FSM au niveau logique 77
7.5.2 FSMs au niveau RTL 77
7.6 Simulation en VHDL **79**
7.6.1 Flot de conception général 79
7.6.2 Bancs de tests ou *Testbench* pour la vérification 80
7.6.3 Modèles de référence 80
7.7 Fonctionnement interne d'un simulateur VHDL **81**
7.8 Utilisation du simulateur GHDL **82**
7.8.1 Introduction 82
7.8.2 Commandes essentielles 82
7.9 Conclusion **83**

7.1 Avant propos

Ce chapitre est une introduction à VHDL. Ce langage sera étudié et utilisé plus en détail en deuxième année. Plusieurs cours universitaires découplent totalement l'enseignement de

l'Electronique numérique et celui de VHDL (ou Verilog). C'est une démarche qui a du sens : elle permet notamment de raisonner sur les concepts de l'Electronique, avant de s'engouffrer dans les particularismes de tel ou tel langage. Toutefois, notre expérience nous laisse penser qu'il est frustrant de s'arrêter si près d'expériences possibles, simples et concrètes, qui permettent au futur ingénieur de mieux se projeter dans ce domaine passionnant que sont les systèmes embarqués numériques. Nous avons donc choisi une *approche intermédiaire* qui consiste à se limiter à un survol de VHDL, *restreint à cette seule UV 1.5 de l'ENSTA Bretagne*. Nous nous bornerons :

- A la description de systèmes d'équations booléennes sous la forme d'*assignations concurrentes* de VHDL.
- A la description de bascule D sous la forme de *processus "clockés"*
- A la description de quelques bancs de test (testbench) qui illustrent la démarche de vérification par simulation.

Cette restriction laisse de côté les aspects les plus productifs de VHDL et notamment la maîtrise de la notion d'inférence au niveau transfert de registres (RTL). Cela sera étudié de manière poussée en deuxième année. Les plus impatients pourront consulter différentes ouvrages [2], [3]. Dans les flots de conception traditionnels, notre restriction correspond stricto-sensu au niveau logique.

7.2 Introduction : les langages de description matérielle

Naissance de VHDL 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, resistances, 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 (Elctronic 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 décrire 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.

Particularités de VHDL : deux sémantiques 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 en ce qui concerne le *temps*. 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.

Démarche de modélisation Ecrire des programmes VHDL nécessite d'adhérer à une démarche de modélisation très différente de l'écriture de programmes traditionnels (Python, Java, C etc). Comme leur nom l'indique, les HDL sont utiles à la *description* de circuits : il est inutile d'écrire des programmes VHDL en imaginant qu'il correspondront "par miracle" à une circuit. Il faut avoir à l'inverse le circuit sur papier ou tout au moins bien en tête, avant de se lancer dans sa description. Le modèle séquentiel des langages cités précédemment laisse *peut-être* l'opportunité de programmer "au fil de l'eau" : ce n'est pas le cas ici. Dans notre utilisation de VHDL dans cette UV, il s'agit d'abord d'obtenir les équations logiques, puis de les transcrire en VHDL. Ce sera également le cas en deuxième année, mais on se situera à un niveau d'abstraction (RTL) plus confortable que ces simples équations logiques : il existe des construction du langage qui nosu affranchissent de l'écriture explicite des équations. Mais nous respectons ici le partie-pris pédagogique de restriction du langage.

7.3 La structure globale d'un programme VHDL

Les programmes que nous allons écrire présente une structure systématique :

1. Déclaration de **librairies** et de **packages** utilisés
2. **Entity** : c'est la description des entrées-sorties du circuit, et de paramètres génériques éventuels (taille des données etc).
3. **Architecture** : description de l'organisation interne du circuit.

Ces déclarations apparaissent généralement dans cet ordre, dans un seul et même fichier. Notons que ce n'est toutefois pas une obligation : par exemple, une entité et son architecture peuvent être décrites dans des fichiers séparés. Il s'agit d'*unités de compilation* : le compilateur VHDL est capable de les compiler individuellement et de les assembler au final.

7.3.1 Déclaration de librairies et packages

Un exemple est donné ici. Il fait appel à deux librairies fournies par l'IEEE. Ces librairies sont incontournables.

- La librairie **std_logic_1164** permet d'utiliser les types les plus standard de VHDL, à savoir le type std_logic et le type std_logic_vector. Ce sont simplement l'incarnation du "bit" et d'un ensemble de bits, mais cette librairie permet de bénéficier d'autres valeurs que '0' et '1'. Ces autres valeurs sont d'une grande aide lors de la simulation du circuit et permettent de trouver au plus tôt des cas litigieux. En pratique, il est par exemple intéressant de pouvoir savoir si un fil est effectivement piloté ou non par une tension : dans le cas où aucune tension ne pilote le fil, on affaire à une tension 'u' (unknown). Cette valeur apparait en simulation lorsque le concepteur a oublié de connecter ("driver") le fil (signal) concerné. De même, le concepteur a peut-être connecté deux sources sur un même fil : ce dernier est alors en conflit (valeur 'x'). Ce dernier cas survient lorsque deux ou plusieurs sources tentent d'imposer leur tension. De même une équipotentielle peut se trouver dans un état de haute impédance (valeur 'z'). Le type bit existe nativement (sans appel à des librairies) dans VHDL, mais n'est guère utilisé, car il a la faucheuse tendance de masquer les non-connexions précédentes ou les conflits.

- La librairie **numeric_std** : c'est la librairie qui nous permet entre autre de faire des calculs sur des nombres signés ou non signés. Le type std_logic_vector précédent n'autorise pas d'emblée ce type de calculs arithmétiques (c'est un tableau sans arithmétique associée). Un ensemble de conversions très utiles sont également disponibles dans cette librairie.

```
librairie ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

7.3.2 Notion d'entité

L'entité permet de décrire les **ports** d'entrées-sorties du circuit, ainsi que d'éventuels paramètres dit "génériques". C'est l'**enveloppe extérieure** du composant. Pour être correctement utilisée, cette entité devra être instanciée en respectant ces entrées-sorties : il faudra venir connecter des **signaux** à ces ports.

Un premier exemple de description d'entité est donné ici, sans paramètre.

