Système digital : de l'algorithme au circuit

Jean Vuillemin

21 juin 2015

Département d'Informatique
Ecole Normale Supérieure

Al Khowarizmi (étymologie du mot algorithme) était un savant persan du IX^{e} siècle. Son ouvrage *Al Jahbar* (étymologie du mot algèbre), regroupe les connaissances mathématiques de l'époque, d'origines grecque, arabe et indienne. C'est un recueil de méthodes de calculs, illustré d'exemples significatifs : notre modèle !
L'*astrolabe* permet de connaître sa position sur terre, en la comparant à celle des astres vus dans le ciel. Cet instrument sera remplacé d'abord par le *sextant*, puis maintenant par le GPS - *Geographic Positional System*. Le GPS combine trois technologies de pointe - électronique, espace et télécommunication - et il tient aussi dans la main.

Planche 1 – Al Khowarizmi tenant l'astrolabe

Plan

Introduction

D'après un haut relief de Saqqara en Egypte. Bec au vent, les oiseaux montrent que *l'ordre de lecture* va de droite à gauche. Ces hiéroglyphes *comptent* les tributs payés à Pharaon, lors d'une campagne particulièrement victorieuse.
Chaque signe vaut une puissance de dix : 1 pour la *barre simple*, 10 pour le *fer à cheval*, 100 pour le *serpent*, 1 000 pour le *lotus*, 10 000 pour l'*obélisque*, et 100 000 pour la *salamandre*.
Les quatre nombres qui figurent ici s'écrivent, en décimal : 11 110_{10} (haut gauche), 121 200_{10} (haut droit), 111 200_{10} (bas gauche) et 121 022_{10} (bas droit).

Planche 2 – Nombres hiéroglyphes

1 Contexte historique

L'*Informatique* (science du traitement automatique, de la mémorisation, et de la communication de l'*information*) présente un cas unique dans notre histoire intellectuelle, industrielle et sociale.

1.1 Numérations écrites

L'histoire des numérations écrites est longue et variée : lire [1] et [2] qui font autorité en ce domaine. La base 10 est, de loin, la plus commune - Egypte, Inde, Chine. Les Mayas d'Amérique utilisaient la base 20. Les civilisations de Sumer et Babylone se servaient de la base 60 (*planche* 4), qui subsiste dans nos mesures du temps, et des angles.

Dans une *numération alphabétique* - Egypte, Grèce, Chine - chaque symbole possède une valeur numérique qui est indépendante de sa position dans le nombre

Décomposition numérique de l'œil d'Horus, d'après K. Sethe.

Planche 3 – L'œil d'Horus

(*planche* 2). L'ordre d'écriture des chiffres n'a pas d'importance. On a pas besoin de notation pour zéro. Observez pourtant que les hiéroglyphes de la *planche* 2 sont proprement alignés : comme dans une *numération de position*, et comme sur un abaque de calculs.

Dans une *numération de position* - Babylone, Inde, Maya - la valeur d'un symbole/chiffre est multiple d'une puissance de la base. Son exposant est donné par la position du chiffre dans le nombre. Il faut alors une représentation explicite du zéro, pour marquer la position d'une colonne vide dans l'abaque. Un simple espace est la façon naturelle de marquer le zéro. L'écriture du symbole 0 évite les ambiguïtés de lecture que l'on rencontre en utilisant l'espace : caractère *invisible*. Cette écriture explicite de 0 apparaît plus tard, en Inde (*planche* 5).

Cette tablette - d'après YBC 7289, collection babylonienne de Yale - présente trois nombres écrits à la plume (on dit cunéiforme) sur une argile, cuite il y a près de 4000 ans. Ce sont : 1/2, $\sqrt{2}$ et $1/\sqrt{2}$. La précision est de six chiffres décimaux : c'est plus que bien des mesures de la physique moderne.

Planche 4 – Tablette babylonienne

Nombres de Babylone

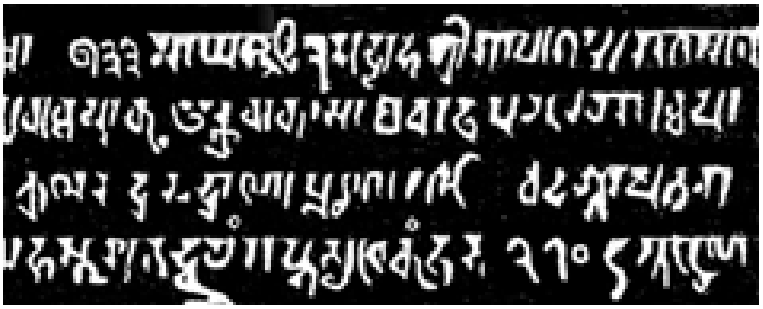
Les numérations de Sumer et Babylone sont *sexagésimales* : base 60.

Les chiffres s'écrivent avec deux symboles : encoche avec la plume verticale pour les unités, horizontale pour les dizaines. Ainsi, le chiffre en haut à gauche de la *planche* 4 vaut 30. Le nombre du centre est formé de la suite de chiffres sexagésimaux [1 24 51 10] et celui du bas de [42 25 35], en lisant de gauche à droite des chiffres forts vers les chiffres faibles. Les nombres de Babylone s'écrivent donc avec deux symboles, comme le binaire. Il convient d'y ajouter l'espace, qui sert à marquer le zéro - avec toutes les ambiguïtés de lecture qu'une telle convention peut amener.

L'espace sert aussi à marquer la position du point fractionnaire : on doit reconstruire sa position d'après le *sens* des valeurs trouvées. Heureusement, elles sont claires pour l'exemple choisi. Pour interpréter correctement la tablette de Yale (en se laissant guider par le dessin du carré et de ses diagonales qui accompagne l'image) il faut ajouter un point fractionnaire : en tête du nombre en haut, en bas, et après l'unité du nombre central.

On trouve alors, en base 60 et 10 :

$$\begin{aligned} [0 \cdot 30]_{60} &= 0 \cdot 5_{10} \\ [1 \cdot 24\ 51\ 10]_{60} &= 1 \cdot 414213 \cdots_{10} \\ [0 \cdot 42\ 25\ 35]_{60} &= 0 \cdot 707106 \cdots_{10} \end{aligned}$$



Ce détail d'une inscription sanskrite contient la plus ancienne écriture attestée de nombres décimaux, avec représentation explicite du zéro : 933 sur la première ligne (à gauche) et 270 sur la dernière (à droite). Guitel [1] donne 876 pour date.

Planche 5 – Stèle de Gwalior, Inde 876

Nombres décimaux

L'écriture décimale naît en Inde au VIII^{e} siècle. Elle représente les nombres entiers avec dix symboles, dont le (célèbre) zéro, qui reçoit enfin sa première écriture explicite attestée - *planche* 5.

L'écriture décimale indienne est ensuite adaptée pour la langue arabe. C'est au travers de la civilisation arabe que l'écriture décimale parvient de l'Inde en Occident. Il ne s'impose définitivement en Europe que vers le XV^{e} siècle, pour y supplanter la numération romaine. Au XXI^{e} siècle, on écrit les nombres de la même façon du Tibet à la Patagonie : chacun sait lire les nombres de l'autre. Le système décimal est, de ce fait, la toute première *norme internationale de communication*.

L'arabe s'écrit de droite à gauche, à l'inverse du sanskrit. Pourtant, l'ordre indien avec le chiffre le plus significatif à gauche est conservé dans l'écriture arabe. Pour lire et écrire l'arabe, il faut donc changer de sens, entre le texte et les chiffres ! Dans la transcription latine qui suit, les scribes ne prennent pas plus de risques : ils gardent intact l'ordre d'écriture des chiffres arabes. Il en résulte, après ces deux transpositions, que nous lisons et écrivons les chiffres dans le même ordre que le texte, comme en sanskrit. Le fait que nous sachions lire ce fragment (*planche* 5) de la stèle de Gwalior est une pure coïncidence : les autres chiffres du document ne sont lisibles que par les érudits. Remarquons quand même que, seul le chiffre zéro est resté inchangé : il s'écrit pareil, en sanskrit, arabe, chinois, et romain moderne.

Calculs de romains

La numération romaine - utilisée en Occident jusqu'à la *Renaissance* - sert encore pour les dates et certaines paginations. Sa complexité apparaît lors de la convertion d'un entier naturel N, en la chaîne de caractères $\mathcal{R}(\mathrm{N})$ qui représente son écriture en chiffres romains.

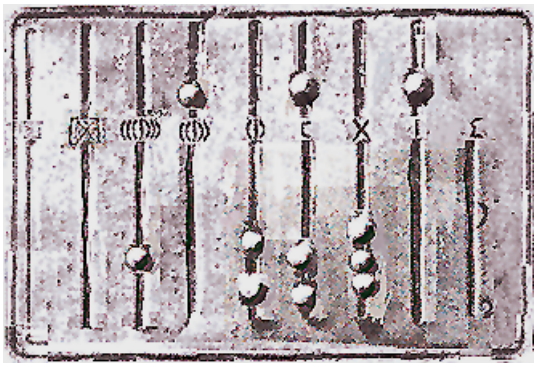
Algorithme 1 (Ecriture romaine des entiers) *On représente l'entier* $\mathrm{N} \in \mathbf{N}$ *sous forme d'une suite de chiffres romains, en appliquant les règles :*

$$\begin{array}{rcrcl}
1000 \leq \mathrm{N} < 6000 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{M}\ \mathcal{R}(\mathrm{N}-1000) \\
900 \leq \mathrm{N} < 1000 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{CM}\ \mathcal{R}(\mathrm{N}-900) \\
500 \leq \mathrm{N} < 900 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{D}\ \mathcal{R}(\mathrm{N}-500) \\
400 \leq \mathrm{N} < 500 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{CD}\ \mathcal{R}(\mathrm{N}-400) \\
100 \leq \mathrm{N} < 400 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{C}\ \mathcal{R}(\mathrm{N}-100) \\
90 \leq \mathrm{N} < 100 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{XC}\ \mathcal{R}(\mathrm{N}-90) \\
50 \leq \mathrm{N} < 90 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{L}\ \mathcal{R}(\mathrm{N}-50) \\
40 \leq \mathrm{N} < 50 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{XL}\ \mathcal{R}(\mathrm{N}-40) \\
10 \leq \mathrm{N} < 40 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{X}\ \mathcal{R}(\mathrm{N}-10) \\
9 \leq \mathrm{N} < 10 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{IX}\ \mathcal{R}(\mathrm{N}-9) \\
5 \leq \mathrm{N} < 9 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{V}\ \mathcal{R}(\mathrm{N}-5) \\
4 \leq \mathrm{N} < 5 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{IV}\ \mathcal{R}(\mathrm{N}-4) \\
1 \leq \mathrm{N} < 4 & : & \mathcal{R}(\mathrm{N}) & = & \mathbf{I}\ \mathcal{R}(\mathrm{N}-1) \\
0 \leq \mathrm{N} < 1 & : & \mathcal{R}(\mathrm{N}) & = &
\end{array}$$

L'algorithme 1 d'écriture en chiffres romains est présenté ici par des règles de calcul, à utiliser pour remplacer l'expression de gauche par celle de droite quand la condition sur N est satisfaite. Ainsi l'écriture de la date décimale 1999_{10} demande 4 applications des règles :

$$\mathcal{R}(1999) \Rightarrow \mathbf{M}\mathcal{R}(999) \Rightarrow \mathbf{MCM}\mathcal{R}(99) \Rightarrow \mathbf{MCMXC}\mathcal{R}(9) \Rightarrow \mathbf{MCMXCIX}.$$

Songez aux problèmes pratiques de calculs posés par cette écriture ! On se fera une idée précise de la difficulté en réalisant le comput ecclésiastique de l'an 800 avec les moyens de l'époque. Ceux-ci comprennent les chiffres romains et l'algorithme du calendrier Julien, qu'on trouve en section 5.4.1 ; l'utilisation d'abaque de calcul est autorisée, mais pas la notation moderne des classes de congruences (modulo), que l'on doit à Gauss (fig. 2.2).



Les chiffres sont marqués par des cailloux. Le nom latin pour caillou est *calculus* - étymologie du mot calcul.
Les lettres au centre indiquent la valeur des cailloux placés en bas de colonne ; un caillou placé en haut de colonne vaut cinq fois plus. Sauf pour celle de droite qui contient des nombres fractionnaires, chaque colonne correspond à une puissance de dix, de 1 à 10 millions. L'abaque ci-dessus représente l'entier décimal $152\,735_{10}$.

Planche 6 – Boulier romain

1.2 Calculs manuels

Lucy - notre ancêtre supposée de mille siècles - avait dix doigts. Elle s'en servait pour *compter* : probablement comme vous et moi. Si ce n'est d'elle, c'est de ses descendants que nous vient le *calcul décimal* à deux mains.

L'homme apprend ensuite à mener des calculs plus complexes, en déplaçant des cailloux sur un abaque. Le boulier (*planches* 6 et 4.3) est la forme moderne de ces outils primitifs. Il est encore largement utilisé dans le monde : ce compromis harmonieux entre ergonomie, simplicité et coût, reste hors de portée de la concurrence électronique, pour bien des cas utiles.

En contraste avec cette variété des écritures de nombres, les représentations sur abaque de calcul utilisent directement le système décimal, ou ses variantes simples.

L'histoire des calculs sur abaques, qui est moins bien documentée que celle des numérations écrites [1], semble pourtant beaucoup plus simple. Tous les instruments de calcul manuel dont nous ayons gardé trace et dont nous comprenons la fonction, reposent sur le système décimal de position, au codage des chiffres décimaux et au support près - sable, bois, pierre, papier.

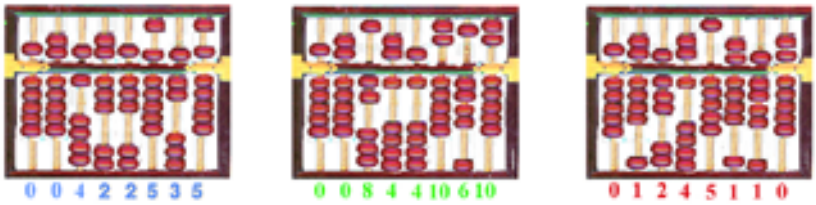
Zéro se marque (invariablement et économiquement) par une colonne vide.

Les abaques de calcul des Romains (nous venons de critiquer leur numération écrite) sont semblables à celles des Grecs (leur numération n'est d'ailleurs pas meilleure que celle de Rome), Egyptiens, Chinois, Indiens, et des autres. L'utilisation du système décimal dans les calculs pratiques semble précèder son écriture, explicite et conservée à nos jours, de plusieurs millénaires.

Vérifions, à l'aide d'un boulier contemporain, l'affirmation de la *planche* 4 :

$$2 \times [0 \cdot 42\,25\,35]_{60} = [1 \cdot 24\,51\,10]_{60}.$$

Ceci se fait en trois étapes. Partant de la configuration initiale $I = [0 \cdot 42\,25\,35]$, on double chaque chiffre pour représenter : $2I = [0 \cdot 84\,50\,70]$. En prenant alors soin de reporter les retenues à 6 sur les colonnes paires, et à 10 sur les impaires, on trouve : $2I = [1 \cdot 24\,51\,10]$.



1.3 Algorithmes

La vitesse des opérations sur les bouliers stagne pendant plusieurs millénaires. Les seuls progrès significatifs viennent des *algorithmes*, c'est à dire des méthodes de calcul utilisées. L'histoire de ce sujet est aussi riche qu'ancienne.

Par exemple, l'algorithme de multiplication binaire par additions et décalages, qui se pratique dans presque tous les microprocesseurs modernes, n'a pas changé depuis l'Egypte antique - *planche* 4.2. Plus jeune d'un bon millénaire, l'algorithme d'Euclide ne varie pas, lui non plus, avec le temps. Il en est ainsi des algorithmes *fondamentaux* : ceux qui sont indispensables, et pour toujours.

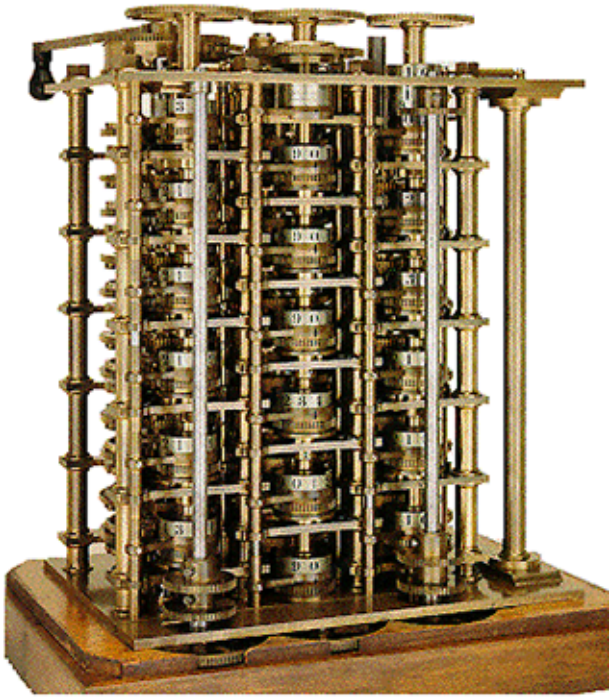
Grace aux algorithmes, les applications du calcul se multiplient, au fil des âges. Il faut calculer le calendrier, les impôts, les récoltes, les plans, les marées, les cartes, les trajectoires, et bien d'autres choses. La *planche* 1 illustre l'étymologie des deux mots : *algorithme* et *algèbre*. Elle évoque le rôle prépondérant que joue la civilisation arabe dans le développement du calcul décimal, et de ses applications en occident jusqu'au XIV^e siècle. L'adoption du système décimal permet de reproduire sur du papier les calculs du boulier et de les conserver pour référence. Ceci ouvre la porte à des calculs plus longs et plus fiables. L'algèbre permet alors de spécifier l'algorithme par une suite d'expressions qui économise les opérations à calculer, sur chaque jeu d'entré.

L'homme, qui calcule les opérations de ces algorithmes, est la première forme de *machine universelle* : il sait réaliser tout calcul, si on lui donne du temps, des crayons et du papier. Les marins, les astronomes, les banquiers et les savants du XVI^e siècle font tous leurs calculs à la main. Ils s'aident de tables numériques, qui sont propres aux *algorithmes* de chaque métier.

La limite de cette méthode ne vient pas la nature des calculs, mais de leur longueur. Si certains opérateurs humains réussissent à calculer plus d'un chiffre d'addition décimale par seconde, la fatigue amène *toujours* des erreurs, à la longue. Les tables dont ils se servent sont aussi calculées à la main, et elles contiennent

donc aussi des erreurs. D'autant plus que le typographe en rajoute, avant même qu'elles ne soient reproduites par l'imprimerie : il doit en effet disposer, sur une matrice en bois, de pleines pages de chiffres en plomb. Comment ne *pas faire d'erreur*, en copiant *à la main*, la table sur 40 pages des logarithmes (plus de 100 000 chiffres) ? Tout calcul manuel devient forcément faux, au-delà d'une certaine longueur.

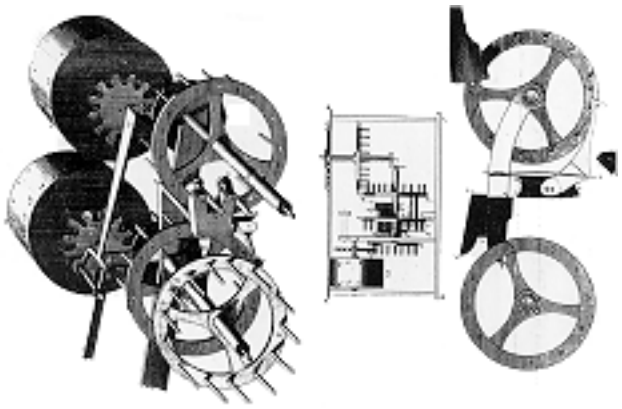
Calculs mécaniques



Cette partie de la machine différentielle de Babbage est assemblée en 1832 par Clement. Elle comprend plus de 2 000 engrenages, et a coûté 20 000 livres à l'amirauté britannique. Une réplique marche impeccablement au British Museum ; mais ce n'est qu'une partie du plan initial de Babbage.
Sa fonction est d'évaluer des polynômes par différences finies, pour construire automatiquement *et sans erreur* les tables de calcul.

Planche 7 – Machine différentielle de Babbage, 1832

C'est pour tenter de réduire les erreurs, et donc de permettre des calculs plus longs, que les savants et les ingénieurs réalisent des machines mécaniques, comme la *Pascaline* en 1642 - *planche* 4.1. L'une après l'autre, les quatre opérations de l'arithmétique décimale sont *représentées* par le mouvement d'engrenages, coordonnés par des roues dentées - *planche* 8. Ces techniques d'*horlogerie* culminent



Extrait de l'Encyclopédie de Diderot, rubrique arithmétique et calcul.

Planche 8 – Report des retenues dans la machine de Pascal

en 1832, avec la réalisation de la machine différentielle de Babbage - *planche* 7. C'est une machine spécialisée dans l'évaluation de polynômes de degré 6, aux valeurs entières consécutives. Sa fonction est de calculer, *automatiquement et sans erreur*, les tables dont se sert l'amirauté britannique. Il marche encore mais son développement n'a pas suivi : il suffit d'un tour de manivelle pour voir s'afficher l'entrée suivante de la table, mais il reste encore à reporter - à la main et sur du papier - les chiffres lus aux cadrans de la machine. C'est long, et des erreurs s'introduisent. Au-delà des succès techniques et scientifiques bien reconnus, toutes ces machines à calcul *mécanique* sont des échecs économiques : trop cher, trop fragile et trop lent pour remplacer le papier et le crayon, manipulatés de main d'homme. Ces calculateurs mécaniques à engrenages ne sont véritablement utilisés - dans le commerce comme caisses enregistreuses - que pendant la première moitié du XX^{e} siècle.

Près d'un siècle après la machine à additionner de Pascal, Jacquard invente un dispositif à aiguilles et cartons perforés qui permet de contrôler automatiquement (numériser, programmer) les motifs à réaliser par le métier à tisser - *planches* 9 et 10. En 1801 le *métier Jacquard* - muni de 24 000 cartes - tisse le portrait de son concepteur.

Cette machine spécialisée dans la reproduction d'images sur la soie connaît un grand succès : Lyon compte plus de 30 000 métiers Jacquard à la mort de son inventeur en 1834. Avec la mécanisation vient aussi la révolte sociale des *canuts*. Pour chaque machine introduite, un emploi est supprimé : Jacquard fut sauvé plusieurs fois par la police de tentatives contre sa vie. Plus calme socialement est le disque rigide à ergots, qui remplace le carton perforé dans les automates issus de l'horlogerie helvétique. Pour la bonne vitesse de rotation, on fait jouer de la musique numérique par des marionnettes animées et des pianos mécaniques : premier multimédia digital en temps réel ? En combinant le lecteur de cartes perforées à un compteur mécanique, Hollerith gagne pourtant le contrat des machines de recense-

Dans le métier à tisser, le motif à réaliser est codé sur des cartes perforées - *planche* 10 - qui commandent directement la mécanique servant à lever les trames. Suivant qu'une trame est levée ou non, le fil attaché à la navette passe au-dessus ou au-dessous du point correspondant qui devient visible ou non dans l'image tissée au final.

Planche 9 – Métier à tisser Jacquard

ment aux Etats Unis ; il fonde en 1896 une compagnie, dont le nom deviendra en 1924 - après quelques fusions *IBM - International Business Machines*.

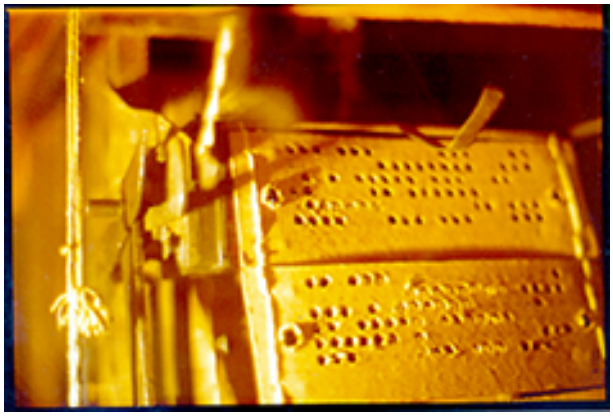


Planche 10 – Carte perforée du métier à tisser.

Machines universelles

Les formes que prennent les machines à calculer se multiplient autant que leurs fonctions. Pendant la réalisation de sa machine *différentielle* - *planche* 7 - Babbage perd son intérêt pour ce projet.

Il a en effet une *vision* de calcul beaucoup plus générale et élégante, et il passe le reste de sa vie sur les plans de l'*Analytical Engine* - *planche* 5.1. C'est la première machine à *usage universel* : elle est capable d'effectuer des calculs automatiques, et d'en contrôler l'enchaînement par *programmes*. La comtesse *Ada de Lovelace* - fille de *Lord Byron* qui donne son prénom au langage ADA - conçoit divers programmes pour cette machine, dont les premiers pour la composition musicale.

Sa machine va le ruiner, car le concept de Babbage est trop en avance sur les techniques mécaniques de son époque. Sa vision nécessite plus de 30 000 engrenages et cartes perforées : trois tonnes de pièces mobiles. Elle ne sera réalisée que partiellement - et quarante ans après la mort de Charles - par son fils Henry : *planche* 5.1.

Le sens qu'il faut donner au mot *universel* devient clair en 1936, avec la *thèse* de Church et Turing, qui caractérise mathématiquement ce qui est *calculable automatiquement*, et ce qui ne l'est *pas* - chapitre 6. La machine *analytique* est une forme particulière machine à calculer, capable de remplir à elle seule *toutes* les fonctions du calcul automatique, à la taille et à la vitesse près.

Calculs électroniques, et système binaire

C'est le développement au début du XX^e siècle de l'électromécanique et de l'électronique - *planche* 11 - qui rend enfin possible la construction machines à calculer complexes : Allemagne (Zuse), France (Coufignal) et Etats Unis (Stibitz).

L'usage du système binaire est fécond au travers des mathématiques antiques - *planches* 3 et 4.2. Il faut pourtant attendre 1703 pour que Leibnitz montre expli-

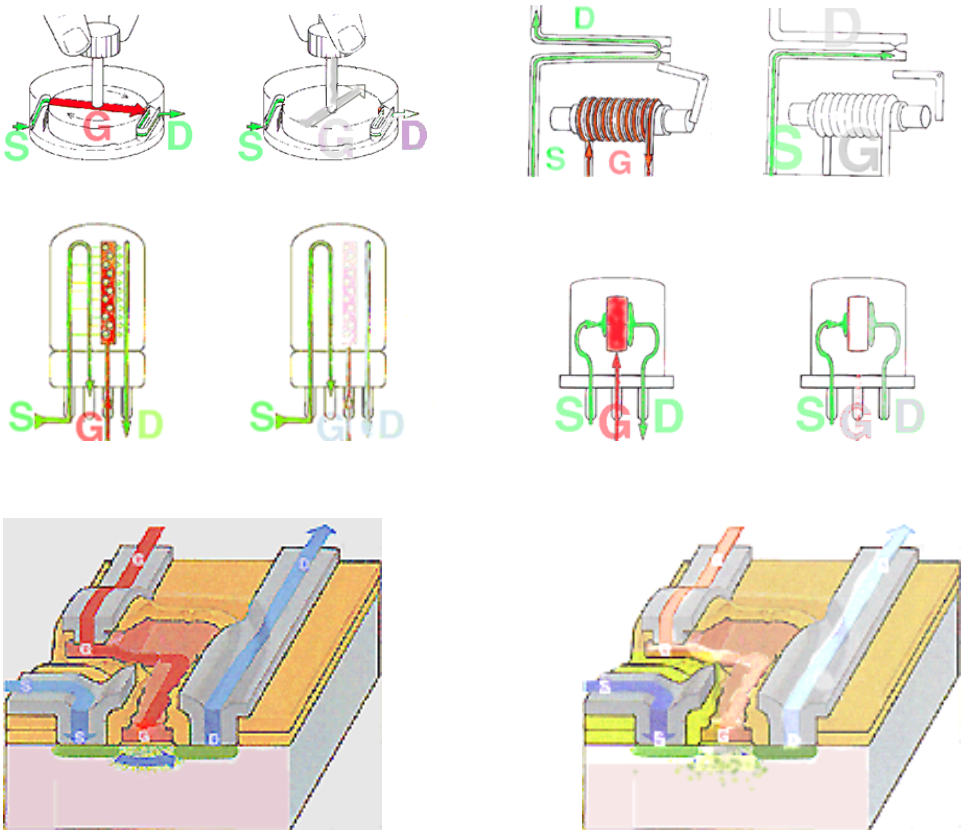


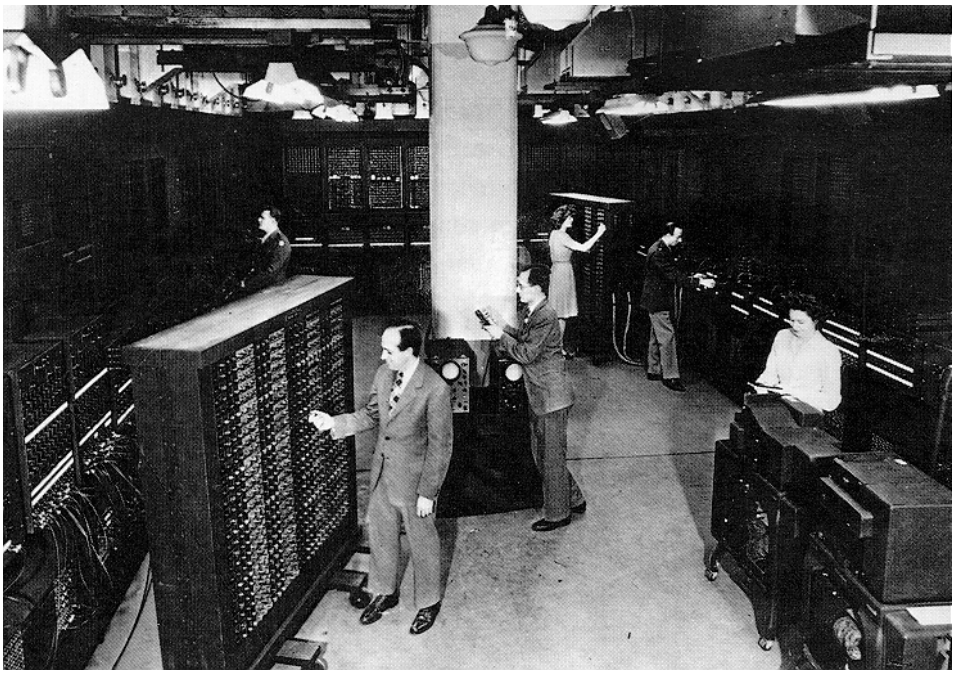
Planche 11 – Interrupteurs à commande

citement la profonde simplicité que prennent les quatre opérations $+, -, \times, \div$ de l'arithmétique, dans cette numération. Entré dans la pratique courante depuis lors, le système binaire se marie aux techniques électroniques - relais, lampes à vide, transistor - pour devenir le codage de choix dans les machines à calculer du XX^{e} siècle.

Pourtant, jusqu'à la dernière guerre mondiale, les questions de calcul ne sont primordiales ni dans les sciences, ni dans les techniques. Les besoins qui en calculs militaires - radar, sonar, cryptographie, bombe, balistique - changent radicalement la situation.

1.4 Hier

Après la guerre, l'Université de Pennsylvanie utilise des tubes à vide pour réaliser l'ENIAC, premier ordinateur électronique - *planche* 12. Dans une première version de l'ENIAC, le contrôle du chemin des données était câblé en fonction de l'algorithme en cours d'exécution. Cette exécution terminée, on arrêtait la machine pour procéder au câblage nécessaire à l'algorithme suivant. *Cela prenait des*



L'ENIAC - *Electronic Numerical Integrator And Calculator* - à l'Université de Pennsylvanie. Au premier plan, les deux concepteurs : Eckert et Mauchly.

Planche 12 – ENIAC, 1947

heures ! C'est à *von Neumann* que l'on attribue l'idée de *programme enregistré*, qui résout le problème. Le programme est enregistré dans la même mémoire que les données ; ceci se fait dans le mode de configuration de la machine, avec chargement automatique des données initiales. Une fois configurée, la machine passe dans son mode standard : exécution automatique du programme enregistré sur les données en mémoire. L'exécution terminée, on lit les résultats du calcul à partir de la mémoire.

La structure générale de l'ENIAC (unité de calcul, mémoire digitale commune aux données et programmes - *planche* 12) donne les grandes lignes de celle des ordinateurs qui lui succèdent, de 1950 à nos jours. La machine dite de *von Neumann* est née.

En 1947, trois physiciens des laboratoires Bell - *Bardeen, Shockley, Brittain* - inventent le *transistor* - *planche* 13. Il permet de contrôler la valeur du courant qui passe dans un *semi-conducteur*, et réalise ainsi un *interrupteur programmable* : plus petit, plus fiable, et moins cher que les *lampes à vide* et les *relais électromagnétiques*, jusqu'alors en usage - *planche* 11.

C'est à partir de 1947 - année de naissance du transistor, de l'ordinateur programmable et de la théorie mathématique de la communication - que s'amorce un phénomène de *boule de neige*, entre science, technologie et économie, qui fait basculer le monde dans un nouvel *Age*, celui de l'*information* et de ses *calculs*.

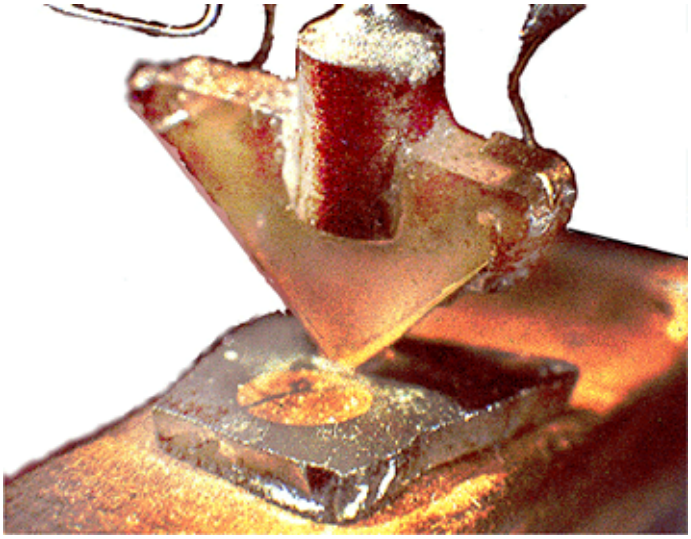


Planche 13 – Le premier transistor, 1947

Dans les années 50, la radio à transistors, économique et portable, offre au monde la communication de masse par les ondes. Les bénéfices en retour servent à financer la mise au point de la technologie suivante. C'est le *circuit intégré* des années 60 - *planche* 15.

Bientôt, on *intègre* de plus en plus de transistors connectés sur le même support physique à état solide. En combinant tous ces transistors sur une seule *puce*, on peut réaliser des fonctions de plus en plus complexes, comme par exemple celles de la *calculatrice* électronique. Fin 60, elle remplace la règle à calcul dans les lycées, et rentre dans l'électronique *grand public*. Dans le même temps, la radio bénéficie aussi des circuits intégrés ; elle étend son marché, par des modèles de haute fidélité, comme par des modèles miniature à portée de toutes les bourses de la planète.

Il en va de même pour l'ordinateur qui avance à grand pas de *puce*, avec la gamme *System/360* de la compagnie *IBM* ; cette société devient alors l'une des premières compagnies mondiales, et elle domine l'informatique pendant plus de vingt ans. La grande nouveauté de ses systèmes est de permettre l'exécution des logiciels sur toutes les machines de la gamme. Ceci permet aux clients de conserver et d'accumuler leurs propres applications, au travers des fréquents changements de matériels auxquels ils vont finir par s'habituer.

Pour répondre aux besoins de ses clients - fabricants japonais de calculatrices, *Intel* développe en 1971 le *I4004*, premier microprocesseur 4 bits - *planche* 16. Il intègre sur une même *puce* les trois composantes d'une machine *universelle* (chapitre 5) : unité de calcul UAL, mémoire RAM et contrôle par mémoire ROM. Cette combinaison unique de composants remplace par autant de programmes les opérateurs câblés qui se multiplient alors pour les calculettes : addition, soustraction, multiplication, division, racine, exposant, logarithme, ...

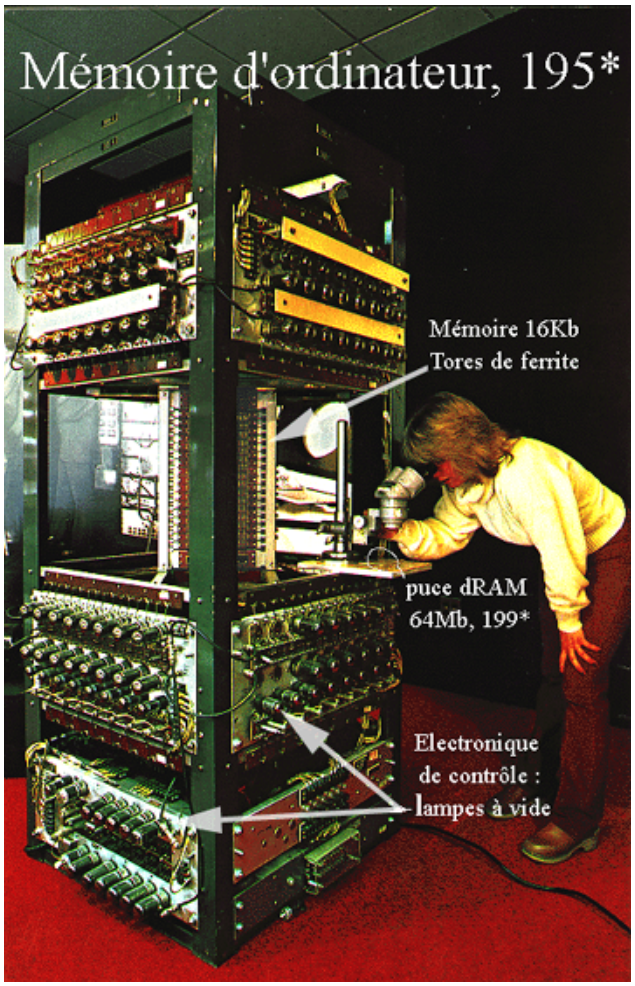
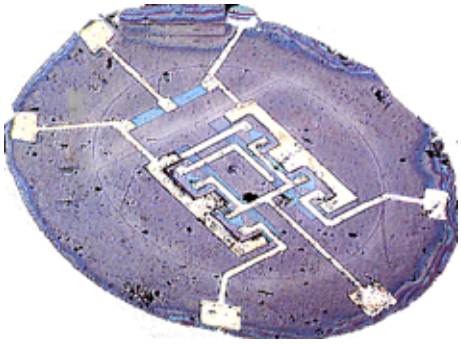


Planche 14 – Mémoire magnétique, années 50

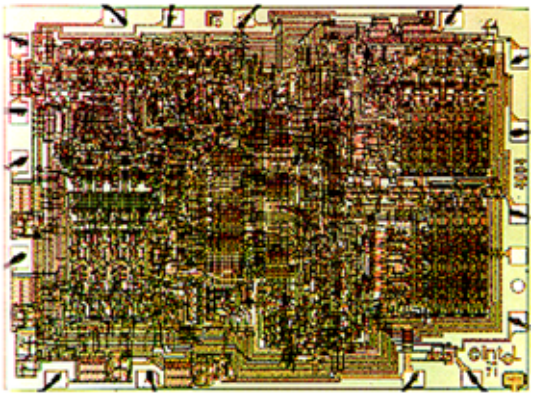
C'est dans la conquête de l'espace, le contrôle des machines à laver, et d'autres applications tout aussi inattendues que cet outil *à tout faire* connaît ses premiers succès industriels. Deux ans après son introduction, le microprocesseur connaît plus de 2000 applications, dont aucune n'avait été prévue par ses concepteurs. Il faut pourtant attendre 1979 - *planche* 18 - pour que le microprocesseur commence à trouver aussi sa place actuelle, au cœur de la structure des ordinateurs.

La multiplication des applications augmente d'autant les volumes de production des circuits intégrés ; ceci permet de diminuer les coûts unitaires, et de financer les investissements nécessaires pour réduire encore la taille des composants. Les avantages techniques et économiques de cette *miniaturisation* sont alors clairs. En 1973, G. Moore - président et fondateur d'*Intel* - prédit que le nombre de transistors par puce va doubler tous les dix huit mois. Cette prédiction, connue sous le nom de "loi de Moore", s'est révélée correcte depuis près de trente ans. En devenant plus



Ce circuit de FAIRCHILD est une mémoire 1 bit du type *Set/Reset flip-flop*.

Planche 15 – Premier circuit intégré planaire, 1960



Ce processeur 4 bits I4004 de la compagnie *Intel* contient 2 300 transistors. Gros comme un ongle de bébé, il coûte $200 à l'époque et livre plus de calculs que l'ENIAC.

Planche 16 – Le premier microprocesseur, 1971

petit, le circuit intégré gagne en vitesse, en fiabilité, et il devient moins cher. Ceci permet de baisser continuellement les coûts des applications existantes. Chaque gain de puissance ouvre la porte de nouvelles applications. Chaque baisse dans le coût des calculs élimine des techniques concurrentes, qu'elles soient mécaniques ou humaines.

Un autre développement majeur des années 70 est rendu possible par la combinaison du circuit intégré et des télécommunications numériques : le commutateur téléphonique *automatique*. C'est un progrès technique important : d'objet de frustration perpétuelle (Allo ! le 22 à Asnière ?), le téléphone devient progressivement fiable, et beaucoup plus utile.

Ce progrès a aussi des conséquences sociales majeures : le métier d'*opérateur téléphonique* disparaît. Des centaines de milliers de gens, surtout des femmes, doivent changer d'emploi. C'est la première fois que l'informatique déplace aussi

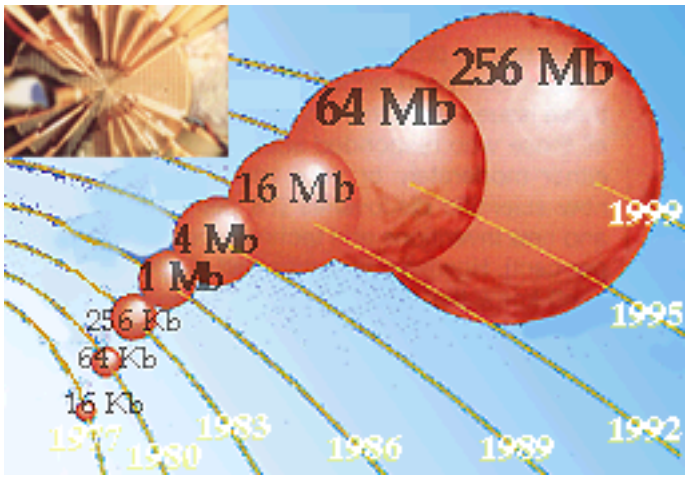


Planche 17 – Evolution des mémoires dRAM

brutalement un métier. Ce n'est pas la dernière. Les emplois peu qualifiés sont remplacés par une solution automatique dès que celle-ci devient économiquement compétitive. En échange, on crée des emplois plus qualifiés, dans la conception, la programmation, la fabrication, la vente et la maintenance de ces solutions, à base de calcul automatique.