```
entity example1 is
   port(
     reset_n       : in  std_logic;
     clk           : in  std_logic;
     a             : in  std_logic;
     b,c,d         : in  std_logic;
     data1,data2   : in  std_logic_vector(31 downto 0);
     data3,data3   : in  unsigned(7 downto 0);
     f             : out std_logic;
     dataout       : out std_logic_vector(15 downto 0)
   );
end entity;
```

Cette entité présente un **nom** ("example1"), puis un ensemble de **ports**, qui sont des entrées ("in") ou des sorties ("out"). Notons qu'il existe aussi le moyen de décrire des ports bidirectionnels ("inout"), qui ne seront pas décrits ni utilisés ici ; ils nécessitent des buffers trois-états. On remarque immédiatement la syntaxe exigeante de VHDL, avec la présence de nombreuses parenthèses, signes de ponctuations (";","") et certaines redondances dans l'utilisation des mots clés ("entity"). Les bons éditeurs de textes comme Emacs fournissent une aide substantielle dans l'écriture du VHDL, en offrant des auto-complétions puissantes, voire la génération de gabarits (squelettes de code) complets, déclenchées sur des racourcis claviers.

7.3.3 Notion d'architecture

L'architecture est le contenu réel du circuit. Si l'entité était un capot de voiture, l'architecture représente le moteur, sous le capot. Une architecture est explicitement associée à une entité unique. Notons par contre qu'une entité peut être associée à plusieurs architectures : cela restitue l'idée que derrière l'apparence externe d'un composant (derrière l'entité), on peut trouver plusieurs organisations différentes, de la même manière que derrière un capot de voiture de course, on peut trouver soit un moteur performant, soit un un moteur plus banal etc.

```
architecture logique of example1 is
  -- ici les declarations necessaires a l'architecture :
  -- ...signaux notamment.
  signal w1 : std_logic;
  -- etc...
begin
```

```
  -- ici le contenu effectif
end logique;
```

Dans l'exemple donné, le nom de l'architecture est "logique" (choix évidemment libre), et le nom de l'entité associée est "example1". L'architecture est elle-même structurée : une première partie permet de placer les déclarations locales à l'architecture. Le corps véritable de l'architecture se situe entre les mots clés "begin" et "end". Noter que le "end" est suivi du nom de l'architecture.

7.4 Les éléments clés de l'architecture

C'est dans le corps de cette architecture que le concepteur peut réelement décrire son circuit en détail. Plusieurs nouveaux concepts sont mis à disposition, associés à des éléments de syntaxe que nous allons détailler. La chose importante à comprendre est qu'au sein de cette architecture, l'ordre d'écriture de ces concepts n'a pas d'importance : ceci diffère de ce que vous avez l'habitude d'écrire (python, java, etc). Ceci restitue pourtant naturellement l'idée de *décrire* le circuit (imaginé, dessiné ou plus rarement existant), à la manière d'un naturaliste qui décrirait un fleur en se laissant le droit de commencer par la tige, les racines ou les pétales.

7.4.1 Assignations concurrentes des signaux

Les assignations concurrentes s'effectuent sur des *signaux* : une assignation est la manière d'affecter un signal. Un signal peut être vu comme un fil (une équipotentielle). Les assignations s'effectuent en *parallèle* : en conséquence, un ensemble d'assignations concurrentes peuvent être vues comme un ensemble de définitions d'équations. Leur ordre n'a pas d'importance, ce qui tranche avec la notion d'assignation de variable dans un programme impératif classique. Dans l'exemple qui suit, la partie de droite de 'affectation du signal $w1$ est une expression booléenne. Dans la seconde assignation, la partie droite est plus complexe car elle est conditionnelle, mais in fine, le synthétiseur logique synthétisera 8 équations logiques (une pour chaque fil du signal $w2$), après avoir *inféré* un réseau booléen constitué d'un additionneur, d'un multiplieur 8 bits et de multiplexeurs.

```
architecture logique of example1 is
  signal w1 : std_logic;
  signal w2 : unsigned(7 downto 0);
begin
  -- ici deux assignations concurrentes

  w1 <= (a or b) and c;

  w2 <= (a+b) when cmd="00" else
        (a*b) when cmd="01" else
        to_unsigned(0,32);

end logique;
```

7.4.2 Processus

Une autre manière d'effectuer des actions parallèles, qui figure dans le corps d'une architecture, est le recours aux *processus*. Un processus est un "bout de code", évalué de manière séquentielle par le simulateur. Vous pouvez ainsi décrire des "programmes" séquentiels qui s'exécutent en parallèle et interagissent : leur moyen d'interagir est d'affecter des signaux. Le signal peut être vue comme des canaux de communication entre processus. Une des grandes difficultés de VHDL est que ces processus, *évalués* de manière séquentielle par le simulateur, peuvent être utilisés

à la description de dispositifs électroniques qui ne sont pas forcément séquentiels. Ainsi, selon l'écriture, un processus résultera, après traitement par le synthétiseur, en un circuit combinatoire ou séquentiel. Etudions quelques exemples, afin de nous permettre d'y voir plus clair.

Exemple 1 processus générant un circuit combinatoire Nous allons décrire le même circuit que précédemment, à l'aide d'un seul processus.

```
p1 : process(a,b,c)
begin
  w1 <= (a or b) and c;
  w2 <= 1;
end process;
```

Il s'agit du même exemple que précédemment codé avec des assignations concurrentes. Le concepteur a la choix de la forme qu'il utilise pour décrire son futur circuit. Ce processus (précédé d'un label optionnel –ici "p1"–), est structuré comme un programme séquentiel : chaque instruction du processus est d'ailleurs suivi d'un ";", marqueur traditionnel de séquentialité dans les langages comme C ou Java. On y retrouve un "if"..."elsif". Le simulateur (lui-même séquentiel) profitera de cette forme séquentielle pour simuler aisément le circuit ; le simulateur calcule par contre qu'aucun temps physique significatif ne s'est ici écoulé entre les instructions au début du processus et celles de la fin. Le synthétiseur, quant à lui, est capable d'inférer un matériel parfaitement combinatoire. Ces deux manières de voir le processus sont ainsi cohérentes.

Processus générant un circuit séquentiel Test

```
architecture ex2 of circuit is
  signal d1,d2,q : std_logic;
begin
  p1 : process(clk)
  begin
    if rising_edge(clk) then
      q1 <= a;
    end if;
  end process;

  p2 : process(reset_n,clk)
  begin
    if reset_n='0' then
     q2 <= "000";
    elsif rising_edge(clk) then
      if enable='1' then
        q2 <= b;
      end if;
    end if;
  end process;

end logique;
```

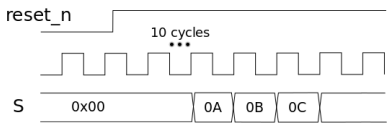
Dans cet exemple, le concepteur a cherché à décrire deux bascules D, à l'aide de 2 processus. Le *principe fondamental* pour comprendre l'inférence de bascules D est le suivant :

Toute assignation d'un signal, sous la condition d'un front d'horloge, résulte en une bascule D.

Exemple 3 : processus utilisé en banc de test Les processus sont également utilisés pour simuler des environnement de tests : ils peuvent décrire des générateurs de signaux, ou à l'inverse des instruments d'observation de signaux. Ces processus ne servent que dans des *bancs de test virtuels* (testbench) sur lesquels nous allons revenir. Ces processus n'ont pas alors vocation à être synthétisable, et l'on peut donc utiliser pleinement le langage VHDL. On qualifiera ces processus de "comportementaux".

Dans l'exemple suivant, on décrit la génération d'un signal (ou "stimulus"), à l'aide d'une processus qui opère sur plusieurs cycles d'horloge

```
architecture bhv of circuit is
  signal s : unsigned(7 downto 0);
begin
  stimulus : process(clk)
  begin
    s <=x"00";
    report "waiting for reset";
    wait until reset_n='1';
    for i in 0 to 10 loops
      wait until rising_edge(clk);
    end loop;
    report "starting sequence";
    s <= x"0A";
    wait until rising_edge(clk);
    s <= x"0B";
    wait until rising_edge(clk);
    s <= x"0C";
    wait until rising_edge(clk);
    s <= x"00";
    report "done."
    wait; --forever
  end process;

  -- autres processus ...

end logique;
```



7.4.3 Instanciation de composants et d'entités

L'instanciation consiste à utiliser un modèle de circuit, précédemment décrit. Sans cette instanciation, un couple entité-architecture ne peut être simulé : il n'est pas "incarné", mais reste un modèle imaginé. Une analogie naturelle peut être faite avec la programmation orientée objet, qui distingue bien la notion de *classe* et d'*objets* (appelés "instances"). Il faut dans notre cas imaginer que l'on dispose d'une réserve infinie de composants sur étagère,et que l'on vient construire un système en les "posant" sur une table imaginaire. Cette table imaginaire est l'architecture associée à une entité : l'instanciation se fait donc en parallèle des assginations concurrents vues précédemment. Elle permet en outre de construire des circuits hiérarchiques, à la manière des poupées russes (ou poupées gigognes) : un composant peut être composé de composants etc...

Deux formes d'instanciations sont possibles : l'instanciation de *component* et l'instanciation d'*entity*. Une fois posés, ces composants devront être connectés aux signaux en présence : horloge, reset, entrées, sorties, signaux locaux. Cette connection s'appelle le "port map". Comme nous allons le voir, il existe là aussi deux formes de port map : explicite ou positionnel.

Instanciation de composants La première forme a tendance a disparaître des pratiques, car elle requière notamment plus de texte : au sein de l'architecture receptacle, il est nécessaire de rappeler l'interface de l'entité, sous la forme d'un "component". Supposons que l'on dispose d'un modèle entité-architecture d'un circuit "PGCD", dans un fichier "pgcd.vhd". Pour instancier ce circuit, il est nécessaire de procéder comme suit explicité dans l'exemple.

```
architecture bhv of circuit is

  component MonCircuit is
    port(
      reset_n       : in  std_logic;
      clk           : in  std_logic;
      a,b           : in  std_logic_vector(31 downto 0);
      f             : out std_logic_vector(31 downto 0);
    );
  end component;

  signal aa,bb, f1,f2 : std_logic_vector(31 downto 0);

begin -- instanciation de 2 composants

  -- port map explicite, de la forme :
  --     port formel => signal
  inst_1: MonCircuit
    port map (
      reset_n => reset_n,
      clk     => clk,
      a       => aa,
      b       => bb,
      f       => f1
    );

  -- port map positionnel
  inst_2: MonCircuit
    port map (reset_n,clk,aa,bb,f2);

end bhv;
```

Instanciation d'entités L'instanciation d'entité est syntaxiquement voisine, mais plus succinte, puisque nous n'avons pas à déclarer l'existence d'un "component", comme c'était le cas précédemment.