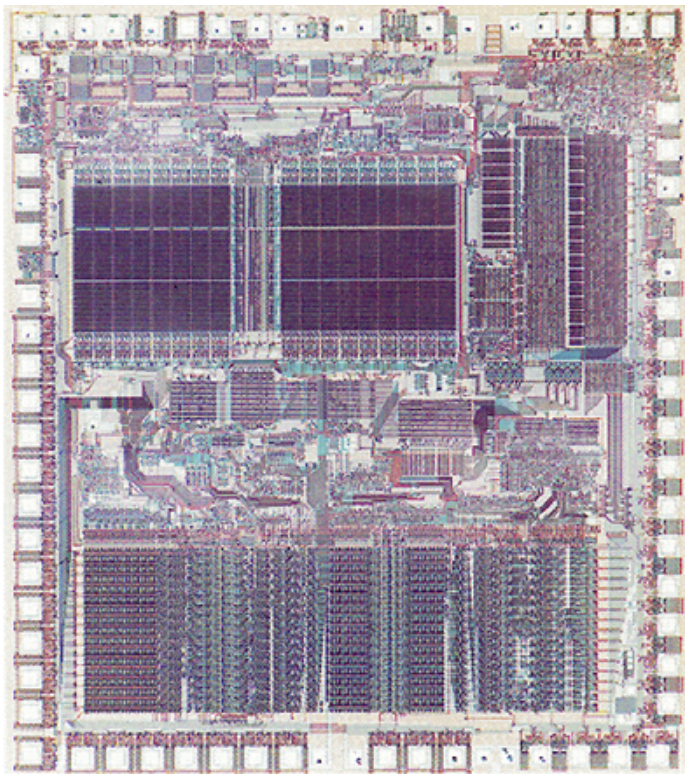
Souvent, ces nouveaux métiers de l'informatique sont tout aussi éphémères : on a vu passer les perforateurs de cartes, les pupitreurs, les opérateurs d'ordinateurs, les techniciens de maintenance, . . .Ces métiers ont disparu presque aussi vite qu'ils sont venus. Leurs titulaires ont dû acquérir de nouvelles qualifications : expert en traitement de texte, programmeur, ingénieur système, maintenance par réseaux, etc. Les programmeurs COBOL ont dû apprendre le langage C++, et maintenant JAVA.

Dans les années 80, le mini-ordinateur VAX de *Digital* rend le calcul accessible à une multitude de chercheurs, enseignants, ingénieurs, gestionnaires, dans tous les métiers de l'homme. Ceci prépare la multiplication - qui va suivre - des applications de l'informatique à chacune de nos professions : de l'avocat au biologiste et au livreur de pizza.

Les *réseaux d'ordinateurs* arrivent à maturité durant la même décennie. Ils permettent depuis lors de réserver un vol d'avion - en temps réel - partout sur le globe. Ils permettent aussi de contrôler des chaînes d'assemblage automatiques, dans lesquelles des robots travaillent. Les humains, ici encore, sont déplacés par la machine : des emplois de basse qualification disparaissent ; d'autres, plus qualifiés mais moins nombreux, apparaissent.

Dans le même temps, le microprocesseur - *planche* 18 - couplé à la mémoire RAM - *planche* 17 - devient l'outil de base de toute l'informatique *embarquée* : dans les avions, les autos, les satellites, les missiles, les bateaux, les robots, les satellites, les téléphones, . . .

La part de l'informatique dans le coût d'une voiture monte régulièrement :



Le MC68000 de *MOTOROLA*.

Planche 18 – Microprocesseur 16 bits, 1979

pour approcher aujourd'hui le quart dans le haut de gamme ! Fin 70, on prédisait que l'automobile fournirait le plus important marché de l'électronique : il se vendait alors beaucoup plus de voitures que de microprocesseurs. Il est vrai maintenant que toute voiture neuve comporte plusieurs processeurs - plus de 50 pour certains modèles récents. C'est aujourd'hui un marché substantiel. Il est pourtant bien moins important que celui des millions de processeurs que l'on vend chaque jour sur les cartes à puces, ou chaque mois dans les PC. Le marché de l'automobile est maintenant très petit, face au gigantesque marché du téléphone portable. Songez de plus que la Chine est devenue en 2003 le premier marché mondial : plus de 200 millions de ces objets. Ces circuits sont au cœur de la communication sur la planète toute entière.

Le pilote d'un avion moderne - *fly by wire* - conduit avant tout un ordinateur complexe, qui est maintenant seul aux commandes *directes* des ailerons, des moteurs, de la radio, météo et navigation. Le pilote prend seul chaque décision, mais la réalisation est automatique et (cela reste à voir) sans faute.

L'argent électronique stocké dans les ordinateurs du monde a une somme bien supérieure à tout ce qu'on garde dans tous les porte-monnaie et coffres-forts de la planète, pour un volume bien moindre. On estime que la valeur des échanges jour-

$30t$	$4000\ lampes$	$30mn$ entre pannes
$2 \times 10^5 Watt$	$10km$ de fils	
$140m^2$	$> 1M$ US$	40 multiplications$/s$
Ordinateur 1950 (ENIAC).		
$200gr$	4 circuits	6 mois entre pannes
$3 \times 10^{-6} W$	$10cm$ de fils	
$20cm^3$	100 FF	10^3 multiplications$/s$
Calculette 1980.		

Planche 19 – Trente ans de progrès technique

naliers d'argent électronique est au moins dix fois celle des réserves des banques centrales de tous les états de la planète.

Prises dans leur ensemble, les technologies de l'information sont devenues en cinquante ans la toute première industrie de l'homme. Nous sommes passés du millénaire de l'industrie à celui du numérique.

> The overall electronics industry represents the United States' largest manufacturing business (1989 revenue : $ 300 billion), bigger than steel, aerospace and automobile combined.
>
> *Time magazine, Dec. 4, 1989.*

1.5 Aujourd'hui

Le *personal computer* PC est introduit par *IBM* en 1981 : 16 KB de mémoire, et un processeur 8 bits à 300 KHz. Il coute 2500 US$; pour 2000$ de plus, on a 256KB de mémoire et un moniteur alphanumérique en couleur. Tout ceci est bien peu de chose en regard de la puissance d'alors des *mainframe* ordinateurs d'IBM et des *minis* de Digital. Moins de dix ans plus tard, le rapport des forces est inverse : il se vend maintenant plus de PC que tous les autres types d'ordinateurs d'usage général combinés ; les écarts de performance diminuent chaque jour. Pour suivre cette montée en puissance du PC, les fabriquants d'autres types d'ordinateurs doivent développer leurs propres microprocesseurs et néanmoins réduire dramatiquement leurs coûts pour survivre. Comme ils n'ont pas les économies d'échelles propres au PC, beaucoup disparaissent ; d'autres tentent de changer de métier.

Par une ironie de l'économie, l'introduction du PC et son succès auprès du grand public dans les années 80 marque la fin du monopole insolent qu'IBM a gardé pendant près de quarante ans sur l'informatique. Il est remplacé dans les années 90 par un *duopoly*, entre *Microsoft* et *Intel* - dit *Wintel*. Combien de temps

durera t-il ? Une lectrice aura-t-elle la vision du futur pour faire à *WinTel* ce que Bill Gates fit à IBM en 10 ans ? Les lecteurs peuvent essayer aussi.

L'imprimerie, par laquelle Gutenberg a changé le monde, est désormais digitale : presque tout document reproduit aujourd'hui passe par une représentation numérique. Il y a cinquante ans, la page que vous lisez aurait demandé plusieurs tonnes d'équipements (cadres tenant le négatif de chaque caractère pour chaque page), et des centaines d'heures de travail avant de pouvoir être imprimée sur papier. Elle a été composée sur un PC [1] portable (de moins de cinq kilogrammes), en moins d'une heure. Elle est imprimée sur une imprimante de bureau en couleur qui revient - dès 1997 - moins cher à l'année d'usage que le téléphone - achat, papier et encre comprises. Par les réseaux numériques - au travers d'un MODEM [2] - on peut aussi imprimer toutes ces pages à distance, de chez soi comme des antipodes. En 2009, le texte de ce cours est envoyé par *mél* chez l'imprimeur au format PDF ; le paiement de cette transaction est aussi numérique.

En devenant digital, le téléphone revient aux principes du télégraphe, son ancêtre. Et le son prend d'autres dimensions : il devient aussi image, film et texte. On le transmet d'ordinateur en ordinateur, à une vitesse souvent proche de celle de la lumière. La communication, à toute distance et sous presque toute forme, est maintenant numérique. Les barrières historiques entre télécommunication et informatique, entre *Electrical Engineering* et *Computer Science* sont caduques. Les idées nouvelles et intéressantes sont presque toutes à cheval entre ces domaines, souvent jaloux l'un de l'autre. Pourtant, il n'est plus de réseau sans ordinateur, plus d'ordinateur sans réseau, et le logiciel est la colle numérique qui fait tenir le tout ensemble.

Internet L'*hypertexte* est inventé par Tim Berners-Lee en 1980. Grace à lui, le CERN [3] est en 89 le plus gros noeud Internet en Europe. L'hypertexte permet en effet à plus de 100 000 physiciens du monde entier de contribuer aux projets de recherches du Centre. Tim Berners-Lee programme en 1990 le premier "lecteur hypertexte". Netscape impose le concept du logiciel gratuit et domine les lecteurs Internet jusqu'en 99. Après un échec commercial, il reste le lecteur Internet le plus populaire en 2010 sous le nom de Mozilla.

Louis Monier [4] écrit, sur un PC de son garage, le premier "robot Internet" en 92. Ce programme visite systématiquement le graphe Internet : les noeuds sont des pages ; les pages contiennent leurs données propres (texte, images, ...), et des liens hypertexte qui renvoient vers d'autres pages. Le robot "rampe 24h/24 sur l'Internet " ; au fil de son voyage, il tient à jour un index unique, de son garage vers les quelques millions de liens Internet d'alors. Louis Monier et Michael Burrows programment ensuite AltaVista, le premier moteur de recherches Internet : l'index permet de lister les pages contenant les mots cherchés. Le succès est spectaculaire :

1. Personal Computer
2. Modulator & Demodulator
3. Centre Européen de Recherche Nucléaire.
4. La première thèse que j'ai dirigée est celle de Louis Monier, à Orsay en 1980.

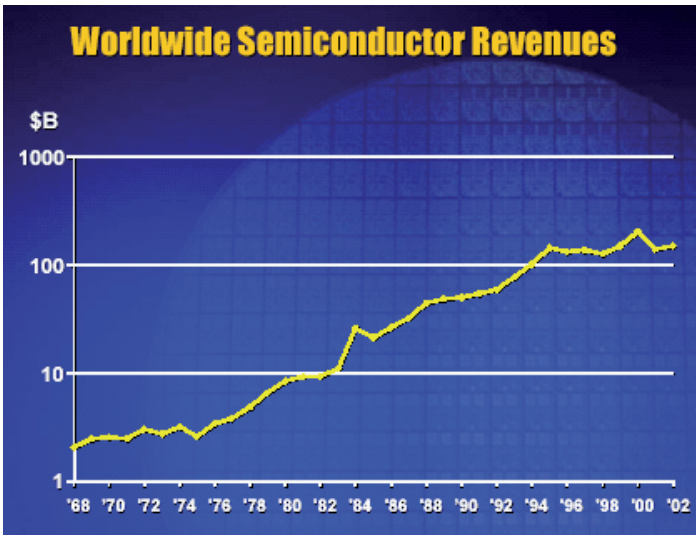


Planche 20 – Chiffres d'affaires de l'industrie micro-électronique jusqu'en 02 ; en 09, c'est 212 B$.

le nombre d'utilisateurs d'AltaVista se multiplie par 10 tous les 2 mois pendant plus d'un an ; sans publicité, par des méls entre individus, dans la "Silicon Valley" d'abord, vite dans le monde entier. Mais Digital ne saura jamais tirer profit de ce succès technique.

Deux ans plus tard, l'Internet a 10 fois plus de pages statiques, et une infinité de pages dynamiques : chaque instance en est différente, car générée par code. Une recherche retourne typiquement des milliers de pages, et il est fort long d'en extraire celles qui nous intéressent. Pendant leur Ph. D. à l'université de Stanford, Larry Page et Sergey Brin inventent une méthode qui associe un rang unique à chaque index de page et les ordonne par fréquence d'accès décroissante. Ils soumettent un papier sur le " page rank " qui est rejeté : " utilisation intéressante de techniques matricielles bien connues " ! Plutôt que de finir leur Ph.D., Page et Brin fondent Google en 98. Ils font un tabac en bourse et dans les ménages, avec des millions d'utilisateurs assidus. Ils survivent au " dot.com crash " de 2001 : une des pires dépressions dans l'histoire de l'informatique - cf. *planche* 20.

Au début, le " page rank " de Google est public (les données à partir desquelles on le calcule restent secrètes) : on minimise le temps moyen de chaque recherche. Avec l'arrivée du PDG Eric Schmidt , le " page rank " devient secret. Le " bizness model " optimise le profit Google, en vendant aux enchères aveugles tous les liens Internet indexés, seconde par seconde. Et Google devient un animal à part dans notre zoo numérique. La compagnie possède de loin le plus puissant centre de calcul au monde. C'est un " nuage ", formé de " grappes " distribuées autour de la planète. Chaque grappe comprend des dizaines de milliers de " serveurs " : des PC 64b parmi les plus performants du moment, avec un maximum de

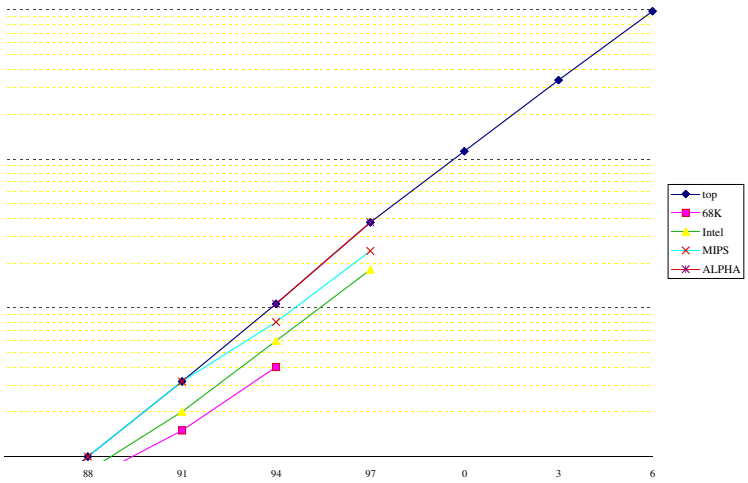


Planche 21 – Evolution de la puissance de calcul des microprocesseurs.

mémoire. Ces PC vont par dizaine sur des cartes, et des dizaines de cartes s'empilent dans des placards. La grappe abrite des centaines de placards dans un local climatisé, proche d'une source d'électricité peu chère pour alimenter les calculs en énergie. Le nuage formé par les grappes de recherches Internet consomme près de 2% de l'électricité mondiale. Les grappes du nuage sont toutes reliées entre elles par les meilleures communications disponibles. Le " cloud computing " de Google répond à des millions de recherches Internet chaque seconde. C'est rapide et gratuite pour chacun. En 2010, Google possède aussi une gigantesque base de données sur les utilisateurs Internet, et contrôle la majorité des tarifs de publicité sur Internet. Directement ou indirectement, Google peut imposer ses conditions à tout " fournisseur de contenu " au monde (sauf encore en Chine). Changer le " page rank " d'une société peut lui signifier sa fortune, ou sa mort. Au début de ce brave nouveau millénaire numérique, beaucoup s'inquiètent de cette nouvelle forme de monopole, et des utilisations malveillantes de données sur les " habitudes Internet " de chacun d'entre nous.

1.6 Demain

L'âge de l'Information souffle vigoureusement ses cinquante bougies ; il va croître et embellir pendant de nombreuses années encore. Tout devient numérique. Tout devient connecté. Les applications se multiplient. Les technologies convergent. Ce que la *forme du calcul* gagne - taille, vitesse, coût - la *fonction du calcul* le récupère en généralité et (quelques fois) en simplicité d'usage.

Combien de temps la spirale informatique va-t-elle encore tourner à cette vitesse ?

La "loi de Moore" dit que la finesse de gravure sur silicium se divise par deux tous les trois ans ; elle deviendra forcement fausse, car c'est le lot de tout

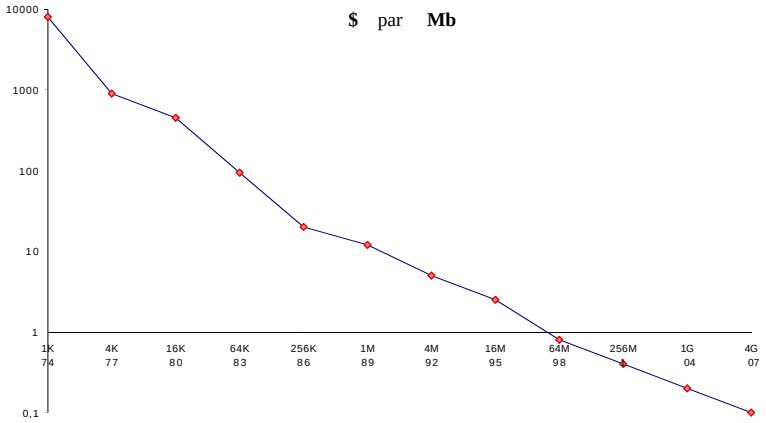


Planche 22 – Coût en $ du **Mb** de mémoire dRAM.

phénomène naturel, dont la croissance initiale semble exponentielle. Pourtant, en 30 ans, la *densité du calcul* sur silicium s'est multipliée par plus de 1000. Parions ici que la "loi de Moore" va s'appliquer pendant au moins trois générations technologiques, soit plus de 10 ans. C'est en tous cas ce que prédisent les tables de cette industrie, dont l'objectif est 0.1 μ en 2006, et 0.05 μ en 2012 !

En 1997, les motifs les plus fins des circuits commerciaux mesurent 0.3 μ. Comme cette taille est déjà inférieure à la longueur d'onde de la lumière blanche, il faut remplacer celle-ci, dans les phases de gravure optique, par quelque chose de plus précis : la lithographie UV, à rayons X, ou par faisceaux d'électrons sont des candidats possibles. L'épaisseur du diélectrique qui isole, dans un transistor, la grille du canal, correspond à 20 couches atomiques pour 0.3 μ, et 3 atomes pour 0.03 μ. Quand le nombre d'électrons utilisés pour stocker le 1 logique tombe de quelques milliers à quelques centaines, les lois de la mécanique statistique cèdent le pas à celles de la mécanique quantique. Les formules que nous donnons au chapitre 3 ne s'appliquent plus à cette échelle. Par exemple, il n'y a plus de gain sur la vitesse d'horloge, en retour de la miniaturisation. Ces phénomènes nous attendent aux alentours de 0.05 μ. Dans 15/20 ans ? On trouvera des références sur les limites physiques intrinsèques aux calculs MOS en [5, 3].

Pour quelques années encore, ce ne sont donc pas les difficultés de la fabrication qui vont limiter les possibilités de la miniaturisation des circuits. Par exemple, la compagnie NEC réalise en 1997 la première mémoire dRAM de 4 G bits sur une seule puce ; sa commercialisation n'est envisagée que vers l'an 2005 ! Ce sont les lois de l'économie, avant celles de la technologie, qui actionnent et régulent la "loi de Moore". Une usine de silicium *state of the art* coûtait déjà plus d'un milliard de $ en 1995 ; le prix double maintenant avec chaque génération de technologie. Pendant combien de temps la croissance du marché justifiera de la rentabilité d'investissement aussi colossaux, avant que le phénomène ne ralentisse ?

La radio et la télévision convergent vers le *tout numérique* avant le millénaire ; le cinéma devra suivre vite, ou mourir. La téléconférence et la commande vocale

sont maintenant possibles en temps réel. La télévision numérique haute définition sur grand écran peut consommer, quant à elle seule, tous les cycles de calcul qu'on saura lui livrer.

On ne parle jusqu'ici que des technologies et d'applications bien connues en 2010. Il y a gros à parier que nos *machines à tout calculer* trouveront des applications nouvelles, aussi importantes économiquement que celles dénichées à ce jour, et que personne n'avait prévu. De plus, les calculs optiques, quantiques, biologiques ou autres pourraient devenir un jour plus rentables que les calculs sur silicium, et contribuer à amplifier le progrès global.

Le couplage d'un réseau de communication mondial avec une puissance de calcul à coût infinitésimal ouvre des possibilités fantastiques. Le développement de la vente directe de biens et de produits sur *Internet* remet en cause la structure du commerce, de la distribution, et de la fabrication ! Dans les braves nouveaux mondes digitaux que l'on imagine alors, l'informatique va encore déplacer de multiples métiers dans les services ; ce sont en partie ceux qui avaient remplacé les métiers disparus lors de l'informatisation des machines des industries de production. Des métiers de basse qualification vont encore disparaître. De nouveaux métiers - de plus haute qualification - vont aussi résulter de cette nouvelle mutation technologique.

A la fin de la journée, notre seule certitude est qu'un scientifique ayant la *compétence informatique* n'aura que l'embarras du choix de son métier, pour longtemps. Pour réussir, il faut allier trois types de connaissances :

- les principes fondamentaux du sujet ; ceux qui restent invariants du temps et des technologies de calcul. C'est le propos de cet ouvrage.
- la pratique expérimentale des meilleurs logiciels sur les meilleures machines du moment. Il faut adapter en permanence cette pratique à l'apparition prévisible des nouvelles technologies.
- l'aptitude à concilier principes théoriques et réalisations pratiques, pour savoir au mieux appliquer l'informatique, au bénéfice de chaque science et de chaque technique.

I

Principes

Chapitre 1

Circuit mathématique

Contents

Dans un circuit électronique en fonctionnement, les paramètres physiques qui régissent son comportement dynamique - potentiels, courants, champs - varient continûment avec le temps réel $t \in \mathbf{R}$. Qui plus est, ils varient avec la position géométrique du point d'observation sur le circuit actif, à cause des *pertes en ligne*. Deux copies physiques du même circuit, opérant sur les mêmes entrées, n'ont pas forcement les mêmes sorties ! En variant d'un circuit à l'autre, même faiblement, les paramètres analogiques [1] peuvent induire d'importantes variations sur le comportement en sortie, après passage dans des amplificateurs non linéaires. On comprend que la conception, l'analyse et la mise au point des circuits *analogiques* soit un art *difficile* ; il faut prendre en compte et limiter tous ces *bruits*, alors qu'ils empêchent la reproductibilité exacte, d'une expérimentation sur l'autre.

Le concept de *circuit digital synchrone* - CDS présenté ici - est une idéalisation mathématique d'une certaine classe de circuits électroniques. Dans cette abstraction, le circuit CDS est défini par un système fini d'équations entre des variables mathématiques - qui symbolisent ici les équipotentielles électriques. Par construction, la solution au système d'équations définissant le CDS est *unique*. Pour chaque variable v, on trouve une suite

$$[\![v]\!] \;=\; v(0)v(1)\cdots v(t)\cdots$$

de bits $v(t) \in \mathbf{B} = \{0, 1\}$, qui se calcule de proche en proche, à partir des valeurs des entrées du circuit aux cycles successifs d'horloge : pour $t = 0$, $t = 1 \cdots$ et pour tout $t \in \mathbf{N}$. C'est précisément le calcul qui est effectué, des milliards de fois par seconde, par le circuit électronique que l'on réalise automatiquement au chapitre 3, à partir des équations du CDS.

Du temps *continu* des circuits analogiques - mesuré par un nombre réel $t \in \mathbf{R}$ - nous passons ici au temps *discret* des CDS - mesuré par un nombre entier $t \in \mathbf{N}$. Ce temps discret est *synchrone* : c'est le même pour toutes les variables, et tous les composants des circuits CDS. Le retour au temps continu sera fait au chapitre 3, lors de la réalisation électronique des CDS. Tout se passe alors comme si :

1. à chaque instant $t \in \mathbf{R}$, la valeur $v(t) \in \mathbf{B}$ d'une variable v est *digitale* - soit $v(t) = 0$, soit $v(t) = 1$;
2. une variable *synchrone* ne peut changer de valeur - passer de 0 à 1, ou l'inverse - qu'aux instants discrets correspondants aux valeurs entières $t \in \mathbf{N}$ du temps réel $t \in \mathbf{R}$.

En d'autres termes, la valeur $v(t) = v(n)$ d'une variable *synchrone* est constante pendant chaque période d'horloge : $n \leq t < n + 1$. L'entier $n = \lfloor t \rfloor \in \mathbf{N}$ est la partie *plancher* du réel $t \in \mathbf{R}$. La donnée des valeurs $v(t) \in \mathbf{B}$ d'une variable digitale synchrone aux instants entiers $t \in \mathbf{N}$ est alors équivalente, à celle de ses valeurs aux instants réels $t \in \mathbf{R} \geq 0$.

En adoptant la méthodologie de conception *digitale et synchrone*, on obtient des circuits électroniques dont le *comportement logique*, est parfaitement reproductible d'une réalisation sur l'autre. Même lorsque deux incarnations physiques

1. géométrie exacte des fils et des transistors, température locale, réflexions d'ondes, ...

d'un unique circuit CDS sont réalisées dans deux technologies différentes, ils effectuent exactement le même calcul mathématique quand ils opèrent sur les mêmes entrées. Les différences éventuelles de comportement ne concernent que la taille, la vitesse, la consommation et le prix des deux réalisations.

Le circuit digital permet des calculs *arbitrairement longs*, tout en restant indéfiniment reproductibles. Le bruit intrinsèque des circuits analogiques limite la durée du calcul, jusqu'à obtenir des résultats aléatoires.

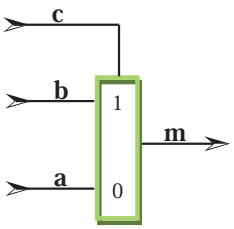
En s'y prenant bien, les circuits digitaux synchrones peuvent aussi être *corrects par construction*. Ce n'est pas rien, quand on doit assembler plusieurs millions de portes sans faute !

1.1 Composants de base

Tout circuit CDS est un assemblage de *composants de base*. La base choisie ici comprend deux éléments : le *multiplexeur* **mux** et le *registre synchrone* **reg** . D'autres choix possibles, pour cette *base*, sont présentés plus loin. Les composants de base sont aux CDS ce que les atomes sont à la chimie : on les combine pour former les circuits complexes, et ils sont insécables ; du moins jusqu'au chapitre 3, où leur structure électronique est révélée, en termes d'un composant encore plus élémentaire : le transistor.

1.1.1 Multiplexeur : mux

Icône



Equation $m = \mathbf{mux}(c, b, a)$

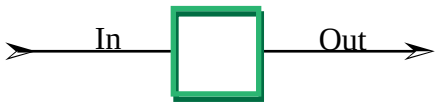
Solution
$$m(t) = \begin{cases} b(t) & \text{si } c(t) = 1, \\ a(t) & \text{si } c(t) = 0. \end{cases}$$

Le *multiplexeur* dispose de trois variables d'entrée, nommées ici c, b, a et d'une sortie nommée m. La valeur $m(t) \in \mathbf{B}$ de la sortie est égale à celle de l'entrée b, soit $m(t) = b(t)$ quand $c(t) = 1$; elle est égale à celle de l'entrée a, soit $m(t) = a(t)$ quand $c(t) = 0$. La valeur du contrôle $c(t)$ détermine sur quelle entrée - b ou a - est connectée la sortie m, à chaque instant $t \in \mathbf{N}$.

Le multiplexeur est un circuit dit *combinatoire*, c'est à dire sans mémoire. La réponse $o(t)$ en sortie d'un circuit combinatoire est *instantanée* : elle dépend exclusivement de la valeur $i(t)$ des entrées à l'instant présent t ; elle ne dépend ni valeurs $i(t')$ entrées dans le passé ($t' < t$), ni dans le futur ($t' < t$).

1.1.2 Registre synchrone : reg

Icône



Equation $out = \mathbf{reg}(in)$

Solution $out(t+1) = in(t), out(0) = 0$

Le *registre* synchrone possède une entrée in et une sortie out. Au temps $t = n$, le registre *mémorise* sa valeur d'entrée $in(n)$; il la restitue en sortie au cycle suivant : $out(n+1) = in(n)$. La sortie du registre est nulle $out(0) = 0$ pendant le cycle initial $t = 0$. Remarquons que l'*horloge* est ici implicite : tous les registres de tous les CDS changent de valeur au *même* instant abstrait $t \in \mathbf{N}$. La réalisation électronique d'une horloge synchrone et son lien avec le temps réel $t \in \mathbf{R}$ de la physique sont vus au chapitre 3.

1.1.3 Alimentations : gnd et vdd

Tout circuit CDS comporte deux entrées constantes, nommées ici **gnd** - pour *ground* - et **vdd** - pour *voltage distribution* :

- la *terre* **gnd** vaut $\mathbf{gnd}(t) = 0$ à tout instant $t \in \mathbf{N}$;
- l'*alimentation* **vdd** vaut $\mathbf{vdd}(t) = 1$ à tout instant $t \in \mathbf{N}$.

Comme pour l'horloge, ces deux entrées sont *implicites* dans notre formalisme.
On identifie **gnd** avec la constante 0 : $[\![\mathbf{gnd}]\!] = 0 = {}_2(0)$, soit gnd(t)=0.
L'alimentation **vdd** s'identifie au chapitre 2 avec la constante *moins un* : $[\![\mathbf{vdd}]\!] = -1 = {}_2(1)$, soit vdd(t)=1 pour $t \in \mathbf{N}$.
La constante *un* est un circuit différent, tel que $[\![\mathbf{un}]\!] = 1 = {}_2 1(0)$. Sa sortie vaut 1 au cycle initial $un(0) = 1$, et 0 ensuite : $un(t) = 0$ pour $t \in \mathbf{N} + 1$.

1.1.4 Portes logiques : not and or xor nor

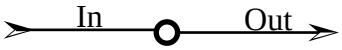
Montrons comment réaliser les portes *logiques*, à partir de la seule primitive combinatoire : **mux**, et des alimentations **gnd** et **vdd** . Ces portes, *combinatoires* par construction, servent toutes par la suite ; nous introduisons leurs *icônes graphiques*, pour les représenter dans les schémas, ainsi que les *symboles* mathématiques pour les représenter dans les formules.

Inverseur : not

Un *inverseur* comprend deux variables - digitales, synchrones - nommées ici in et out. Comme son nom l'indique, in est l'*entrée* de l'inverseur, dont la valeur $in(t)$ à tout temps $t \in \mathbf{N}$ est imposée par le monde extérieur : elle peut être n'importe quelle suite de bits, $in(t) \in \mathbf{B} = \{0, 1\}$ pour $t \in \mathbf{N}$.

L'autre variable, de nom ici out, est l'unique sortie de l'inverseur. Sa valeur $out(t)$ est, à tout temps $t \in \mathbf{N}$, inverse logique de l'entrée : $out(t) = 1 - in(t)$.

En faisant abstraction du temps, on note la même chose par l'équation : $out =$ **not**(in). Cette formulation privilégie la lecture de ces 11 caractères ASCII par un ordinateur. Dans les formules, destinées à l'œil et à l'esprit du lecteur, nous utilisons la notation, équivalente et plus concise : $out = \neg in$.



L'icône représentant un inverseur dans nos schémas est une simple bulle. On la précéde parfois d'un triangle, qui symbolise un *amplificateur*.

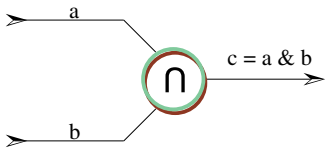
A ce point, nous connaissons la forme - entrée in, sortie out - et la fonction $out = \neg in$ de l'inverseur. C'est tout ce qu'il faut savoir pour s'en servir : l'inverseur est vu comme une *boite noire* dont on ignore la structure interne. Celle-ci peut prendre diverses formes, qui n'affectent en rien la logique des montages réalisés. On fait ainsi *abstraction* de la représentation interne - mathématique ici, physique au chapitre 3 - de l'inverseur.

Seul le concepteur et le réalisateur de circuits sont réellement concernés par la structure interne de l'inverseur - délai, surface, consommation, prix ; contentons-nous de le définir ici, en terme de **mux** et des alimentations, par l'équation de principe :

$$out = \neg in = \textbf{mux}(in, \textbf{gnd}, \textbf{vdd}).$$

Porte ET : and

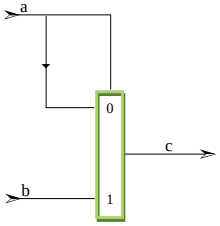
Icône



Equation $c = a\&b$

Solution $c(t) = a(t) \times b(t)$, soit $c = a \cap b$

Schémas



Définition $_2$**Z** Dans le langage $_2$**Z** utilisé ici pour décrire les circuits, on peut définir la porte **and** par l'expression suivante.

```
    and(a,b) = c                    // c = and(a,b) = a & b
 where
    c = mux(a,b,a)
 end where;
```

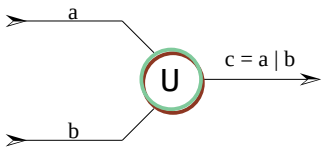
La définition qui suit est équivalente mathématiquement à celle donnée au-dessus. Elle conduit pourtant à une structure physique différente pour la porte **and** réalisée :

```
and(a,b) = mux(a,b,0)
```
// c = **and**(a,b) = $a \,\&\, b$

De toutes façons, la porte **and** est aussi une primitive du langage $_2$**Z** - inutile donc de la redéfinir. Elle dispose d'une syntaxe *infixe*, où **and** est noté - comme dans le langage C - par le symbole & et commercial. On peut ainsi écrire $c = a\&b$ aussi bien que $c =$ **and**(a,b). Pour le lecteur humain - pas informatique - nous écrirons de préférence $c = a \cap b$.

Porte OU : or

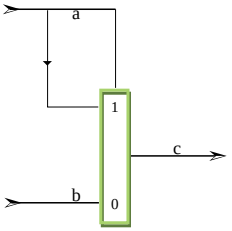
Icône



Equation $c = a|b$

Solution $c(t) = a(t) + b(t) - a(t) \times b(t)$, soit $c = a \cup b$

Schémas



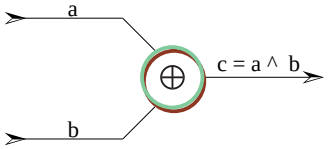
Définition $_2$**Z**

```
or(a,b) = mux(a,a,b)
```
// c = **or**(a,b) = $a \mid b$

La porte **or** dispose aussi d'une syntaxe *infixe* avec le symbole | barre verticale. On peut ainsi écrire $c = a|b$ aussi bien que $c =$ **or**(a,b). Nous écrivons aussi $c = a \cup b$.

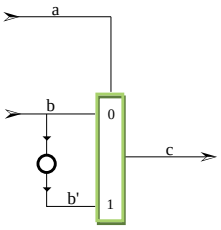
Porte OU exclusif : xor

Icône



Equation $c = a \hat{} b$

Solution $c(t) = a(t) + b(t) - 2a(t) \times b(t)$, soit $c = a \oplus b = a + b \pmod 2$.



Schémas

Définition $_2$Z

```
    xor(a,b) = c                    // définition de c = xor(a,b)
 where
    c = mux(a,b',b)                 // équation de c
    b' = not(b)                     // équation de b'
 end where;
```

On réserve le symbole ^ pour la syntaxe *infixe* de **xor** . On écrit donc $c = a \hat{} b$ aussi bien que $c = \mathbf{xor}(a, b)$; de préférence $c = a \oplus b$.

Dans chaque ligne de code $_2$**Z** , le texte trouvé à droite de // est un *commentaire*, à l'usage du lecteur.

1.2 Forme des circuits digitaux synchrones

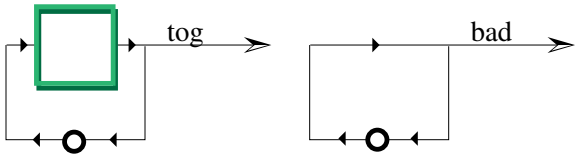


Planche 1.1 – Bonne et mauvaise boucle

1.2.1 Circuit à cycle

Tous les circuits vus à ce point sont dépourvus de cycles. Leurs schémas sont dessinés de façon que les *flèches* aillent de gauche à droite ou de haut en bas. En suivant les flèches, de porte en porte à partir d'une entrée, on finit donc nécessairement sur une sortie externe, sans possibilité de boucle. Afin de comprendre la contrainte que nous allons imposer sur les circuits à boucles - pas de cycle combinatoire - examinons deux équations simples.

```
bad = not(bad);                     // boucle combinatoire !
tog = reg(not(tog)).                // tog = -2/3 = 2(01)
```

Dans les schémas correspondants, on reboucle la sortie sur l'entrée, de l'inverseur pour bad, du registre composé d'un inverseur pour tog - regarder la planche 1.1.

Cycle à mémoire

Pour résoudre l'équation $tog = \mathbf{reg}(\mathbf{not}(tog))$, substituons les variables dans les définitions du registre et de l'inverseur, soit : $tog(0) = 0$ et $tog(t+1) = \neg tog(t)$, pour $t \in \mathbf{N}$. La solution en est : $tog(2t) = 0$ et $tog(2t+1) = 1$, pour tout $t \in \mathbf{N}$. La suite $[\![tog]\!] = 010101\cdots$ des valeurs de tog est *périodique*, de période 2, ce que nous écrivons :

$$[\![tog]\!] = (01).$$

La réalisation de l'équation tog par un circuit électronique - suivant la technique du chapitre 3 - ne pose aucun problème : sa sortie alterne simplement entre 0 et 1, à chaque cycle d'horloge.

Cycle combinatoire

Pour résoudre l'équation $bad = \mathbf{not}(bad)$, substituons les variables $in \Rightarrow bad$ et $out \Rightarrow bad$ dans la définition de l'inverseur, soit : $bad(t) = 1 - bad(t)$, pour $t \in \mathbf{N}$. L'unique solution à cette équation est $bad(t) = 0.5$, qui n'est *pas* digitale : ni 0, ni 1.

Quand un circuit électronique connecte la sortie d'un inverseur à son entrée, on observe des oscillations à haute fréquence entre deux tensions min et max, comprises entre les tensions d'alimentation - voir la simulation de la planche 1.2. Plus grande est la fréquence d'oscillation, plus faible est l'excursion $max - min$ du signal. Il se stabilise à la limite sur un potentiel moyen constant, modulo un *bruit* de très haute fréquence.

Le monde *digital* - mathématique comme physique - *rejette* les circuits comme bad, qui comportent une *boucle combinatoire* : ils sont mal formés. Si toutes les *boucles* du circuit comportent un registre, comme c'est le cas dans tog, le circuit est bien formé.

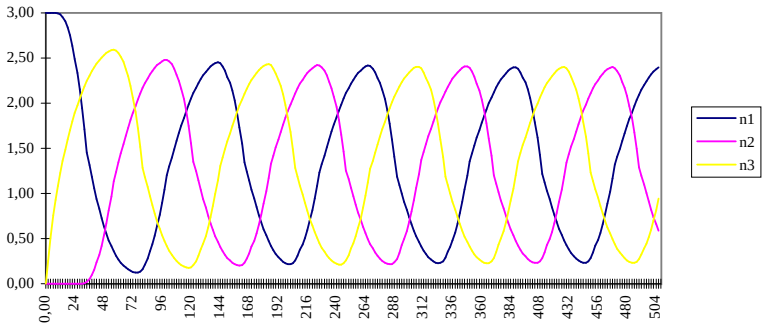


Planche 1.2 – Simulation de trois inverseurs en boucle

1.2.2 Circuit digital synchrone CDS

Donnons ici une définition mathématique de notre objet de base.

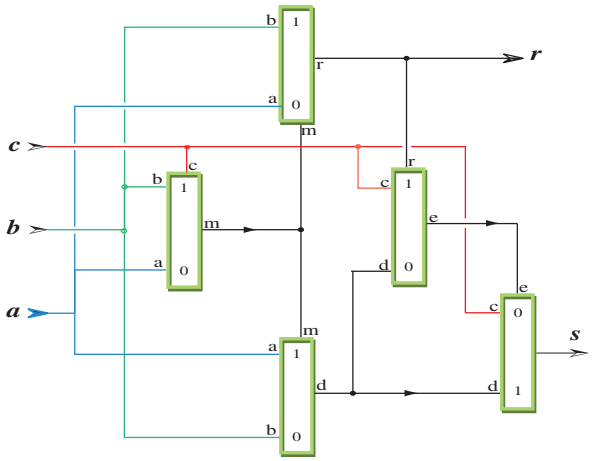
Définition 1 (CDS) *Un* circuit digital synchrone $C \in \mathcal{C}_{ds}$ *est défini par un ensemble* $\mathcal{E} = \mathcal{E}(C)$ *d'*équations *entre des* variables $v \in \mathcal{V} = \mathcal{V}(C)$.

1. *Les variables* $\mathcal{V} = \mathcal{I} \cup \mathcal{G} = \mathcal{I} \oplus \mathcal{G}$ *sont, soit des* entrées $\mathcal{I} = \mathcal{I}(C)$ *(y compris les constantes* **vdd** *et* **gnd** *), soit des* portes $\mathcal{G} = \mathcal{G}(C)$.
2. *Chaque porte* $v \in \mathcal{G}$ *est définie par une* équation unique*, qui peut prendre deux formes, registre* $\mathcal{R} = \mathcal{R}(C)$ *ou multiplexeurs* $\mathcal{M} = \mathcal{M}(C)$ *:*
   (a) $v = \mathbf{reg}(v')$*, pour* $v \in \mathcal{R}$ *et* $v' \in \mathcal{V}$ *;*
   (b) $v = \mathbf{mux}(v_c, v_b, v_a)$*, pour* $v \in \mathcal{M}$ *et* $v_a, v_b, v_c \in \mathcal{V}$.
3. *Les* sorties $\mathcal{O}$ *du circuit sont un sous-ensemble des variables* $\mathcal{O} \subseteq \mathcal{V}$*. Avec les entrées, les sorties* $\mathcal{O} \cap \mathcal{G}$ *sont les seules variables visibles de l'extérieur de* C*. Les autres variables* $\mathcal{O} \oplus \mathcal{G}$ *sont* internes *à* C.
4. *Le circuit* C *est bien formé - c'est à dire que* $C \in \mathcal{C}_{ds}$ *- si le graphe des équations* $\mathcal{E}$ *de définition ne comporte* aucun *chemin combinatoire infini. Ceci implique que tout cycle dans le graphe des équations* $\mathcal{E}$ *de définition comporte au moins un registre (pas de cycle combinatoire).*

Pour compléter cette définition des CDS, il reste à préciser que le *graphe* d'un ensemble d'équations $\mathcal{E}$ possède un *nœud* par variable $v \in \mathcal{V}$; il a une *arête*, allant de $v_e \in \mathcal{V}$ vers $v_s \in \mathcal{V}$, chaque fois que l'équation définissant v_s admet v_e comme entrée : soit $v_s = \mathbf{reg}(v_e)$, soit $v_s = \mathbf{mux}(v_c, v_a, v_b)$ avec $v_e \in \{v_a, v_b, v_c\}$. Un *cycle* (ou boucle) dans ce graphe doit nécessairement traverser au moins un registre. Il est équivalent, dans le cas de circuits finis, de dire que le graphe est dépourvu de cycle combinatoire - ne traversant que des multiplexeurs. La formulation donnée permet de considérer aussi des circuits *infinis*.

Nous décrivons les circuits qui suivent de trois façons : schéma, équation et solution. Ces trois notations sont codifiées au fil des exemples, et elles sont logiquement équivalentes : chacune spécifie le même système d'équations que les deux autres, et donc le même comportement dynamique. Il y a des raisons à cette redondance, qui est inutile du strict point de vue mathématique. La première est un souci de clarté : chaque notation sert à expliquer, et à définir l'autre. La seconde est de préparer la traduction, au chapitre 3, du circuit mathématique en sa réalisation sur un semi-conducteur, qui calcule automatiquement les solutions du système d'équations mathématiques par un dispositif électronique. Les *schémas* contiennent de l'information *logique* - qui spécifie le comportement - et de l'information *géométrique* - placement des portes et dessin du routage des variables entre les portes - qui permettra de dessiner automatiquement les plans de fabrication du circuit physique final.

Pour illustrer la définition 1, considérons deux exemples : un circuit combinatoire - *sans* mémoire - et un circuit séquentiel - *avec* mémoire.



Cette version d'*additionneur binaire complet*, en 5 **mux** , est due à l'ordinateur de J. C. Madre.

Planche 1.3 – Additionneur binaire complet

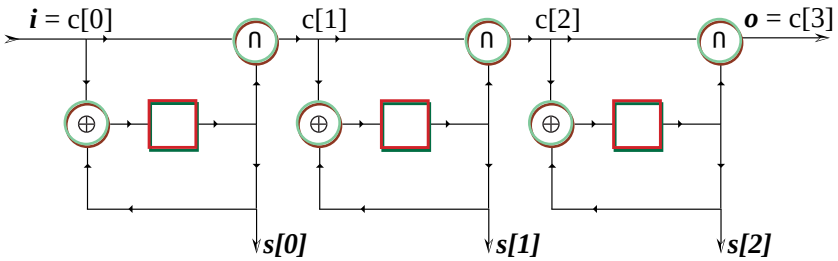
Un additionneur binaire complet

Le circuit **abc** de la planche 1.3 est un *additionneur binaire complet* **abc** . Il dispose de trois entrées a, b, c, et calcule deux sorties combinatoires s, r par le réseau de **mux** sans cycle dont on trouve les équations ci-dessous.

```
abc(a,b,c) = (s,r)
where
 m=mux(c,b,a);                    // profondeur 1
 r=mux(m,b,a);                    // profondeur 2
 d=mux(m,a,b);                    // profondeur 2
 e=mux(r,c,d);                    // profondeur 3
 s=mux(e,d,c);                    // profondeur 4
end where;
```

Compteur 3 bits

Le circuit $Count3a$ est un *compteur* 3 bits que l'on représente par le schéma :



Ce circuit a une variable d'entrée unique, de nom i - pour incrément. Les sorties du compteur comprennent un *vecteur* $s[0..2]$ de 3 variables $s[0]$, $s[1]$, et $s[2]$ qui représente à tout instant $t \in \mathbf{N}$ le *compte* courant $S = S(t)$ en binaire, soit :

$$S = s[0] + 2s[1] + 4s[2].$$

Les sorties $S = S(t)$ au temps t donnent la représentation binaire - sur 3 bits - du nombre de fois (modulo 8) où la variable d'entrée i prend la valeur $i(k) = 1$, entre le cycle initial $k = 0$ et le dernier cycle $k = t - 1$. La sortie complémentaire, de nom o - pour *overflow*, détecte les débordements arithmétiques, modulo 8. La fonction du compteur est de maintenir les invariants suivants :

$$\begin{array}{rcl} S(t) & = & \sum_{0 \le k < t} i(k) \pmod 8, \\ o(t) & = & (S(t) + i(t)) \div 8, \\ S(t+1) & = & (S(t) + i(t)) \cdot|\cdot 8. \end{array} \tag{1.1}$$

Un exemple de circuit $Count3a \in \mathcal{C}_{ds}$ qui respecte ces invariants est donné par les équations de base qui suivent.

```
Count3a(i) = (s[0],s[1],s[2],o)      // Nom(entrées)=(sorties)
where                                // l'entrée i commande l'incrémentation
  s[0]  = reg(x[0]);                 // Tranche 0 : sortie, bit 0
  x[0]  = mux(i,s'[0],s[0]);         // x[0] = i ⊕ s[0]
  s'[0] = mux(s[0],0,-1);            // s'[0] = ¬ s[0]
  c[1]  = mux(i,s[0],0);             // c[1] = i ∩ s[0]
  s[1]  = reg(x[1]);                 // Tranche 1 : sortie, bit 1
  x[1]  = mux(c[1],s'[1],s[1]);      // x[1] = c[1] ⊕ s[1]
  s'[1] = mux(s[1],0,-1);            // s'[1] = ¬ s[1]
  c[2]  = mux(c[1],s[1],0);          // c[2] = c[1] ∩ s[1]
  s[2]  = reg(x[2]);                 // Tranche 2 : sortie, bit 2
  x[2]  = mux(s[2],c'[2],c[2]);      // x[2] = c[2] ⊕ s[2]
  s'[2] = mux(s[2],0,-1);            // s'[2] = ¬ s[2]
  o     = mux(c[2],s[2],0);          // Retenue : o = c[2] ∩ s[2]
end where;
```

Ce programme définit le circuit$Count3a \in \mathcal{C}_{ds}$ dans le langage ${}_2\mathbf{Z}$, que nous présentons ici. La première ligne spécifie l'*interface* du circuit : son nom $Count3a$, son entrée $\mathcal{I} = \{i\}$, et ses sorties $\mathcal{O} = \{s[0], s[1], s[2], o\}$. Les lignes entre les mots where et end where donnent les douze équations de base $\mathcal{E}$ de ce circuit :

– trois registres $\mathcal{R} = \{s[0], s[1], s[2]\}$;
– neuf multiplexeurs $\mathcal{M} = \{s'[0], s'[1], s'[2], x[0], x[1], x[2], c[1], c[2], o\}$.

Soient douze portes $\mathcal{G} = \mathcal{R} \cup \mathcal{M}$ et treize variables $\mathcal{V} = \mathcal{I} \cup \mathcal{G}$, dont quatre sorties $\mathcal{O}$ - restent neuf variables internes $\mathcal{V} \oplus \mathcal{O}$.

1.2.3 Equivalence syntaxique

Une syntaxe plus concise pour décrire les compteurs binaires se trouve en section 4.1, pour tout nombre de bits ; au-delà de ces simplifications à venir de la syntaxe, le circuit $Count3a$ ci-dessus et celui $Count(3)$ de la section 4.1 ont la même *structure*, équation pour équation.

Définition 2 *Deux circuits $\mathcal{C}_{ds}$sont* syntaxiquement équivalents *s'ils ont la même structure, c'est à dire les mêmes équations de base, mêmes entrées et mêmes sorties aux noms des variables et à l'ordre des équations près.*

1.2.4 Tri topologique

La *profondeur combinatoire* d'une variable dans un circuit est une notion *syntaxique*, c'est à dire calculable au vu de la forme du circuit, indépendamment de toute évaluation.

Définition 3 (Profondeur combinatoire) *Soit $C \in \mathcal{C}_{ds}$ un circuit CDS. La* profondeur $prof(v) \in \mathbf{N}$ *d'une variable $v \in \mathcal{V}(C)$ est la longueur du* plus long *chemin combinatoire (ne passant que par des* **mux** *) entre v et, soit une entrée $\mathcal{I}(C)$, soit une sortie de registre $\mathcal{R}(C)$. La* profondeur combinatoire $prof(C) \in \mathbf{N}$ *du circuit C est celle de sa variable la plus profonde : $prof(C) = \mathbf{max}\{prof(v) : v \in \mathcal{V}(C)\}$.*

Par construction, un circuit $C \in \mathcal{C}_{ds}$ ne comporte pas de cycle combinatoire. Tous les chemins combinatoires dans le graphe de C sont donc finis, et $prof(v)$ est la longueur du plus long chemin combinatoire qui arrive à v. La profondeur combinatoire des entrées $v \in \mathcal{I}(C)$ est nulle $prof(v) = 0$, comme celle des sorties de registres $v \in \mathcal{R}(C)$. Seules les sorties de multiplexeurs $v \in \mathcal{M}(C)$ ont une profondeur non nulle. Si $v \in \mathcal{M}$ est donné par l'équation $v = \mathbf{mux}(v_c, v_a, v_b)$, on a par définition :

$$prof(v) = 1 + max\{prof(v_c), prof(v_b), prof(v_a)\}.$$

Cette formule permet de calculer les profondeurs de proche en proche ; en parcourant systématiquement le graphe du circuit, à partir des nœuds de profondeur 0

(entrées et registres), puis de leurs voisins immédiats de profondeur 1, et ainsi de suite jusqu'à épuiser tous les multiplexeurs du circuit, par profondeur croissante. Cet algorithme de parcours d'un graphe est nommé *tri topologique*. Le tri topologique ordonne les équations du circuit par profondeur croissante. Voir ainsi la formule donnée ci-dessus de l'**abc** de la planche 1.3.

Pour $Count3$, on trouve la forme suivante après tri topologique :

```
                                  // présentation de Count3, après tri topologique
Count3t(i) = (s[0],s[1],s[2],o)
where                                          // profondeur combinatoire 0
   s[0]  = reg(x[0]);                                    // sortie, bit 0
   s[1]  = reg(x[1]);                                    // sortie, bit 1
   s[2]  = reg(x[2]);                                    // sortie, bit 2
                                               // profondeur combinatoire 1
   c[1]  = mux(i,s[0],0);                          // c[1] = i ∩ s[0]
   s'[0] = mux(s[0],0,-1);                         // s'[0] = ¬ s[0]
   s'[1] = mux(s[1],0,-1);                         // s'[1] = ¬ s[1]
   s'[2] = mux(s[2],0,-1);                         // s'[2] = ¬ s[2]
                                               // profondeur combinatoire 2
   c[2]  = mux(c[1],s[1],0);                       // c[2] = c[1] ∩ s[1]
   x[0]  = mux(i,s'[0],s[0]);                      // x[0] = i ⊕ s[0]
   x[1]  = mux(c[1],s'[1],s[1]);                   // x[1] = c[1] ⊕ s[1]
                                               // profondeur combinatoire 3
   x[2]  = mux(c[2],s'[2],s[2]);                   // x[2] = c[2] ⊕ s[2]
   o     = mux(c[2],s[2],0);                       // o = c[2] ∩ s[2]
end where;
```

Les circuits $Count3a$ et $Count3t$ sont équivalents syntaxiquement : on ne change pas la fonction calculée par un circuit en changeant l'ordre de ses équations.

1.3 Fonction des circuits

Nous disposons maintenant d'une *syntaxe*, qui permet de décrire un circuit $C \in \mathcal{C}_{ds}$ par un système d'équations de base - sans boucle combinatoire - entre des variables. Cette forme des circuits est assez précise pour permettre de *compiler automatiquement* les masques d'un circuit intégré ; elle se prête également aux raisonnements et constructions mathématiques qui suivent.

Définissons pour cela la *sémantique* des circuits, qui associe à chaque $C \in \mathcal{C}_{ds}$ une fonction spécifique, des entrées $\mathcal{I}(C)$ vers les sorties $\mathcal{O}(C)$. Le calcul de cette fonction peut être réalisé par un humain méticuleux, un ordinateur séquentiel, comme par un circuit intégré parallèle : tous le font aussi bien, et ils trouvent le même résultat ; tous ne vont pas aussi vite que les autres.

Théorème 1 (CDS) *Soit $C \in \mathcal{C}_{ds}$ un circuit CDS décrit par un système $\mathcal{E}(C)$ d'équations de base entre les variables $\mathcal{V} = \mathcal{V}(C)$. Fixons, pour chaque variable d'entrée $v \in \mathcal{I}(C)$, une valeur binaire $v(t) \in \mathbf{B}$ à chaque instant $t \in \mathbf{N}$. Le système $\mathcal{E}$ admet alors une* solution unique *; la valeur binaire $v(n) \in \mathbf{B}$ obtenue au temps $t = n$ pour chaque variable $v \in \mathcal{V}$ dépend exclusivement des valeurs des entrées, prises du cycle initial $t = 0$ au cycle courant $t = n$.*

Preuve : Prouvons ce résultat de trois façons équivalentes ; chacune correspond à un mécanisme *différent* d'évaluation des équations. Pourtant, tous définissent la même sémantique : la valeur calculée $v(t) \in \mathbf{B}$ de chaque variable $v \in \mathcal{V}$ à chaque cycle $t \in \mathbf{N}$ est identique, pour les trois méthodes.

1.3.1 Evaluation parallèle

Ce mécanisme idéalise la propagation des élecrons dans le silicium, et toutes les portes sont évaluées *en parallèle, à chaque instant*. Pour les besoins de cette simulation, le temps discret $t \in \mathbf{N}$ est divisé, entre chaque phase d'horloge de $t = n$ à $t = n + 1$, en un temps fractionnaire $t = n + \epsilon$, $t = n + 2\epsilon$, ... ; ϵ est ici un nombre suffisamment petit. On dispose le calcul dans un tableau dont le nombre de colonnes $v = |\mathcal{V}|$ est égal au nombre de variables dans le circuit à simuler, prises dans un *ordre arbitraire*. Les lignes du tableau correspondent aux étapes de temps successives, fractionnaires et entières. La simulation procède comme suit.

1. Dans la première ligne, correspondant à $t = 0$, on place les valeurs connues $v(0) \in \mathbf{B}$ des entrées $v \in \mathcal{I}$; on range la valeur initiale $v(0) = 0$ en place des registres $v \in \mathcal{R}$; on remplit ensuite toutes les autres cases - celles des multiplexeurs $v \in \mathcal{M}$ - par le symbole *, qui signifie ici une valeur arbitraire, soit 0, soit 1.

2. Pour passer de la ligne correspondant au temps t, à la ligne suivante, correspondant au temps $t + \epsilon$, on reporte les valeurs $v(t + \epsilon) = v(t)$ des entrées $v \in \mathcal{I}$ et les registres $v \in \mathcal{R}$ de la ligne précédante. Pour les multiplexeurs $v \in \mathcal{M}$, on se reporte à l'équation définissant chaque variable, soit $v = \mathbf{mux}(v_c, v_b, v_a)$. On pose donc $v(t + \epsilon) = v_b(t)$ quand $v_c(t) = 1$, $v(t + \epsilon) = v_a(t)$ quand $v_c(t) = 0$, et $v(t + \epsilon) = *$ quand $v_c(t) = *$.
   Avec $Count3a$ pour exemple, ce calcul donne :

t	i	s[0]	x[0]	s'[0]	c[1]	s[1]	x[1]	s'[1]	c[2]	s[2]	x[2]	s'[2]	o
0	1	0	*	*	*	0	*	*	*	0	*	*	*
ϵ	1	0	*	1	0	0	*	1	*	0	*	1	*
2ϵ	1	0	1	1	0	0	0	1	0	0	*	1	*
3ϵ	1	0	1	1	0	0	0	1	0	0	0	1	0
4ϵ	1	0	1	1	0	0	0	1	0	0	0	1	0
5ϵ	1	0	1	1	0	0	0	1	0	0	0	1	0

   Le calcul est *stable* à partir de $t = 3\epsilon$: les lignes suivantes donnent la même valeur aux variables.

En généralisant à partir de cet exemple, on voit que, pour un circuit arbitraire $C \in \mathcal{C}_{ds}$, le calcul se stabilise au plus tard à l'étape $t = n + p\epsilon$ ou $p = prof(C)$ est la profondeur combinatoire de C, et $n \in \mathbf{N}$ est le cycle courant.

3. Une fois la stabilisation combinatoire acquise, au temps fractionnaire $t \geq n + p\epsilon$, on passe au cycle d'horloge suivant $t = n + 1$. Pour cela, on calcule les multiplexeurs comme ci-dessus ; les valeurs $v(n + 1) \in \mathbf{B}$ des entrées $v \in \mathcal{I}$ sont mises à jour ; la valeur de chaque registre $v \in \mathcal{R}$ est mise à jour à partir de son équation définissante : pour $v = \mathbf{reg}(v_e)$, on pose $v(n + 1) = v_e(n + p\epsilon)$.

En reprenant la simulation de $Count3a$ au cycle 3ϵ, il vient :

t	i	s[0]	x[0]	s'[0]	c[1]	s[1]	x[1]	s'[1]	c[2]	s[2]	x[2]	s'[2]	o
3ϵ	1	0	1	1	0	0	0	1	0	0	0	1	0
1	1	1	1	0	1	0	0	1	0	0	0	1	0
1+ϵ	1	1	0	0	1	0	1	1	0	0	0	1	0
1+2ϵ	1	1	0	0	1	0	1	1	0	0	0	1	0
1+3ϵ	1	1	0	0	1	0	1	1	0	0	0	1	0
2	1	0	0	0	1	1	1	1	0	0	0	1	0
2+ϵ	1	0	0	1	0	1	1	0	1	0	0	1	0
2+2ϵ	1	0	1	1	0	1	1	0	0	0	1	1	0
2+3ϵ	1	0	1	1	0	1	1	0	0	0	0	1	0
3	1	1	1	1	0	1	1	0	0	0	0	1	0
3+ϵ	1	1	1	0	1	1	1	0	0	0	0	1	0
3+2ϵ	1	1	0	0	1	1	0	0	1	0	0	1	0
3+3ϵ	1	1	0	0	1	1	0	0	1	0	1	1	0
4	1	0	0	0	1	0	0	0	1	1	1	1	0
4+ϵ	1	0	0	1	0	0	0	1	0	1	1	0	1
4+2ϵ	1	0	1	1	0	0	0	1	0	1	1	0	0
4+3ϵ	1	0	1	1	0	0	0	1	0	1	1	0	0

La simulation continue ainsi pour un nombre de cycles arbitraire. Dans les simulations de compteur, l'entrée i est systématiquement mise à 1 ; en effet, pour $i = 0$, rien ne se passe d'un cycle sur l'autre, et les variables gardent la même valeur, conformément aux invariants du compteur. Introduire des cycles pour lesquels $i = 0$ prendrait donc inutilement du temps et de la place.

Signalons au passage l'apparition de *phénomènes transitoires*, comme par exemple sur la retenue sortante o, qui passe de 0 à 1 au cycle 4+ϵ, pour revenir et se stabiliser à 0 au cycle 4+2ϵ ; on observe le même type de phénomène dans les circuits électroniques. Les transitoires ne sont pas gênants, pourvu que le nombre de pas fractionnaires soit choisi assez grand (supérieur à $prof(C)$) pour les faire disparaître. La condition équivalente pour les circuits électroniques est d'opérer à une fréquence d'horloge suffisamment basse, en dessous de la *fréquence critique* propre à chaque circuit - voir chapitre 3.

La simulation parallèle serait proche de la réalité physique si tous les multiplexeurs avaient le même délai, et si les temps de propagation dans les fils étaient tous négligeables. Ce n'est pas le cas en général, et les calculs intermédiaires aux temps fractionnaires de la simulation parallèle n'ont pas de véritable signification physique. C'est pourquoi on les élimine dans la présentation finale des résultats, pour ne garder que les cycles entiers $t \in \mathbf{N}$; en cachant aussi les variables internes, le résultat de la simulation parallèle du compteur 3 bits donne finalement :

t	i	s[0]	s[1]	s[2]	o
0	1	0	0	0	0
1	1	1	0	0	0
1	1	0	1	0	0
3	1	1	1	0	0
4	1	0	0	1	0
5	1	1	0	1	0
6	1	0	1	1	0
7	1	1	1	1	1
8	1	0	0	0	0
9	1	1	0	0	0

(C3 de 0 à 9)

1.3.2 Evaluation séquentielle

Si elle présente l'avantage pédagogique d'être proche de la réalité physique, la simulation parallèle n'est efficace, ni pour l'humain, ni pour l'ordinateur. Pour simuler n cycles d'un circuit $C \in \mathcal{C}_{ds}$, ayant $v = |\mathcal{V}|$ variables et profondeur $p = prof(C)$, il faut remplir un tableau de $n \times v \times p$ cases ; en particulier, chaque multiplexeur est évalué p fois durant chaque cycle d'horloge. Pour corriger cela, disposons les variables en *ordre topologique* dans la ligne ; ceci veut dire que la colonne attribuée à tout multiplexeur $v \in \mathcal{M}$ défini par $v = \textbf{mux}(v_c, v_b, v_a)$ se trouve nécessairement à *droite* des colonnes choisies pour v_a, v_b et v_c - voir section 1.2.4. Avec cette disposition des variables dans la ligne, la simulation procède sans introduire de temps fractionnaire, et l'évaluation des expressions se fait à la suite, l'une après l'autre, de gauche à droite dans chaque ligne.

1. Dans la première ligne, correspondant à $t = 0$, on met les valeurs connues $v(0) \in \mathbf{B}$ des entrées $v \in \mathcal{I}$, et on range la valeur initiale $v(0) = 0$ en place des registres $v \in \mathcal{R}$.

2. Une fois en place la ligne $t = n$, on calcule les valeurs des multiplexeurs $v \in \mathcal{I}$ de gauche à droite ; pour $v = \textbf{mux}(v_c, v_b, v_a)$, on pose $v(t) = v_b(t)$ quand $v_c(t) = 1$ et $v(t) = v_a(t)$ quand $v_c(t) = 0$. Remarquons que les valeurs $v_a(t), v_b(t), v_c(t)$ sont connues à ce point, puisque situées à gauche de v dans le tableau de simulation.

3. Pour passer de la ligne $t = n$ à la suivante $t = n + 1$, on reporte les valeurs connues des entrées, et on met à jour les valeurs de registres, à partir de la ligne précédante $t = n$ du tableau.

Le nombre d'opérations dans l'évaluation en série est $n \times v$, où $n \in \mathbf{N}$ est le nombre de cycles et v le nombre de variables. C'est p fois moins que pour l'évaluation parallèle, avec $p = prof(C)$.

Avec $Count3t$ pour exemple, le calcul d'évaluation à la suite donne :

t	i	s[0]	s[1]	s[2]	c[1]	s'[0]	s'[1]	s'[2]	c[2]	x[0]	x[1]	x[2]	o
0	1	0	0	0	0	1	1	1	0	1	0	0	0
1	1	1	0	0	1	0	1	1	0	0	1	0	0
2	1	0	1	0	0	1	0	1	0	1	1	0	0
3	1	1	1	0	1	0	0	1	1	0	0	1	0
4	1	0	0	1	0	1	1	0	0	1	0	1	0
5	1	1	0	1	1	0	1	0	0	0	1	1	0
6	1	0	1	1	0	1	0	0	0	1	1	1	0
7	1	1	1	1	1	0	0	0	1	0	0	0	1
8	1	0	0	0	0	1	1	1	0	1	0	0	0
9	1	1	0	0	1	0	1	1	0	0	1	0	0

En enlevant les colonnes qui ne correspondent pas à des sorties du compteur, on vérifie que ce calcul est bien identique à celui du tableau (C3 de 0 à 9).

1.3.3 Evaluation symbolique

Dans cette dernière méthode, l'évaluation d'un circuit $C \in \mathcal{C}_{ds}$ se fait en deux temps.

1. On élimine, par substitution algébrique, toutes les variables de type multiplexeur dans les équations qui définissent chaque registre. C'est possible car le circuit n'a pas de boucle combinatoire. Soit $r = |\mathcal{R}|$ le nombre de registres, $i = |\mathcal{I}|$ le nombre d'entrées et $o = |\mathcal{O}|$ le nombre de sorties dans le circuit C. Après élimination des multiplexeurs, les équations du circuit prennent la forme $\mathbf{r}[k] = \mathbf{reg}(F_k(\mathcal{R}, \mathcal{I}))$, pour $0 \leq k < r$, où F_k est une fonction combinatoire de $r + i$ entrées (au plus). On élimine aussi, par substitution, tous les multiplexeurs qui ne sont pas des variables de sortie. Il ne reste ainsi, après élimination, plus que $o = |\mathcal{O}|$ équations combinatoires, une par sortie ; elles prennent la forme

$$\mathbf{o}[j] = G_j(\mathcal{R}, \mathcal{I}),$$

pour $0 \leq j < o$ et les fonctions combinatoires G_j ont même arguments que les F_k.

2. Définissons l'*état* $S = S(t)$ du circuit C au temps $t \in \mathbf{N}$ par le nombre entier dont on trouve la représentation binaire dans l'état courant des registres $\mathbf{r}[k] \in \mathcal{R}$:

$$S(t) = \sum_{0 \leq k < r} 2^k \mathbf{r}[k](t). \tag{1.2}$$

L'état initial, à $t = 0$, est nul : $S(0) = 0$, par définition du registre. A tout instant $t \in \mathbf{N}$, l'état du circuit est un entier compris entre 0 et $2^r - 1$. La donnée de l'état $S(t) \in \mathbf{Z}_{2^r} = \{0, 1, \cdots, 2^r - 1\}$ au temps $t \in \mathbf{N}$ est équivalente à celle des valeurs $\mathbf{r}[k](t)$ de chaque registre : on retrouve ces valeurs en écrivant le nombre $S(t)$ en binaire, sur r bits - cf. chapitre 2. De la même façon, on représente le vecteur d'entrée au temps t par le nombre entier

$$I(t) = \sum_{0 \leq k < i} 2^k \mathbf{i}[k](t),$$

et le vecteur de sortie par l'entier

$$O(t) = \sum_{0 \leq k < o} 2^j \mathbf{o}[k](t).$$

Soient $S < 2^r$ et $I < 2^i$ deux entiers, dont les représentations binaires respective - sur r et i bits - sont $S = {}_2 s_0 \cdots s_{r-1} = \sum_k s_k 2^k$, et $I = {}_2 i_0 \cdots i_{i-1} = \sum_j i_j 2^j$. Aux r fonctions booléennes $F_0, \cdots, F_{r-1}$, associons la *fonction de transition* du circuit C par la formule $F(S, I) = N$, avec $N = \sum_k n_k 2^k$ et $n_k = F_k(s_0 \cdots s_{r-1}, i_0 \cdots i_{i-1})$. Définissons de même la *fonction de sortie* du circuit C par la formule $G(S, I) = M$, avec $M = \sum_j m_j 2^j$ et $m_j = G_j(s_0 \cdots s_{r-1}, i_0 \cdots i_{i-1})$). La fonction F est une application des paires d'entiers - le premier modulo 2^r, le second modulo 2^i - dans les entiers $\mathbf{Z}_{2^n}$ modulo 2^r. On écrit $F \in \mathbf{B}^r \times \mathbf{B}^i \mapsto \mathbf{B}^r$, en tablant sur l'identification faite au chapitre 2 : $\mathbf{Z}_{2^n} = \mathbf{B}^n$. Le type de G est : $G \in \mathbf{B}^r \times \mathbf{B}^i \mapsto \mathbf{B}^o$.

Forts de ces notations, nous pouvons maintenant décrire le circuit C au moyen des trois *équations de comportement*, valables pour tout $t \in \mathbf{N}$:

$$\begin{array}{rcl} S(0) & = & 0, \\ S(t+1) & = & F(S(t), I(t)), \\ O(t) & = & G(S(t), I(t)). \end{array} \tag{1.3}$$

En éliminant l'état des équations (1.3), nous trouvons la valeur $O(t)$ des sorties au temps t, en fonction des valeurs $I(0), I(1), \cdots, I(t)$ des entrées vues à ce point :

$$\begin{array}{rcl} O(0) & = & G(0, I(0)), \\ O(1) & = & G(F(0, I(0)), I(1)), \\ O(2) & = & G(F(F(0, I(0)), I(1)), I(2)), \\ O(3) & = & G(F(F(F(0, I(0)), I(1)), I(2)), I(3)), \\ & & \cdots \end{array}$$

Reprenons cette technique d'évaluation symbolique, sur l'exemple du compteur 3 bits. Après élimination des multiplexeurs, nous trouvons :

// présentation de $Count3$, après élimination des multiplexeurs

```
Count3e(i) = (s[0],s[1],s[2],o)
```

```
where

   s[0]  = reg(mux(i,not s[0] ,s[0]));

   s[1]  = reg(mux(mux(i,s[0],0),not s[1],s[1]));

   s[2]  = reg(mux(mux(mux(i,s[0],0),

                          s[1],0),not s[2],s[2]));

   o     = mux(mux(mux(i,s[0],0),s[1],0),s[2],0);

end where;
```

En utilisant les identités de l'algèbre de Boole - chapitre 2 - on peut simplifier les expressions **mux** et écrire :

$$\begin{aligned} s[0] &= \mathbf{reg}(i \oplus s[0]), \\ s[1] &= \mathbf{reg}((i \cap s[0]) \oplus s[1]), \\ s[2] &= \mathbf{reg}((i \cap s[0] \cap s[1]) \oplus s[2]), \\ o &= i \cap s[0] \cap s[1] \cap s[2]. \end{aligned}$$

En utilisant maintenant la syntaxe des opérateurs logiques disponibles dans le langage ${}_2\mathbf{Z}$ [2], on peut décrire le même circuit par :

```
                                  // Compteur 3 bits, après élimination des mux
                                                  // et simplifications logiques.
Count3s(i) = (s:[3],o)
where
  s[0]  = reg(i ^ s[0]);                                  // i ⊕ s[0]
  s[1]  = reg((i & s[0]) ^ s[1]);          // (i ∩ s[0]) ⊕ s[1]
  s[2]  = reg((i & s[0] & s[1]) ^ s[2]);
  o     = i & s[0] & s[1] & s[2];
end where;
```

Notons l'utilisation de la déclaration de vecteur $s : [3]$, en place de ses composants explicites $s[0], s[1], s[2]$.

L'*état* du compteur est le nombre, déjà r encontré : $S = s[0] + 2s[1] + 4s[2]$. Quant à la *fonction de transition*, elle semble à *priori* complexe. Nous montrons pourtant, au chapitre 4, qu'elle prend une forme remarquablement simple, dans le cas des compteurs :

2. la syntaxe ${}_2\mathbf{Z}$ pour les opérateurs logiques est reprise de celle du langage C

$$F(S,i) = (i+S) \cdot|\cdot 8.$$

Il en va de même pour la fonction de sortie qui vaut $G(S,i) = S + 8o$, avec $o = (i+S) \div 8$. Les invariants (1.1) du compteur sont donc bien maintenus.

Ceci termine la preuve du théorème 1. **Q.E.D.**

1.3.4 Equivalence sémantique

La définition 2 donne une première notion d'équivalence (dite structurelle) entre circuits. Pour qualifier, deux circuits doivent avoir la même structure, aux noms des variables et à l'ordre des équations près. Ceci implique que les deux circuits électroniques réalisés au chapitre 3 à partir de ces équations sont identiques. C'est par exemple le cas de nos deux premiers compteurs 3 bits, $Count3a$ et $Count3t$. Quant à $Count3e$, il sera *syntaxiquement* équivalent aux deux précédents si on utilise dans la traduction la représentation donnée dans ce chapitre aux portes logiques **and** et **xor** . Il sera structurellement différent si on utilise une autre représentation de ces portes logiques, et il en est de nombreuses. Ceci n'empêche en rien de tels circuits d'avoir le même comportement dynamique, c'est à dire d'être *sémantiquement* équivalents.

Définition 4 *Deux circuits $C, C' \in \mathcal{C}_{ds}$ sont* sémantiquement équivalents *si :*

1. *il existe une bijection β entre leurs entrées : $\beta(\mathcal{I}(C)) = \mathcal{I}' = \mathcal{I}(C')$ - avec $\beta^-(\mathcal{I}') = \mathcal{I}$ - et leurs sorties $\beta(\mathcal{O}) = \mathcal{O}' = \mathcal{O}(C')$ - avec $\beta^-(\mathcal{O}') = \mathcal{O}$;*
2. *ils calculent la même fonction ; pour toute valeur commune des entrées - soit $v(t) = v'(t)$ pour $t \in \mathbf{N}$, $v \in \mathcal{I}$, $v' \in \mathcal{I}'$ et $v' = \beta(v)$ - les sorties sont identiques - soit $v(t) = v'(t)$ pour $t \in \mathbf{N}$, $v \in \mathcal{O}$, $v' \in \mathcal{O}'$ et $v' = \beta(v)$.*

Deux circuits qui sont syntaxiquement équivalents sont bien entendu sémantiquement équivalents. La réciproque est fausse en général, et nous montrons par la suite de nombreux exemples de circuits qui ont le même comportement, sans avoir les mêmes équations. Dans la traduction du chapitre 3, on obtient alors des circuits électroniques différents : leur comportement logique est le même, leur comportement physique (taille et vitesse d'horloge par exemple) ne l'est pas.

1.4 Montre digitale

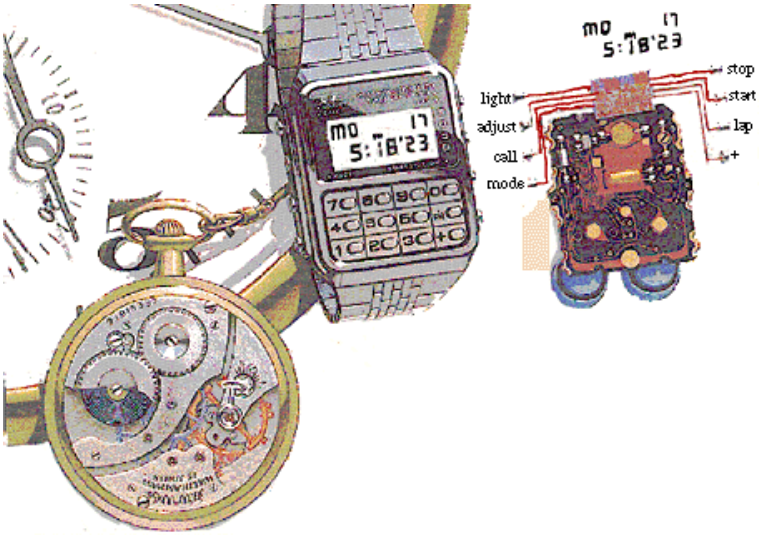


Planche 1.4 – Montres

Les liens entre la mesure du temps et le calcul numérique sont demeurés étroits, de Babylone à la Silicon Valley. Les numérations de Sumer et de Babylone remontent aux origines de l'écriture, sur tablettes d'argile et papyrus, il y a plus de 4000 ans - planche 4. Elles utilisent la base 60, dite *sexagésimale*. C'est à travers l'astronomie grecque d'abord, arabe ensuite, que le système sexagésimal a survécu jusqu'à nos jours, dans les mesures de temps et d'angles. La base $60 = 2^2 \times 3 \times 5$ présente l'avantage de se subdiviser facilement : une heure (respectivement minute) est un multiple entier de 30, 20, 15, 12, 10, 6, 5, 4, 3 et 2 minutes (respectivement secondes).

Les techniques d'engrenages issues de l'horlogerie calquent mécaniquement le système sexagésimal. Au début des années 70, la montre numérique devient la première application destinée au *grand public* des circuits intégrés. Au travers de ces bouleversements techniques, le décompte du temps en base 60 n'a pratiquement pas changé depuis Babylone, et il reste aujourd'hui d'usage universel sur la planète Terre. Songez que cet algorithme est exécuté au moins soixante fois par seconde, au poignet de chacun des centaines de millions de possesseurs de montre digitale. C'est un calcul qui compte beaucoup !

Comme illustration des circuits digitaux synchrones, nous réalisons une montre numérique pour compter le temps en secondes, minutes et heures. La représentation des nombres utilisée dans cet appareil est un mélange de binaire, décimal et sexagésimal. Son origine historique en fixe clairement la *fonction* ; sa *forme numérique* en résulte naturellement.

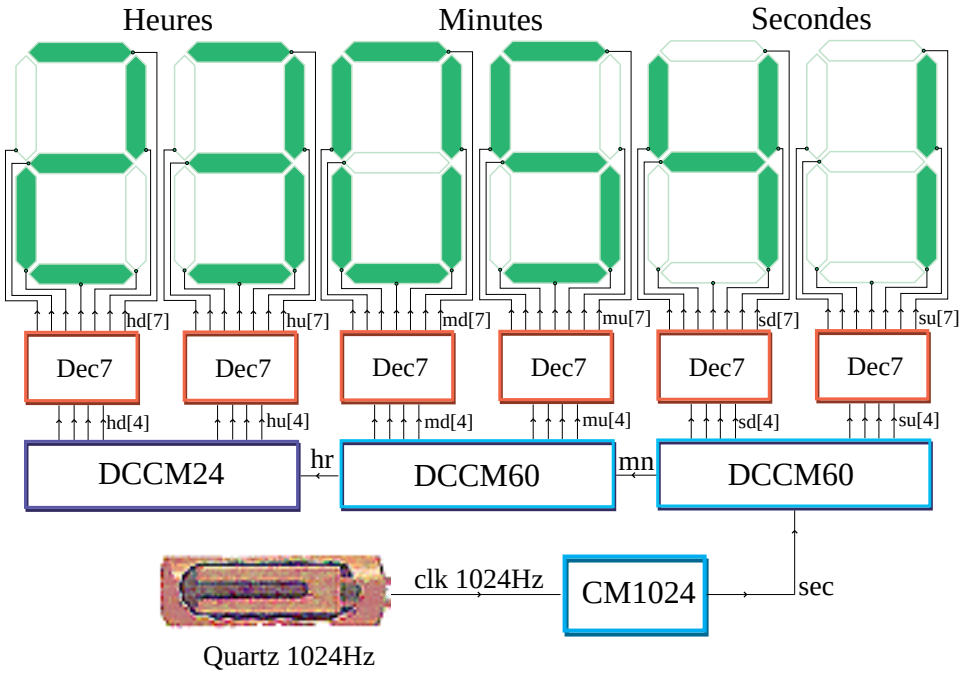


Planche 1.5 – Schéma d'ensemble d'une montre numérique

1.4.1 Structure d'ensemble

Le schéma d'ensemble de la montre numérique se trouve dans la planche 1.5. C'est une chaîne qui va du générateur d'horloge $Q1KHz$, vers 8 dispositifs d'affichage à *cristaux liquides*. Seul le cœur numérique de l'appareil est représentable par un circuit CDS, dont on trouve aussi la description dans le langage ${}_2\mathbf{Z}$ - planche 1.5. Les deux extrémités de la chaîne - générateur d'horloge et affichage sur cristaux liquides - sont des circuits *analogiques*.

- La fonction du générateur $Quartz\ 1KHz$ est de produire l'horloge synchrone clk de l'ensemble du système. Sa fréquence, régulée par un *quartz*, est précisément 1024 fois par seconde. Nous donnons au chapitre 3 des indications sur la réalisation d'un tel *montage analogique*. Si, d'aventure, la fréquence n'est pas *exactement* 1K=1024 Hertz, l'ensemble fonctionne quand même sans problème *logique* : on a juste une montre, qui avance ou retarde.
- Le cœur du système est le circuit $Montre \in \mathcal{C}_{ds}$, défini par les schémas de la planche 1.5 ou, de façon équivalente, par le code ${}_2\mathbf{Z}$ qui suit.

```
  Montre()=(s1:[7],s10:[7],
            m1:[7],m10:[7],
            h1:[7],h10:[7])
where
  sec = CM1K();                          // Divise la fréquence par 1024
  (su,sd,mn) = CM60(sec);                // Compte les secondes
  (mu,md,hr) = CM60(mn);                 // Compte les minutes
     (hu,hd) = CM24(hr);                 // Compte les heures
                  // Décodages de décimal codé binaire vers sept segments :
  s1  = Dec7(su);                        // Unités de secondes
  s10 = Dec7(sd);                        // Dizaines de secondes
  m1  = Dec7(mu);                        // Unités de minutes
  m10 = Dec7(md);                        // Dizaines de minutes
  h1  = Dec7(hu);                        // Unités d'heures
  h10 = Dec7(hd);                        // Dizaines d'heures
end where;
```

L'*interface* du circuit $Montre \in \mathcal{C}_{ds}$ est donné dans la première ligne :
Montre() = (s1 :[7],s10 :[7],m1 :[7],m10 :[7],h1 :[7],h10 :[7])
Cette ligne déclare que le circuit $Montre$ n'a pas d'entrée ; ou plutôt que son unique entrée, l'horloge clk issue de $Quartz\ 1KHz$, est implicite, et sert à synchroniser *tous* les registres qui apparaissent dans le montage. Le circuit $Montre$ dispose de 6 sorties, chacune étant un vecteur composé de 7 bits. Ces 42 bits de sortie servent à contrôler, en final, les 6 afficheurs sept segments.

La structure interne de $Montre$ est faite de quatre compteurs et 6 décodeurs sept segments. Le premier compteur $CM1K$ divise la fréquence d'horloge par 1024, pour produire en sortie la variable sec, qui bat la seconde. Le second $CM60$ compte les secondes, modulo 60 et produit trois sorties : le vecteur $su[0..3]$ code en binaire les unités de secondes, sur 4 bits ; $sd[0..3]$ compte les dizaines de secondes, et la sortie mn bat la minute. Le troisième compteur $CM60$, alimenté par mn, est une copie du précédent, cette fois pour les minutes. Ses sorties $mu[0..3]$ et $md[0..3]$ codent les minutes, et hr bat l'heure. Le compteur $CM24$ donne l'heure modulo 24, en unités $hu[0..3]$ et dizaines $hd[0..3]$. La structure de ces compteurs est détaillée en section 1.4.2.

Chacun des 6 vecteurs de 4 bits su, sd, mu, md, hu, hd passe enfin au travers d'un *décodeur sept segments* $Dec7$, ce qui donne en sortie 6 vecteurs de 7 bits chacun $s1, s10, m1, m10, h1, h10$. Ce sont les sorties numériques finales du circuit $Montre$, qui servent à alimenter les afficheurs à cristaux liquides.

– L'affichage final de l'heure, à usage humain, se fait au moyen de 6 afficheurs à cristaux liquides, suivant le codage des chiffres décimaux sur sept segments défini dans la planche 1.7.

1.4.2 Les compteurs du temps

La montre numérique comporte trois types de compteurs, dont nous donnons ici une représentation sous forme de circuit CDS.

Compteur modulo 1024

Le diviseur de fréquence $CM1K$ est un compteur sur 10 bits, étant donné que $1024 = 2^{10}$. En se reportant à la définition de $Count[n]$ donnée en section 4.1.2, on le définit par :

```
CM1K()=sec
where
                    // Compteur 10 bits en roue libre : i = vdd = −1 = 2(1).
  (sum,sec)=Count(10)(-1);
end where;
```

Ce compteur 10 bits est incrémenté à chaque cycle (soit $i = -1 =$ **vdd**), et on ignore les 10 sorties S de somme, pour ne conserver que la retenue sortante sec.

Compteur modulo 60

Le compteur $CM60$ modulo 60 s'obtient en composant un compteur $CM10$ modulo 10 avec un compteur $CM6$ modulo 10.

```
CM60(i)=(u:[4],d:[4],o)
where
  (u,o10)=CM10(i);                    // Compteur modulo 10
  (d ,o) =CM6(o10);                   // Compteur modulo  6
end where;
```

Le nombre 10 s'écrit en binaire sur 4 bits, et 6 sur 3 bits. En se reportant à la définition de $CountMod[m]$ donnée en section 4.1.4, on définit $CM10$ par :

```
CM10(i) = (c:[4],o)
where
 (c,o) = CountMod(10)(i);             // Compteur modulo 10
end where;
```

Quant à $CM6$, on doit étendre le résultat de $CountMod[6]$ sur 4 bits, soit :

```
CM6(i)=(c:[4],o)
where
 (c[0..2],o)=CountMod(6)(i);          // Compteur modulo 6
        c[3] = 0;                     // poids fort nul
end where;
```

Compteur modulo 24

Ce compteur ne change de valeur qu'une fois toutes les heures, soit tous les $1024 \times 3600 = 4\,849\,200$ cycles de l'horloge $t1KHz$. C'est pourtant le plus complexe à réaliser, pour des raisons arithmétiques, et non physiques : en effet, 10 ne divise pas 24 ! Le compteur $CountMod[24]$ donne un résultat sur 5 bits, qu'il faut ensuite traduire, par le circuit $Dec24$, en *décimal codé binaire*, sur deux chiffres de 4 bits pour l'affichage final de l'heure.

```
CM24(i)=(u:[4],d:[4])
where
 (c,o) = CountMod(24)(i);      // Compteur modulo 24, sur 5 bits
 (u,d) = Dec24(c);                    // Codage binaire sur 8 bits
end where;
```

Le circuit combinatoire $Dec24 \in \mathbf{B}^5 \mapsto \mathbf{B}^8$ a pour en-tête :

Dec24(c :[5]) = (u :[4],d :[4])

Pour le décrire, et ensuite le réaliser, le plus simple est de donner sa *table de vérité*. En binaire, celle-ci comporte $32 = 2^5$ lignes - une pour chacune des 5 bits d'entrée c :[5] - et 8 colonnes - 4 pour la sortie des unités u :[4] et 4 pour la sortie des dizaines d :[4]. Pour économiser le papier comme pour tester les aptitudes du lecteur à la conversion binaire/décimale, donnons la table en décimal : $c_{10} = c[0] + 2c[1] + \cdots + 32c[4]$ ainsi que pour u_{10} et d_{10}. Ignorons aussi les entrées non significatives de l'intervalle $24 \leq c_{10} < 32$; pour ces valeurs d'entrée, on peut choisir les valeurs des sorties u, d de façon *arbitraire*.

c_{10}	0	1	2	3	4	5	6	7	8	9	10	11
u_{10}	0	1	2	3	4	5	6	7	8	9	0	1
d_{10}	0	0	0	0	0	0	0	0	0	0	1	1

c_{10}	12	13	14	15	16	17	18	19	20	21	22	23
u_{10}	2	3	4	5	6	7	8	9	0	1	2	3
d_{10}	1	1	1	1	1	1	1	1	2	2	2	2

Nous montrons en section 2.3 comment traduire *automatiquement* de telles tables en un circuit combinatoire. Une autre approche à la réalisation de $Dec24$ est proposée en section 3.3.3, au moyen d'une mémoire ROM - *Read Only Memory*.