```
architecture bhv2 of circuit is

  signal aa,bb, f1,f2 : std_logic_vector(31 downto 0);

begin -- instanciation d'une entite

  inst_1: entity work.MonCircuit(RTL)
    port map (
      reset_n => reset_n,
      clk     => clk,
      a       => aa,
      b       => bb,
      f       => f1
    );
```

```
end bhv2;
```

7.5 Décrire des machines d'états finis

7.5.1 FSM au niveau logique

Nous restreignons ici le style de description à l'UV1.5. D'autres styles de descriptions des FSMs, plus légers, seront vus en 2ième année, et sont évoqués dans la section suivante.

Comme étudié au chapitre précédent, on peut désormais décrire des FSMs à l'aide des seuls concepts d'équations logiques et de bascule D. Nous proposons le code VHDL de l'automate dont nous avons précédemment trouvé les équations.

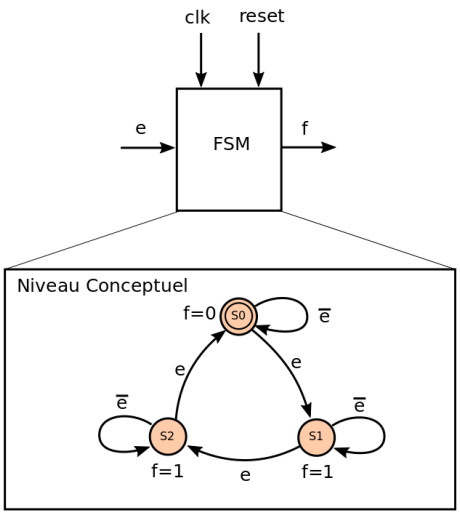
```
library ieee;
use ieee.std_logic_1164.all;

entity FSM is
 port(
     reset_n : in std_logic;
     clk     : in std_logic;
     e       : in std_logic;
     f       : out std_logic
   );
end FSM;

architecture logic of FSM is
begin

  -- description des bascules d'etat
  state: process(reset_n,clk)
  begin
    if reset_n='0' then
      q0 <= '0';
      q1 <= '0';
    elsif rising_edge(clk) then
      q0 <= d0;
      q1 <= d1;
    end if;
  end process;

  -- next state logic :
  d0 <= (q0 and not(e)) or (not(q1) and not(q0) and e);
  d1 <= (q1 and not(e)) or (q0 and e);

  -- output logic :
  f  <= q1 or q0;

end logique;
```

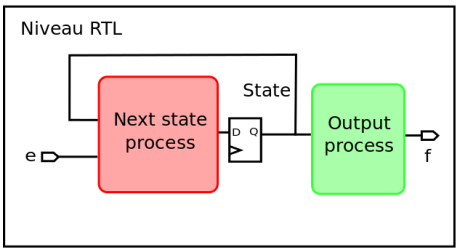
7.5.2 FSMs au niveau RTL

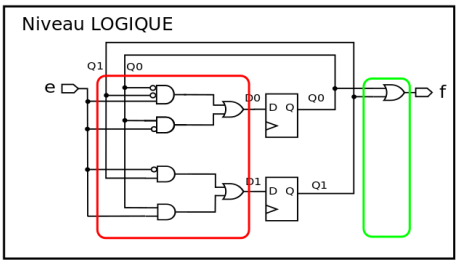
Fort heureusement, il est possible de décrire l'automate précédent de manière plus intuitive qu'au niveau logique. Le niveau d'abstraction s'appelle alors le niveau RTL : register transfert level. Le niveau RTL est associé à un ensemble de pratiques de codage, qui permettent de s'affranchir de

la description des équations logiques précédentes. Les synthétiseurs RTL sont capables d'inférer les équations logiques sous-jacentes aux descriptions RTL. Par exemple, l'automate suivant est équivalent au précédent. Pour obtenir strictement le même circuit logique, il faudra toutefois veiller à imposer d'une manière ou d'une autre l'encodage des états,ici laissé libre dans la description. Le synthétiseur réalise une exploration afin d'optimiser la réalisation, pour une cible technologique donnée. Sans contrainte d'encodage des états, il est possible qu'il choississe un circuit logique différente de celui que nous avons péniblement calculé et décrit précédemment.

```
architecture RTL of FSM is
begin

  -- description des bascules d'etat
  state: process(reset_n,clk)
  begin
    if reset_n='0' then
      state <= S0;
    elsif rising_edge(clk) then
      state <= next_state;
    end if;
  end process;

  -- transition 'function'
  next_state_logic : process(state,e)
  begin
    next_state <= state; --default
    case state is
    when S0 =>
      if e='1' then
        next_state <= S1;
      end if;
    when S1 =>
      if e='1' then
        next_state <= S2;
      end if;
    when S2 =>
      if e='1' then
        next_state <= S0;
      end if;
    when others =>
      null;
    end case;
  end process;

  -- output 'function' :
  output_logic : process(state)
  begin
    if state=S0 then
      f <='0';
    else
      f <= '1';
    end if;
  end process;

end logique;
```







Cette description est plus longue que la précédente, mais ce n'est pas le cas en général. La description RTL est généralement plus concise et surtout beaucoup plus explicite.

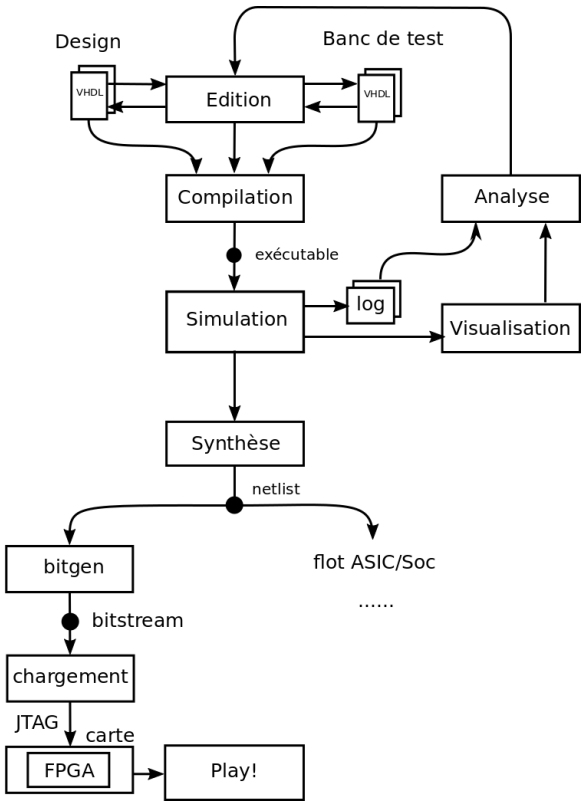
Notons que l'entité n'a pas besoin d'être à nouveau déclarée : elle est commune aux deux architectures. Dit autrement : à cette entité correspondent deux architectures (ou réalisations) différentes.

7.6 Simulation en VHDL

Nous présentons ici un survol des premiers principes de simulation en VHDL. Bien évidemment, la simple édition et compilation d'un design VHDL ne garantit pas le bon fonctionnement final du circuit : il est nécessaire de vérifier, à l'aide de ces simulations, que le comportement effectif du circuit, correspond bien aux attentes du concepteur.

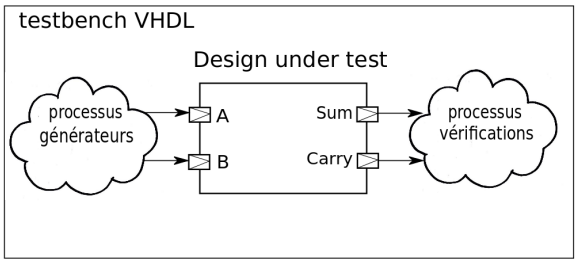
7.6.1 Flot de conception général

La simulation s'inscrit dans un *flot de conception* plus large. On utilise ce terme pour rendre compte de la succession d'étapes lors de la conception, en partant des étapes amont, vers des étapes aval. Les grandes étapes de ce flot sont outillées et donnent lieu à des spécialisations de la part des concepteurs : il est désormais impossible de maîtriser dans son ensemble les compétences nécessaires au bon déroulement de ce flot, essentiellement du fait de la multiplicités des outils mis en jeu. Ce flot est illustré sur la figure suivante. La simulation correspond à la partie amont du flot, alors que la synthèse (RTL et logique, non séparées ici) apparaît dans la partie aval.



7.6.2 Bancs de tests ou *Testbench* pour la vérification

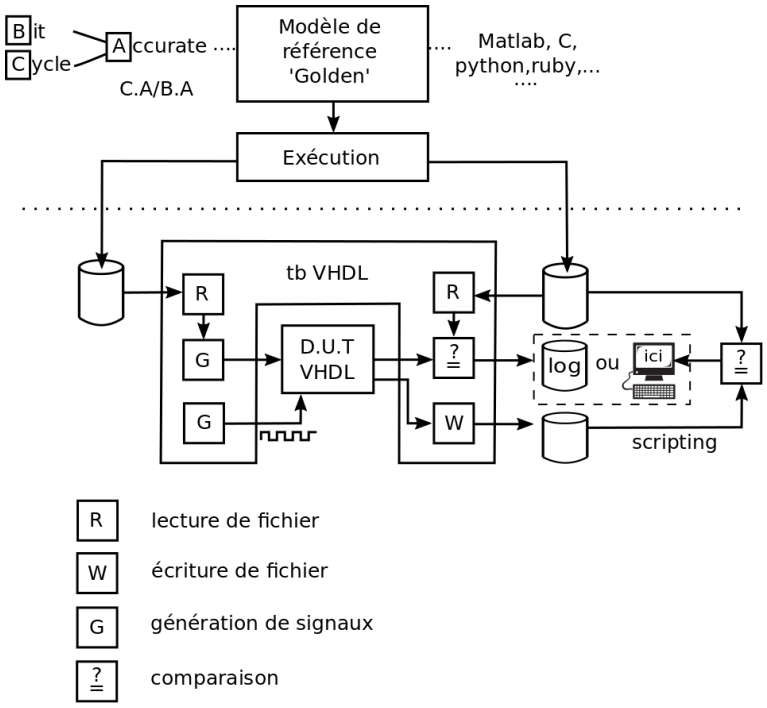
Les bancs de tests VHDL (ou testbenches en anglais) permettent de se doter d'*instruments virtuels*, également décrits en VHDL. Ces instruments sont de deux types : générateurs de signaux, et analyseurs de signaux. De manière simplifiée, et en première approximation, on peut considérer que chaque instrument revient à écrire un processus VHDL.



Génération de l'horloge

7.6.3 Modèles de référence

Le schéma précédent n'illustre pas complètement la difficulté à réellement vérifier le bon fonctionnement du circuit. Vérifier un circuit ne peut se pas se limiter à une simple "vérification visuelle" de la bonne forme de certains signaux : les designs sont généralement trop complexes pour se contenter d'une une simple "opinion". Il est nécessaire de développer des méthodologies qui permettent de *comparer* le comportement simulé avec un comportement supposé correct. Le schéma suivant illustre à l'inverse le recours à un modèle supposé correct : il est d'usage d'appeler ce modèle un *golden model*. C'est un modèle qu'on ne remettra pas en question lors de la vérification.



Ce modèle est issu le plus généralement d'équipes d'algorithmiciens. A l'issu de leur activité, il leur est possible de mettre à disposition des fichiers d'entrées-sorties de leur algorithme. La difficulté est de s'entendre sur le contenu de ces fichiers. Par exemple, les algorithmiciens n'ont pas la même perception du temps : il est rare qu'ils soient en mesure de fournir des données de référence exacte au cycle près (*cycle-accurate models*). Le plus souvent les données restent cependant précises au bit près (*bit-accurate models*). Toute sorte de modèles intermédiaires sont imaginables.

7.7 Fonctionnement interne d'un simulateur VHDL

Idée de causalité Comment fonctionne un simulateur VHDL ? Nos équations logiques et nos structures séquentielles se présentent comme un vaste enchevêtrement de signaux, interconnectés les uns aux autres, sans ordre évident. Un simulateur de circuits doit simuler l'existence de phénomènes temporels logico-physiques, qui se déroulent parfois les uns à la suite des autres, ou parfois de manière parallèle. Mais, dans tous les cas, le simulateur est un algorithme qui s'exécute sur un PC, qui est essentiellement séquentiel : l'algorithme de simulation doit être en mesure de correctement restituer l'idée de *causalité*. Le respect de la causalité est essentiel : notamment on comprend bien qu'une modification d'un signal à un temps t ne peut avoir de conséquence à un temps antérieur.

Simulateur à événements discrets L'algorithme le plus flexible pour accomplir cette prouesse s'appelle un simulateur à événements discrets. En VHDL, un *événement* est un *changement de valeurs* d'un signal donné. Cet évenement est associé à un *timestamp*, c'est-à-dire une information sur la date où doit survenir cet événement. Le travail du simulateur va donc être de gérer une vaste liste d'événements planifiés. A partir d'une première collecte effectué au temps simulé $t = 0$, un premier ensemble d'événements est enregistré. Ce premier ensemble va agir par "effet d'avalanche" : ces événements vont déclancher de nouveaux événements au fur-et-à-mesure de l'exécution du simulateur. Ces évenements "primitifs" proviennent d'une écriture spécifique utilisée par le concepteur VHDL : il est par exemple possible en VHDL d'exprimer les modifications

qu'un signal rencontrera au cours du temps. Ainsi le code suivant indique que simulateur qu'à $t = 0$ le signal s prend la valeur 1. Toutefois, il est d'ores-et-déjà enregistré que dans le futur, à $t = 12ns$, ce signal vaudra '0' etc.