1.4.3 Affichage sur cristaux liquides

Dans un *cristal liquide*, on injecte de multiples petites particules magnétiques. En l'absence de champ magnétique, ces particules s'orientent de façon aléatoire dans le liquide, et le cristal est *opaque* à la lumière. Si on applique maintenant un champ magnétique de valeur convenable, les particules s'orientent toutes dans le même sens, et le cristal devient alors *transparent*, au moins dans une plage de fréquences optiques proches du *vert*. On éclaire l'arrière du cristal, et on commande

Planche 1.6 – Détail d'un afficheur à cristaux liquides

le champ électrique par un signal *digital* : quand il vaut 0, la zone cristalline semble éteinte et opaque, vu de l'avant ; quand la commande vaut 1, la zone s'allume en prenant une teinte verte caractéristique.

En disposant, sur un support adéquat, des segments de cristaux liquides, on obtient un afficheur d'images digitales rudimentaires. La disposition classique, dite *sept segments* - chacun commandé par un bit de contrôle - permet de représenter les chiffres de 0 à 9 suivant le dessin de la planche 1.7.

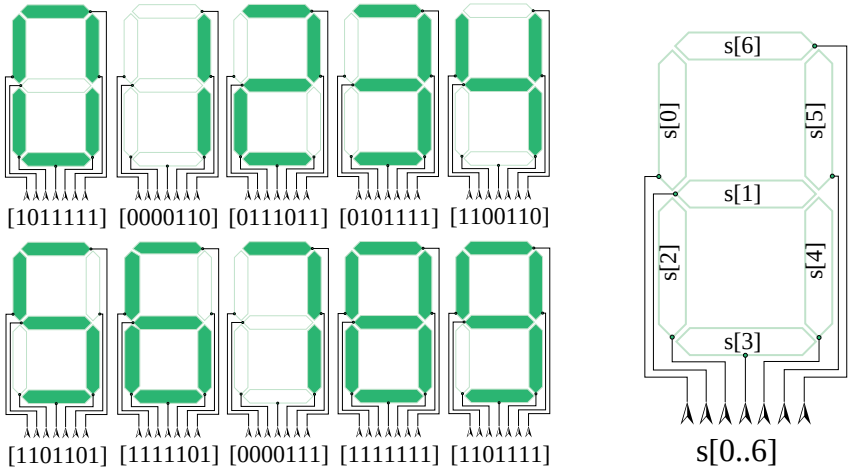


Planche 1.7 – Code sept segments des chiffres décimaux

Décodeur sept segments

On connecte enfin chacune des sorties sur 4 bits des compteurs de la section qui précède à un affichage sur sept segments par cristaux liquides, en passant au travers d'un décodeur $Dec7$. La *table de vérité* de cette fonction booléenne $Dec7 \in \mathbf{B}^4 \mapsto \mathbf{B}^7$, avec 4 bits d'entrée et 7 bits de sortie se dérive en lisant la planche 1.7.

e_{10}	0	1	2	3	4	5	6	7	8	9
e_2	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
$s[0]$	1	0	0	0	1	1	1	0	1	1
$s[1]$	0	0	1	1	1	1	1	0	1	1
$s[2]$	1	0	1	0	0	0	1	0	1	0
$s[3]$	1	0	1	1	0	1	1	0	1	1
$s[4]$	1	1	0	1	1	1	1	1	1	1
$s[5]$	1	1	1	1	1	0	0	1	1	1
$s[6]$	1	0	1	1	0	1	1	1	1	1

Les entrées $e[0..3]$ sont données en décimal e_{10} et en binaire e_2. La table est *partielle* : les valeurs des sorties correspondant aux entrées non utilisées $e_{10} \in \{10, 11, 12, 13, 14, 15\}$ peuvent être choisies arbitrairement. Nous montrons en section 2.3 comment traduire automatiquement cette table pour la réaliser par un circuit CDS combinatoire ou par une mémoire ROM en section 3.3.3.

Tout ceci nous permet de construire, à l'aide des techniques du chapitre 3, une montre digitale rudimentaire : il lui manque un dispositif de remise à l'heure, ainsi qu'un dispositif de calcul et d'affichage de la date.

Pour un véritable fabricant de montres, dont les volumes de production se chiffrent en millions d'unités par an, la structure proposée ici à fins pédagogiques est loin de l'optimal. On peut en effet réaliser la même fonction par des circuits beaucoup plus petits physiquement, et donc moins chers à construire par unité. Les circuits électroniques modernes fonctionnent en effet avec des périodes d'horloge proches de la *nano seconde* - 1 ns = 10^{-9} sec. - soit une fréquence de l'ordre du *giga* Hertz. Un microprocesseur qui opère à la fréquence de 1 GHz est capable de calculer 10^9 opérations par seconde - soit mille millions ! C'est *largement* assez pour calculer en séquence - et non en parallèle comme nous le faisons ici - la centaine d'opérations binaires à calculer par seconde, pour tenir à jour l'affichage du temps à l'échelle humaine. Il reste beaucoup de marge pour calculer en prime la date, et surtout, pour réduire la *consommation d'énergie* : la pile d'alimentation d'une montre a un volume de quelques mm^3, et elle doit durer plusieurs années. Nous reprenons ceci au chapitre 5, comme exemple de programmation d'un microprocesseur.

Chapitre 2

Algèbre binaire

Contents

Au chapitre 1, nous dotons les circuits CDS d'une syntaxe et d'une sémantique. Prenons, pour simplifier, le cas d'un circuit $C \in \mathcal{C}_{ds}$ ayant une seule entrée $i = \mathcal{I}(C)$, et une seule sortie $o = \mathcal{O}(C)$. Sa fonction $[\![C]\!]$ est une *application* de la suite des bits d'entrée

$$[\![i]\!] = i(0), i(1), \cdots, i(t), \cdots$$

dans la suite des bits de sortie

$$[\![o]\!] = o(0), o(1), \cdots, o(t), \cdots$$

Mathématiquement, une suite infinie de bits est une application de l'ensemble $\mathbf{N}$ des entiers naturels dans l'ensemble binaire $\mathbf{B} = \{0, 1\}$, ce que nous écrivons $[\![i]\!] \in \mathbf{N} \mapsto \mathbf{B}$ et $[\![o]\!] \in \mathbf{N} \mapsto \mathbf{B}$. Quant à la fonction $[\![C]\!]$ du circuit, c'est une application de $\mathbf{N} \mapsto \mathbf{B}$ dans $\mathbf{N} \mapsto \mathbf{B}$, soit

$$[\![C]\!] \in (\mathbf{N} \mapsto \mathbf{B}) \mapsto (\mathbf{N} \mapsto \mathbf{B}).$$

L'objet de ce chapitre est d'étudier les propriétés *algébriques* des suites infinies de bits $\mathbf{N} \mapsto \mathbf{B}$ - qui représentent les *valeurs* des variables des circuits CDS - ainsi que celles des applications sur de telles suites $(\mathbf{N} \mapsto \mathbf{B}) \mapsto (\mathbf{N} \mapsto \mathbf{B})$ - qui représentent les *fonctions* des circuits CDS.

Nous considérons ici trois *interprétations* principales des suites binaires :

(1) comme *ensemble d'entiers* ;

(2) comme *entier 2-adique* ;

(3) comme *série formelle* à coefficients dans le corps à deux éléments.

Avec chaque interprétation vient naturellement une famille d'opérateurs, ensemblistes $\neg, \cup, \cap$ pour (1), et arithmétiques $+, -, \times$ pour (2), et $\oplus, \otimes$ pour (3). Ceci donne aux suites binaires $\mathbf{N} \mapsto \mathbf{B}$ une structure triple : à la fois une *algèbre de Boole*, un *anneau entier* et un *anneau* de polynômes, ce que nous nommons une emphalgèbre binaire.

L'étude de l'algèbre binaire permet de comprendre la sémantique des opérateurs que l'on trouve dans certains *langages de programmation* comme C. Elle s'applique aussi à l'analyse et à la synthèse des circuits CDS, combinatoires (sans mémoire) en section 2.3 et séquentiels (avec mémoire) en section 6.2.

2.1 Numérations de position

2.1.1 Ecriture et lecture des entiers

Les règles de lecture et d'écriture des nombres décimaux sont un cas particulier - pour $b = 10$ - de celles des *numérations de position*, définies pour toute base entière $b \in \mathbf{N} + 2$, soit $b \geq 2$.

Algorithme 2 (Ecriture par les poids faibles) *L'écriture en base b de l'entier positif $a \in \mathbf{N} + 1$ est la suite finie de* chiffres $0 \leq a_k < b$ *:*

$$\mathcal{E}_b(a) = [a_{n-1}a_{n-2}\cdots a_1a_0]_b$$

que l'on calcule de proche en proche, à partir des poids faibles. On pose $A_0 = a$, et on calcule, pour $k \in \mathbf{N}$:

$$\begin{aligned} a_k &= A_k \cdot|\cdot b, \\ A_{k+1} &= A_k \div b, \end{aligned}$$

jusqu'à trouver $A_n = 0$. Le chiffre le plus significatif $a_{n-1} > 0$ ne peut donc être nul.

Nous utilisons $a \div b$ pour représenter le *quotient* $a \div b$ de la *division entière* de a par b, et $a \cdot|\cdot b$ pour représenter le *reste* ; par définition, on a :

$$a = b \times (a \div b) + (a \cdot|\cdot b), \text{ avec } a \div b \in \mathbf{Z} \text{ et } 0 \leq a \cdot|\cdot b < |b|. \qquad (2.1)$$

La formule (2.1) définit donc la *division entière* de deux nombres réels $a, b \in \mathbf{R}$, avec $b \neq 0$ non nul.

Algorithme 3 (Lecture par les poids forts) *La valeur du nombre entier $a \in \mathbf{N}$ dont la représentation en base b est $[a_{n-1}a_{n-2}\cdots a_1a_0]_b$ se calcule à partir des poids forts, à l'inverse de l'écriture. On pose $A_n = 0$, et on calcule, pour $k > 0$:*

$$A_{k-1} = b \times A_k + a_{k-1},$$

jusqu'à trouver $a = A_0$.

On constate que les nombres A_k rencontrés dans l'algorithme 2 de lecture et dans l'algorithme 3 d'écriture sont les *mêmes* ; on trouve ainsi :

$$a = [a_{n-1}\cdots a_1a_0]_b = \sum_{0 \leq k < n} a_k b^k. \qquad (2.2)$$

Cette formule définit la *sémantique* de la représentation en base b, c'est à dire la valeur qu'il convient de donner à cette suite de chiffres. En éliminant les nombres A_k dans la définition de l'algorithme 3, on trouve une formule équivalente à (2.2), dite *schéma de Horner* :

$$[a_{n-1} \cdots a_1 a_0]_b = a_0 + b \times (a_1 + b \times (a_2 + \cdots + b \times a_{n-1})).$$

Proposition 1 *La* longueur $n = |a|_b$ *de la représentation* $[a_{n-1} \cdots a_0]_b$ *en base* b *de l'entier positif* $a > 0$ *est donnée par la formule :*

$$\begin{aligned} n \ = \ |a|_b \ &= \ log_b(a+1) \div 1 = 1 + log_b(a) \div 1 && (2.3) \\ &= \ \lceil log_b(a+1) \rceil = 1 + \lfloor log_b(a) \rfloor. && (2.4) \end{aligned}$$

Preuve : Dans la formule (2.2), le chiffre le plus significatif $a_{n-1} \geq 1$ n'est pas nul, donc : $b^{n-1} \leq a = \sum_k a_k b^k < b^n$. En prenant le logarithme en base b, il vient : $n - 1 \leq log_b(a) < n$. Le nombre entier $n - 1 \in \mathbf{N}$ est donc la partie entière (dite *plancher*) du nombre réel $log_b(a)$, soit $n - 1 = \lfloor log_b(a) \rfloor$. L'entier n est aussi le *plafond* $\lceil log_b(a+1) \rceil$, i.e. le plus petit entier supérieur ou égal au réel $log_b(a+1)$. Par définition de la division entière (2.1), on voit que le *plancher* d'un réel est égal au quotient de la division par 1, soit $\lfloor log_b(a) \rfloor = log_b(a) \div 1$. **Q.E.D.**

2.1.2 Ordre d'écriture

Certains algorithmes - addition, multiplication - traitent les chiffres en procédant des poids faibles vers les poids forts. D'autres algorithmes - comparaison, division - traitent les chiffres en procédant à l'inverse, des poids forts vers les poids faibles. Ceci nous amène naturellement à écrire les nombres dans un sens ou dans l'autre, suivant le propos. Afin d'éliminer toute dyslexie, convenons d'écrire explicitement la base b de numération en indice du *chiffre de poids faible*, chaque fois que se présente un risque d'ambiguïté. A titre d'illustration, écrivons le nombre douze de six façons :

$$12_{10} \ = \ 1010_2 \ = \ {}_20101 \ = \ 110_3 \ = \ {}_3011 \ = \ \mathbf{XII}.$$

2.1.3 Changement de base

Considérons un entier positif $n \in \mathbf{N} + 1$, et écrivons ce nombre dans diverses bases de numération. Par exemple :

$$\begin{aligned} &1997_{10} = 2658_9 = 3715_8 = 5552_7 = 13125_6 \\ &= 30442_5 = 133031_4 = 2201222_3 = 11111001101_2. \end{aligned}$$

Quels liens existent-ils entre les suites de chiffres obtenues dans chaque base ?

Par la formule (2.3), nous connaissons les longueurs de ces représentations (la vérifier pour n=1997). Les relations entre les chiffres de ces diverses numérations sont complexes si les bases sont premieres entres elles : comme $b = 2$ et $b = 3$. La relation est simple quand une base b^l est puissance de l'autre b, pour l entier.

Proposition 2 *Si la représentation de l'entier n en base b est donnée par les $k = |n|_b$ chiffres $n = [n_{k-1} \cdots n_0]_b$, sa représentation en base $b' = b^l$ est donnée par les $k' = |n|_{b^l} = (k + l - 1) \div l$ chiffres $n = [m_{k'-1} \cdots m_0]_{b'}$, avec $m_j = [n_{lj+j-1} \cdots n_{lj}]_b = \sum_{0 \leq m < l} n_l 2^l$, en étendant la représentation par des chiffres non significatifs $n_h = 0$ pour $h \geq k$.*

Cette proposition explique le lien simple entre les représentations de l'entier $n = 1997$ dans les bases 2 et 4 à savoir : $n = 1\,;11\ 11\ 00\ 11\ 01_2 = 133031_4$; et, dans la base 8 : $n = 11\ 111\ 001\ 101_2 = 3715_8$.

2.2 Nombre binaire fini

Etudions ici la structure algébrique des nombres que l'on peut représenter avec un nombre fini $n \in \mathbf{N}$ de bits, soit l'ensemble $\mathbf{Z}_{2^n}$ des entiers modulo 2^n. Nous étendrons ces résultats dans la section 2.4 à l'ensemble ${}_2\mathbf{Z}$ des entiers 2-adiques, dont les représentations binaires sont *infinies*.

2.2.1 Entiers n bits

Soit $B = \{0, 1\}$ les bits, et $\mathbf{B}^n$ l'ensemble des suites binaires de longueur n. Les 2^n éléments de $\mathbf{B}^n$ permettent de *coder* toute structure finie, ayant au plus 2^n éléments. Considérons ici deux de ces codages.

1. L'écriture binaire des entiers, étendue et tronquée à n bits, établit une bijection $\mathbf{B}^n \rightleftharpoons \mathbf{Z}_{2^n}$ entre $\mathbf{B}^n$ et l'ensemble $\mathbf{Z}_{2^n} = \{0, \cdots, 2^n - 1\}$ des 2^n premiers entiers naturels. La correspondance est donnée par la formule explicite :

$$ {}_2a_0a_1 \cdots a_n = \sum_{0 \leq k < n} a_k 2^k, \tag{2.5}$$

   dans laquelle le nombre binaire qui figure en membre de gauche est écrit avec les poids faibles en tête, ce qui correspond à l'ordre naturel de traitement par les algorithmes qui vont suivre.

2. En considérant une suite de n bits comme le *vecteur caractéristique* d'un ensemble, on établit une bijection $\mathbf{B}^n \rightleftharpoons \wp(\mathbf{Z}_n)$ entre $\mathbf{B}^n$ et l'ensemble $\wp(\mathbf{Z}_n)$ des *parties* de $\mathbf{Z}_n = \{0, \cdots, n - 1\}$, c'est à dire l'ensemble des ensembles d'entiers inférieurs à n. Cette correspondance est donnée par la formule :

$$ {}_2a_0a_1 \cdots a_n = \{k \in \mathbf{N} \ : k < n \text{ et } a_k = 1\} \tag{2.6}$$

B^4	$\mathbf{Z}_{16}$	$\wp(\mathbf{Z}_4)$
$_2$0000	0	$\{\}$
$_2$1000	1	$\{0\}$
$_2$0100	2	$\{1\}$
$_2$1100	3	$\{0,1\}$
$_2$0010	4	$\{2\}$
$_2$1010	5	$\{0,2\}$
$_2$0110	6	$\{1,2\}$
$_2$1110	7	$\{0,1,2\}$

B^4	$\mathbf{Z}_{16}$	$\wp(\mathbf{Z}_4)$
$_2$0001	8	$\{3\}$
$_2$1001	9	$\{0,3\}$
$_2$0101	10	$\{1,3\}$
$_2$1101	11	$\{0,1,3\}$
$_2$0011	12	$\{2,3\}$
$_2$1011	13	$\{0,2,3\}$
$_2$0111	14	$\{1,2,3\}$
$_2$1111	15	$\{0,1,2,3\}$

Planche 2.1 – Entiers sur 4 bits

La planche 2.1 donne la table des bijections $\mathbf{B}^n \rightleftharpoons \mathbf{Z}_{2^n} \rightleftharpoons \wp(\mathbf{Z}_n)$, pour $n = 4$. La formule (2.5) permet de définir les opérations *arithmétiques* $+, -, \times$ sur les suites binaires $\mathbf{B}^n$. La formule (2.6) permet de définir les opérations *ensemblistes* $\neg, \cap, \cup$ sur les mêmes suites binaires $\mathbf{B}^n$.

2.2.2 Anneau commutatif

L'allemand Gauss est le *prince* des mathématiciens. Son traité "Disquisitiones Aritmeticæ", publié en 1801, demeure une référence fondamentale de l'arithmétique. Il y introduit, entre autres, la notion de classe de nombres modulaires, dont nous faisons un usage intensif.

Planche 2.2 – Karl Freidrich Gauss

On sait que les nombres $\mathbf{Z}_p = \{0, 1, \cdots, p-1\}$ m unis des opérations $+, -, \times$ prises *modulo* p, forment un *anneau*, pour tout $p \in \mathbf{N} + 2$.

Définition 5 *Un* anneau commutatif $(E, 0, 1, -, +, \times)$ *est composé d'un ensemble* E *comprenant* $0, 1 \in E$*, l'opération unaire* $- \in E \mapsto E$ *et les deux opérations binaires* $+, \times \in E^2 \mapsto E$*. Pour tout* $a, b, c \in E$*, on a les égalités suivantes.*

$$\begin{array}{rcl} (a+b)+c & = & a+(b+c) \\ a+b & = & b+a \\ a+(-a) & = & 0 \\ a+0 & = & a \end{array} \qquad \begin{array}{rcl} (a\times b)\times c & = & a\times(b\times c) \\ a\times b & = & b\times a \\ a\times(b+c) & = & (a\times b)+(a\times c) \\ a\times 1 & = & a \\ a\times 0 & = & 0 \end{array}$$

L'anneau $\mathbf{Z}_{2^n}$ n'est pas *intègre*, car il contient en général des diviseurs non triviaux de zéro ; par exemple, $2 \times 2 = 4 = 0 \pmod 4$. L'anneau $\mathbf{Z}_p$ est intègre si et seulement si p est un nombre *premier*. En particulier, $\mathbf{Z}_2 = \mathbf{B}$ est le (seul) *corps* à deux éléments.

2.2.3 Algèbre de Boole

Le mathématicien anglais Boole étudie la structure algébrique qui porte désormais son nom dans son ouvrage "The Mathematical Analysis of Logic", en 1847.

Planche 2.3 – Georges Boole

On sait depuis Boole (planche 2.3) que l'ensemble $\wp(X)$ des parties d'un ensemble arbitraire X forme une *algèbre de Boole*.

Définition 6 (Algèbre de Boole) *Une* algèbre de Boole $(E, \emptyset, \overline{\emptyset}, \neg, \cap, \cup)$ *est composée d'un ensemble E comprenant deux éléments distingués $\emptyset, \overline{\emptyset} \in E$, d'une opération unaire $\neg \in E \mapsto E$ et de deux opérations binaires $\cap, \cup \in E^2 \mapsto E$. Pour tout $a, b, c \in E$, on a les égalités suivantes.*

$$\begin{array}{rcl} (a\cap b)\cap c & = & a\cap(b\cap c) \\ a\cap b & = & b\cap a \\ a\cap(b\cup c) & = & (a\cap b)\cup(a\cap c) \\ a\cup\emptyset & = & a \\ a\cap\neg a & = & \emptyset \end{array} \qquad \begin{array}{rcl} (a\cup b)\cup c & = & a\cup(b\cup c) \\ a\cup b & = & b\cup a \\ a\cup(b\cap c) & = & (a\cup b)\cap(a\cup c) \\ a\cap\overline{\emptyset} & = & a \\ a\cup\neg a & = & \overline{\emptyset} \end{array}$$

Partant des 10 *axiomes* qui caractérisent ici l'algèbre de Boole, on peut dériver de nombreuses autres formules dont l'interprétation ensembliste est intuitive. Par exemple, on prouve que

$$a \cap \emptyset \;=\; \emptyset, \qquad a \cup \overline{\emptyset} \;=\; \overline{\emptyset},$$

en écrivant : $a \cap \emptyset = (a \cap \emptyset) \cup \emptyset = (a \cap \emptyset) \cup (a \cap \neg a) = a \cap (\emptyset \cup \neg a) = a \cap \neg a = \emptyset$.

On dérive de façon similaire les lois de *DeMorgan* :

$$\neg(a \cap b) \;=\; (\neg a) \cup (\neg b), \qquad \neg(a \cup b) \;=\; (\neg a) \cap (\neg b). \tag{2.7}$$

On peut illustrer ces lois par deux paires de schémas logiques, qui sont *sémantiquement équivalents* - planche 2.4.

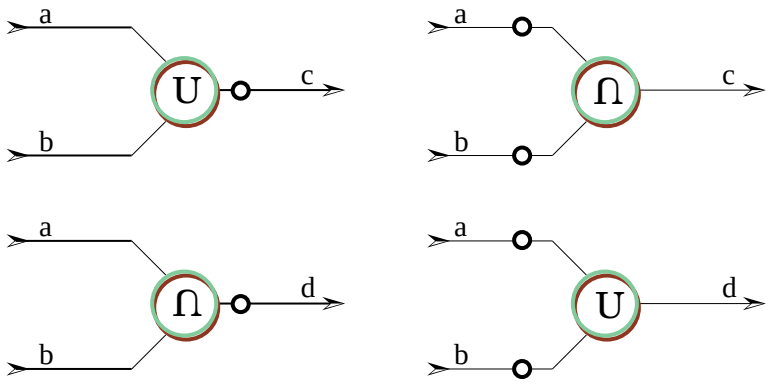


Planche 2.4 – Lois de DeMorgan

Remarquons que les égalités de la définition 6 sont données par *paires*, et que le système reste *invariant* quand on échange $\cap$ avec $\cup$, et $\emptyset$ avec $\overline{\emptyset}$. Donc, si une égalité est dérivable à partir des axiomes, il en va de même pour la proposition *duale*, obtenue en échangeant dans celle-ci $\cap$ avec $\cup$, et $\emptyset$ avec $\overline{\emptyset}$. C'est le *principe de dualité* de l'algèbre de Boole. Ainsi, de l'égalité $a \cap \emptyset = \emptyset$ prouvée ci-dessus, on déduit par dualité que $a \cup \overline{\emptyset} = \overline{\emptyset}$. Il résulte par dualité que : si $(E, \emptyset, \overline{\emptyset}, \neg, \cap, \cup)$ est une algèbre de Boole, alors, $(E, \overline{\emptyset}, \emptyset, \neg, \cup, \cap)$ est *aussi* une algèbre de Boole.

Exemples

- La plus simple des algèbre de Boole est $(\mathbf{B}, 0, 1, \mathbf{not}, \mathbf{and}, \mathbf{or})$. Ici, $\mathbf{B} = \{0, 1\}$ est l'ensemble des bits et les opérateurs logiques $\mathbf{not}, \mathbf{and}, \mathbf{or}$ sont définis au chap. 1. La vérification des 10 axiomes est laborieuse, mais sans difficulté.
- Soit X un ensemble arbitraire, et $\wp(X) = \{A \;:\; A \subseteq X\}$ l'ensemble de ses *parties* (sous-ensembles de X). Alors, $(\wp(X), \emptyset, X, \neg, \cap, \cup)$ est une

algèbre de Boole. Ici, $\emptyset$ est l'ensemble vide et $\overline{\emptyset} = X$ son complémentaire ; l'opérateur $\cap$ dénote l'intersection d'ensembles $A \cap B = \{c \in X : c \in A \text{ et } c \in B\}$; l'opérateur $\cup$ est l'union d'ensembles $A \cup B = \{c \in X : c \in A \text{ ou } c \in B\}$ et $\neg$ dénote le complément à X, soit $\neg A = \{c \in X : c \notin A\}$. Quand $X = \{1\}$ n'a qu'un élément, on retrouve (à un isomorphisme près) l'algèbre de Boole à deux éléments $\mathbf{B} = \{0, 1\}$.
Quand X est un ensemble *fini* à $n = |X|$ éléments, l'algèbre de Boole sur $\wp(X)$ possède 2^n éléments. La proposition 3 montre que toute algèbre de Boole finie est de ce type.

– L'ensemble des applications $f : \mathbf{B}^n \mapsto \mathbf{B}$ muni des opérations

$$(\neg f)(x) = \neg f(x),\ (f \cap g)(x) = f(x) \cap g(x) \text{ et } (f \cup g)(x) = f(x) \cup g(x)$$

est une algèbre de Boole, isomorphe à celle des parties $\wp(\mathbf{B}^n)$ de l'ensemble des suites de n bits. Cette algèbre possède 2^{2^n} éléments.

– Soit $D6 = \{1, 2, 3, 6\}$ l'ensemble des diviseurs du nombre 6, $PGCD(a, b)$ le *plus grand commun diviseur* des entiers $a, b \in \mathbf{N}$, et $PPCM(a, b)$ le *plus petit commun multiple*. Alors, $(D6, 1, 6, I6, PGCD, PPCM)$ est une algèbre de Boole, pour $I6(n) = 6/n$. Elle est isomorphe à l'algèbre de Boole des parties $\wp(\mathbf{B})$ de l'ensemble à 2 éléments. En revanche, si on considère l'ensemble $D4 = \{1, 2, 4\}$ des diviseurs de 4, avec $I4(n) = 4/n$, la structure $(D4, 1, 4, I4, PGCD, PPCM)$ ***n'est pas*** une algèbre de Boole ; en effet, comme $I4(2) = 2$, on a $PPCM(2, I4(2)) = PGCD(2, I4(2)) = 2$, qui est contraire aux équations $a \cap \neg a = \emptyset$ et $a \cup \neg a = \overline{\emptyset}$ de la définition 6. Plus généralement, la structure $(Dk, 1, k, Ik, PGCD, PPCM)$ des diviseurs de l'entier positif $k \in \mathbf{N} + 1$, avec $Ik(n) = k/n$, est une algèbre de Boole si et seulement si le nombre k est *sans carré*, c'est à dire produit de nombres premiers tous distincts.

– Soit $[0, 1] = \{x \in \mathbf{R} : 0 \leq x \leq 1\}$ l'intervalle réel unitaire, que l'on dote des opérations $\sim (x) = 1 - x$, $x \wedge y = xy$ et $x \vee y = x + y - xy$. La structure $([0, 1], 0, 1, \sim, \wedge, \vee)$ satisfait les premiers axiomes de la définition 6, mais pas les suivants ; en particulier : $x \wedge x = x^2 \neq x$, sauf pour $x = 0$ et $x = 1$.

Algèbre de Boole finie

Proposition 3 *Toute algèbre de Boole* finie $(E, \emptyset, \overline{\emptyset}, \neg, \cap, \cup)$ *possède* 2^n *éléments, pour quelque entier* $n \in \mathbf{N}$. *Elle est* isomorphe *à l'algèbre des parties* $\wp(Z_n)$*, où* $\mathbf{Z}_n = \{0, 1, \cdots, n - 1\}$.

Preuve : Un *atome* de l'algèbre de Boole est un élément $a \in E$ non nul $a \neq \emptyset$ tel que, pour tout $x \in E$, on a : soit $y \cap a = a$, soit $y \cap a = \emptyset$. On montre alors que l'algèbre E est isomorphe à celle des parties $\wp(A)$, où A est l'ensemble des atomes. Pour une preuve explicite de ce résultat de logique élémentaire, consulter par exemple le chap. 2 de Gilbert [6]. **Q.E.D.**

Treillis des nombres

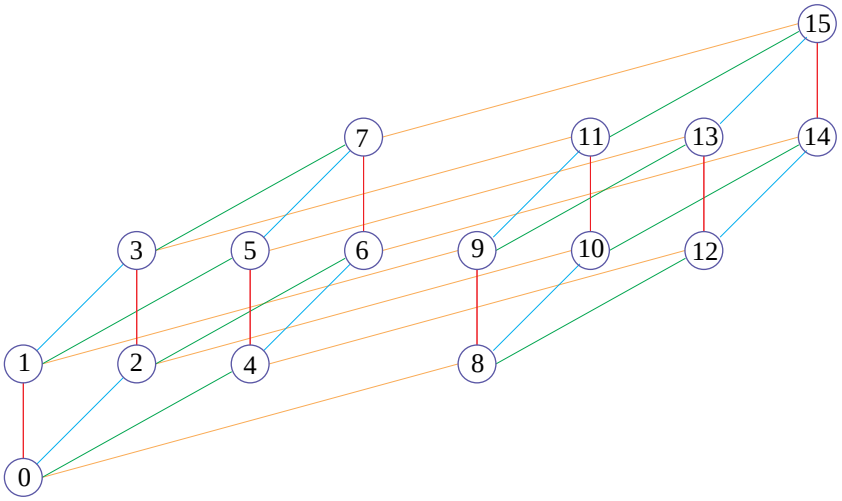


Planche 2.5 – L'hyper-cube H4

Soit $(E, \emptyset, \overline{\emptyset}, \neg, \cap, \cup)$ une algèbre de Boole. On définit la relation $\subseteq$ par :

$$a \subseteq b \text{ si et seulement si } a \cap b = a,$$

ou de façon équivalente, par $a \subseteq b$ si et seulement si $a \cup b = b$. Si $E = \wp(X)$ est l'ensemble des parties de X, la relation $\subseteq$ est l'inclusion ensembliste classique - lire a est un sous-ensemble de b. La relation $\subseteq$ est un *ordre partiel* sur E, c'est à dire que :

1. $a \subseteq a$; (réflexivité)
2. si $a \subseteq b$ et $b \subseteq a$, alors $a = b$; (anti-symétrie)
3. si $a \subseteq b$ et $b \subseteq c$, alors $a \subseteq c$. (transitivité)

Un ordre partiel sur un ensemble fini E peut être visualisé par son *graphe*, dont chaque nœud est un élément $a \in E$ et chaque arête entre a et b (orientée de bas en haut) indique que $a \subseteq b$ hors transitivité, c'est à dire qu'il n'existe pas d'élément $c \notin \{a, b\}$ tel que $a \subseteq c \subseteq b$. Pour une algèbre de Boole sur l'ensemble $\wp(X)$ des parties de X, ce graphe est connu sous le nom d'*hyper-cube* de dimension $n = |X|$. L'hyper-cube H4 de dimension 4 associé à l'algèbre de Boole sur les entiers $\mathbf{Z}_{16} = \{0, 1, \cdots, 15\}$ se trouve en planche 2.2.3. Au vu de telles figures, il est clair que l'ordre partiel $\subseteq$ sur les éléments d'une algèbre de Boole possède des propriétés très particulières.

1. C'est un *treillis*. Ceci signifie que toute paire d'éléments $a, b \in E$ possède un *plus petit majorant* $j = a \cup b$, et un *plus grand minorant* $i = a \cap b$. Le plus petit majorant est tel que $a \subseteq j$, $b \subseteq j$ et, pour tout c tel que $a \subseteq c$,

$b \subseteq c$, on a $j \subseteq c$. Le plus grand minorant est tel que $i \subseteq a$, $i \subseteq b$ et, pour tout c tel que $c \subseteq a$, $c \subseteq b$, on a $c \subseteq i$.

2. Le treillis est *borné*, c'est à dire qu'il contient deux éléments $\emptyset$ et $\overline{\emptyset}$ tels que $\emptyset \subseteq a \subseteq \overline{\emptyset}$, pour tout $a \in E$.
3. Le treillis est *distributif*, soit $a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$ et $a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$.
4. Tout élément $a \in E$ admet une *négation* $\neg a \in E$ telle que $a \cap \neg a = \emptyset$ et $a \cup \neg a = \overline{\emptyset}$.

Ces quatre propriétés caractérisent l'ordre partiel $\subseteq$ associé à une algèbre de Boole. On peut les utiliser en place de la définition 6, pour arriver au même résultat.

Anneau booléen

Soit $(E, \emptyset, \overline{\emptyset}, \neg, \cap, \cup)$ une algèbre de Boole. On définit l'*union exclusive* $a \oplus b = \mathbf{xor}(a, b)$ de deux éléments $a, b \in E$ par la formule :

$$a \oplus b = (a \cap \neg b) \cup (\neg a \cap b).$$

On voit directement que l'opérateur $\oplus$ est commutatif, associatif, et distributif vis à vis de $\cap$:

$$a \cap (b \oplus c) = (a \cap b) \oplus (a \cap c).$$

Définissons ici $-a = a$, et vérifions que $a \oplus (-a) = a \oplus a = \emptyset$. Il en résulte que la structure $(E, \emptyset, \overline{\emptyset}, \oplus, \cap)$ est un *anneau booléen*, c'est à dire un anneau dans lequel $a \cap a = a$ pour tout $a \in E$. Inversement, soit $(E, 0, 1, +, -, \times)$ un anneau booléen, avec $a \times a = a$ pour tout $a \in E$. En définissant alors $a \cap b = a \times b$, $a \cup b = a + b - a \times b$ et $\neg a = 1 + a$, on vérifie sans peine que $(E, 0, 1, \neg, \cap, \cup)$ est une algèbre de Boole. Cette correspondance est biunivoque entre algèbre de Boole et anneau booléen.

2.2.4 Algèbre binaire finie

Il est temps de regrouper les observations de cette section sous une forme plus synthétique. Soient $a = {}_2a_0 \cdots a_{n-1}$ et $b = {}_2b_0 \cdots b_{n-1}$ deux suites de n bits. A l'aide des correspondances $\mathbf{B}^n \rightleftharpoons \mathbf{Z}_{2^n} \rightleftharpoons \wp(\mathbf{Z}_n)$, données par les formules (2.5) et (2.6), définissons précisément les opérations arithmétiques et logiques dont nous avons doté les suites de n bits :

- la *somme* $a + b = s = {}_2s_0 \cdots s_{n-1}$ est donnée par la formule $\sum_{0 \le k < n} s_k 2^k = \sum_{0 \le k < n} (a_k + b_k) 2^k \pmod{2^n}$;
- la *différence* $a - b = d = {}_2d_0 \cdots d_{n-1}$ est donnée par la formule $\sum_{0 \le k < n} d_k 2^k = \sum_{0 \le k < n} (a_k - b_k) 2^k \pmod{2^n}$;
- le *produit* $a \times b = p = {}_2p_0 \cdots p_{n-1}$ est donnée par la formule $\sum_{0 \le k < n} p_k 2^k = (\sum_{0 \le k < n} a_k 2^k) \times (\sum_{0 \le k < n} b_k 2^k) \pmod{2^n}$;

- la *négation* $\neg a = n = {}_2 n_0 \cdots n_{n-1}$ est donnée par la formule $n_k = \mathbf{not}(a_k) = 1 - a_k$, pour $0 \leq k < n$;
- l'*intersection* $a \cap b = i = {}_2 i_0 \cdots i_{n-1}$ est donnée par la formule $i_k = \mathbf{and}(a_k, b_k) = a_k \times b_k$, pour $0 \leq k < n$;
- l'*union* $a \cup b = u = {}_2 u_0 \cdots u_{n-1}$ est donnée par la formule $u_k = \mathbf{or}(a_k, b_k) = a_k + b_k - a_k \times b_k$, pour $0 \leq k < n$;
- l'*union exclusive* $a \oplus b = x = {}_2 x_0 \cdots x_{n-1}$ est donnée par la formule $x_k = \mathbf{xor}(a_k, b_k) = (a_k + b_k) \cdot\!|\cdot 2$, pour $0 \leq k < n$;
- l'*inclusion* $a \subseteq b = c$ vaut $c = 1$ si $a_k \leq b_k$, pour tout $0 \leq k < n$; elle vaut $c = 0$ sinon.

L'opposé arithmétique unaire - dit *complément à deux* - se définit par $-a = 0 - a$ à partir de la différence binaire. La négation logique - dite *complément à un* - est telle que $a + \neg a = {}_2 11 \cdots 1 = 2^n - 1 = -1 \pmod{2^n}$. On a donc :

$$-a = (\neg a) + 1. \tag{2.8}$$

Chacune des 8 opérations ci-dessus possède un nombre *fini* de bits d'entrée (soit n, soit $2n$), et un nombre *fini* de bits de sortie (1,n ou $2n$). Chacune est calculable par un circuit CDS *combinatoire*, ce qui est fait au chap. 4 sur l'arithmétique binaire.

Les suites de n bits munies des opérations définies ci-dessus ont une structure algébrique caractéristique, que nous nommons *algèbre binaire*.

Théorème 2 (Algèbre binaire) *Les entiers* $\mathbf{Z}_{2^n} = \{0, 1, \cdots, 2^n - 1\}$ *représentables sur* n *bits forment une* algèbre binaire*, pour tout* $n \in \mathbf{N}$.

1. *C'est un* anneau commutatif $(\mathbf{Z}_{2^n}, 0, 1, +, -, \times)$.
2. *C'est une* algèbre de Boole $(\mathbf{Z}_{2^n}, 0, 2^n - 1, \neg, \cap, \cup)$.
3. *C'est un* anneau booléen $(\mathbf{Z}_{2^n}, 0, 2^n - 1, \oplus, \cap)$.
4. *C'est un* treillis $(\mathbf{Z}_{2^n}, 0, 2^n - 1, \subseteq, \neg)$ complet*,* distributif *et* complémenté*.*

Par la proposition 3, nous savons qu'une algèbre binaire *finie* est isomorphe à $\mathbf{Z}_{2^n}$, pour un n entier. Ceci caractérise donc *toute* algèbre binaire finie, à un isomorphisme près.

L'application $f : \mathbf{Z}_{2^m} \mapsto \mathbf{Z}_{2^n}$ définie, pour $m > n$, par $f(x) = x \cdot\!|\cdot 2^n$ est un *morphisme d'algèbre*. Utilisons le signe $\odot$ pour désigner l'une des six opérations binaires (avec deux arguments) définies ci-dessus, à l'exclusion de $\subseteq$. Pour ces six opérations, on a

$$f(a \odot b) = f(a) \odot f(b)$$

pour tout $a, b \in \mathbf{Z}_{2^m}$; dans cette formule l'opération $\odot$ de gauche se passe sur m bits, celle de droite sur n bits. Pour la négation, on a de même $f(\neg a) = \neg f(a)$. La signification *pratique* de cette observation est simple. Si l'on veut par exemple simuler les opérations d'un ordinateur dont le mot machine comporte n=8 bits avec un ordinateur ayant m=64 bits, il suffit de faire tous les calculs sur 64 bits, et de

zéro	0000	0	0	0	0
et	0001	$x \cap y$	$x \times y$	$\mu(x, y, 0)$	xy
non-et	0010	$\overline{x} \cap y$	$(1-x) \times y$	$\mu(x, 0, y)$	$\bar{x}y$
projection	0011	y	y	y	y
et-non	0100	$x \cap \overline{y}$	$x \times (1-y)$	$\mu(y, x, 0)$	$x\bar{y}$
projection	0101	x	x	x	x
ou exclusif	0110	$x \oplus y$	$(x-y)^2$	$\mu(x, \mu(y, 0, 1), y)$	$x\bar{y} \cup \bar{x}y$
ou	0111	$x \cup y$	$x + y - x \times y$	$\mu(x, 1, y)$	$x \cup y$
non-ou	1000	$\overline{x \cup y}$	$(1-x) \times (1-y)$	$\mu(x, 0, \mu(y, 0, 1))$	$\bar{x}\bar{y}$
équivalent	1001	$x \equiv y$	$1-(x-y)^2$	$\mu(x, y, \mu(y, 0, 1))$	$xy \cup \bar{x}\bar{y}$
non	1010	$\overline{x}$	$1-x$	$\mu(x, 0, 1)$	$\bar{x}$
implique	1011	$x \supset y$	$1 - x \times (1-y)$	$\mu(x, y, 1)$	$y \cup \bar{x}$
non	1100	$\overline{y}$	$1-y$	$\mu(y, 0, 1)$	$\bar{y}$
implique	1101	$y \supset x$	$1-(1-x) \times y$	$\mu(y, x, 1)$	$x \cup \bar{y}$
non-et	1110	$\overline{x \cap y}$	$1 - x \times y$	$\mu(x, y, 1)$	$\bar{x} \cup \bar{y}$
un	1111	1	1	1	1

Présentations par : nom usuel ; table de vérité ; symbolisme logique usuel ; expression arithmétique ; expression $\mu(c, b, a) = \mathbf{mux}(c, b, a)$; forme normale disjonctive DNF, avec $\overline{a} = \neg a$, $ab = a \cap b$.

Planche 2.6 – Les 16 applications booléennes $\mathbf{B}^2 \mapsto \mathbf{B}$

tronquer le résultat final à 8 bits. Sa signification *théorique* est moins banale ; elle nous permet d'étendre, dans la section 2.4, les résultats acquis aux suites binaires *infinies*.