```
s <= '1', '0' after 12 ns, '1' after 34 ns;
```

Cette première collecte s'effectue sur l'ensemble du système numérique décrit. Au sein des processus VHDL, cette évaluation s'arrête localement aux instructions "wait". Le processus continuera d'être évalué, au cours de la simulation, lorsque la condition du wait sera effectivement remplie. Lorsque l'évaluation atteint la fin d'un processus, le simulateur considère qu'il doit reprendre l'évaluation du processus à partir de son début.

Au cours de la collecte, le simulateur a également analysé les listes de sensibilité des processus : il sait donc que tel processus, sensible à un signal s, devra être réévalué dès lors qu'un nouvel événement survient sur s. A l'issue de cette première collecte, le simulateur cherche le timestamp t le plus proche dans le futur.

Il exécute tous les processus sensibles à ce signal, ou les reprend au point d'exécution où il s'était précédemment arrêté. C'est au cour de cette exécution que tout un ensemble de nouveaux événements sera planifié *dans le futur*.

Temps physique et délai delta L'ordonnancement des événements se fait non seulement sur le temps physique, mais également sur un temps appelé "delta", qui correspond à un temps δt infinitésimal. Ce temps est ajouté au timestamp d'un signal, lors de l'assignation de ce signal. Ce "delta delay" permet de maintenir l'idée de précédence et de causalité au sein des affectations s'effectuant au même pas de temps physique.

7.8 Utilisation du simulateur GHDL

7.8.1 Introduction

GHDL est un simulateur open source développé par Tristan Gingold, un ingénieur français. A la différence d'outils professionnels comme Modelsim de la société Mentor Graphics, GHDL s'utilise uniquement en ligne de commande. GHDL peut être être utilisé en synergie avec un second outil appelé GTKWave dans le but de visualiser des chronogrammes. Nous rappelons ici les commandes utiles qui permettent de simuler un circuit simple. D'autres commandes sont disponibles mais ne seront pas traitées ici.

7.8.2 Commandes essentielles

GHDL s'utilise en ligne de commande comme n'importe quel compilateur. Le 'G' de GHDL fait référence au projet GNU lié à l'Open source dont le fer de lance est GCC, le compilateur pour le langage C (et bien d'autres langages !). D'ailleurs GHDL utilise le même backend que GCC : il génère les mêmes fichiers binaires "*.o". A la date de rédaction (2017), une version de GHDL génère un bitcode LLVM, alternative à GCC.

Les options les plus utiles sont les suivantes, utilisée dans cet ordre :

- **ghdl a nom_fichier.vhd** : Analyse le fichier et génère un binaire (faites ls -a nom_fichier.o pour le voir si vous le souhaitez). Généralement il faut compiler plusieurs fichiers avant de passer à l'étape suivante.

- **ghdl e nom_de_l'entité_simulable** : Elaboration (ou Edition de lien), qui crée le simulateur compilé : c'est un exécutable comme un autre. Il embarque un moteur de simulation à événement discret.

- **ghdl r nom_de_l'entité_simulable** : Run de l'exécutable précédent. Il est tout à fait possible de lancer l'exécutable sans passer par cette commande.

Généralement, on cherche à enregistrer les formes d'ondes (waveforms ou chronogrammes) dans un fichier, pour une visualisation post-mortem (c'est un désavantage de ghdl par rapport à Modelsim, qui permet d'arrêter une simulation et de la reprendre etc). La manière de faire consiste à modifier la dernière commande précédente :

```
$ ghdl -r entite_simulable --wave=fichier.ghw}
```

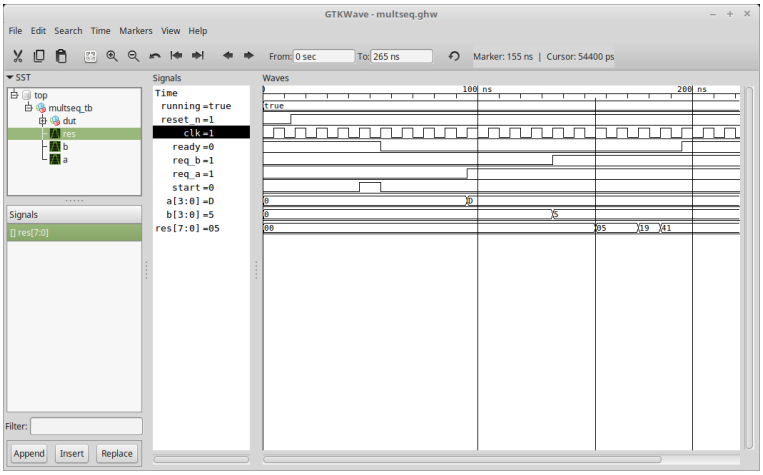
Le fichier au format ghw est ensuite lisible par le logiciel Gtkwave.

```
$ gtkwave fichier.ghw
```

Le logiciel Gtkwave permet de naviguer dans la hiérarchie de votre design, et de sélectionner les signaux que vous souhaitez visualiser. Cette sélection peut être enregistrée pour une visualisation future, en sauvant un fichier .sav (passer par le menu de gtkwave). On pourra alors relancer la simulation en demandant à GTKWave d'afficher automatiquement ces signaux.

```
gtkwave fichier.ghw signaux.sav
```

Au final, il est possible d'automatiser le processus de compilation-simulation-visualisation en copiant les commandes précédentes dans un script. N'oubliez pas de rendre ce script exécutable, en lui donnant les bons droits (chmod +x nom_script).



7.9 Conclusion

Ce chapitre nous a permis d'aborder VHDL de manière simplifiée, et d'appréhender la difficulté de mettre au point un circuit un tant soit peu complexe. La simulation requiert une forte discipline non seulement pour le concepteur, mais également pour un environnement d'ingénierie plus vaste encore, faisant intervenir d'autres spécialistes. Nous avons cantonné la simulation au niveau logique et débordé sur le niveau RTL (concernant les automates). VHDL permet d'aller plus loin dans la simulation , en simulant les délai des portes logiques en présence. VHDL permet également de remonter en abstraction : les bancs de test illustrent cette remontée, puisqu'il est possible d'y utiliser des constructions de langage plus habituels dans un "langage traditionnel". Cette richesse sera étudiée en deuxième année.

Chapitre 8

Synthèse sur FPGA

Dans ce chapitre, nous présentons la synthèse de descriptions VHDL. Notre cible est un circuit FPGA, embarqué sur une carte électronique. Dans notre cas, nous utiliserons un FPGA Nexys4DDR de la société Xilinx, implanté sur une carte Digilent.

8.1 FPGA : un circuit reconfigurable

Définition d'un FPGA Un FPGA est l'acronyme de *field programmable gate array*. C'est un circuit numérique qui peut être vu comme un vaste receptacle d'équations logiques de de bascule D. Un FPGA peut désormais en contenir plusieurs millions. Mais le plus intéressant tient au fait que les FPGA sont des circuits reconfigurables : cela signifie qu'ils peuvent être reprogrammé à volonté. Cette "reprogrammabilité" leur confère des traits généralement associés au domaine du logiciel, (s'exécutant sur un processeur versatile par nature). Les FPGA ont donc des caractéritiques de performances (dûes au parallélisme des équations fonctionnant simultanément) et de reprogrammabilité.

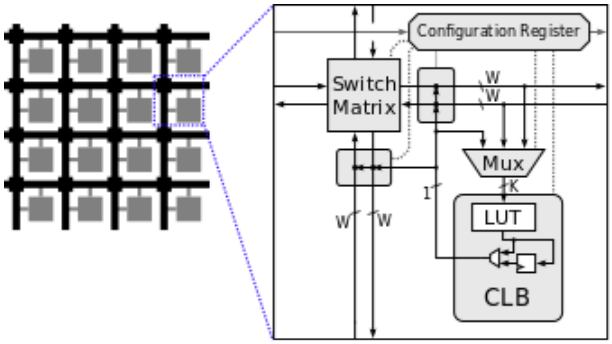


FIGURE 8.1: Architecture des FPGA : routeurs, BLE, connection box et configurations

Architecture interne L'architecture d'un FPGA se présente comme un réseau à deux dimensions [1] de blocs configurables appelés CLB (*configurable logic blocks*) interconnectés par des *routeurs* (ou matrice d'interconnexion ou *switch matrix*). [2]. Ils sont eux-mêmes constitués d'une

[1] Des travaux de recherche avancés proposent la 3D...
[2] Ces CLB ont été inventés par Page et Peterson en 1985

Look-up table (LUT) à n entrées : c'est une mémoire qui permet d'enregister les valeurs attendues de n'*importe quelle fonction booléenne de n entrées*. C'est l'équivalent d'une petite table de vérité.

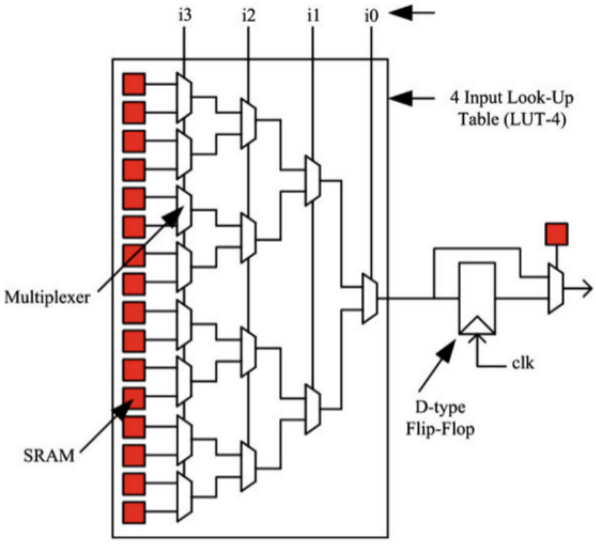


FIGURE 8.2: Architecture d'une table de lookup à 4 entrées (LUT4)

Ces éléments (routeur, connexion box, LUT) sont présentés succintement sur les figures 8.2 et 8.3

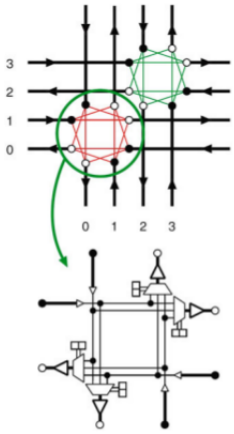


FIGURE 8.3: Architecture d'un routeur

Comparaison avec les ASICs Pour rappel, les circuits logiques peuvent également être fondus sur des circuits d'un autre type : les ASICs. Ces ASICs (application specific integrated circuits) sont plus performants que des FPGA en terme de densité de portes logiques, de consommation et de fréquence d'exécution. Mais, comme leur nom l'indique, les ASICs ne sont pas fondamentalement reconfigurables (pour qu'ils le soient il faut les concevoir comme tels). Ils visent des applications déterminées et fixées une bonne fois pour toutes. Le coût de fabrication complet d'un ASIC est prohibilitif dès lors que seul un faible nombre de pièces sont sensées être produites : pour qu'il soit rentable, il faut viser plusieurs dizaines à plusieurs millions de pièces. A noter que le coût

brut du Silicium n'est pas le budget le plus important. C'est essentiellement la fabrication des *masques lithographiques* qui coûte cher.

Les FPGA ont tendance à s'opposer aux ASICs de ce fait. Toutefois, dans les faits, FPGAs et ASICs se révèlent complémentaires : tout d'abord les FPGA permettent de *prototyper* tout ou partie des futurs ASICs, dans des conditions proches du produit final (mêmes captures VHDL, mêmes chaînes de synthèse etc). Le coût à la pièce d'un FPGA est plus élevé qu'un ASIC, mais sa disponibilité immédiate est intéressante. Certains domaines comme le militaire ou le médical profitent donc avantageusement des capacités de FPGA.

8.2 Flot de synthèse

Le flot de synthèse que nous allons expérimenter est présenté sur le schéma suivant. A partir des sources VHDL et de contraintes diverses (mises en relation des entrées et sorties des entités VHDL et des ports physiques du FPGA, etc), nous synthétisons un bitstream : c'est un fichier binaire capable de programmer le réseau précédemment étudié (CLB, configurations, Switch matrix etc). Nous avons alors recours à un second outil, qui permet de programmer la carte effectivement.

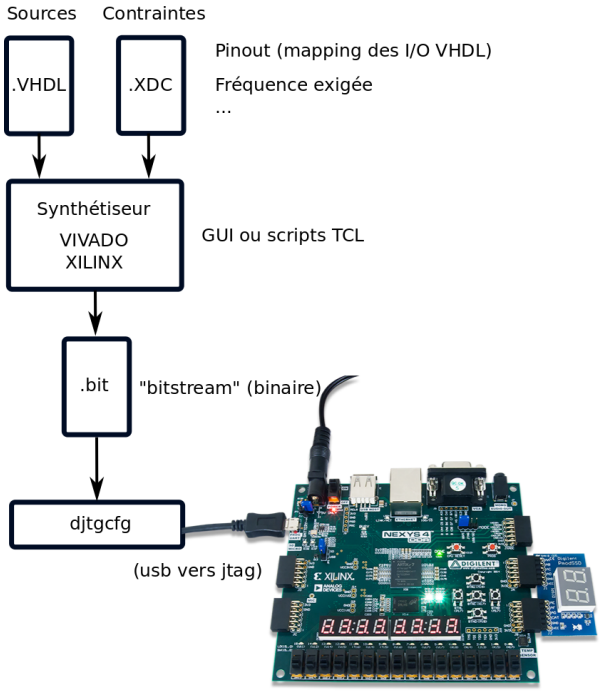


FIGURE 8.4: Aperçu du fLot de synthèse

8.3 Expérience pratique

En séance de travaux pratiques, nous allons synthétiser un algorithme sur FPGA. Cet algorithme consiste à réaliser une multiplication par additions itérées, mais la démarche vaudrait pour un algorithme un peu plus complexe. Pour passer à de "véritables" algorithmes, nous devrons nous doter de mémoires (non vues ici en première année) et d'un codage plus approprié que nos simples équations logiques : il faut passer au niveau dit "RTL", où les équations logiques sont avantageusement remplacées par des constructions syntaxiques de VHDL, intuitives et

automatiquement traduites en équations logiques par les outils (notion d'inférence). Ceci sera étudié en deuxième année. Le présent exercice a tout de même l'intérêt de "démontrer" si besoin que nos équations logiques sont justes ! Tant les FSM que les chemins de données sont correctement réalisées avec des équations que nous avons eu le mérite d'écrire et de simplifier "à la main".

La synthèse sur la cible FPGA Xilinx se fait à l'aide d'un script écrit par nos soins dans le language TCL. Ce langage pilote la plupart des outils de CAO moderne. Ce script se lance en ligne de commande comme ceci :