2.3 Fonction combinatoire

Nous pouvons maintenant exploiter la structure d'algèbre binaire pour analyser, simplifier et synthétiser les circuits CDS. Considérons d'abord les *circuits combinatoires*, c'est à dire sans mémoire. La fonction d'un circuit combinatoire $C = (\mathcal{I}, \mathcal{E}, \mathcal{O})$, avec $n = |\mathcal{I}|$ entrées et $m = |\mathcal{O}|$ sorties est une *application booléenne* (dite aussi *fonction combinatoire*) :

$$[\![C]\!] \in \mathbf{B}^n \mapsto \mathbf{B}^m.$$

A chaque instant $t \in \mathbf{N}$, le circuit calcule le vecteur de m bits de sortie $O(t) = [\![C]\!](I(t))$ en réponse au vecteur $I(t)$ de n bits d'entrée.

2.3.1 Table de vérité

On classe les applications booléennes $f \in \mathbf{B}^n \mapsto \mathbf{B}$ par leur nombre n d'arguments, dit *arité*. Ainsi, l'ensemble $\mathbf{B} \mapsto \mathbf{B}$ des applications *unaires* d'arité un contient quatre éléments, la fonction nulle $f(x) = 0$, l'identité $f(x) = x$, la négation logique $f(x) = \neg x = 1 - x$ et la fonction unité $f(x) = 1$. Chacune de ces fonctions peut se décrire par sa table de vérité :

x	$f(x) = 0$
0	0
1	0

x	$f(x) = x$
0	0
1	1

x	$f(x) = \neg x$
0	1
1	0

x	$f(x) = 1$
0	1
1	1

De la même façon, une application booléenne à deux arguments

$$f \in \mathbf{B}^2 \to \mathbf{B}$$

se représente par une table de vérité comprenant les quatre valeurs

$$[f(0,0)\, f(1,0)\, f(0,1)\, f(1,1)].$$

Les $2^4 = 16$ applications $\mathbf{B}^2 \to \mathbf{B}$ sont identifiées dans la planche 2.6.

Définition 7 *La* table de vérité *d'une application $f \in \mathbf{B}^n \mapsto \mathbf{B}$ est le vecteur de 2^n bits $[f_0 \cdots f_{2^n-1}]$ dont on calcule la composante $f_k \in \mathbf{B}$ d'index $k \in \mathbf{Z}_{2^n}$ en écrivant k en binaire $k = \sum_{1 \le i < n} k_i 2^i = {}_2[k_0 \cdots k_{n-1}]$ et en posant $f_k = f(k_0, \ldots, k_{n-1}) \in \mathbf{B}$.*

Inversement, on associe à toute table de vérité $F = [F_0 \cdots F_{2^n-1}] \in \mathbf{B}^{2^n}$ l'application $f \in \mathbf{B}^n \mapsto \mathbf{B}$ définie par $f(b_0, \cdots, b_{n-1}) = F_b$, en posant $b = \sum_{1 \le i \le n} b_i 2^i$. Il y a donc *bijection* entre les applications $\mathbf{B}^n \mapsto \mathbf{B}$ et les suites $\mathbf{B}^{2^n}$: nous pouvons représenter les applications booléennes par des suites de bits, et réciproquement. En particulier, nous voyons que l'ensemble $\mathbf{B}^n \mapsto \mathbf{B}$ contient exactement 2^{2^n} applications.

Nous montrons dans la section 3.3.3 comment réaliser une application arbitraire $f \in \mathbf{B}^n \mapsto \mathbf{B}^m$ en inscrivant les $m2^n$ bits de la table de vérité de f dans une *mémoire morte* ROM - *Read Only Memory*.

Dans bien des cas, la réalisation des fonctions par ROM est la méthode de choix, pour le concepteur de circuits comme pour le programmeur.

- Pour $n \le 8$, on peut représenter chaque application booléenne $\mathbf{B}^n \mapsto \mathbf{B}$ par sa table de vérité, qui contient au plus 256 bits. Rangés au plus serré dans une ROM, ces 256 bits occupent souvent moins de place que les circuits concurrents, composés de portes discrètes.
- Quand la fonction à calculer n'a pas de *structure* évidente, la représentation par tables est souvent la seule option réaliste. C'est le cas, par exemple du décodeur $Dec24 \in \mathbf{B}^5 \mapsto \mathbf{B}^8$ de la section 1.4.2, qui transforme la représentation binaire sur 5 bits d'un nombre en ses deux chiffres décimaux,

codés en binaire sur 4 bits chacun. La ROM permettant de réaliser cette fonction n'a que $8 \times 2^5 = 256$ bits. Un autre exemple vient du décodeur sept segments $Dec7 \in \mathbf{B}^4 \mapsto \mathbf{B}^7$ - au paragraphe 1.4.3 - que l'on peut réaliser avec une ROM de $7 \times 2^4 = 112$ bits. Nous comparons plus loin avec le coût de réalisation de $Dec7$ par portes logiques.

Bien entendu, cette méthode de *force brute* a ses limites. La table de vérité d'une application $\mathbf{B}^{32} \mapsto \mathbf{B}^{32}$ comprend 32×2^{32} bits, soit 32 Gb (*giga* bits) ; c'est plus que la mémoire disponible dans la majorité des ordinateurs de 1997. Il est donc nécessaire de rechercher des représentations plus concises pour nos applications $f : \mathbf{B}^n \mapsto \mathbf{B}$ dès que n est grand. Pour cela, on cherche à exploiter la *structure* spécifique de chaque application f, pour la décomposer en opérations plus simples.

Quand une fonction d'arité faible apparaît souvent dans un même circuit, il vaut la peine d'en réduire la taille ; le gain, même s'il est faible, est multiplié par le nombre d'occurrences. Ainsi, dans la montre digitale de la section 1.4, le décodeur $Dec7$ apparaît 6 fois, et vaut plus d'effort que $Dec24$ dont on ne trouve qu'un seul exemplaire.

Au chap. 4, nous exploitons la structure des entiers binaires pour produire des circuits *arithmétiques* bien plus petits que toute mise en table, en les réduisant *tous* à des assemblages d'une fonction de base particulièrement importante, l'*additionneur binaire complet*. Nous l'étudions ici de près, car c'est la porte qui nous ouvre le passage de la logique à l'arithmétique.

Définition 8 *Un* additionneur binaire complet *(*Full Adder*) est un circuit réalisant la fonction* $\mathbf{abc} \in \mathbf{B}^3 \mapsto \mathbf{B}^2$*, de trois entrées* $a, b, c \in \mathbf{B}$ *et deux sorties* $r, s \in \mathbf{B}$ *définies par l'équation arithmétique :*

$$a + b + c \quad = \quad s + 2r. \tag{2.9}$$

Une solution arithmétique à l'équation (2.9) est donnée par :

$$\begin{aligned} s &= (a + b + c) \cdot|\cdot 2, \\ r &= (a + b + c) \div 2. \end{aligned}$$

Ceci nous permet de calculer la table de vérité de l'additionneur binaire complet.

a	01010101
b	00110011
c	00001111
s	01101001
r	00010111

(2.10)

Remarquons au passage que la table de vérité nous permet de numéroter les applications booléennes. Pour abc, on trouve par exemple :

$$\begin{aligned} s &= {}_2 01101001 = 150_{10}, \\ r &= {}_2 00010111 = 232_{10}. \end{aligned}$$

Signalons enfin que **abc** est une *fonction symétrique* de ses variables : par exemple, **abc** (a,b,c)=**abc** (b,c,a) ainsi que pour chacune des 6 permutations des trois variables. Pour une fonction symétrique, la table de vérité ne dépend pas de l'ordre des variables d'entrée. Ceci n'est pas vrai des fonctions non symétriques.

2.3.2 Expressions logiques

Soit une fonction combinatoire $f \in \mathbf{B}^n \mapsto \mathbf{B}^m$, donnée par les $m2^n$ bits de sa table de vérité, et que l'on cherche à exprimer par une expression composée d'opérateurs plus simples. Pour cela, il convient d'abord de choisir les opérateurs de base.

Base combinatoire Une *base combinatoire* est un jeu d'opérateurs permettant d'exprimer *toute* application booléenne. Il y en a beaucoup ; certaines viennent de la logique ; d'autres, de l'arithmétique ; d'autres enfin de la technologie. Toutes sont équivalentes : à partir des opérateurs de l'une, on peut exprimer ceux de l'autre, et réciproquement. Dans la liste qui suit, $a, b, c \in \mathbf{B}$ sont des variables booléennes arbitraires.

- $(0, -1, \mathbf{mux})$ est la base choisie au chap. 1. Nous montrons au paragraphe suivant que c'est effectivement une base.
- $(\neg, \mathbf{mux})$ est une variante : $0 = \mathbf{mux}(a, \neg a, a)$ et $-1 = \mathbf{mux}(a, a, \neg a)$.
- $(\neg, \cap, \cup)$ est la base de l'algèbre de Boole : $0 = a \cap \neg a$, $-1 = a \cup \neg a$ et $\mathbf{mux}(c, b, a) = (c \cap b) \cup ((\neg c) \cap a)$.
- $(-1, \cap, \oplus)$ est la base de l'anneau de Boole : $0 = a \oplus a$ et $\mathbf{mux}(c, b, a) = (c \cap b) \oplus (c \cap a) \oplus a$.
- $(0, -1, \mathbf{abc})$ est la base arithmétique : $(a \oplus b, a \cap b) = \mathbf{abc}(a, b, 0)$.
- **nor**, avec $\mathbf{nor}(a, b) = \neg(a \cup b)$ est une base des circuits MOS (cf. chap. 3) ; partant des lois de De Morgan (2.7), on trouve $\neg a = \mathbf{nor}(a, a)$, $a \cup b = \neg\mathbf{nor}(a, b)$ et $a \cap b = \mathbf{nor}(\neg a, \neg b)$. L'écriture de **mux** en terme de **nor** , toutes substitutions faites, est une expression bien lourde !
- **nand**, avec $\mathbf{nand}(a, b) = \neg(a \cap b)$ est la base *duale* des circuits MOS : $\neg a = \mathbf{nand}(a, a)$ et $\mathbf{nor}(a, b) = \neg\mathbf{nand}(\neg a, \neg b)$, par dualité.
- $\mathbf{muxnot}$, avec $\mathbf{muxnot}(c, b, a) = \neg\mathbf{mux}(c, b, a)$ est une autre base dont le générateur est unique.

Note : $(\cap, \cup)$ est une base des fonctions *monotones*, et **mux** est une base des fonctions *strictes*. Une fonction combinatoire f est : (a) monotone si $a \subseteq b$ implique $f(a) \subseteq f(b)$; (b) stricte si $f(\emptyset) = \emptyset$ et $f(\overline{\emptyset}) = \overline{\emptyset}$.

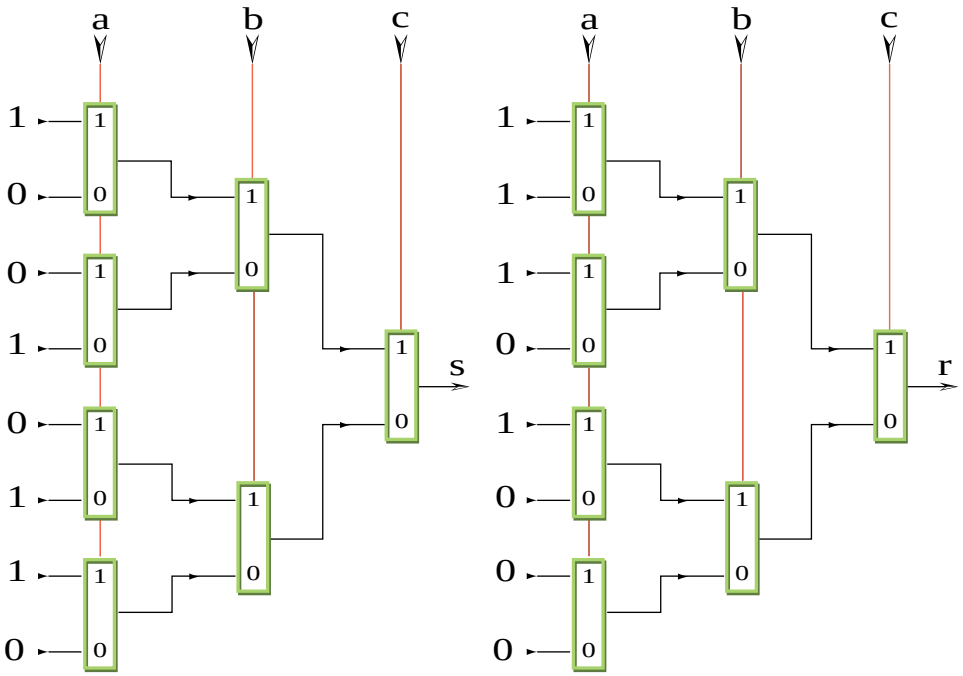


Planche 2.7 – Formule de Shannon pour abc

Diagramme de décision binaire : BDD

Soit $f \in \mathbf{B} \mapsto \mathbf{B}$ une fonction booléenne, donnée par sa table de vérité $f(0), f(1)$. On a, pour $x \in \mathbf{B}$:

$$f(x) = \mathbf{mux}(x, f(1), f(0)).$$

On voit que la fonction du multiplexeur est d'*appliquer* f à son argument x, pour calculer $f(x)$. En général, on trouve la *formule de Shannon* :

$$f(x_0, x_1 \cdots, x_{n-1}) = \mathbf{mux}(x_0, f_1(x_1 \cdots, x_{n-1}), f_0(x_1 \cdots, x_{n-1})).$$

Elle exprime f en terme de **mux** , et de deux fonctions d'arité $n-1$:
$f_0(x_1 \cdots, x_{n-1}) = f(0, x_1 \cdots, x_{n-1})$ et
$f_1(x_1 \cdots, x_{n-1}) = f(1, x_1 \cdots, x_{n-1})$. En décomposant alors *récursivement* f_0 et f_1 par la formule de Shannon, on termine finalement sur une expression de f en arbre binaire, dont les $2^n - 1$ nœuds sont des **mux** , et les 2^n feuilles contiennent la table de vérité de f.

Pour une fonction binaire $f \in \mathbf{B}^2 \mapsto \mathbf{B}$, on trouve :

$$f(x, y) = \mathbf{mux}(x, \mathbf{mux}(y, f(1,1), f(1,0)), \mathbf{mux}(y, f(0,1), f(0,0))).$$

Ceci est suffisant pour montrer que $(0, 1, \mathbf{mux})$ est bien une *base combinatoire*.

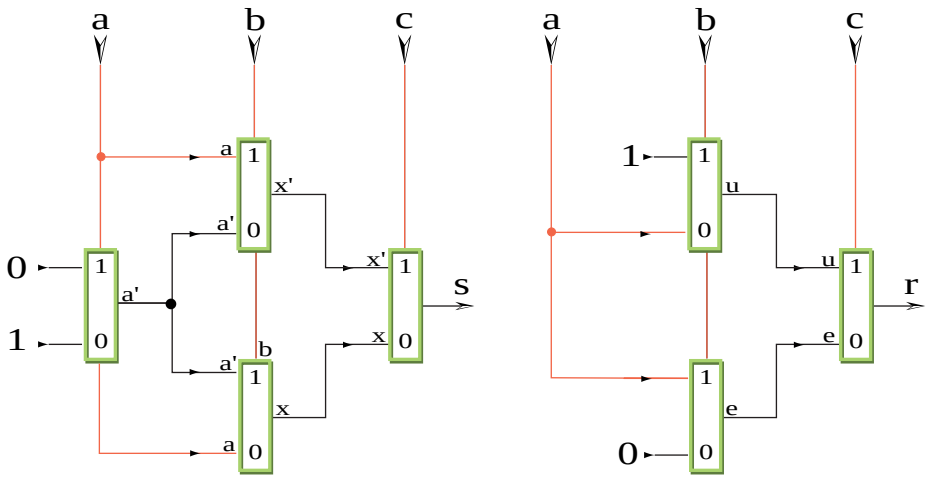


Planche 2.8 – Synthèse BDD de l'additionneur binaire complet

Au passage, la construction nous livre un circuit combinatoire $f \in \mathcal{C}_{ds}$ qui calcule f.

Par exemple, en partant de la table (2.10), la décomposition de Shannon donne le circuit *abc* qui suit - schémas en planche 2.7.

```
abc(a,b,c) = (s,r)                    // Full Adder, formule de Shannon
where
 s=mux(c,mux(b,mux(a,-1,0),mux(a,0,-1)),
         mux(b,mux(a,0,-1),mux(a,-1,0)));        // somme
 r=mux(c,mux(b,mux(a,-1,-1),mux(a,-1,0)),
         mux(b,mux(a,-1,0),mux(a,0,0)));         // retenue
end where;
```

Une fois ces expressions compilées en *équations de base*, on trouve 14 **mux** .

```
abc(a,b,c) = (s,r)                    // Full Adder : 14 équations de base
where
 s1=mux(a,-1,0);                                  // s1 = a
 a'=mux(a,0,-1);                                  // a' = ¬ a
 s3=mux(a,0,-1);                                  // s3 = ¬ a
 s4=mux(a,-1,0);                                  // s4 = a
 x'=mux(b,s1,a');                                 // x' = ¬ (b ⊕ a)
 x=mux(b,s3,s4);                                  // x = b ⊕ a
 s=mux(c,x',x);                                   // s = a ⊕ b ⊕ c
 r1=mux(a,-1,-1);                                 // r1 = 1
 r2=mux(a,-1,0);                                  // r2 = a
 r3=mux(a,-1,0);                                  // r3 = a
 r4=mux(a,0,0);                                   // r4 = 0
```

```
  u=mux(b,r1,r2);                                  // u = a ∪ b
  e=mux(b,r3,r4);                                  // e = a ∩ b
  r=mux(c,u,e);                                    // r = ab ∪ bc ∪ ca
end where;
```

Ce deuxième circuit **abc** $\in \mathcal{C}_{ds}$ est *syntaxiquement équivalent* au premier.

Il ne reste plus qu'à le simplifier, en *partageant* systématiquement les expressions *sémantiquement égales* qui apparaissent dans le code. C'est ici facile, grâce aux *commentaires* que nous donnons à droite des signes // qui suivent chaque équation. Le résultat final ne comporte plus que sept équations de base - schémas en planche 2.8, équations ci-dessous.

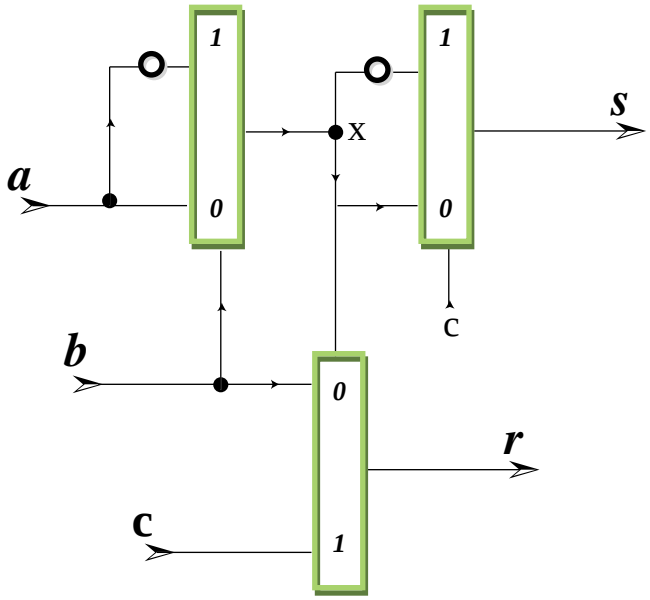
```
abc(a,b,c) = (s,r)                    // Full Adder, synthèse par BDD
where
  a'=mux(a,0,-1);                                  // a' = ¬ a
  x'=mux(b,a,a');                                  // x' = ¬ (a ⊕ b)
  x=mux(b,a',a);                                   // x = a ⊕ b
  s=mux(c,x',x);                                   // s = a ⊕ b ⊕ c
  u=mux(b,-1,a);                                   // u = a ∪ b
  e=mux(b,a,0);                                    // e = a ∩ b
  r=mux(c,u,e);                                    // r = ab ∪ bc ∪ ca
end where;
```

Ce troisième circuit est *sémantiquement* équivalent aux deux précédents, mais pas *syntaxiquement*. Comme **abc** est une fonction symétrique, le BDD construit ne dépend pas de l'ordre des variables d'entrée. Ceci n'est plus vrai dans le cas général. La méthode illustrée par cet exemple est connue sous le sigle BDD, pour *Binary Decision Diagram*.

Algorithme 4 (Synthèse BDD de circuit combinatoire)
Soit $f \in \mathbf{B}^n \to \mathbf{B}^m$ *une application booléenne ; on construit un circuit combinatoire* $BDD(f) \in \mathcal{C}_{ds}$ *qui* calcule f.

1. *Ecrire chaque composante de* f *par la formule de Shannon, et compiler la formule en* $m(2^n - 1)$ *équations* **mux** .
2. *Partager* toutes *les variables sémantiquement égales qui apparaissent dans le processus, et éliminer les équations redondantes.*

La procédure BDD permet de représenter toute fonction sur la base combinatoire $(0, -1, \mathbf{mux})$. Pour passer à la base $(\mathbf{not}, \mathbf{mux})$, il suffit de partager les expressions égales ainsi que leur négation logique, en introduisant alors un inverseur. Par exemple, la synthèse BDD'(abc) - avec négation - donne :




```
abc(a,b,c) = (s,r)          // Full Adder, BDD sur la base (¬, mux)
where
 x=mux(a,not b,b);                                  // x = a ⊕ b
 s=mux(c,not x,x);                                  // s = a ⊕ b ⊕ c
 u=mux(b,b,a);                                      // u = a ∪ b
 e=mux(b,a,a);                                      // e = a ∩ b
 r=mux(c,u,e);                                      // r = ab ∪ bc ∪ ca
end where;
```

Application au décodeur sept segments Utilisons l'algorithme 4 pour synthétiser le décodeur sept segments $Dec7$ de la section 1.4.3 à partir de sa table de vérité :

$b[0]$	0101010101	0	1	0	1	0	1
$b[1]$	0011001100	1	1	0	0	1	1
$b[2]$	0000111100	0	0	1	1	1	1
$b[3]$	0000000011	1	1	1	1	1	1
$s[0]$	1000111011	$*$	$*$	$*$	$*$	$*$	$*$
$s[1]$	0011111011	$*$	$*$	$*$	$*$	$*$	$*$
$s[2]$	1010001010	$*$	$*$	$*$	$*$	$*$	$*$
$s[3]$	1011011011	$*$	$*$	$*$	$*$	$*$	$*$
$s[4]$	1101111111	$*$	$*$	$*$	$*$	$*$	$*$
$s[5]$	1111100111	$*$	$*$	$*$	$*$	$*$	$*$
$s[6]$	1011011111	$*$	$*$	$*$	$*$	$*$	$*$

Cette table est *partielle*, puisque nous ne représentons que les chiffres de 0 à 9. Les astérisques $*$ indiquent des valeurs binaires que nous pouvons ici choisir *arbitrairement*. Ou plutôt, on choisit ces valeurs afin de maximiser le nombre d'expressions communes rencontrées dans la construction BDD, ce qui minimise la taille du circuit synthétisé. A vue, nous prenons :

$b[0]$	0101	0101	0101	0101
$b[1]$	0011	0011	0011	0011
$b[2]$	0000	1111	0000	1111
$b[3]$	0000	0000	1111	1111
$s[0]$	1000	1110	1111	1111
$s[1]$	0011	1110	1111	1111
$s[2]$	1010	0010	1010	0010
$s[3]$	1011	0110	1111	1111
$s[4]$	1101	1111	1111	1111
$s[5]$	1111	1001	1111	1111
$s[6]$	1011	0111	1111	1111

Dans une table de vérité comme celle-ci, les blocs de 2 bits consécutifs correspondent aux fonctions partielles du premier argument $b[0]$, et on trouve ici les 4 configurations possibles, soit : ${}_200 = 0$, ${}_201 = b[0]$, ${}_210 = \overline{b[0]} = \neg b[0]$ et ${}_211 = -1$. Les blocs de 4 bits correspondent aux fonctions partielles de $b[0], b[1]$; on ne trouve ici que 6 configurations, parmi les 16 possibles de la planche 2.6. C'est ce *partage* qui réduit la taille du BDD ; alors que la formule de Shannon contient $7 \times (2^4 - 1) = 105$ équations **mux** , le BDD qui suit n'en a que 22.

```
Dec7(b:[4]) = s:[7]                          // Synthèse BDD
where

                                  // Expressions partagées, sur 1 bits
 n=not b[0];                                  // 210 = ¬b[0]
```

```
                                          // Expressions partagées, sur 2 bits
  d1=mux(b[1],0,n);                       //₂1000 = b[0] ∪ b[1]
  d2=mux(b[1],n,-1);                      //₂1110 = b[0] ∩ b[1]
  d3=mux(b[1],n,0);                       //₂0010 = b[0] ∩ b[1]
  d4=mux(b[1],-1,n);                      //₂1011 = b[0] ⊃ b[1]
  d5=mux(b[1],n,b[0]);                    //₂0110 = b[0] ≡ b[1]
  d6=mux(b[1],b[0],-1);                   //₂1101 = b[1] ⊃ b[1]
  d7=mux(b[1],b[0],n);                    //₂1001 = b[0] ⊕ b[1]
  d8=mux(b[1],-1,b[0]);                   //₂0111 = b[0] ∪ b[1]
                                          // Expressions partagées, sur 3 bits
  t1=mux(b[2],d2,d1);                     //₂1000 1110
  t2=mux(b[2],d2,b[1]);                   //₂0011 1110
  t3=mux(b[2],d5,d4);                     //₂1011 0110
  t4=mux(b[2],-1,d6);                     //₂1101 1111
  t5=mux(b[2],d7,-1);                     //₂1111 1001
  t6=mux(b[2],d8,d4);                     //₂1011 0111
                                                        // Sorties
  s[0]=mux(b[3],-1,t1);                   //₂1000 1110 11111111
  s[1]=mux(b[3],-1,t2);                   //₂0011 1110 11111111
  s[2]=mux(b[2],d3,n);                    //₂1010 0010
  s[3]=mux(b[3],-1,t3);                   //₂1011 0110 11111111
  s[4]=mux(b[3],-1,t4);                   //₂1101 1111 11111111
  s[5]=mux(b[3],-1,t5);                   //₂1111 1001 11111111
  s[6]=mux(b[3],-1,t6);                   //₂1011 0111 11111111
end where;
```

Notons que $Dec7$ n'est pas une fonction symétrique des entrées $b[0] \cdots b[3]$; le lecteur peut vérifier que le nombre de **mux** obtenu à l'issu de la procédure BDD change quand on renverse par exemple l'ordre des variables $b[3] \cdots b[0]$. L'algorithme 4 donne une représentation de toute fonction $f \in \mathbf{B}^n \to \mathbf{B}^m$ par un circuit combinatoire $BDD(f) \in \mathcal{C}_{ds}$ qui est déterminé de façon *unique* - à l'équivalence syntaxique près - une fois choisi l'*ordre* d'écriture des variables d'entrée. C'est la *forme normale BDD* de la fonction f. Passons en revues trois autres types de *formes normales*, plus classiques mais généralement moins utiles en synthèse de logique.

Formes normales disjonctives et conjonctives : DNF, CNF

$$\begin{aligned} s &= a\bar{b}\bar{c} \cup \bar{a}b\bar{c} \cup \bar{a}\bar{b}c \cup abc, \\ r &= ab \cup bc \cup ca. \end{aligned}$$

Forme normale exclusive : ENF

$$\begin{aligned} s &= a \oplus b \oplus c, \\ r &= ab \oplus bc \oplus ca. \end{aligned}$$

La forme normale ENF est plus facile à manipuler que DNF car $(\oplus, \cap)$ est un *anneau*. Ceci nous permet de documenter ci-dessous le circuit de Madre - planche 1.3 - et de prouver qu'il s'agit bien d'un additionneur binaire complet, réalisé avec seulement 5 **mux** - dont on prouve une forme d'optimalité en section 6.4.2.

```
abc(a,b,c) = (s,r)                          // Full Adder de Madre
where
 m=mux(c,b,a);                              // m = cb ⊕ ca ⊕ a
 r=mux(m,b,a);                              // r = ab ⊕ bc ⊕ ca
 d=mux(m,a,b);                              // d = r ⊕ a ⊕ b
 e=mux(r,c,d);                              // e = abc ⊕ a ⊕ b
 s=mux(e,d,c);                              // s = a ⊕ b ⊕ c
end where;
```

2.4 Nombre binaire infini

Les *nombres p-adiques* ${}_p\mathbf{Z}$ - pour p premier - sont introduits vers 1900 par le mathématicien allemand K. Hensel, et ils tiennent depuis une place importante en arithmétique. On les obtient en étendant *indéfiniment* l'écriture usuelle des nombres en base p - par l'algorithme 2 dont on oublie la condition de terminaison $A_k = 0$. Nous considérons ici le cas $p = 2$ des *nombres 2-adiques* ${}_2\mathbf{Z}$, dont les propriétés *arithmétiques* sont (presque) semblables à celles des autres nombres p-adiques, mais dont les propriétés *logiques* sont uniques et appropriées à l'étude des circuits CDS.

Algorithme 5 (Ecriture binaire infinie) *L'écriture 2-adique du nombre a est la suite infinie de* bits $a_n \in \mathbf{B}$ *:*

$$\mathcal{E}_2(a) = {}_2a_0a_1 \cdots a_n \cdots$$

que l'on calcule, pour tout $n \in \mathbf{N}$, comme suit.

1. *Le premier bit - de* poids faible - *s'obtient par $a_0 = a \cdot|\cdot 2$.*
2. *Les chiffres suivants s'obtiennent en calculant $\mathcal{E}_2(a \div 2)$ r écursivement par le même algorithme 5.*

Calculons par exemple l'écriture 2-adique du septième nombre entier :

$$\mathcal{E}_2(7) \Rightarrow {}_21\mathcal{E}_2(3) \Rightarrow {}_211\mathcal{E}_2(1) \Rightarrow {}_2111\mathcal{E}_2(0) \Rightarrow {}_21110\mathcal{E}_2(0) \Rightarrow \cdots$$

A ce point, l'algorithme 5 continue à écrire *indéfiniment* des zéros non significatifs, ce qu'on note, en mettant la partie périodique entre parenthèses, par :

$$7 = {}_2111(0).$$

Ecrivons maintenant un nombre négatif, comme -7 ; ce nombre est impair, et on a $-7 = 1 + 2 \times -4$: $\mathcal{E}_2(-7) \Rightarrow {}_21\mathcal{E}_2(-4) \Rightarrow {}_210\mathcal{E}_2(-2) \Rightarrow {}_2100\mathcal{E}_2(-1) \Rightarrow {}_21001\mathcal{E}_2(-1) \Rightarrow {}_210011\mathcal{E}_2(-1) \Rightarrow \cdots$ Soit, en bref $-7 = {}_2100(1)$ en utilisant la convention de parenthèses pour les écritures périodiques.

Définition 9 (Entier 2-adique ${}_2\mathbf{Z}$) *Un* entier 2-adique $B \in {}_2\mathbf{Z}$ *est la limite commune de trois suites infinies équivalentes*

$$B = \lim_{n\mapsto\infty} B(n) = \lim_{n\mapsto\infty} B[n] = \lim_{n\mapsto\infty} B\{n\}$$

définies respectivement, pour tout $n \in \mathbf{N}$, par :

1. *$B(n) = b(0) \cdots b(n-1)$ est la suite des n premiers bits $b(k) \in \mathbf{B}$ de B ;*
2. *$B[n] = \sum_{0\leq k\leq n} b(k)2^k$ est l'entier naturel $B[n] = B \cdot|\cdot 2^n$ dont $B(n)$ est la représentation binaire ;*

3. $B\{n\} = \{k \in \mathbf{N} \ : \ 0 \leq k < n, b(k) = 1\}$ *est le sous-ensemble des entiers* $\{0, 1, \cdots, n-1\}$ *dont* $B(n)$ *est le* vecteur caractéristique.

On muni l'ensemble **Z**, les entiers relatifs, des opérations *logiques* $\neg, \cap, \cup$. Elles sont définies ci-dessus par **not**, **and**, **or** appliqués bit à bit sur la représentation binaire *infinie* des nombres.

La structure $(\mathbf{Z}, 0, -1, \neg, \cap, \cup)$ est une algèbre de Boole. L'ensemble $\mathbf{N} \mapsto \mathbf{B}$ des suites de bits infinies, muni des mêmes opérations (toujours prises bit à bit) est une algèbre de Boole, isomorphe à celle des parties $\wp(\mathbf{N})$ de l'ensemble **N** des entiers naturels.

Chapitre 3

Circuit électronique

Contents

Partant d'un circuit $C \in \mathcal{C}_{ds}$, l'objectif est d'en compiler les équations $\mathcal{E}(C)$, afin de produire *automatiquement* les plans de fabrication d'un circuit qui réalise électriquement la fonction $[\![C]\!]$ du circuit - définition au chapitre 1.

En théorie, on peut appliquer cette démarche à toute technologie de réalisation : que la machine physique visée soit mécanique, optique, ou autre. On se contente ici d'étudier la synthèse de circuits électroniques pour la technologie cMOS - *complementary Metal Oxide Semiconductor*.

Comme on doit l'appliquer à des circuits de plusieurs millions de transistors (bientôt des *milliards*) il est impératif que la méthode choisie soit *correcte par construction*. Ceci veut dire que tous les circuits *fonctionnels* sortis de l'usine de silicium calculent rigoureusement la même fonction, soit $[\![C]\!]$; ceci modulo une *interface*, qui spécifie la représentation des données de cette application *mathématique*, dans l'espace et dans le temps *physique*.

C'est un *calcul* informatique qui réalise le pont entre la spécification mathématique d'une *fonction*, et sa forme physique dans un circuit MOS. L'entrée du calcul est le système d'équations du circuit, sa sortie est un dessin à l'échelle ; il donne le *négatif* numérique des plans de fabrication, pour les copies électroniques du circuit à réaliser.

Ce calcul est souvent *complexe* à mener en pratique. Pour concevoir un circuit qui tire le meilleur parti de la prochaine génération de technologie - il est quatre fois plus gros et deux fois plus rapide que tous ceux disponibles - on doit utiliser les ordinateurs les plus puissants du moment avec des programmes de CAD (*Computer Aided Design*) qui sont parmi les logiciels les plus complexes du marché, et certainement les plus chers. Malgré cela, il faut typiquement plusieurs heures pour simuler sur ordinateur une micro seconde d'un gros circuit moderne. En 97, il a fallu 13 jours de simulation, avec d'énormes moyens de calcul, pour faire dire *hello world* à l'OS (*operating system*) d'un ordinateur basé sur un microprocesseur qui verra le jour en 99, dans une *fonderie de silicium* qui n'existe pas encore. On la met au point en parallèle avec la conception du futur circuit, pour arriver au plus tôt sur le marché.

En dépit de toutes ces difficultés pratiques, la synthèse informatique de plans de circuits ne contient *pas d'erreur*, du moins en théorie. En pratique, il y a toujours place pour que l'erreur trouve quand même à se glisser, dans un travail aussi complexe. Ainsi, le premier *Penthium* arrive sur le marché avec une erreur dans la division flottante ; pourtant, ni la qualité de fabrication des usines *Intel*, ni leur procédure de test ne sont en cause. La réalisation physique était conforme à la spécification ; celle-ci s'est révélée fausse en 94, et elle est corrigée en trois mois pour disparaître des fabrications ultérieures. Certains estiment à plus de 100 M$ le coût de cette *bogue* d'*Intel*.

Dans le passage du monde *parfait*, mais statique des circuits CDS - où les délais sont nuls et le temps entier - au monde *imparfait*, mais dynamique des charges électriques dans un objet fabriqué en masse, d'autres formes erreurs s'introduisent forcement. Elles résultent des nombreux *bruits* physiques rencontrés dans la vingtaine d'étapes du processus de fabrication. De la poussière au mauvais

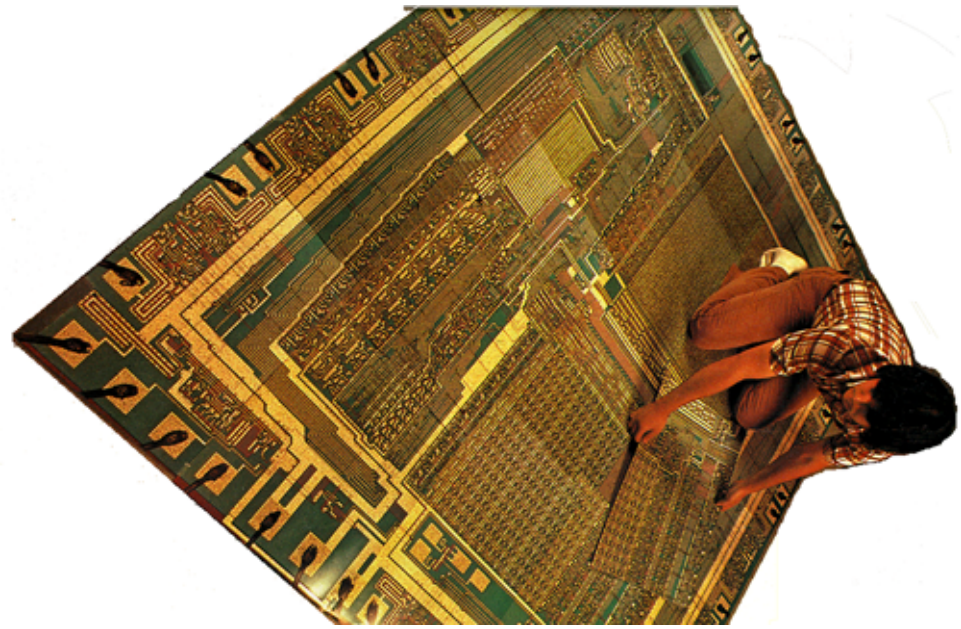
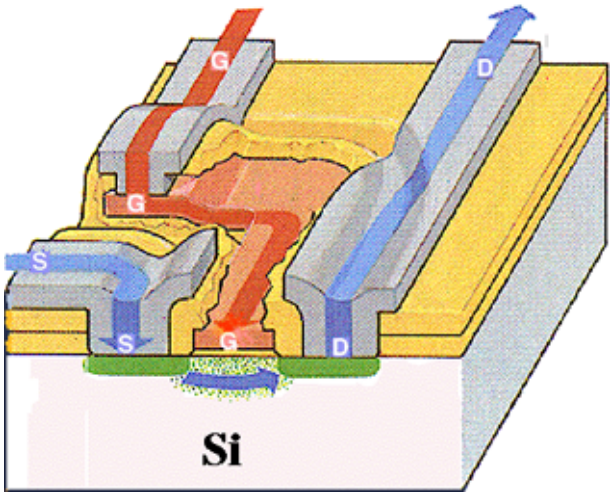


Photo : *National Geographic*
Les plans assemblés ici à une échelle lisible par l'œil humain sont issus de photographies d'une puce concurrente. Des ingénieurs vont les analyser afin de reconstruire la fonction et l'architecture de ce circuit. Ce *reverse engineering* inverse la traduction, de la forme vers la fonction du circuit. C'est une activité parfaitement légale. Le plan à même échelle d'un circuit contemporain passe de 2m×2m en 78, à plus de 200m×200m en 97. C'est dire que l'espionnage industriel de circuits se pratique aujourd'hui sur ordinateur. Ce qui est contraire aux lois sur le *copyright* - comme pour les livres, logiciels et œuvres d'art - c'est d'utiliser les plans ainsi obtenus pour fabriquer soi-même des copies du circuit, et les vendre.

Planche 3.1 – Plan à échelle humaine, 1978

contact en passant par le court-circuit, on trouve toutes sortes d'anomalies dans les hautes technologies. Un circuit sans anomalie est dit *fonctionnel*. Il y va de la survie économique d'une fonderie de silicium d'avoir un bon *rendement* : nombre de circuit fonctionnel par circuit fabriqué. D'où l'importance de la procédure de test en sortie de fabrication, qui vise à séparer les circuits fonctionnels des autres. La nécessité d'un test qui soit exhaustif et rapide impose des contraintes fortes et intéressantes sur la méthodologie de conception des circuits.

3.1 Transistor



Durant la fabrication, la zone dite *canal* située sous la grille G - dans le substrat de silicium - est dopé de charges *négatives* pour un transistor de type n, *positives* pour le type p. En l'absence de champ électrique, la grille est au même potentiel que le substrat ; elle n'est pas conductrice et la source S est isolée du drain D. L'arrivée de charges sur la grille G crée un champ magnétique local, au-delà d'un seuil V_{th} du potentiel de la grille ; ce champ attire les charges opposées du canal vers le voisinage de la surface du substrat, rendant celui-ci conducteur : source et drain sont alors connectés par le transistor qui, en devenant passant, se comporte comme une résistance, dont la valeur est modulable par la tension de grille.

Planche 3.2 – Transistor à effet de champ

Dans la section qui suit, prenons le parti de l'ingénieur, qui veut juste d'abord connaître le *comment*, pas forcement tout le *pourquoi*. Un exposé plus fondamental de cet aspect des circuits - allant plus loin que le pense-bête cMOS donné ici - se trouve dans les cours de physique du solide et d'électronique, voir par exemple [9].

Un circuit intégré est une structure solide, que l'on réalise en couches successives sur un substrat isolant en silicium mono-cristallin. Ces couches servent à réaliser les fils - conducteurs passifs - qui relient les transistors entre eux. Les transistors sont réalisés aussi par le procédé : ils sont tantôt conducteurs, tantôt isolants suivant l'état électronique de leur support dans le cristal de silicium - voir les explications en planche 3.2.

Le canal et ses contacts à la source et au drain sont réalisés directement dans le substrat, en remplaçant sélectivement les atomes de silicium par leurs voisins dans la table de Mendeleev ; l'objectif est de préserver au mieux la structure cristalline, tout en changeant localement la structure électronique d'isolant à semi-conducteur et à conducteur.

En jouant sur les paramètres physiques du transistor de la planche 3.2 (géométrie,

matériaux, tensions et courants) on obtient une immense variété de comportements. Le modèle électrique d'un transistor typique - utilisé par les outils de CAO au niveau analogique - est un réseau complexe d'une vingtaine de résistances, capacités et inductances. Suffisamment complexe en tous cas pour que l'on puisse choisir les paramètres et représenter effectivement *tout* circuit analogique par un réseau équivalent de transistors, jusqu'à capter des photons sur un circuit CCD - planche 7.7. Les équations qui régissent ce monde sont celles de Maxwell, telles qu'on les trouve par exemple chez Feynman [17].

Les avantages en *densité* de calcul de l'analogique sur le digital sont parfois importants ; on consultera à ce sujet le livre de Mead [9], qui présente en conclusion deux circuits analogiques d'envergure - *silicon retina* et *electronic cochlea*. Ces capteurs réalisent une modélisation mathématique de la physiologie de l'œil et de l'oreille humaine par des réseaux électroniques analogiques. On peut bien entendu reproduire de tels circuits au moyen de techniques exclusivement digitales, au prix d'un calcul numérique qui aurait un volume et un débit immense. Bref, lisez Mead [9] pour en savoir plus sur cet aspect du calcul moderne.

Revenons définitivement dans le monde digital. En assemblant convenablement des transistors, et en les reliant par des fils conducteurs, notre objectif est de réaliser tout circuit digital synchrone. Procédons en deux temps. Dans une première passe, nous donnons une idéalisation digitale du comportement d'un transistor. Dans une deuxième passe, nous voyons que cette construction, qui est correcte dans une technologie *idéale*, ne l'est pas dans la technologie cMOS (MOS *complémentaire*) qui domine aujourd'hui, et que nous choisissons d'exposer. Une fois compris et corrigés ces problèmes électriques, nous tenons tous les éléments pour reprendre la compilation des circuits $\mathcal{C}_{ds}$en circuits électroniques corrects par construction.