```
$ vivado -mode tcl -source script.tcl
```

A l'issue de cette synthèse (qui peut durer plusieurs minutes), un fichier de bitstream est produit (*.bit*). C'est ce fichier qui configure le circuit FPGA final. Le chargement du fichier sur la carte se fait ici par un cable USB entre le PC et la carte. Le chargement se fait à l'aide d'un outil logiciel fourni par Digilent.

```
$ djtgcfg prog -d Nexys4DDR -i 0 -f top.bit
```

Votre circuit est alors prêt à l'emploi !

Bibliographie

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Systems : A Programmer's Perspective.* Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[2] Pong P. Chu. *RTL Hardware Design Using VHDL : Coding for Efficiency, Portability, and Scalability.* Wiley-IEEE Press, 2006.

[3] Pong P. Chu. *FPGA Prototyping by VHDL Examples : Xilinx Spartan-3 Version.* Wiley-IEEE Press, 2008.

[4] Alan Clements. *The Principles of Computer Hardware.* Oxford University Press, Inc., New York, NY, USA, 3rd edition, 2000.

[5] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing).* Wiley-Interscience, 2006.

[6] Randy H. Katz. *Contemporary Logic Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[7] M. Morris Mano. *Digital Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2001.

[8] Neil Weste and David Harris. *CMOS VLSI Design : A Circuits and Systems Perspective.* Addison-Wesley Publishing Company, USA, 4th edition, 2010.