3.1.1 Transistor digital

Considérons une technologie complémentaire cMOS *idéale* ; elle existe dans le monde des mathématiques, pas dans celui de la physique. Les potentiels V des conducteurs, de 0 Volt à 3 Volt pour la physique, deviennent $0 \in \mathbf{B}$ pour $V \leq 1.5$ Volt, et $1 \in \mathbf{B}$ pour $V > 1.5$ Volt.

Dans ce monde idéal, on trouve deux types de transistors logiques : n et p. Chaque transistor a trois points de contact électrique, dits *grille*, *source* et *drain*. Le fonctionnement logique des transistors n et p est décrit en planche 3.3. Les transistors de type p sont affublés d'un cercle sur la grille : il symbolise l'inversion logique qui échange la fonction des transistors n et p. Enfin, dans notre logique idéale, le transistor commute, de l'état passant à l'état bloqué, et inversement, *en temps zéro* : pouf, instantané !

Bien entendu, les lois de la physique cMOS ne sont pas faites ainsi. Mais, après un peu de travail, on fera en sorte que *tout se passe comme si* ...

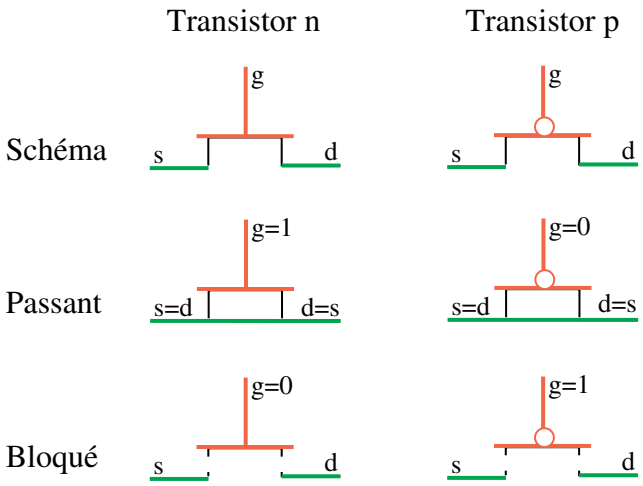


Planche 3.3 – Transistor logique

3.1.2 Transistor analogique

Considérons un transistor $nMOS$ de source s, drain d et grille g. A tout instant, le système se comporte comme une résistance entre s et d, dont on écrit la valeur $R = 1/C$ comme inverse de la *conductance* $C(V_g, V_s, V_d)$, quantité qui a l'avantage ici de rester *finie* quand elle varie avec les tensions de g, s et d sous considération. Elle varie aussi avec les dimensions géométriques du transistor. Fixons celles-ci au transistor *minimal* et considérons d'abord les variations de la conductance avec les tensions de ses trois connecteurs. On peut supposer que $V_{sd} = V_s - V_d > 0$, c'est à dire qu'un courant d'intensité I_{sd} passe de la source au drain. Dans le cas contraire, il suffit d'inverser source et drain, car le transistor est ici parfaitement *symétrique*. Le transistor nMOS fonctionne sous trois régimes.

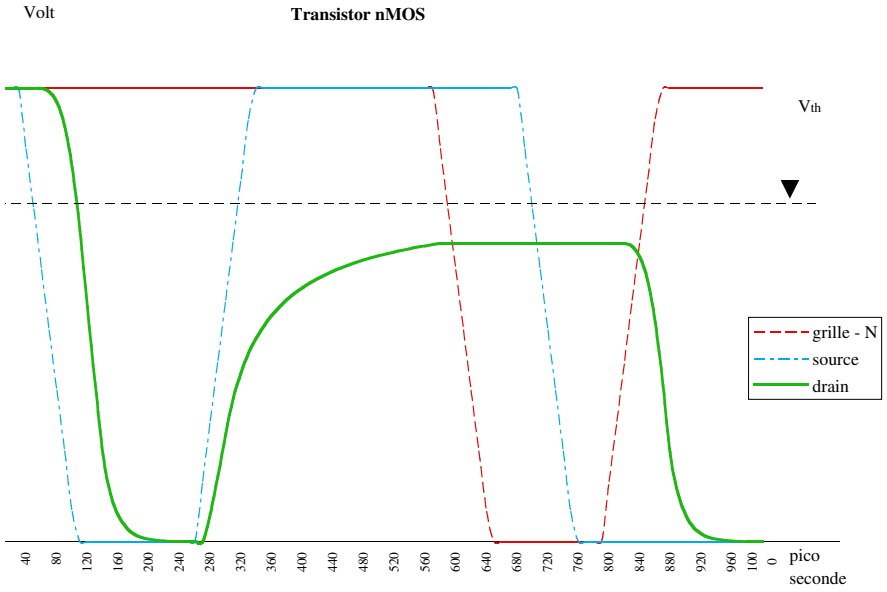
- Quand la tension grille/drain $V_{gd} < V_{th}$ est inférieure à la *tension de seuil* (on a typiquement $V_{th} \approx$ **vdd**$/4$), le transistor est *bloqué* et $I_{sd} = 0$ - le courant ne passe pas.
- Tant que $V_{gd} - V_{th} > V_{sd}$, le transistor est *résistif* - de conductance $C = c_n(V_{gd} - V_{th})$, avec c_n une constante physique - et la loi d'Ohm donne $I_{sd} = C \times V_{sd}$.
- Quand $V_{sd} > V_{gd} - V_{th}$, le transistor est *saturé* : la tension V_s au delà de $V_{gd} - V_{th}$ ne contribue plus rien, et l'intensité du courant est donnée par $I_{sd} = c_n(V_{gd} - V_{th})^2/2$; c'est moitié de la formule précédente dans le cas limite $V_{gd} - V_{th} = V_{sd}$ (voir [9] pour l'explication détaillée).

On peut résumer tous ces cas - et ceux pour $V_{sd} < 0$ - par une formule passe-partout :

$$I_{sd} \quad = \quad c_n \times C \times V \ ;$$

$$\begin{aligned} C &= max\{V_{gd} - V_{th}, V_{gs} - V_{th}, 0\}\ ; \\ V &= sign(V_s - V_d) \times min\{|V_s - V_d|, U/2\}. \end{aligned}$$

Munis de la formule passe-partout et d'un tableur, nous pouvons simuler le comportement dynamique d'un transistor nMOS : on pilote à volonté source et grille pour observer la réponse du drain. Dans cette simulation, le drain du transistor est une simple capacité - typiquement la grille d'un autre transistor - qui se charge et se décharge par le canal.

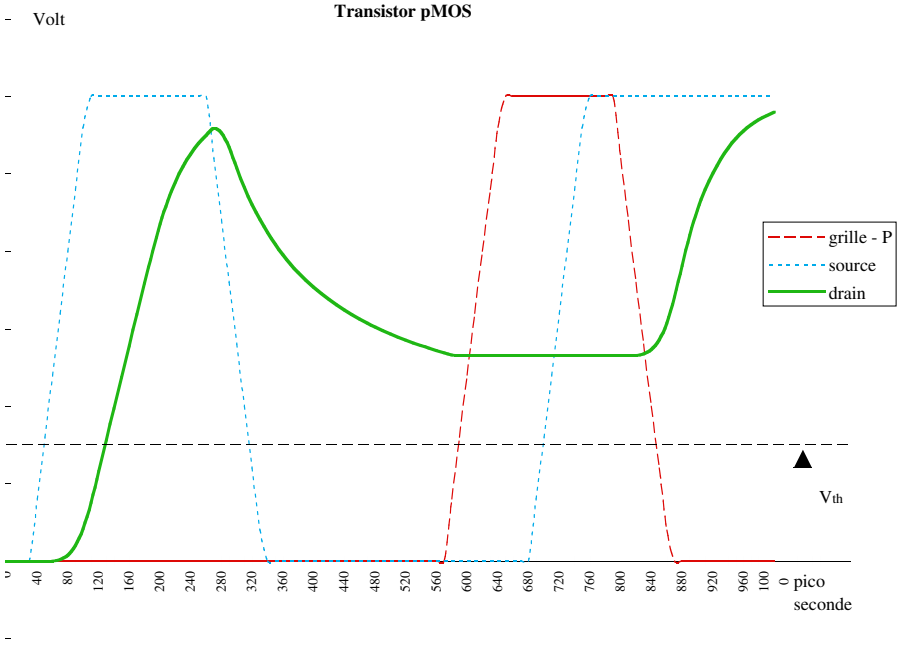


Commentons. Initialement $V_s = V_d = V_g = \mathbf{vdd} = 3$ Volt, et V_s décroît linéairement à partir de 30 ps, pour croiser $V_g - V_{th}$ vers 50 ns. A partir de ce point, le transistor devient résistif, et V_s entraîne V_d dans sa chute. Le premier passe sous 1 Volt à 80 ps, le second à 130 ps. Ce retard de 50 ps de V_d sur V_s à 1 Volt augmente pour atteindre 110 ps à 0.1 Volt. De 260 à 320 ps, la tension V_s remonte de 0 à 3 Volts. Comme la grille est toujours à 3 Volts, le transistor est passant et V_d se doit de suivre V_s : péniblement ! A 1.5 Volt, le retard est de 50 ps ; à 1.8 Volt, il est de 150 ps ; au delà de $\mathbf{vdd} - V_{th}$, il est *infini* : le drain n'atteint jamais cette tension car le transistor se bloque à partir du seuil $\mathbf{vdd} - V_{th} \approx 3\mathbf{vdd}/4$. De 560 à 620 ps, la grille chute de 3 à 0 Volt, ce qui isole maintenant le drain. Le drain *mémorise* alors sa charge ; ceci reste même quand on impose $V_s = 0$ à 740 ps. On voit ensuite V_g remonter et passer par V_{th} à 800 ps, ce qui remet le transistor dans son état résistif. Comme on a toujours $V_s = 0$, la charge stockée précédemment dans V_d s'écoule pour tomber près de zéro vers 1000 ps.

Il s'en passent des choses dans ces quelques microns carrés, en une nano seconde !

Moralité : le transistor nMOS laisse bien passer le 0. Il ampute le 1 d'une tension de seuil ; disons 1- pour ne plus dire 2.25 Volts. Son temps de *descente* est

meilleur que son temps de *montée* : c'est un effet de la tension de seuil, qui limite le potentiel effectif de grille à **vdd** -V_{th}.



La discussion sur le transistor pMOS est *duale*. La formule passe-partout devient ici :

$$\begin{aligned} I_{sd} &= c_p \times C \times V \ ; \\ C &= max\{max\{V_s, V_d, V_g\} - V_g - V_{th}, 0\} \ ; \\ V &= sign(V_s - V_d) \times min\{|V_s - V_d|, U/2\}. \end{aligned}$$

Pour des raisons physiques incontournables, le transistor pMOS est moins conducteur que le nMOS, dans un rapport de l'ordre $c_n \approx \frac{5}{2} c_p$. Contemplons le transistor pMOS en marche, en le stimulant à l'inverse du nMOS précédant.

Le commentaire du chronogramme ci-dessus est dual du précédant. Observons que les pentes du signal pMOS sont moins vives que celles du nMOS ; ceci vient de la plus mauvaise conductivité $c_p < c_n$.

Moralité : le transistor pMOS laisse bien passer le 1. Il augmente le 0 d'une tension de seuil - disons 0+ au lieu de 0.75 Volt. Son temps de *montée* est meilleur que son temps de *descente*. Dans tous les cas, il est plus lent que le transistor nMOS - à taille égale.

3.1.3 Règles électriques

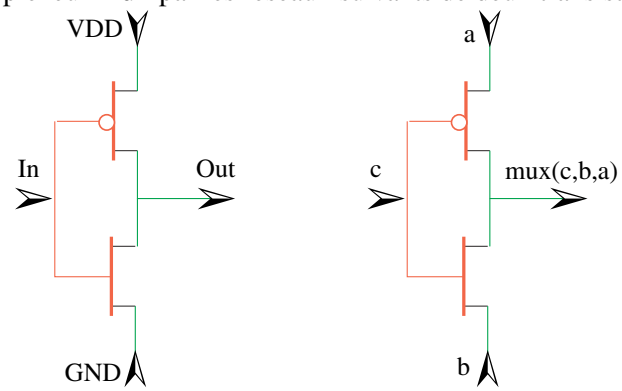
Tirons les conséquences de ces faits, sous forme de *règles électriques* de conception.

1. On peut mettre des transistors nMOS en série : l'entrée à 0 donne une sortie à 0, celle à 1 une sortie à 1- ; comme pour un seul nMOS.
2. On peut mettre des transistors pMOS en série : l'entrée à 1 donne une sortie à 1, celle à 0 une sortie à 0+ ; comme pour un seul pMOS.
3. Ne jamais mettre en série un nMOS avec un pMOS : le signal sortant (mauvais 0+ et mauvais 1-) n'est pas exploitable pour nous.
4. Un 1- peut servir de grille nMOS, pourvu que la source soit connectée à 0.
5. Un 0+ peut servir de grille pMOS, pourvu que la source soit connectée à 1.

Ces règles électriques doivent être respectées tant qu'on ne comprend pas quand et comment on peut les violer : simulation électronique avec les *vrais* paramètres physiques à l'appui - au risque de longues nuits de *déboguage*.

3.1.4 Multiplexeur et inverseur

Pour comprendre les conséquences des règles électriques, réalisons l'inverseur **not** et le multiplexeur **mux** par les réseaux suivants de deux transistors chacun.

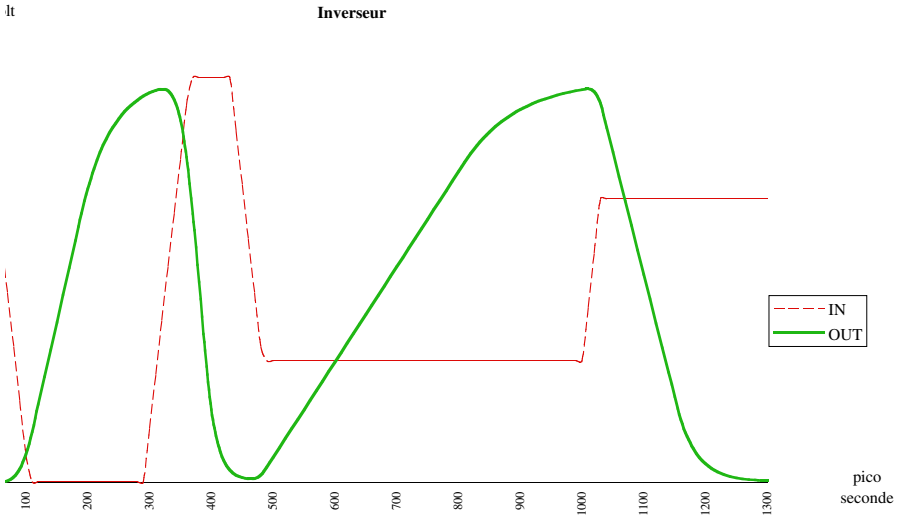


Inverseur Multiplexeur

Les deux sont corrects logiquement. L'inverseur est électriquement correct. Cette forme du multiplexeur ne l'est pas, sauf quand a=**vdd** et b=**gnd** , ce qui nous ramène à l'inverseur.

Inverseur

Contemplons d'abord une simulation de l'inverseur.

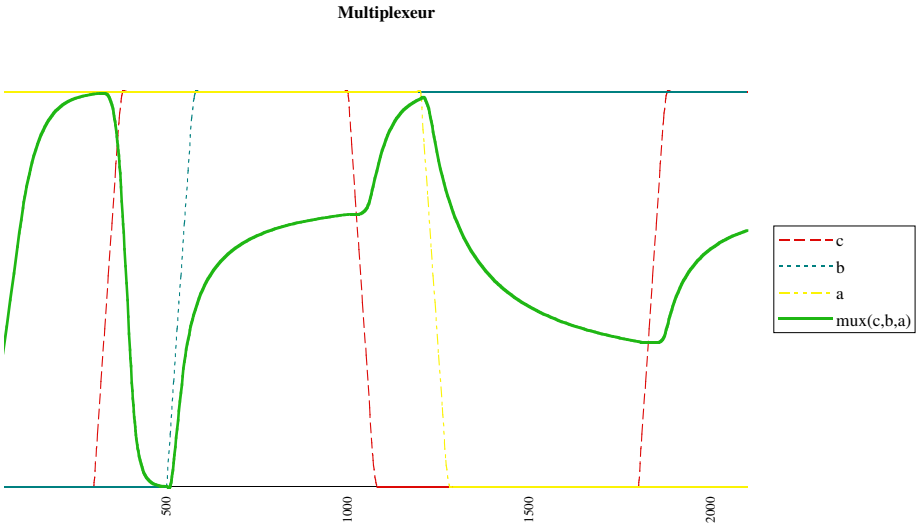


La première descente puis remontée de IN montre que OUT suit en inverse, dès la tension de seuil passée, avec un délai que nous mesurerons plus tard. Observons que les fronts descendants sont plus raides que les fronts montants ; ceci vient encore du rapport des conductances c_n/c_p.

La suite de la simulation montre qu'un inverseur peut servir d'amplificateur - dit alors *sense-amp* - auquel on présente 0+ en entrée pour avoir 1 en sortie, et 1- pour obtenir 0. C'est en contradiction avec la règle électrique 6 ! On le paye de deux façons. Les temps de montée et de descente sont environ 4 fois ceux du mode normal. Ensuite, il y a *dissipation statique* d'énergie - chose mal vue en cMOS - car aucun des deux transistors n'est jamais (complètement) bloqué dans ce fonctionnement. Le drain finit par être convenablement chargé, mais il s'établit en même temps un courant de fuite permanent entre **vdd** et **gnd** . C'est acceptable à petite échelle, par exemple dans les décodeurs des mémoires ; probablement pas dans des structures massivement répétitives dont on risque de voir fondre les fils d'alimentation, par la migration d'atomes dans le métal, entraînés par la violence du courant des électrons.

Multiplexeur

Passons à une simulation du multiplexeur $m = \textbf{mux}(c, b, a)$, et commentons.

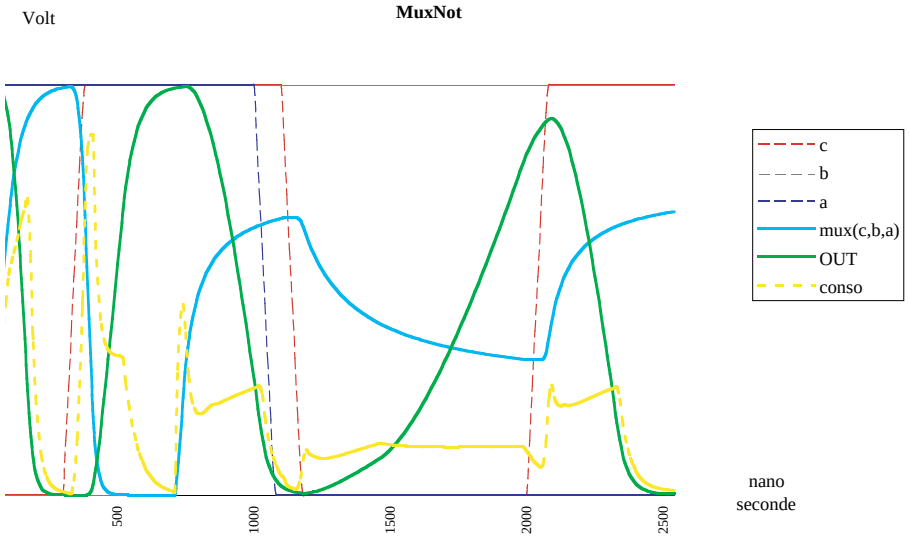


Jusqu'à 500 ps, rien de nouveau : comme $a =$ **vdd** et $b = 0$, le **mux** est un simple inverseur $m = \neg c$. Quand b monte à partir de 500 ps, on voit que m tente de suivre, jusqu'à buter sur la tension de seuil du nMOS - qu'il n'atteint jamais. Comme à ce point $a = b =$ **vdd**, il suffit de faire plonger c vers 0 pour voir monter m à 110 ps, happé vers **vdd** par le pMOS redevenu passant. Maintenant, comme $a = 0$ et $b =$ **vdd**, le **mux** est logiquement un amplificateur $m = c$. Il est mauvais électriquement, car son excursion de sortie se limite de V_{th} à **vdd** -V_{th} ; de plus, il est pratiquement dix fois plus lent que dans le mode inverseur du début.

Conclusion : cette version du multiplexeur en deux transistors ne marche pas ; il faut trouver autre chose. Donnons deux solutions, et comparons.

Multiplexeur Inverseur Plaçons un inverseur - servant de *sense-amp* - en série après le **mux** . Ceci fait un **mux** inverseur **MuxNot** - à lui seul une *base* de l'algèbre de Boole - en 4 transistors.

Son comportement est simulé ci-dessous. Il apparaît que **MuxNot** devient lent dès que l'inverseur passe en mode *sense-amp* : remontée en 600 ps, contre 120 ps en mode normal. De plus, il consomme environ huit fois plus que la solution qui suit - la consommation est indiquée sur ces deux simulations. Pire : il y a un résidu non négligeable de consommation statique. Bref ! La règle 6 nous dit encore de ne pas adopter cette solution.



Multiplexeur cMOS Une bonne réalisation électrique du multiplexeur **mux** utilise deux portes de *transmission cMOS*, avec codage en *rail* de c - où $c' = \neg c$ est explicitement calculé, se trouve dans les schémas de la planche 3.4. Ceci donne un **mux** avec 6 transistors, en comptant les deux qui se cachent dans l'inverseur $c' = \mathbf{not}(c)$. Pour éviter de confondre le cercle de la grille pMOS et celui de l'inverseur dans les schémas, on préfixe le cercle de l'inverseur d'un triangle, qui symbolise un amplificateur électrique dont la fonction logique est l'identité.

Cette porte possède de bonnes caractéristiques électriques, comme le montre la simulation de la planche 3.5. Remarquons qu'elle est *passive*, c'est à dire que les courants induits en sortie doivent venir des entrées au travers de transistors passants. C'est différent d'une porte *régénératrice* comme l'inverseur, dont les courants de sorties viennent des alimentations, sous contrôle des entrées, mais sans échange de charge avec celles-ci. Pour transformer une porte passive en porte régénératrice, il suffit de mettre un amplificateur (soit deux inverseurs) en série sur la sortie.

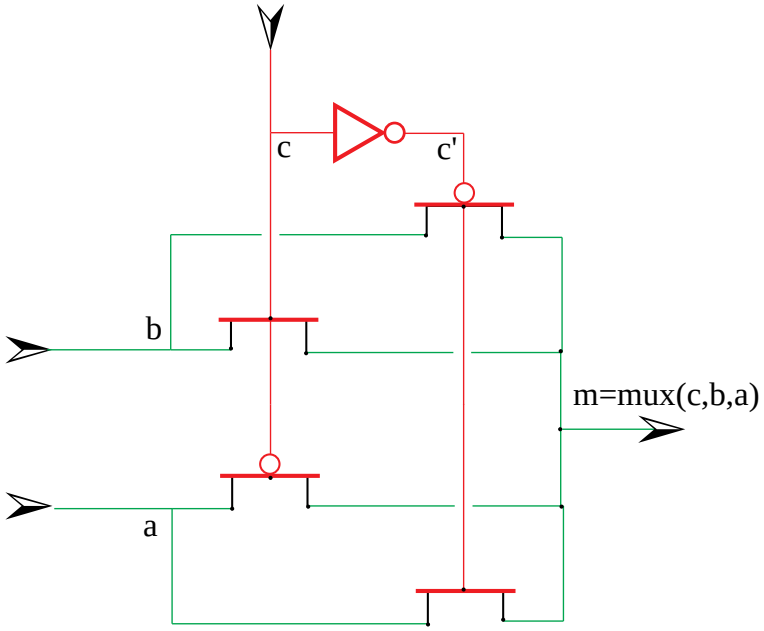


Planche 3.4 – Schémas du multiplexeur cMOS

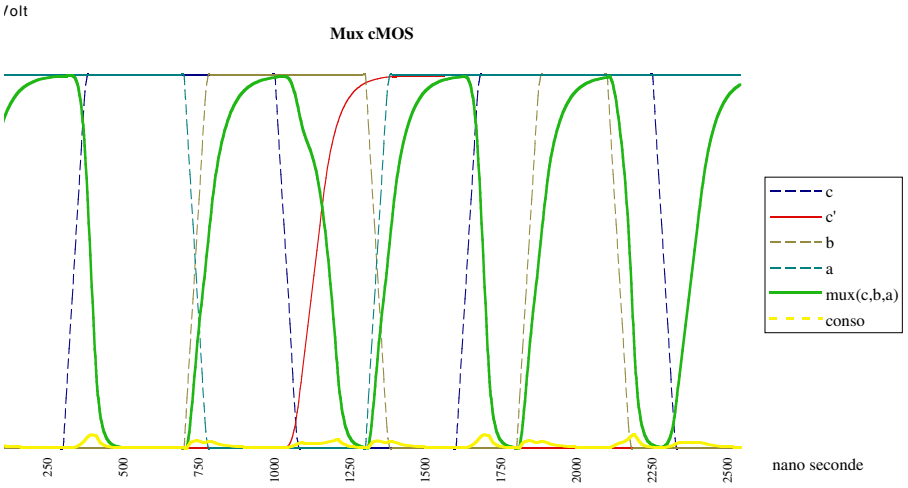


Planche 3.5 – Simulation du multiplexeur cMOS

Multiplexeur à quatre voies On peut combiner les multiplexeurs pour forme une LUT2 - *look-up table* 2 bits, dite aussi *multiplexeur à quatre voies*, comme dans les schémas de la planche 3.6 qui comprennent 3 multiplexeurs et seulement 16 transistors, car deux inverseurs sont partagés.

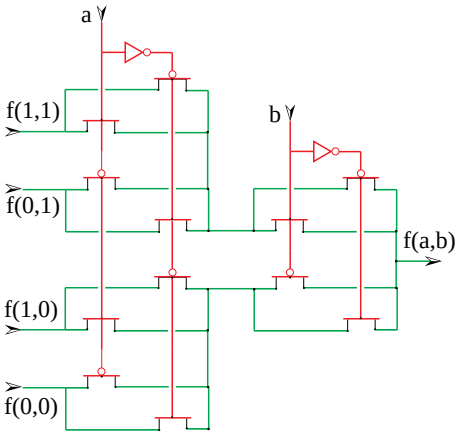
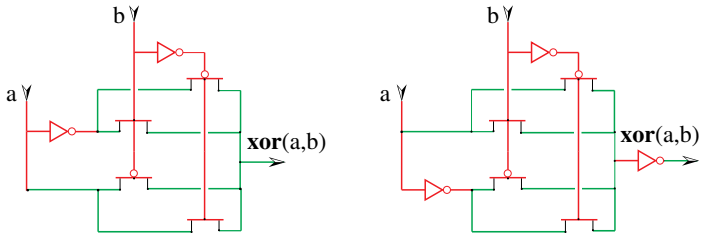


Planche 3.6 – Multiplexeur à quatre voies

Portes à deux entrées En partant de LUT2, et en simplifiant les expressions communes - technique BDD du chapitre 2 - on réalise toute porte combinatoire à deux entrées ; en particulier **xor** se réalise avec 8 transistors - voir la planche 3.7.

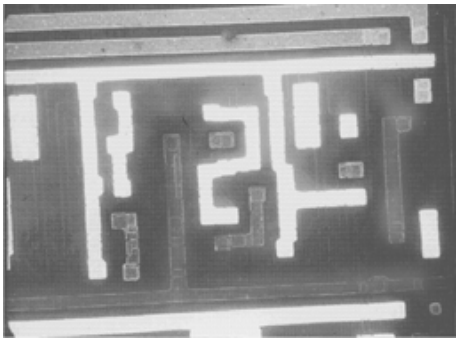


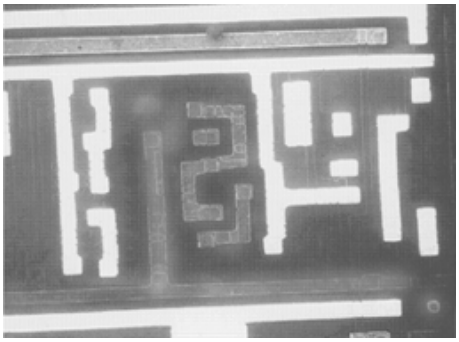
La sortie de la version de gauche est une porte de transmission : pour $b = 1$, c'est une simple résistance entre l'entrée a et la sortie. Dans la version de droite, les charges de sortie viennent des alimentations de l'inverseur, et sont donc découplées des entrées. La porte de gauche est dite *active* ; celle de gauche *passive*.

Planche 3.7 – Ou exclusif

3.1.5 Temps digital et registre synchrone

Envisageons d'abord la question du temps digital et celle du registre dans le monde mathématique - indépendamment de toute technologie de réalisation - puis, ensuite, dans celui de l'électronique cMOS.





La technique de *microscopie par balayage électronique* (MBE) permet de produire des images de circuit en fonctionnement faisant apparaître les équipotentielles à 1 en brillant, celles à 0 en mat. Ces deux images montrent le comportement d'un détail du même circuit lors de deux phases consécutives de l'horloge.

Planche 3.8 – Circuit en fonctionnement

Registre et temps discret

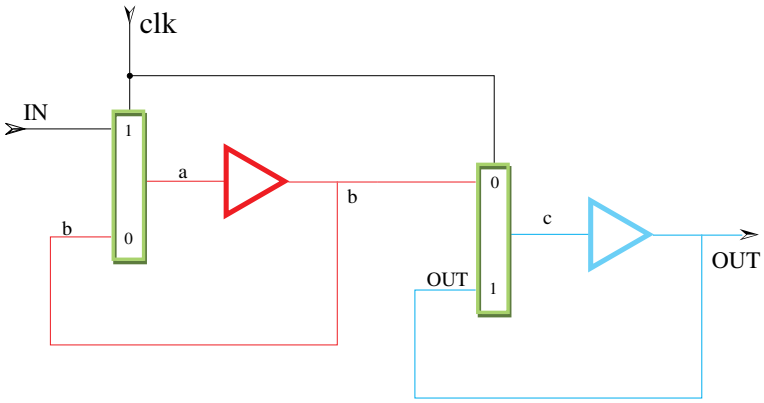


Planche 3.9 – Schéma de principe du registre

On obtient un *registre* 1 bit en mettant en série deux *points mémoire* - dit *latch* - identiques sauf pour la polarité des multiplexeurs qui sont inverses, comme le montre la planche 3.9.

Quand l'horloge clk vaut 0, le premier *latch* est isolé de son entrée IN, et il *mémorise* la valeur logique de b en la faisant circuler dans une *boucle combinatoire*

qui comporte un amplificateur. Contrairement à la boucle bad du chapitre 1 - qui comporte un inverseur - la boucle du latch est *stable*, dès que la valeur de b a été proprement mise en place, soit 0, soit 1. La fonction de l'amplificateur n'est pas logique, mais électrique : il permet de garder indéfiniment la valeur mémorisée, même en présence de perte de charges par le substrat. Toujours avec $clk = 0$, le deuxième *latch* est *transparent* : sa sortie OUT est alors égale à son entrée b, c'est à dire le bit qui est mémorisé dans le premier *latch*.

Quand l'horloge clk passe à 1, le premier *latch* devient *transparent* : sa sortie b est alors égale à son entrée IN. Le deuxième *latch*, quant à lui, mémorise et maintient la valeur OUT acquise dans son dernier mode transparent, juste avant que l'horloge ne monte.

L'horloge clk est un signal dont les valeurs sont alternativement 0 et 1. Chaque transition de 0 vers 1 marque le début d'une nouvelle phase d'horloge : passage du temps digital $t \in \mathbf{N}$ à $t + 1$. La sortie du registre $OUT(t + 1) = IN(t)$ devient alors égale à celle de son entrée lors du cycle précédent.

Dans un circuit mathématique à délais nuls, la phase haute de l'horloge - temps pendant lequel $clk = 1$ - peut être arbitrairement court. A la limite, le *top* d'horloge au temps $t = n$ est donc un *Dirac* [1] $\delta(t - n)$. On peut alors représenter l'*horloge mathématique* par la formule :

$$clk(t) \quad = \quad \sum_{n \in \mathbf{N}} \delta(t - n).$$

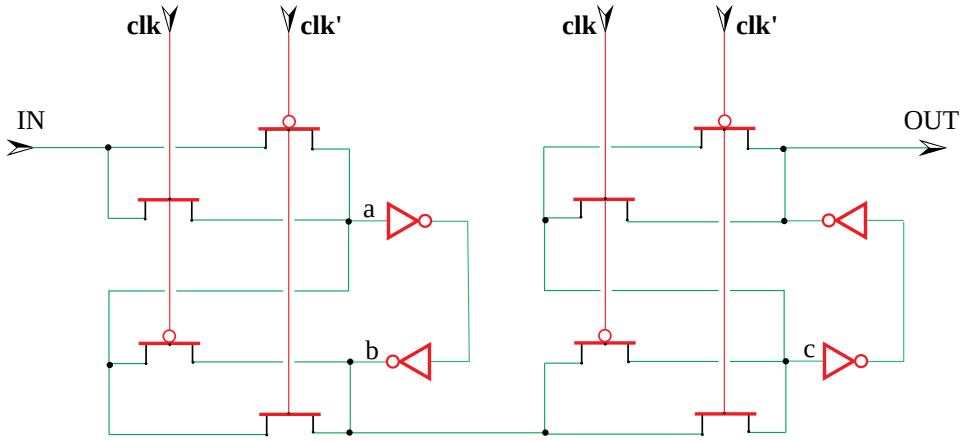


Planche 3.10 – Registre en 16 transistors

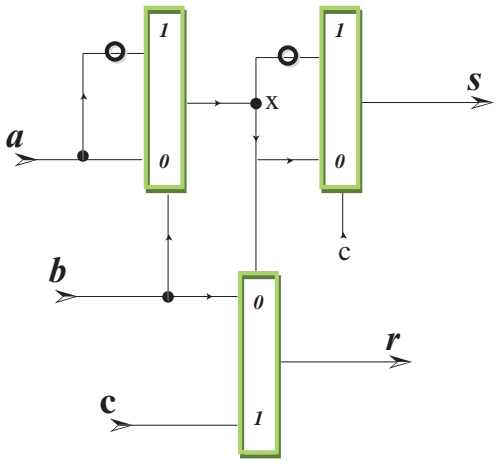
1. La fonction $\delta(x)$ de Dirac vaut $\delta(x) = 0$ pour $x \neq 0$; elle est infinie $\delta(0) = \infty$ pour $x = 0$, mais elle garde une intégrale finie (au sens des distributions) : $\int_{-\infty}^{+\infty} \delta(x)\, dx = 1$.

Registre électronique

Partons des schémas de principe de la planche 3.9. En combinant deux **mux** cMOS et en réalisant chaque amplificateur par deux inverseurs en série, on obtient un registre binaire qui comprend à priori 20 transistors cMOS. Comme l'horloge est distribuée sur deux rails - clk et clk' - on n'a besoin en fait que des 16 transistors de la planche 3.10.

3.2 Conception et réalisation d'un circuit

A ce point, nous savons réaliser toutes les portes de la base $(\mathbf{not}, \mathbf{mux}, \mathbf{reg})$. Ceci suffit pour compiler un circuit CDS arbitraire en sa réalisation électronique cMOS. Cette traduction part d'une description d'un circuit C - en ${}_2\mathbf{Z}$ ou dans tout autre langage - et produit une description de $C \in \mathcal{C}_{ds}$ sous forme d'un ensemble $\mathcal{E}(C)$ d'équations dans la base $(\mathbf{not}, \mathbf{mux}, \mathbf{reg})$ - voir chapitre 1. Notre propos - pour le chapitre présent - commence une fois connues les équations de base du circuit à réaliser.



Pour illustrer le processus, nous utilisons l'additionneur binaire complet abc ci-dessus, donné par cinq équations de base.

```
abc(a,b,c)   =   (s,r)                    // Full Adder en 5 portes
where
 b'= not(b);                              //b' = ¬ b
 x  = mux(a,b',b);                        //x  = a ⊕ b
 r  = mux(x,  c,  b);                     //r  = ab ⊕ bc ⊕ ca
 c'= not(c);                              //c' = ¬ c
 s  = mux(x,c',c);                        //s  = a ⊕ b ⊕ c
end where;
```

3.2.1 De la logique au plan des masques

Schémas électriques

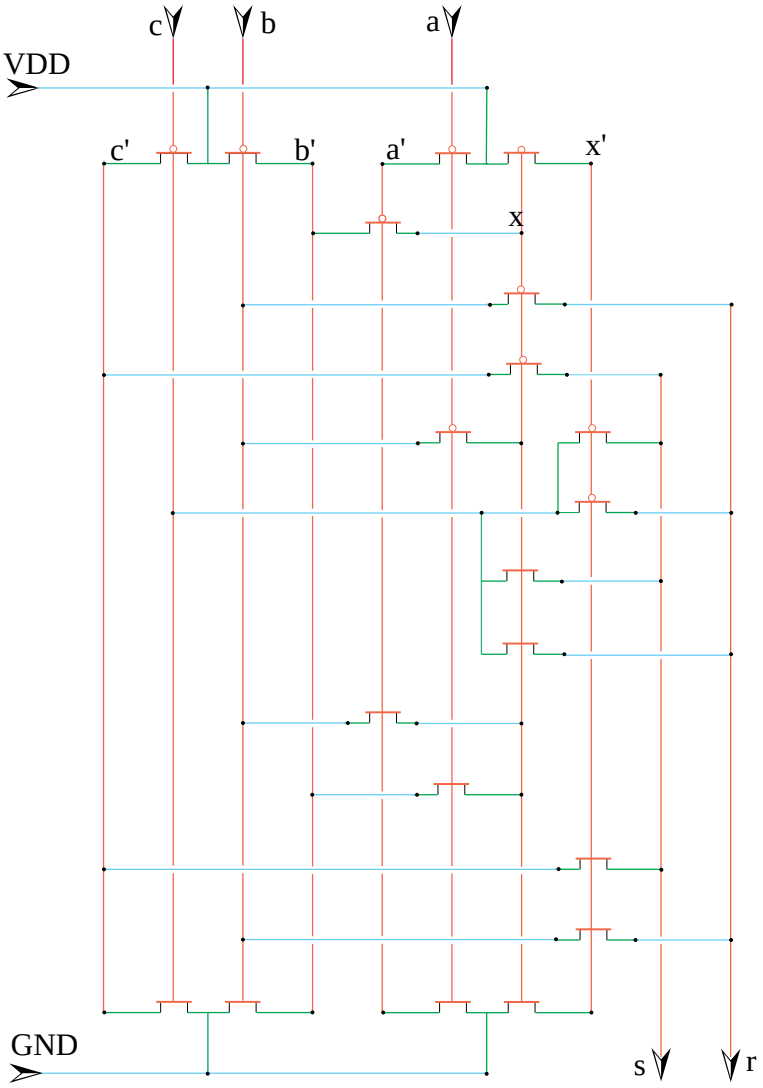
Partant des équations $\mathcal{E}$ du circuit C, on compile une liste des transistors et équipotencielles du circuit - dite *netlist* - en associant à chaque équation les composants des schémas associés dans la section précédente aux primitives (**not**, **mux**, **reg**).

Dans le cas de notre circuit type abc, on a 2 transistors par **not** , et 6 par **mux** , soient 22 transistors à priori. Comme x est grille commune de deux **mux** , on peut partager un inverseur et réduire le compte à 20 transistors. La *netlist* de abc prend alors la forme suivante.

```
abc(a,b,c)   =   (s,r)                    // Full Adder en 20 transistors
where
 transN(b,GND,b');                        // b' = ¬ b
 transP(b,VDD,b');                        // b' = ¬ b

 transN(a,GND,a');                        // a' = ¬ a
 transP(a,VDD,a');                        // a' = ¬ a

 transN(c,GND,c');                        // c' = ¬ c
 transP(c,VDD,c');                        // c' = ¬ c

 transN(x,GND,x');                        // x' = ¬ x
 transP(x,VDD,x');                        // x' = ¬ x

 transN(a,b',x);                     // x = mux (a,b',b)
 transP(a',b',x);                    // x = mux (a,b',b)
 transN(a',b,x);                     // x = mux (a,b',b)
 transP(a,b,x);                      // x = mux (a,b',b)

 transN(x,c,r);                       // r = mux (x,c,b)
 transP(x',c,r);                      // r = mux (x,c,b)
 transN(x',b,r);                      // r = mux (x,c,b)
 transP(x,b,r);                       // r = mux (x,c,b)

 transN(x,c,s);                      // s = mux (x,c',c)
 transP(x',c,s);                     // s = mux (x,c',c)
 transN(x',c',s);                    // s = mux (x,c',c)
 transP(x,c',s);                     // s = mux (x,c',c)
end where;
```

Placement et routage

Partant de la *netlist*, l'étape suivante est de placer les transistors et de router les fils qui constituent chaque équipotentielle. Chaque variable de la *netlist* devient alors un dessin en deux dimensions. Le placement/routage automatique est l'une des composantes importantes des logiciels de CAD. Un placement et routage possible pour notre abc est le suivant.



Ce placement de abc est généré automatiquement. Il n'est pas *optimal* : sa densité de transistors est faible. Un dessinateur compétent peut, en quelques jours, réduire la surface de moitié. Ceci en vaut la peine pour un circuit que l'on compte tirer à un très grand nombre d'exemplaires. Il ne faut pas le faire quand l'objectif est de mettre ce circuit au plus vite sur le marché.

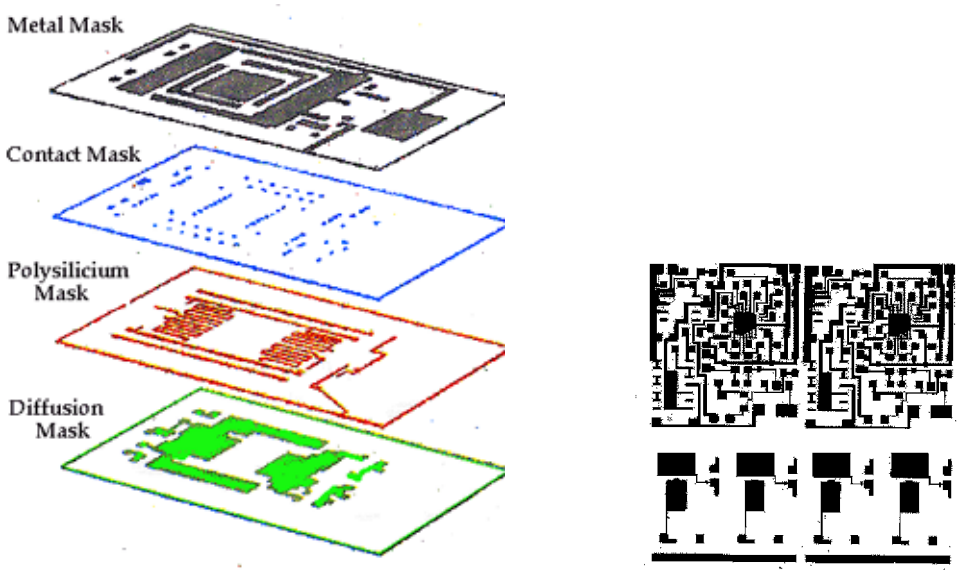


Planche 3.11 – Masques de fabrication

Simulation et optimisation

Ce n'est qu'après le placement et le routage que les paramètres nécessaires à la simulation électrique sont connus. En effet chaque *net* - c'est à dire chaque variable de la *netlist*, c'est à dire chaque équipotentielle du circuit final - est une capacité, composée en partie des transistors dont il est la grille et, pour le reste, des fils de routage qui le composent.

Une fois mesurés ces paramètres électriques, on peut procéder à une simulation réaliste du comportement dynamique du circuit. Le modèle de transistor donné par les formules passe-partout n'est qu'une approximation au premier ordre du comportement, dans laquelle on ignore divers effets parasites qui dégradent les performances - au second ordre. Dans la simulation d'un circuit électronique critique, comme par exemple les circuits analogiques qui suivent, on modélise le transistor par un réseau qui comprend plusieurs dizaines de composants électriques primitifs, à prendre parmi : résistances, capacités, inductances [2], sources de courants, sources de tensions.

L'objet de la simulation est de vérifier les performances a ttendues du circuit - vitesse, consommation - et de les optimiser en jouant sur les dimensions des transistors et les autres paramètres. En effet, quand on double la longueur du canal d'un transistor, on divise sa conductance par deux. A l'inverse, quand on double sa largeur, on multiplie la conductance par deux. Dans les deux cas, on double la capacité de sa grille, ce qui reporte le problème d'optimisation un cran plus haut dans la chaîne logique.

2. rares dans les circuits MOS

Ici encore, les outils de CAD donnent une solution automatique aux dimensions des transistors. Elle est fiable et on l'obtient vite. Elle n'est jamais optimale et le concepteur humain peut l'améliorer, s'il en a le temps.

Dessin des masques

Une fois terminée la conception électrique et géométrique du circuit, il faut en tirer les plans. C'est un dessin informatique à l'échelle de *tous* les fils, contacts et transistors constituant le circuit. A ce stade, chaque variable logique prend une forme géométrique (dessin complet de l'équipotentielle électrique) et des couleurs, qui codent les matériaux de réalisation (planches 3.11, 3.12 et 3.8). Les plans d'un circuit de 100 000 transistors comportent typiquement plusieurs millions de polygones, représentés par quelques centaines de Mb.

Avant de lancer la fabrication, procédé long et coûteux, on procède à de multiples vérifications des plans.

Règles de dessin On vérifie que le dessin de chaque équipotentielle est électriquement connexe, et que les fils qui la composent ont la largeur requise. On vérifie que deux équipotentielles différentes sont espacées d'une garde minimale, pour éviter les courts-circuits. Ces règles sont toutes exprimées en *microns* ($1\mu m = 10^{-6}m$).

Schémas logiques La structure en portes logiques du circuit est reconstruite à partir du dessin des plans, et on vérifie qu'elle est bien isomorphe à la *netlist* de départ. C'est à une activité de même nature que se livre le personnage de la planche 3.1.

Schémas électriques Les caractéristiques électriques - résistance, capacité, ...- de chaque *net* sont calculées à partir du dessin, et des caractéristiques de la technologie de réalisation. On vérifie alors, par des simulations aussi exhaustives que possible, que les performances du circuit (surface, vitesse, consommation) sont bien conformes au cahier des charges.

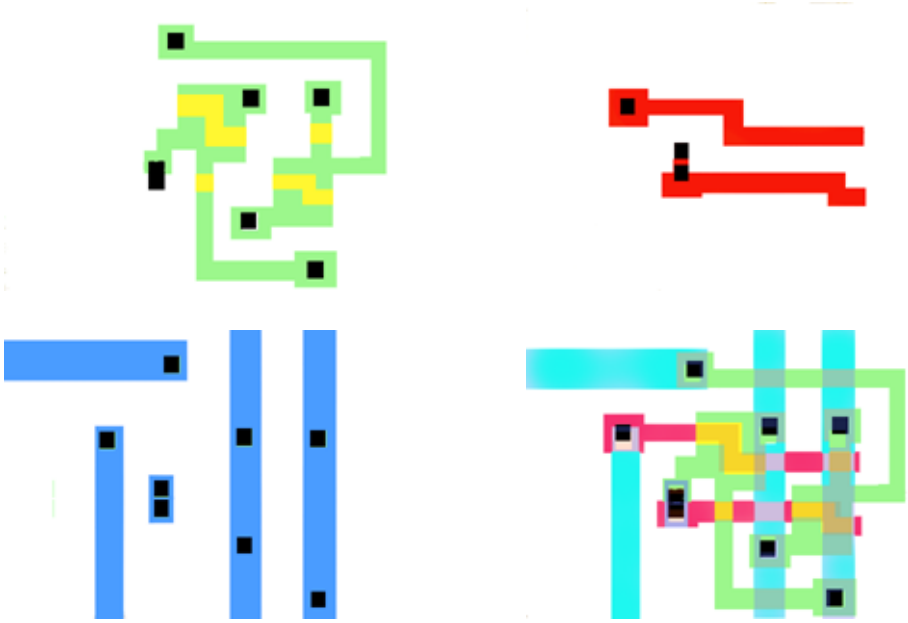
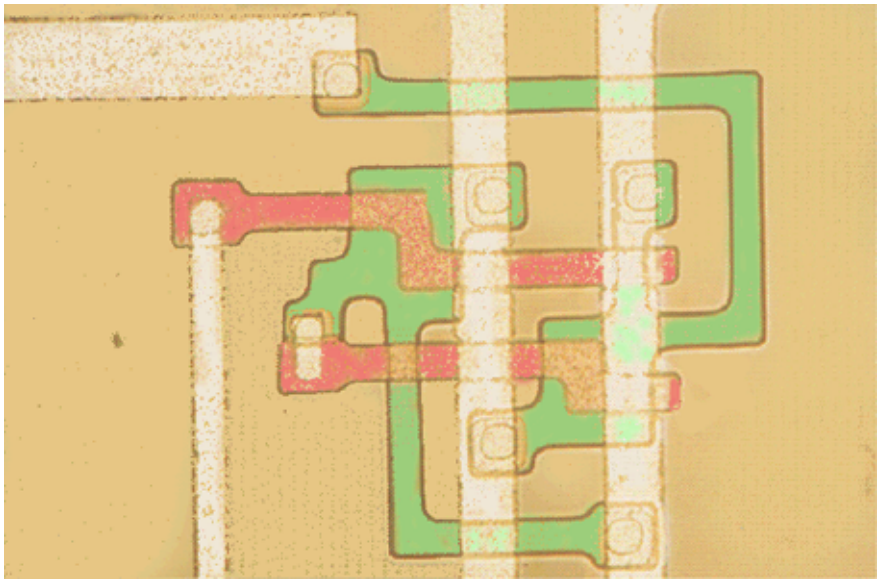


Planche 3.12 – Trois masques et les plans d'une porte



Microphotographie d'une porte, après réalisation d'après les plans de la planche 3.12. Les (fausses) couleurs sont obtenues par un jeu de filtres polarisants.

Planche 3.13 – Porte réalisée

3.2.2 Du silicium au circuit testé

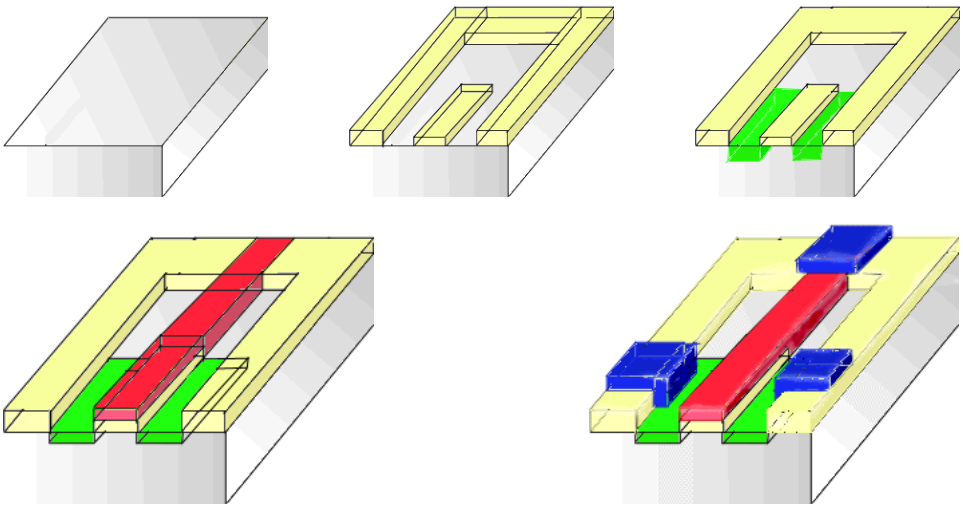


Planche 3.14 – Fabrication d'un transistor

Fonderie de silicium

Toutes vérifications faites, on sépare les plans suivant les diverses couches de matériau constitutif, typiquement de l'ordre d'une dizaine. Le dessin de chaque couche est alors gravé sur du verre, pour constituer le jeu des *masques*, qui est au circuit ce que le *négatif* est à la photographie - planches 3.11, et 3.12.

Le matériau de base des circuits MOS est le silicium mono-cristallin, étiré en lingots puis découpé en galettes - *wafers* - rigoureusement planes et orthogonales à la structure du cristal - planche 3.15. Ces galettes subissent alors une suite de traitements physico-chimiques visant à matérialiser sur le silicium les diverses couches de la structure du circuit défini par le jeu des masques - planches 3.16 et 3.14.

Test et montage

A la fin de ce processus, on a gravé de multiples copies du circuit sur les galettes traitées (planche 3.18). On descend alors des pointes conductrices sur les *pattes* (planche 3.22) de connexion de chaque *puce*, et on injecte des configurations de test visant à séparer les circuits opérationnels des rejets (planche 3.20). On scie alors les galettes et chaque puce réputée bonne est montée sur un support boîtier (planches 3.21 et 3.22). On procède à une dernière série de tests de comportements, et les circuits qui survivent à cette épreuve sont livrés à l'utilisateur.

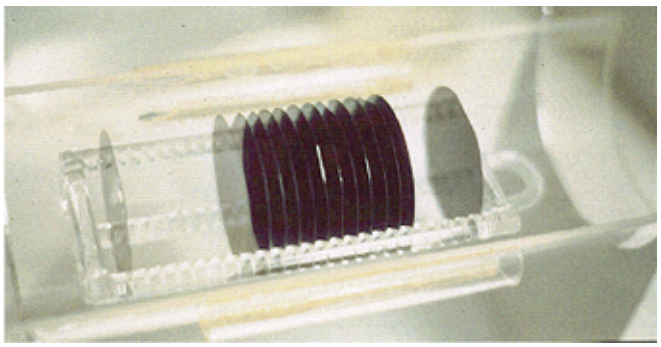
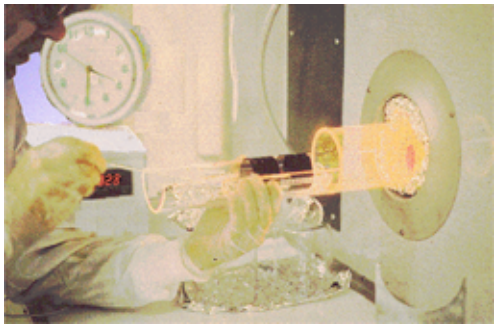


Planche 3.15 – Lingot de silicium mono-cristallin, découpé en galettes



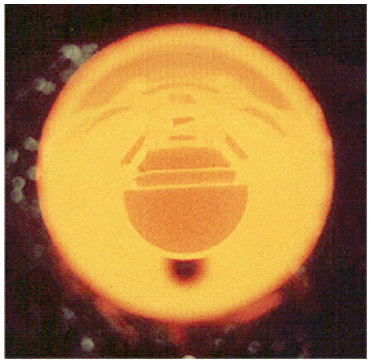
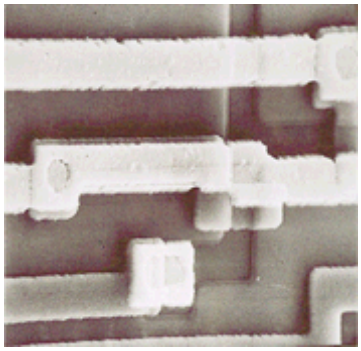


Planche 3.16 – Mise au four des galettes



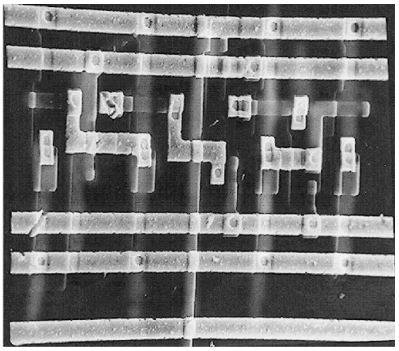


Planche 3.17 – Détails au microscope électronique

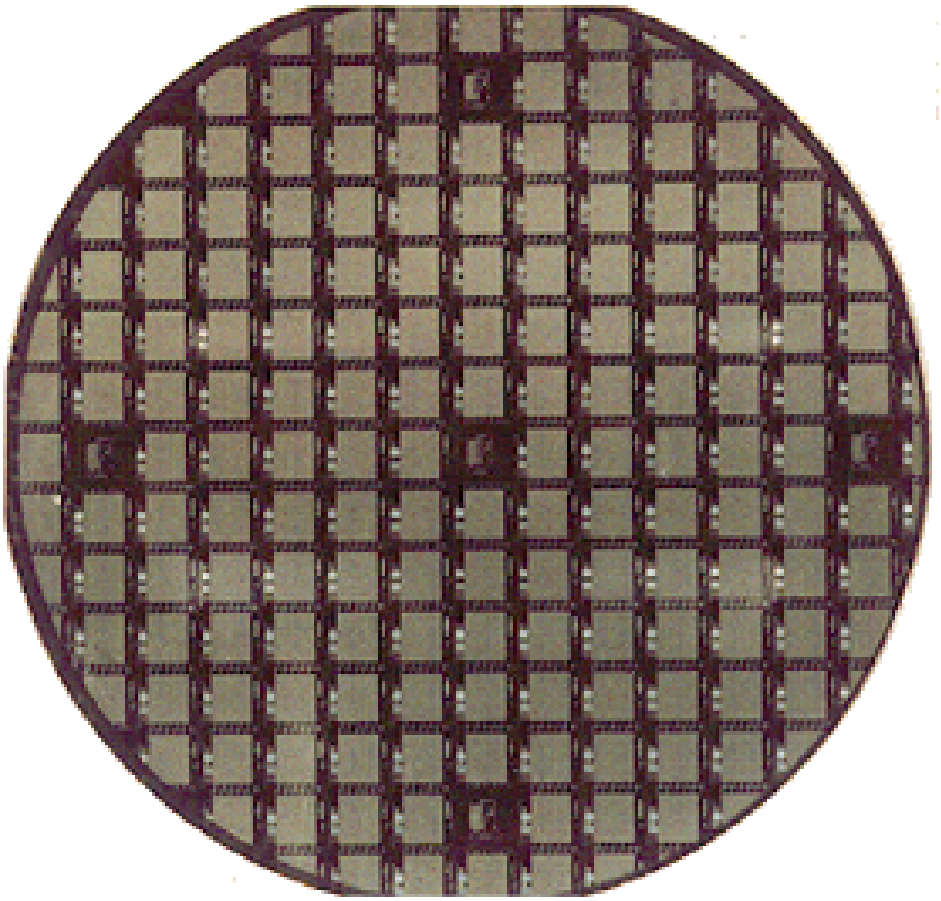


Planche 3.18 – Circuits sur tranche

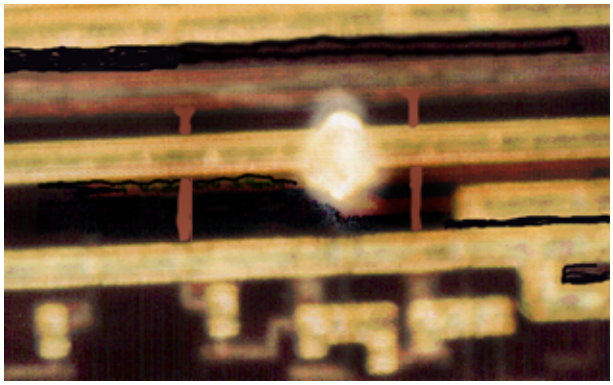


Planche 3.19 – Poussière de 6 μm sur un circuit

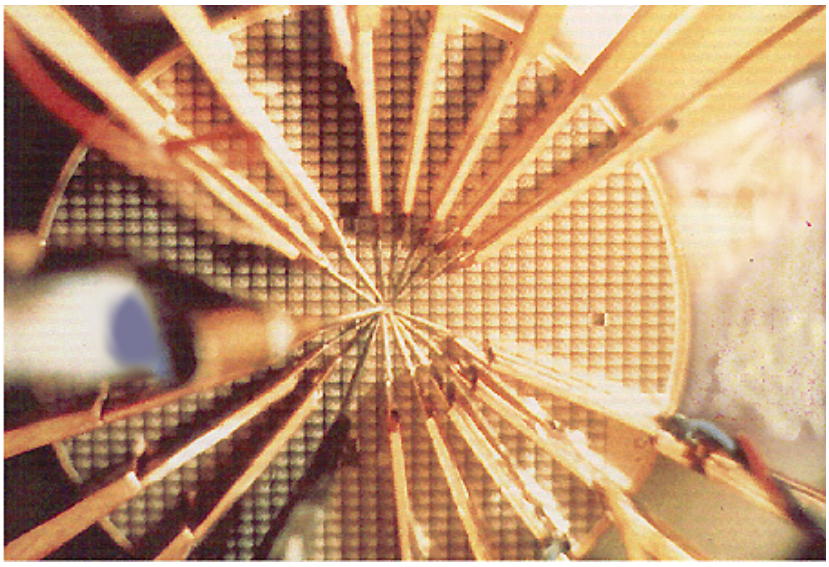


Planche 3.20 – Test sous pointes des circuits

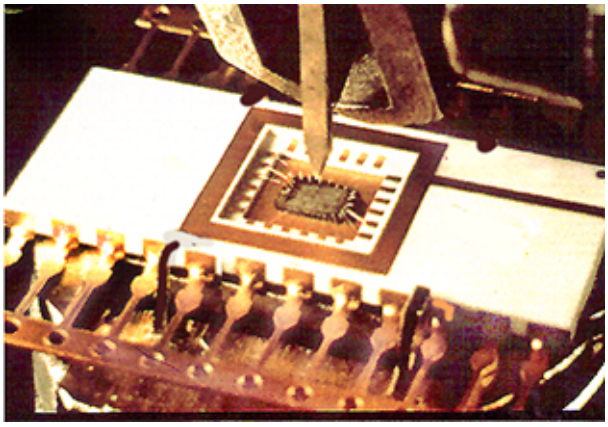


Planche 3.21 – Montage sur boîtier

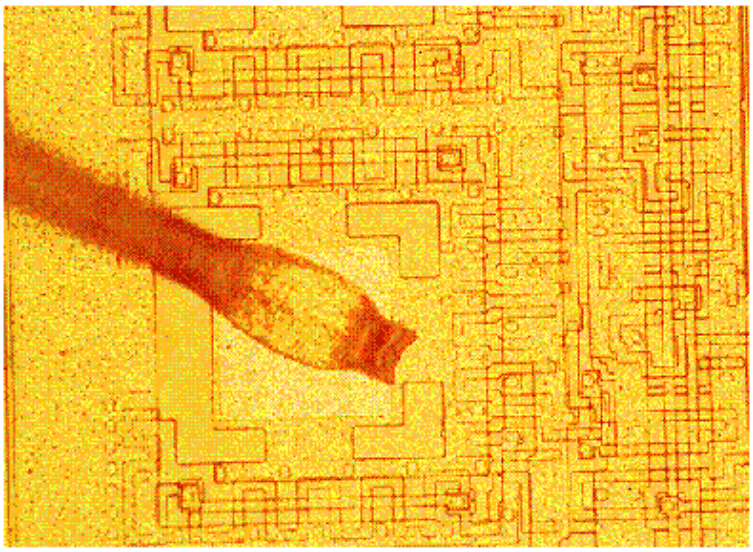


Planche 3.22 – Soudure de patte ($\sim$ 100 μ m)

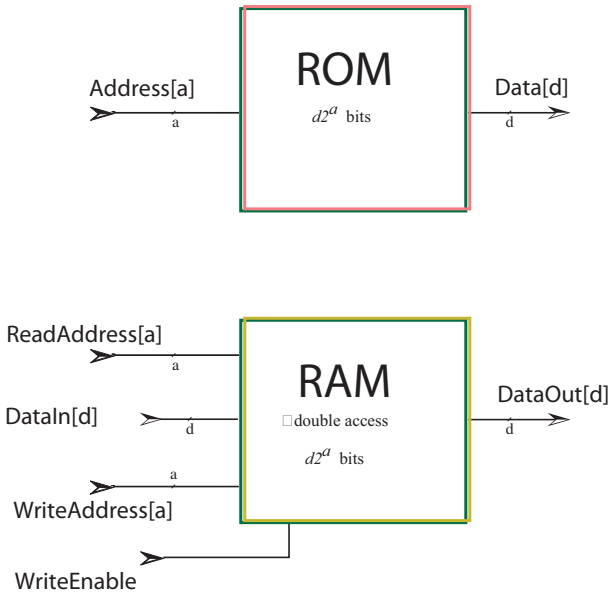


Planche 3.23 – Interfaces de la **ROM** et de la **RAM** double accés

3.3 Mémoires

Etudions quelques techniques pour réaliser les mémoires : *mortes* **ROM** - *read only memory* - et *vives* **RAM** *random access memory*, *statiques* **sRAM** - *static random access memory* - et *dynamiques* **dRAM** - *dynamic random access memory*.

La première est de définir la structure d'une mémoire par un circuit CDS, puis de le traduire en circuit électronique par les techniques de ce chapitre.

La seconde est de considérer les réalisations *analogiques* des mémoires. On utilise alors des montages qu'il est impossible de décrire par un circuit CDS, pour cause de cycles combinatoires ou de bi-directionalité des signaux. Tous ces montages sont *aussi* réalisables par des circuits CDS classiques, plus gros.

La classification des circuits entre *analogique/digital* et *synchrone/asynchrone*, concerne ici seulement la *méthodologie* de conception. Les technologies de fabrication sont largement communes (des phases de traitement spécifiques sont pourtant nécessaires à la réalisation des capacités profondes - *trench capacitor* - du point mémoire dynamique). Une fois réalisé, tout circuit électronique obéit aux lois de l'analogique, c'est à dire celles de *Maxwell*. Par construction, les *circuits mémoire* présentés ici ont cependant un *comportement digital stable* : si l'on stabilise toutes les entrées dans une configuration *digitale* - soit une tension VDD pour (1), soit GND pour (0) - alors, après un délai fini, toutes les sorties se stabilisent aussi sur des configurations *digitales*.

3.3.1 Bus bi-directionnel

Le montage (analogique) qui permet de connecter chaque $bitLine$ à un point mémoire est tellement utile pour la communication des données qu'il est nommé *Bus* - planche 3.24.

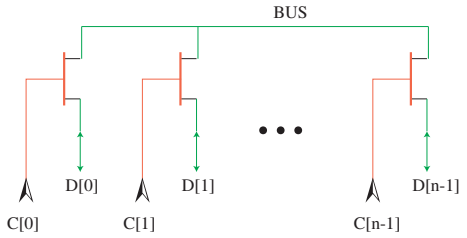


Planche 3.24 – Montage en Bus

Un bus est un fil qui relie n sources (ou destinations) potentielles de données $D[0], \cdots, D[n-1]$ entre elles. Chaque source $D[k]$ est isolée du bus par un transistor dont la commande est le contrôle $C[k]$. Pour éviter un court-circuit comme dans le cas $C[0] = C[1] = 1$, $D[0] =$ **gnd**, $D[1] =$ **vdd** (voir planche 3.24), les contrôles $C[0] \cdots C[n-1]$ doivent avoir la propriété de *un unique* :

$$\sum_{0 \leq k < n} C[k] \leq 1.$$

Cette condition de *un unique* est établie par un *dé-multiplexeur* à 2^n voies. Remarquons que le bus permet une communication *multi-directionnelle*. Chaque $D[k]$ peut être aussi bien source que destination du bus, et ce *dynamiquement*.

3.3.2 Mémoire vive à double accés

On trouve les schémas d'un circuit CDS pour réaliser une mémoire **RAM** à double accés en planche 3.25, pour le cas $a = 3$, $d = 1$. Le code qui suit donne une description paramétrée par a (bits d'adresse) et d (bits de données) :

```
RAM(a,d)(read_adr:net[a],   // Dual ported random access memory
                write_adr:net[a], write_enable:net,
                data_in:net[d])
              = data_out:net[d]
where
  for k<d do
    data_out[k] = RAM1(a)(read_adr,
                                write_adr, write_enable,
                                data_in[k])
  end for;
```

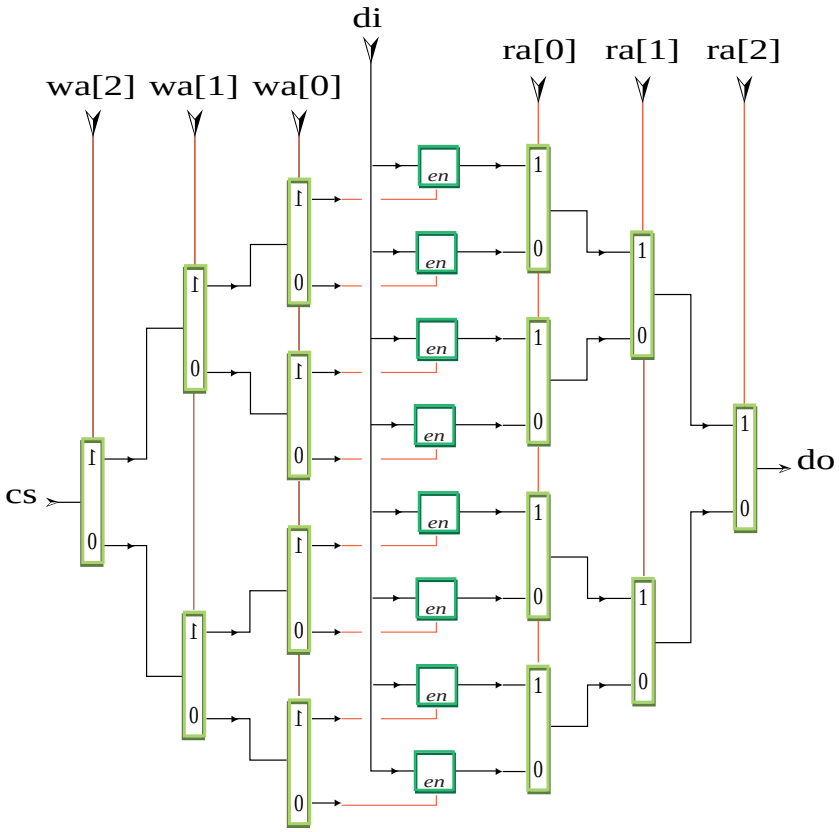


Planche 3.25 – Mémoire **RAM** double accés

```
RAM1(a)(ra, wa, en, din) = dout      // One bit wide RAM block
  where                              // selects the register to be written into
    wordLine = deMuxes(a)(en, wa);
    for k<2**a do
      M[k] = enable(reg(din), wordLine[k])
    end for;                         // selects the register to read from
    dout = Muxes(a)(ra, M)
  end where;
end where;
```

On utilise un *multiplexeur* à 2^a voies pour décoder l'adresse de lecture :

```
Muxes(a)(adr:net[], I:net[2**a]) = out
where          // 2^a ways multiplexor : out = I[ba], ba = ∑_k adr[k]2^k
  if a==0 then
    out = net(-I[0])
  else
```

```
    A'=2**(a-1); A=2*A';
    out  =  mux(adr[a-1], out1, out0);
    out0 = Muxes(a-1)(adr, I[0..A'-1]);
    out1 = Muxes(a-1)(adr, I[A'..A-1]);
  end if;
end where;
```

Le circuit dual - *dé-multiplexeur* à 2^a voies - décode l'adresse d'écriture :

```
    // 2^a ways demultiplexor : wordLine[ba] = en for ba = sum_k adr[k]2^k
          //                    wordLine[ba'] = 0  for ba' ≠ ba
deMuxes(a)(en, adr:net[]) = wordLine:net[2**a]
where
  if a==0 then
    wordLine=[en]
  else
    A=2**a; A'=A div 2;
    (en0, en1)  = demux(adr[a-1], en);
    wordLine[0..A'-1] = deMuxes(a-1)(en0, adr);
    wordLine[A'..A-1] = deMuxes(a-1)(en0, adr);
  end if;                          // one bit demux (dual of mux)
     demux(en, a) = (en & ~ a, en & a);
end where;
```

Le contenu de la **RAM** n'est pas modifié aux cycles $t \in \mathbf{N}$ où la commande $write\_enable_t = 0$ est nulle.

On peut réaliser les transistors du circuit **RAM** (a, d) par synthétise automatique du registre en 16 transistors (planche 3.10), du multiplexeur à 2^a voies (en $2^{a+2} + 2(a-2)$ transistors - planche 3.6), et du dé-multiplexeur (de même taille). Au total, on trouve 24 transistors par bit (16 par registre et 8 par décodeur) dans cette réalisation (naive) de la mémoire vive à double accés.

3.3.3 Mémoire morte ROM

Le contenu d'une mémoire morte - read only memory **ROM** - est gravé une fois pour toutes, durant la fabrication du circuit : chaque bit de la *table* à réaliser se matérialise par le placement d'une structure de contact - sur une piste d'alimentation VDD pour (1), sur une piste de terre GND pour (0) - planche 3.26.

Une **ROM** de taille $N = d2^a$ bits représente une table $M[0..A-1][0..d-1]$ de $A = 2^a$ mots binaires de d bits. Le circuit (planche 3.23) dispose d'une entrée pour l'*adresse de lecture* $adr[0..a-1]$ - sur a bits - et d'une *sortie données* $data[0..d-1]$ - sur d bits. Si $adr_t = \sum_{i<a} adr_t[i]2^i$ est l'adresse présentée en entrée au cycle $t \in \mathbf{N}$, la sortie correspondante est donnée par :

$$data_t = M[adr_t].$$

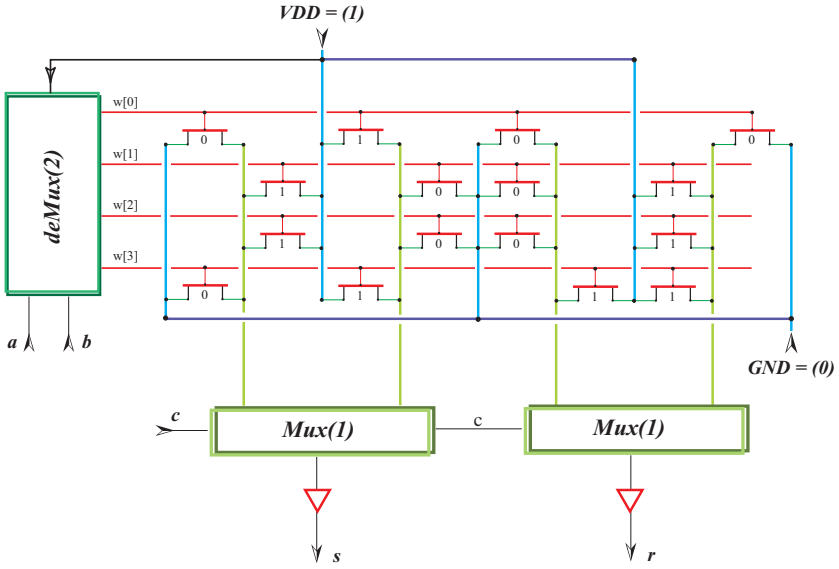


Planche 3.26 – Mémoire **ROM** 8×2 bits, configurée en additionneur binaire complet

En dépit de son nom, la mémoire morte est donc un circuit combinatoire : la donnée qui sort au cycle $t \in \mathbf{N}$ est exclusivement déterminée par l'adresse d'entrée au temps t, quels que soient les accès antérieurs pour $n < t$. Elle ne possède ni état interne, ni registre.

On peut réaliser une mémoire morte **ROM** $(adr[a]) = data[d]$ sur d bits, au moyen de d blocs **ROM** $(adr[a])$ sur un bit :

```
                                        // Read only memory = ROM
ROM(a,d)(M:net[2**a, d])(adr:net[a])=data:net[d]
where
  for k < d do
    data[k] = Muxes(a)(adr, M[0..2**a-1,k])
  end for;
end where;
```

Ce circuit contient $2^{a+2}+2(a-2)$ transistors (taille du multiplexeur à a voies), soit un peu plus de 4 transistors $(4 + 2(a-2)2^{-a})$ par bit mémorisé.

En pratique, on réalise le bloc de mémoire morte **ROM** $(adr[a]) = data$ sur 1 bits par le circuit *analogique* de la planche 3.27, qui ne comprend qu'un (peu plus d'un) transistor par bit. La surface du montage est minimale quand l'adresse a comprend un nombre pair de bits : $a = 2a'$. Le *rapport d'aspect géomètrique* de la puce finale devient alors *carré*.

On coupe l'adresse $adr[0..2a'-1]$ en deux : poids faibles pour la *ligne* $l = adr[0..a'-1]$, et forts pour la colonne $c = adr[a'..2a'-1]$, soit $adr = l + A' \times c$, avec $A' = \sqrt{M}$ et $M = 2^a$.

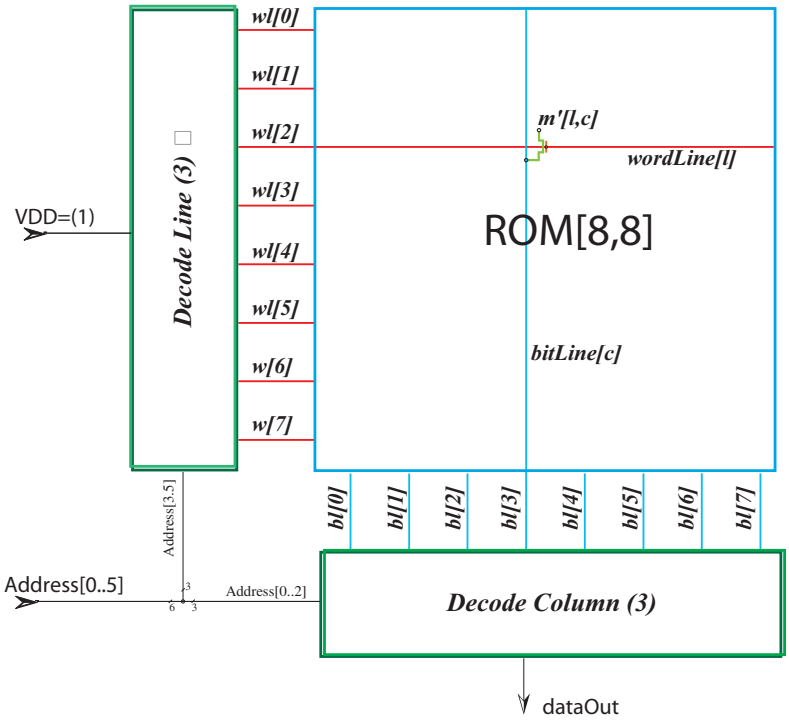


Planche 3.27 – Mémoire **ROM** : 1 transistor par bit

- Les poids faibles $adr[0..a'-1]$ de l'adresse passent dans le *décodeur des lignes*, qui sélectionne une ligne $wordLine[l]$ parmi A', ou zéro quand $WriteEnable = 0$.
- Un transistor de grille $wordLine[l]$ relie le point mémoire $M'[l,c]$ au *bus* (cf. infra) des données $bitLine[c]$, pour toute colonne $0 \leq c < A'$.
- La sortie $data$ est connectée au bus $bitLine[c]$ par le *décodeurs des colonnes* : c'est un *multiplexeur* à A' voies sur les poids forts $adr[a'..a-1]$ de l'adresse.

Au final, la sortie est connectée dynamiquement à l'élément mémoire :

$$data = bitLine[c] = M'[l,c] = M[adr], \text{ avec } adr = l + 2^{a'}c.$$

Pour $a = 2a'$ bits d'adresse et $d = 1$ bit par donnée, ce circuit comporte $2^a + 2(2^{a'+2} + 2(a'-1)$ transistors, soit (à la limite pour a grand) 1 transistor par bit : c'est le transistor qui relie le point mémoire au bus des données ($bitLine$) dans la planche 3.27. Le nombre des transistors situés dans les décodeurs (lignes et colonnes), divisé par le nombre 2^a de points mémoire vaut $\approx 82^{a'}/2^{2a'}$, soit moins de 1/100 quand $a \geq 20$ bits d'adresse pour une **ROM** 1 Mb.

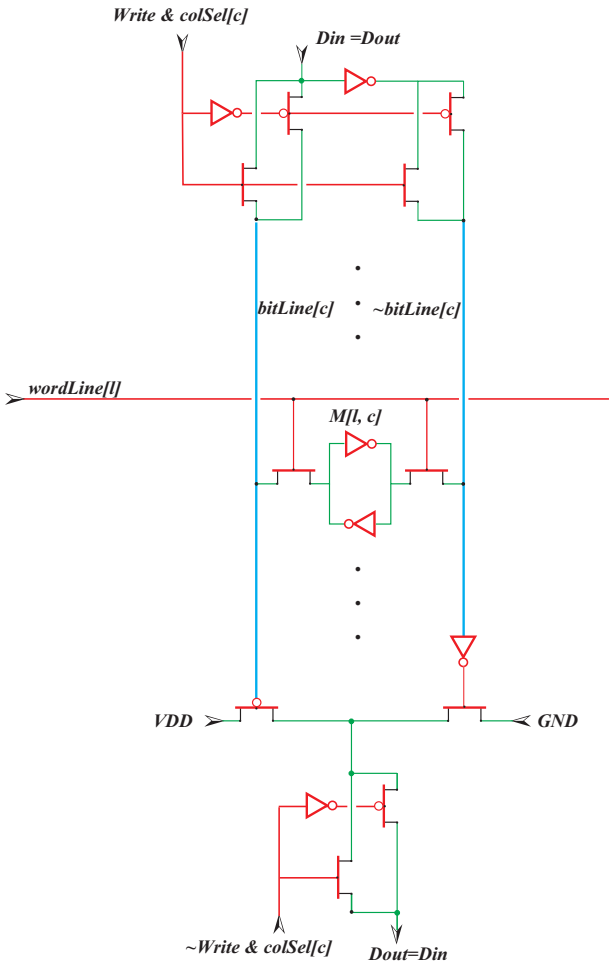


Planche 3.28 – Mémoire statique **sRAM** à 6 transistors par bit

3.3.4 Mémoire statique sRAM

Dans une mémoire vive à simple accés - *single port* **RAM** - on partage les données (soit $Din = Dout$) par un bus, et on confond les adresses (soit $ReadAddress = WriteAddress$) sur une même entrée.

Le signal $WriteNotRead$ contôle l'écriture (quand il vaut 1) et la lecture (quand il vaut 0) dans la mémoire, à chaque cycle.

- Le point mémoire **sRAM** $M[adr] = M'[l, c]$ se réalise avec deux inverseurs en boucle, dont les entrées sont respectivement connectées à $bitLine[c]$ et $\neg bitLine[c]$ par deux transistors commandés par $wordLine[l]$ - planche 3.28. Le rôle du bus différentiel est de permettre une *écriture fiable* dans le point mémoire.
- La *lecture* de l'adresse $adr = l + A'c$ se fait, comme pour la **ROM** ci-dessus, en sélectionnant par le *décodéur des lignes* une $wordLine[l]$ parmi $A' = 2^{a'}$, ainsi qu'une $bitLine[c]$ parmi A'. Le but est de connecter la sortie $data$ avec le contenu $M[adr]$ de la mémoire d'adresse adr :

$$data = bitLine[c] = M'[l, c] = M[adr].$$

  Le décodeur des colonnes est un démultiplexeur sur les poids forts d'adresse $adr[0..a' - 1] = l$, dont la sortie est le tableau $selCol[0..2^{a'} - 1]$ qui sert, dans chaque tranche c, à connecter (ou non) la sortie $Dout$ avec $bitLine[c]$ - voir planche 3.28.
- L'*écriture* du mot de valeur $data$ dans le point mémoire $M[adr] = M'[l, c]$ se fait en déconnectant les bus $bitLine[k]$ et $bitLine'[k]$ du décodeur des colonnes, pour $k \neq c$: ceci laisse les deux bus garder la valeur du point mémoire $M[l, k]$ auquel ils sont connectés. Pour la colonne $k = c$, on force la valeur $data$ de la donnée à écrire sur les bus différentiels $bitLine[c] = data$ et $bitLine'[c] = \neg data$, afin d'imposer la valeur du point mémoire $m[adr] = M'[l, c] = data$ - planche 3.28.

Pour a grand, le nombre des transistors, divisé par le nombre des bits de cette **sRAM** , tend vers 6 : deux inverseurs en boucle, et deux transistors sur le bus différentiel.

Par ses deux inverseurs en boucle (planche 3.28), le point mémoire **sRAM** *garde indéfiniment* sa valeur : une alimentation (GND et VDD) est nécessaire, mais pas l'horloge.

3.3.5 Mémoire dynamique

Le point mémoire dynamique **dRAM** avec 1 transistor/bit (planche 3.29) se réalise en remplaçant, dans le schéma de la mémoire **ROM** (planche 3.27), la connexion à une *alimentation* par une simple *capacité*.

On augmente la charge de cette capacité en la gravant verticalement dans le substrat de silicium - *trench capacitor*.

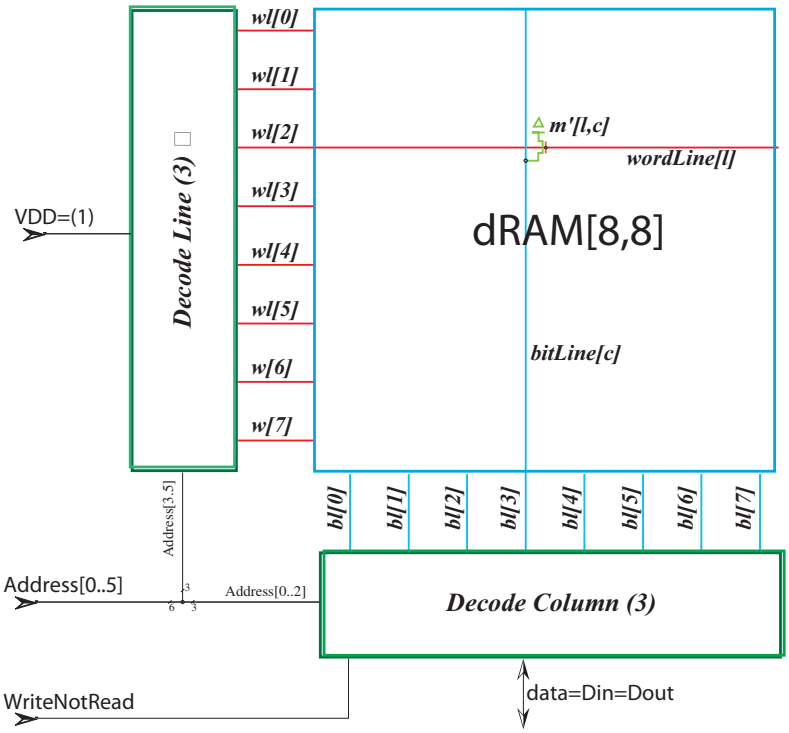


Planche 3.29 – Mémoire dynamique **dRAM** à 1 transistors par bit

Le point mémoire $M'[l,c] = M[l + cA']$ est connecté au bus des données $dataLine[c]$ quand $wordLine[l] = 1$ répond au *décodeur lignes*. Les charges du point mémoire se *partagent* alors avec celles qui était initialement sur le bus. L'effet de ce partage est double :

1. La (faible) différence de potentiel induite sur le bus est captée par un *amplificateur différentiel* A/D - voir planches 3.30 et 7.2 -
2. On perd la charge lue ! Pour corriger, il faut faire systématiquement suivre chaque lecture d'une écriture : celle de la dernière valeur lue.

Le schéma d'un décodeur de colonne se trouve en planche 3.30. Le cycle de lecture et écriture comporte trois temps, réglés par trois signaux $dt0$, $dt1$ et $dt2$ dérivés à partir du *front montant* de l'horloge clk.

La ligne de mémoire $m_l = M'[l, 0..A'-1]$ est transferée dans le *latch statique* $M[0..A'-1]$

1. Pendant $dt0$, les bus de données $bitLine[0..A'-1]$ sont préchargés à une tension intermédiaire $V0$.

2. L'*écriture* du mot de valeur $data[0..A'-1]$ dans le point mémoire $M[adr] = M'[adr_0, adr_1]$ se fait en trois temps :
3. on lit le mot $m[0..A'-1] = M'[l, 0..A'-1]$, en décodant les poids faibles adr_0 de l'adresse adr, dans un registre $m[0..A']$, à situer dans le décodeur des colonnes ;
4. on change la valeur de m en $m'[i] = m[i]$ pour $i \neq adr_1$, et $m'[adr_1]data$, où $data = data_in$ est la la donnée d'entrée ;
5. on écrit le mot $m'[0..A'-1]$ en forçant les valeurs complémentaires ($m'[i]$ et $\not m'[i]$) sur les bus (différentiels) $bitLine[i]$ et $bitLine'[i]$, pour chaque bit $0 \leq i < A'$.

– L'écriture dans la **dRAM** est, au premier abord, semblable à l'écriture dans la **sRAM** . La lecture d'une **dRAM** est plus lente que celle d'une **sRAM** , car les charges mémorisées sont plus faibles, et le bus des données n'est pas différentiel.
– Alors que la lecture d'une **sRAM** laisse la valeur mesurée *intacte*, celle d'une **dRAM** a un effet de bord majeur : perte de la charge lue ! Pour corriger, il faut faire systématiquement suivre chaque lecture d'une écriture, celle de la dernière valeur lue.
– Pire : comme le point mémoire n'est pas activement régénéré, il perd naturellement sa charge après quelques milli-secondes ! On évite ces pertes de charge en venant rafraîchir périodiquement chaque ligne de la **dRAM** , par exemple, en *volant des cycles* au fonctionnement normal.

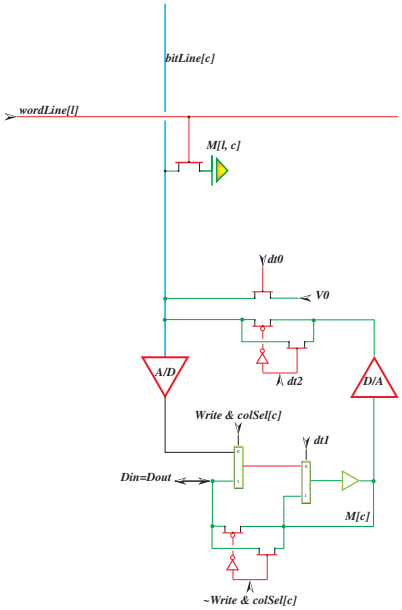
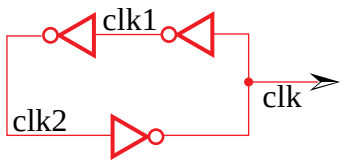


Planche 3.30 – Décodeur ligne d'une **dRAM**

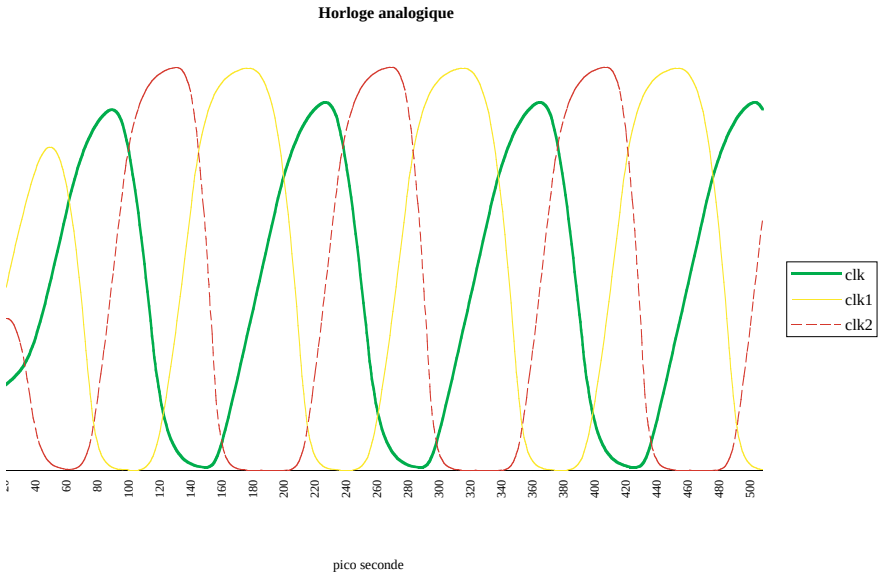
Tout ceci donne une idée de la nature (et de la complexité) des circuits analogiques à mettre au point pour réaliser une mémoire dynamique **dRAM** . la charge stockée dans chaque Ces 4 valeurs analogiques se lors de la lecture au travers d'un sur deux bits (planche 7.3). Il faut aussi savoir lors de l'écriture au travers du convertisseur dual digital/analogique.

Horloge synchrone

Synthèse Une horloge électronique est un *oscillateur* ; son générateur le plus simple est un cycle comprenant un nombre *impair* d'inverseurs - au moins trois.

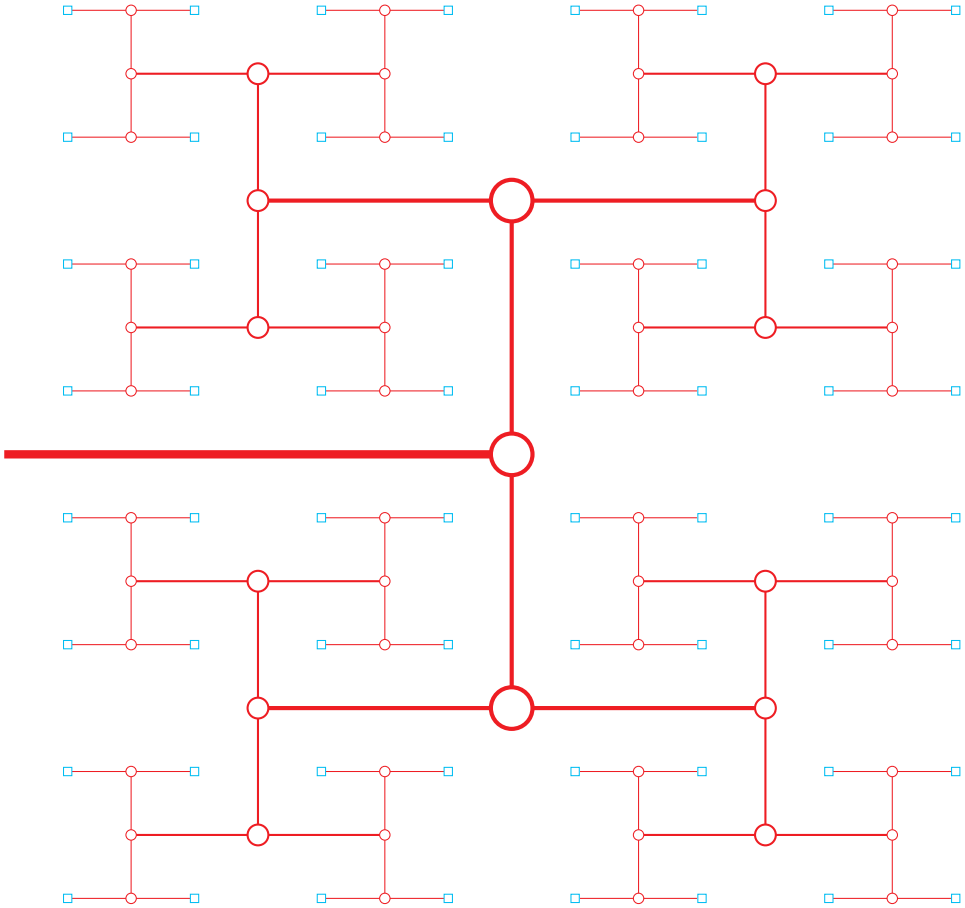


Reconnaissons le circuit *bad* du chapitre 1 dont nous avons rejeté la boucle combinatoire pour être instable. Cependant, la simulation électrique qui suit montre qu'un tel circuit peut servir à générer une horloge.



Distribution synchrone Arbre en H.

$$\begin{aligned}\mathcal{H}(1) &= \{0\},\\ \mathcal{H}(2n) &= (-n+\mathbf{i}\mathcal{H}(n))\cup(n+\mathbf{i}\mathcal{H}(n))).\end{aligned}$$



$$\begin{aligned}\mathcal{H}(1) &= \{-1,+1\},\\ \mathcal{H}(2) &= \{-1-\mathbf{i},-1+\mathbf{i},1-\mathbf{i},1+\mathbf{i}\}\\ &= \{-1,+1\}+\mathbf{i}\{-1,+1\},\\ \mathcal{H}(4) &= \{-3,-1,+1,+3\}+\mathbf{i}\{-3,-1,+1,+3\},\\ \mathcal{H}(2n) &= I(n)+\mathbf{i}I(n),\\ I(n) &= \{k\in 2\mathbf{Z}+1\ :-2n<k<2n\}.\end{aligned}$$

3.4 Progrès technologique

Le nombre de transistors par puce a doublé tous les ans dans les années 60 et début 70. De 1972 à 1996, ce doublement se fait en 18 mois. Certains pronostiquent qu'on verra des *puces* avec un milliard de transistors avant l'an 2000. Les compagnies NEC puis SAMSUNG annoncent, en 1996, l'avoir réalisé au stade expérimental.

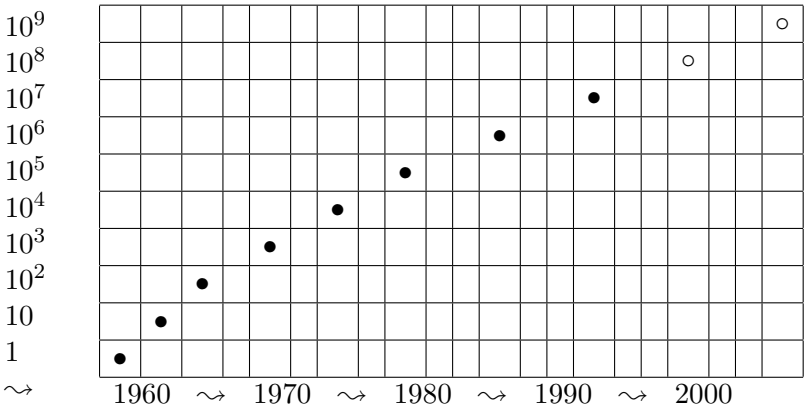
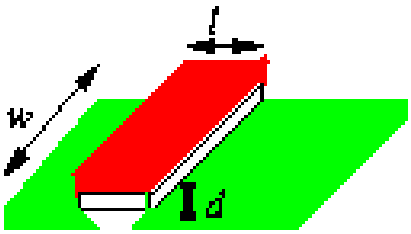
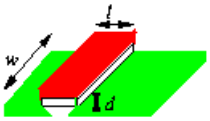


Planche 3.31 – Nombre de transistors par puce

3.4.1 Miniaturisation





Les avantages.
- Vitesse
- Taille
- Energie
- Coût
- Fiabilité

3.4.2 Limites

- Physiques

- Technologiques
- Economiques

La grille de ce transistor experimental fait 0.12 *micron* de large.

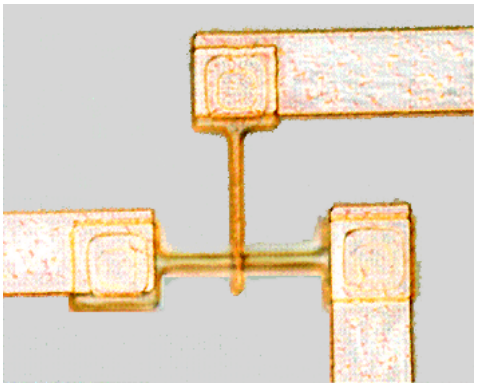


Planche 3.32 – Tout petit transistor, pour 1997

II

Outils

Chapitre 4

Arithmétique sur Silicium

Contents

Blaise Pascal a 19 ans en 1642. Pour faciliter le travail de son père, collecteur d'impôts, il réalise la première machine capable d'additionner et de soustraire automatiquement, par le mouvement d'engrenages et de roues dentées.

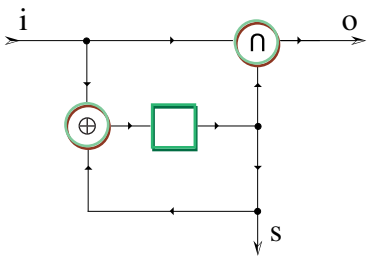
Planche 4.1 – Machine à additionner de Pascal

4.1 Compteurs

Les compteurs, ca compte beaucoup ! C'est la plus rapide des structures digitales. Nous montrons ici qu'il est possible d'opérer un compteur - de taille arbitraire - à une fréquence proche de celle de l'horloge la plus rapide. C'est important, car le compteur est l'outil de base dans la *mesure* digitale de tout signal analogique. Plus le compteur est rapide, plus la mesure est précise.

4.1.1 Compteur modulo 2

Un compteur modulo 2 est la tranche - sur 1 bit - qui sert à assembler les compteurs binaires.



Les invariants du compteur modulo 2 sont les suivants :

$$\begin{aligned} S(t) &= \sum_{k<t} i(k), \\ s(t) &= S(t) \cdot|\cdot 2 = s(t) \oplus i(t), \\ o(t) &= (s(t) + i(t)) \div 2 = s(t) \cap i(t). \end{aligned}$$

4.1.2 Compteur binaire

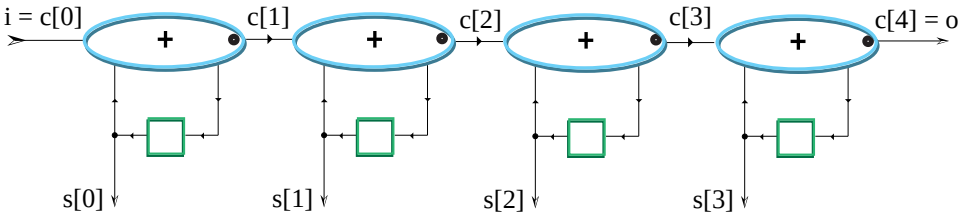


Figure 4.1 – Compteur sur 4 bits

En mettant n compteurs modulo 2 en série, on obtient un compteur $Count(n)$ modulo 2^n sur n bits.

```
Count(n)(i)=(s:[n],o)
where
  c[0]=i;                          // entrée d'incrémentation
  for k<n do                       // faire n fois
    c[k+1]=c[k]&s[k];              // propagation de la retenue
    s[k]=reg(c[k]^s[k])            // mémorisation de la somme
  end for;
  o=c[n]                           // overflow quand S(t)+i(t)=2^n
end where;
```

Les schémas du compteur $Count(4)$ - figure 4.1 - introduisent une icône importante : nous symbolisons - à partir de maintenant - l'additionneur binaire complet **abc** par un ovale autour du signe $+$. Pour éviter toute confusion entre les sorties s - somme et *sum* - et r - retenue et *carry* c - on marque d'un point la sortie retenue.

Quand il manque - comme ici - une entrée à un **abc** , dont on convient quelle vaut implicitement 0, et que l'**abc** se simplifie en un demi-additionneur, comme dessiné avant pour le compteur modulo 2. Les invariants du compteur n bits sont :

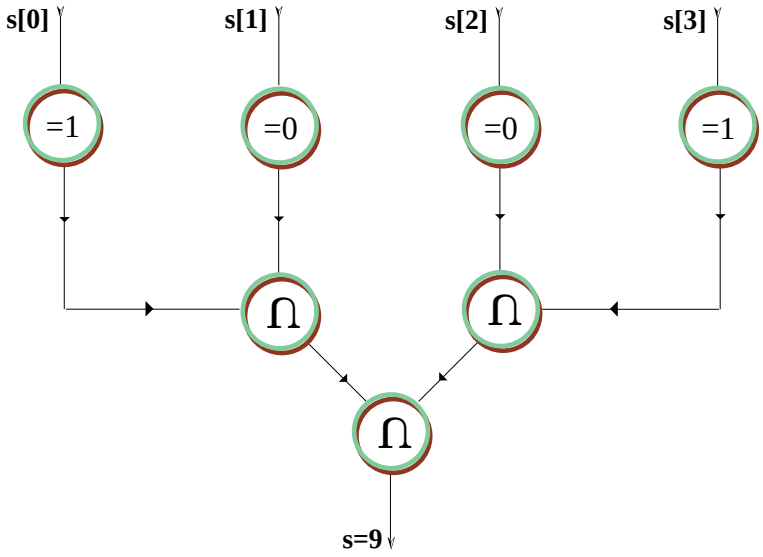
$$\begin{aligned} S(t) &= \sum_{m<n} 2^m s[m](t) = \sum_{k<t} i(k) \pmod{2^n}, \\ S(t+1) &= (S(t)+i(t)) \cdot|\cdot 2^n, \\ o(t) &= (S(t)+i(t)) \div 2^n. \end{aligned}$$

4.1.3 Décodeur de cycle

Il s'agit de détecter le passage du compteur par une valeur donnée $m \in \mathbf{N}$.

```
                          // Sortie seqm = 1 ssi m = S(t) = ∑_{k<n} s[k](t)2^k.
Equal(n,m)(s:[n])=seqm
where
  eq[0] = -1;                                  // égalité initiale
  for k<n do                                   // teste l'égalité bit à bit
    beq[k]=not s[k]^(m##k)                     // beq[k] = ¬ s[k] ⊕ m[k]
  end for
  seqm = Et(n)(beq);                           // seqm = ∩_{0≤ k <n} beq[k]
end where
```

A titre d'exemple, le circuit combinatoire $seq9 = \mathbf{Equal}(4,9)(s:[4])$ teste si le nombre binaire sur 4 bits $s:[4]$ vaut neuf $S = s[0]+2s[1]+4s[2]+8s[3] = m = 9$, soit $S = m = {}_2 1001$.



Remarquons que le calcul d'un et sur n entrées se fait au travers d'un arbre binaire de **and** , dont la profondeur combinatoire est $|n|_2$. On le construit par la définition *récursive* suivante.

```
Et(n)(b:[n])=r                                   // Et à n entrées
where
  if n=1 then
     r=b[0]
  else                                                  // n > 1
     m=n div 2;                                  // diviser pour régner
     r=Et(m)(b[0..m-1])&Et(n-m)(b[m..n-1])
  end if
end where
```

Exercice 1 *Réaliser un compteur* up-down *dont la valeur $S(t+1)$ est liée à l'incrément $i \in \mathbf{B}$ et à la valeur courante $S(t)$ par :*

$$S(t+1) = S(t) + i - (1-i) = S(t) + 2i - 1 \pmod{2^n}.$$

La sortie $S[0, n-1]$ est présentée sur n bits en complément à deux.

Exercice 2 *Réaliser un compteur dont la valeur $S(t+1)$ est liée à la valeur courante $S(t)$ et à celle de l'incrément $I(t)$ par :*

$$S(t+1) = S(t) + I(t) \pmod{2^n}.$$

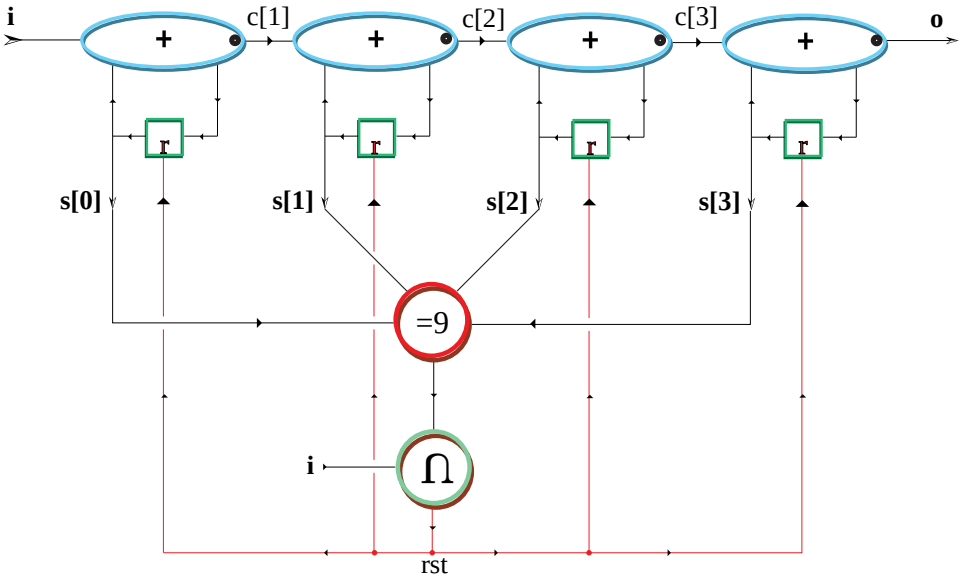
L'entrée $I(t) = \sum 0 \leq k \leq m-2 I[k] 2^k - I[m-1] 2^{m-1}$ est un nombre présenté sur $m < n$ bits $I[0, m-1]$ en complément à deux. La sortie $S(t) = \sum 0 \leq k \leq n-2 S[k] 2^k - S[n-1] 2^{n-1}$ est aussi présentée sur n bits $S[0, n-1]$ en complément à deux. Doter le compteur d'une sortie qui détecte le passage par $S = 0$.

4.1.4 Compteur modulo m

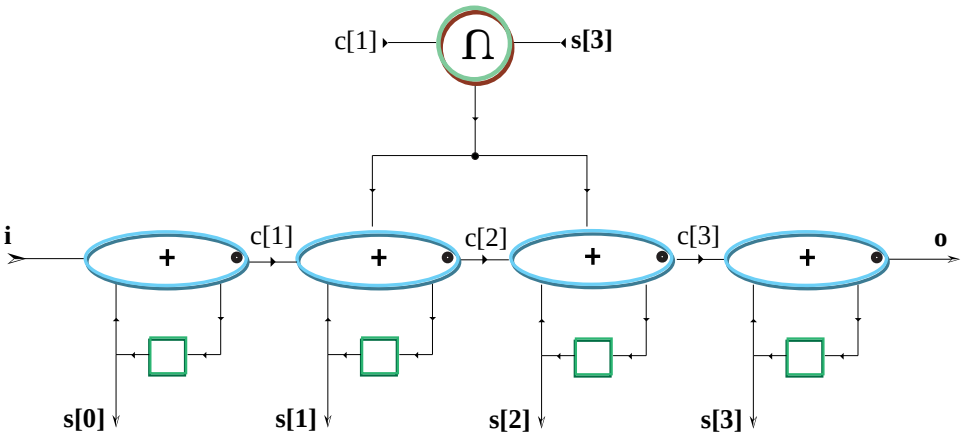
On obtient un compteur modulo m en munissant un compteur binaire sur $n = |m|_2$ bits d'un système de remise à zéro - *reset* - au cycle m. Le signal rst de remise à zéro détecte le passage du compteur par la valeur $m-1$, pour remettre tous les registres à zéro au prochain cycle d'incrément $i = 1$.

```
CountMod(m)(i)=(n,s:[n],o)                    // Compteur modulo m
where
  n=prefixLength(m);          // Taille du compteur, en bits : n = |m|_2.
                    // Signal de remise à zéro, quand S(t) = m − 1 et i = 1
  rst=i&Equal(n,m-1)(s);
  reset s:[n],o by rst do                     // remise à zéro par rst
    (s,o) = Count(n)(i)                       // Compteur  n bits
  end reset;
end where;
```

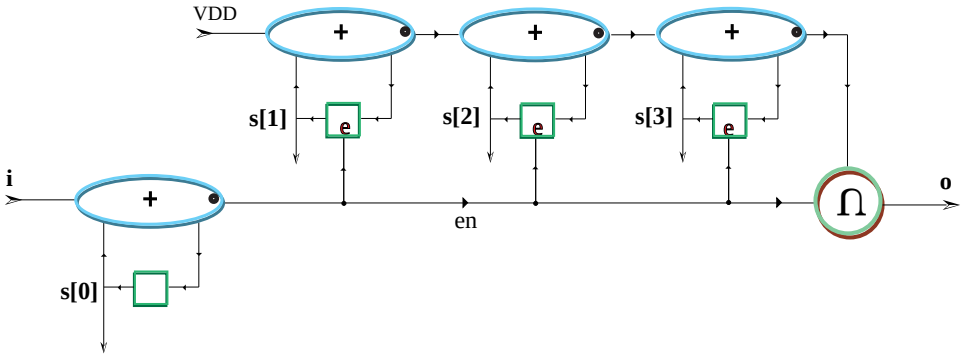
Application au compteur modulo 10 $CM10$.



Une autre approche pour obtenir un compteur modulo m est d'augmenter la valeur du compteur de $2^n - m$ quand $c = m - 1$ et $i = 1$. Après simplifications, on trouve les schémas suivants pour $CM10$.



4.1.5 Compteur rapide




```
Count(n)(i) = (s:[n],o)          // Compteur binaire, version rapide.
where
  k=1;          // Paramètre statique, à optimiser pour la technologie visée.
  if n<=k then                                          // Cas n ≤ k
    (s,o) = Count(n)(i);                 // Compteur binaire sur k bits
  else                                  // Cas n > k, poids faibles du compteur
    (s[0..k-1],e) = Count(k)(i);                  // Compteur k bits.
                  // La retenue e valide l'horloge du compteur des poids forts.
    enable s':[n-k],cn by en do
                                  // compteur en roue libre sur n − k bits
      (s',cn)=Count(n-k)(-1);                    // i=vdd =-1=(1)2
    end enable;
    o=cn&en;                                              // retenue sortante
    s[k..n-1]=s';                                 // poids forts du compteur
  end if;
end where;
```

4.2 Addition

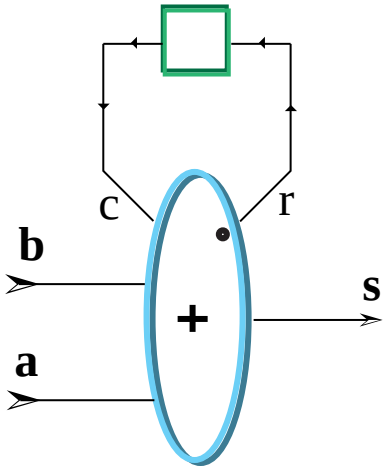


Figure 4.2 – Additionneur binaire en série

4.2.1 Addition série

En binaire

Par sa définition 8, l'*additionneur binaire complet* maintient l'invariant suivant (2.9) :

$$a + b + c \quad = \quad s + 2r.$$

Où plutôt, on devrait écrire $a(t) + b(t) + c(t) = s(t) + 2r(t)$ puisque notre invariant combinatoire porte sur les bits $a(t), b(t), c(t), s(t), r(t) \in \mathbf{B}$. Si on ajoute maintenant un registre, d'entrée r et de sortie c, on a, par l'invariant de **reg** - chapitre 2 :

$$c \quad = \quad 2r.$$

En substituant dans (2.9), les variables de retenue c et r s'éliminent, et il reste :

$$a + b \quad = \quad s.$$

Cet invariant s'écrit, en faisant intervenir explicitement le temps :

$$\sum_{t \in \mathbf{N}} a(t)2^t \; + \; \sum_{t \in \mathbf{N}} b(t)2^t \; = \; \sum_{t \in \mathbf{N}} s(t)2^t.$$

Ceci montre que le circuit de la figure 4.2 calcule bien la *somme 2-adique* $s = a+b$ de ses entrées. Cet *additionneur binaire* calcule - en série par les poids faibles - un bit $s(0)s(1)\cdots s(t)\cdots$ de la somme $s = a + b$ par cycle $t \in \mathbf{N}$. L'icône utilisée ici pour symboliser l'additionneur série de la figure 4.2 est un cercle tracé autour du signe $+$; cette porte a deux entrées a et b et une sortie somme s.

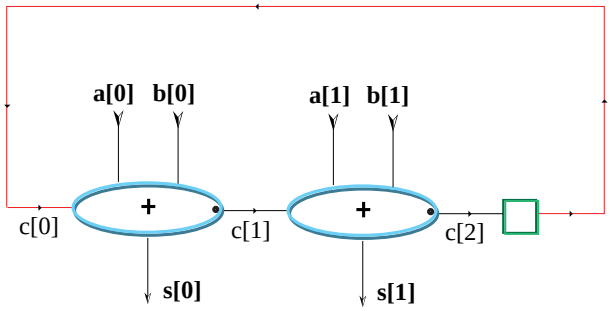


Figure 4.3 – Additionneur série, en base 4

Contemplons le calcul de la somme $a + b = s$, avec les entrées $a = 19_{10} = (0)10011_2$ et $b = 23_{10} = (0)10111_2$, :

$$\begin{array}{l} (0)10011_2 \\ (0)10111_2 \end{array} \boxed{0}\, 0_2 \;\Rightarrow\; \begin{array}{l} (0)1001_2 \\ (0)1011_2 \end{array} \boxed{1}\, 10_2 \;\Rightarrow\; \begin{array}{l} (0)100_2 \\ (0)101_2 \end{array} \boxed{1}\, 010_2 \Rightarrow$$

$$\begin{array}{l} (0)10_2 \\ (0)10_2 \end{array} \boxed{1}\, 1010_2 \;\Rightarrow\; \begin{array}{l} (0)1_2 \\ (0)1_2 \end{array} \boxed{0}\, 01010_2 \;\Rightarrow\; \begin{array}{l} (0)_2 \\ (0)_2 \end{array} \boxed{1}\, 101010_2 \Rightarrow$$

$$\begin{array}{l} (0)_2 \\ (0)_2 \end{array} \boxed{0}\, 0101010_2 \;\Rightarrow\; \begin{array}{l} (0)_2 \\ (0)_2 \end{array} \boxed{0}\, 00101010_2 \;\Rightarrow\; \cdots \Rightarrow (0)101010_2.$$

On trouve bien $s = 42_{10} = (0)101010_2$ pour sortie finale.

Changement de base

Soient $a = {}_2a(0)\cdots a(t)\cdots$ et $b = {}_2b(0)\cdots b(t)\cdots$ deux nombres 2-adiques, avec $a(t), b(t) \in \mathbf{B}$ pour $t \in \mathbf{N}$. Dans la base 4, ces nombres s'écrivent $a = {}_4A(0)\cdots A(t)\cdots$ et $b = {}_4B(0)\cdots B(t)\cdots$ avec, pour chiffres de rang $t \in \mathbf{N}$: $A(t) = a(2t) + 2a(2t+1) \in \mathbf{Z}_4$ et $B(t) = b(2t) + 2b(2t+1) \in \mathbf{Z}_4$. Il en va de même pour la somme 4-adique $s = {}_4S(0)\cdots S(t)\cdots$ avec $S(t) = s(2t) + 2s(2t+1) \in \mathbf{Z}_4$. En reprenant l'invariant *binaire* (2.9) de **abc** aux instants $2t$ et $2t+1$, puis en éliminant r_{2t+1}, on trouve l'invariant 4-adique de l'addition série :

$$A(t) + B(t) + c(t) \quad = \quad S(t) + 4r(t).$$

Dans cette expression, $A(t), B(t)$ sont les chiffres en base 4 des opérandes, et $S(t)$ ceux de la somme. Les *bits* $c(t), r(t) \in \mathbf{B}$ sont les retenues, entrante et sortante. Pour éliminer les retenues, il faut convenir que $c = \mathbf{reg}(r)$ correspond à $c = 4r$, comme il se doit en arithmétique 4-adique. Plus généralement, en arithmétique 2^n-adique, on traite n bits à chaque cycle, et **reg** correspond à la multiplication par la base 2^n, soit un décalage d'un chiffre - vers les poids forts. Ceci nous amène

à l'additionneur série de la figure 4.3, qui lit 2 bits a_0, a_1 de A, deux bits b_0, b_1 de B et calcule 2 bits s_0, s_1 de S à chaque cycle de l'horloge. On obtient $2n$ bits de somme en n cycle, soit deux fois plus qu'avec l'additionneur binaire. L'addition en base 4 semble donc *deux fois* plus rapide que celle en base 2 ; pour un circuit *deux fois* plus gros. Attention ! Ce raisonnement ne vaut que pour une période d'horloge supérieure aux *délais critiques* de chacun des deux circuits - sinon, nos circuits produisent un résultat aléatoire. Le délai critique de l'additionneur binaire de la figure 4.2 est égal - en supposant que les délais RC distribués dans les fils sont négligeables - à la somme des délais $\tau(\mathbf{abc}) + \tau(\mathbf{reg})$ de l'additionneur binaire complet en série avec le registre. Le délai de l'additionneur en base 4 de la figure 4.3 est $2\tau(\mathbf{abc}) + \tau(\mathbf{reg})$. En base 2^n, on trouve $n\tau(\mathbf{abc}) + \tau(\mathbf{reg})$. Dans notre technologie simplificatrice, nous avons $\tau(\mathbf{abc}) = \tau(\mathbf{reg}) = 4\tau$, avec $\tau = \tau(\mathbf{mux}) = \tau(\mathbf{not})$. Ceci donne une période critique pour l'horloge de 8τ en binaire ; en base 4, on trouve 12τ pour un circuit de taille double ; en base 2^n, le délai est $4(n+1)\tau$ pour n fois la surface. Si l'on veut utiliser la technologie disponible à plein rendement, il faut opérer chacun de ces additionneurs à sa fréquence critique. Comparons alors la *densité de calcul* de ces structures d'additionneurs ; pour cela, comptons le nombre de bits calculés par unité de surface λ^2 et de temps τ. Si on normalise cette mesure à 1 pour l'additionneur binaire, on trouve que la densité de calcul de l'additionneur en base 4 est $2/3 = 0 \cdot 6(6)_{10}$, et $2/17 = 0 \cdot 23\cdots_{10}$ sur 16 bits - base $b = 2^{16} = 65536$. La technique de changement de base - montrée ici pour l'addition - est générale. Elle permet de déployer tout circuit CDS de la base 2 à la base 2^n ; on obtient un circuit n fois plus gros, qui calcule n bits par cycle. Si on peut opérer à période d'horloge constante, c'est le compromis *idéal* entre surface et temps, à densité de calcul *constante*. La section qui suit montre que l'on peut approcher cet idéal, en simplifiant et en accélérant les additionneurs combinatoires.

4.2.2 Addition parallèle

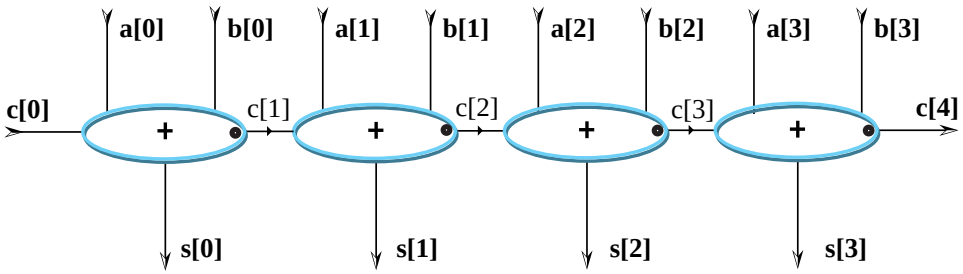


Figure 4.4 – Additionneur à propagation des retenues

Propagation des retenues

Soient $A = {}_2[a_0 \cdots a_k \cdots a_{n-1}], B = {}_2[b_0 \cdots b_k \cdots b_{n-1}]$ deux entiers binaires sur n bits. Soient $S = {}_2[s_0 \cdots s_k \cdots s_n], R = {}_2[r_0 \cdots r_k \cdots r_n]$ les sommes et retenues (sur $n+1$ bits) définies par l'algorithme d'addition binaire ; on les calcule au moyen des formules :

$$r_0 = 0,$$
$$s_k = (a_k + b_k + r_k) \cdot|\cdot 2, \quad r_{k+1} = (a_k + b_k + r_k) \div 2,$$

pour $0 \leq k < n$ et $s_n = r_n$. Ces équations se traduisent directement par le circuit de la figure 4.4, formé de n additionneurs binaires complets **abc** , dont la chaîne des retenues est connectée en série.

```
Add(n)(a:[n],b:[n],c) = s:[n+1]
  where
    c[0]=c;                                    // retenue entrante
    for k<n do
      (s[k],c[k+1])=abc(a[k],b[k],c[k])
    end for;
    s[n]=c[n]                                  // retenue sortante
  end where;
```

Invariant :

$$c + \sum_{i<n} a[i]2^i \; + \; \sum_{i<n} b[i]2^i \; = \; \sum_{i \leq n} s[i]2^i.$$

Le délai combinatoire de l'additionneur à propagation de retenues sur n bits est $n \times \tau(\mathbf{abc})$. Sa surface est $n \times \lambda^2(\mathbf{abc})$.

Version électronique

On calcule pour chaque bit la variable $p[k] = a[k] \oplus b[k]$ qui *propage* la retenue quand elle vaut un, et la variable $g[k] = a[k] \cap b[k]$ *génère* la retenue quand elle n'est pas propagée. Avec p, g, on réalise le circuit de propagation des retenues de la figure 4.5, dans lequel une retenue traverse exactement un transistor par bit. C'est bien sûr très rapide. A cause des délais RC distribués dans la chaîne des retenues, celle-ci ne doit pas être trop longue [1]. En pratique, on introduit donc un inverseur tous les 4 bits, afin de découpler les chaînes de retenues.

4.2.3 Addition par dichotomie

Pour rapide qu'il soit, le délai de propagation de la retenue est proportionnel au nombre de bits n, car sa profondeur combinatoire est égale à n - au moins

1. Le délai équivalent de n RC en série est $\frac{n}{2}$ RC

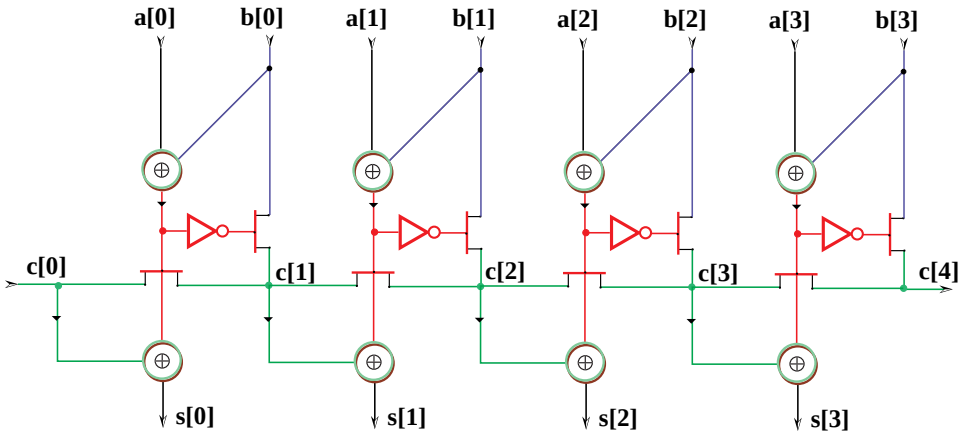


Figure 4.5 – Propagation de retenue par un transistor

pour la version électronique, en comptant le nombre maximal de transistors en série. Existe-t-il des additionneurs moins *profonds* ? Depuis les travaux de von Neumann sur l'unité arithmétique de l'ENIAC (figure 12), on connaît des circuits d'addition dont la profondeur combinatoire est proche du logarithmique binaire $|n|_2 = log_2(n+1) \div 2$ du nombre n de bits.

Algorithme

Algorithme 6 (Addition en temps parallèle logarithmique)

Soient $A = {}_2[a_0 \cdots a_{n-1} a_n \cdots a_{2n-1}]$ *et* $B = {}_2[b_0 \cdots b_{n-1} b_n \cdots b_{2n-1}]$ *deux entiers binaires de* $2n = 2^p$ *bits, que l'on sépare en deux moitiés de* $n = 2^{p-1}$ *bits chacune :*

$$A^0 = {}_2[a_0 \cdots a_{n-1}],\ A = A^0 + 2^n A^1,\ A^1 = {}_2[a_n \cdots a_{2n-1}];$$
$$B^0 = {}_2[b_0 \cdots b_{n-1}],\ B = B^0 + 2^n B^1,\ B^1 = {}_2[b_n \cdots b_{2n-1}].$$

On calcule simultanément

$$\begin{aligned} S &= A + B + 0 = {}_2[s_0 \cdots s_{2n}], \\ S' &= A + B + 1 = {}_2[s'_0 \cdots s'_{2n}], \end{aligned}$$

de la manière suivante :

1. *Pour* $p = 0$*, on calcule* $S = [s_0 s_1]$ *et* $S' = [s'_0 s'_1]$ *comme unique solution booléenne* $s_0, s_1, s'_0, s'_1 \in \{0, 1\}$ *de l'équation (a) :*

$$1 + s_0 + 2s_1 = s'_0 + 2s'_1 = a_0 + b_0 + 1.$$

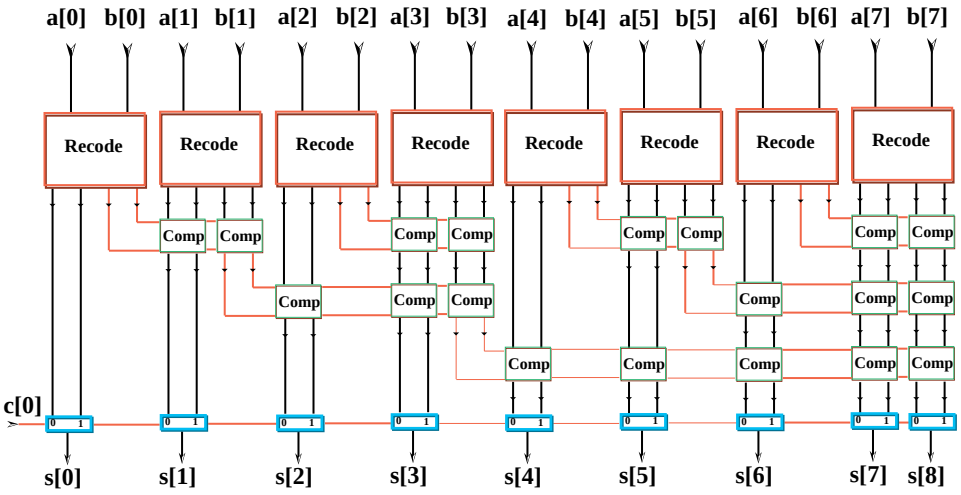


Figure 4.6 – Additionneur à délai logarithmique

2. *Pour $p > 0$, on coupe $A = A^0 + 2^n A^1$ et $B = B^0 + 2^n B^1$ en deux pour calculer,* récursivement et en parallèle *:*

$$\begin{array}{rcccl} S^0 & = & A^0 + B^0 + 0 & = & {}_2[s_0^0 \cdots s_n^0], \\ S^1 & = & A^1 + B^1 + 0 & = & {}_2[s_0^1 \cdots s_n^1], \\ S'^0 & = & A^0 + B^0 + 1 & = & {}_2[s'^0_0 \cdots s'^0_n], \\ S'^1 & = & A^1 + B^1 + 1 & = & {}_2[s'^1_0 \cdots s'^1_n]. \end{array}$$

Les sommes finales S, S' se forment à partir des quatre sommes partielles S^0, S'^0, S^1, S'^1 par les règles (b) :

$$\begin{array}{llllll} \textit{Si} & s_n^0 = 0 & \textit{alors} & S = & S^0 + 2^n S^1 = & {}_2[s_0^0 \cdots s_{n-1}^0 \quad s_0^1 \cdots s_n^1]; \\ \textit{Si} & s_n^0 = 1 & \textit{alors} & S = & S^0 + 2^n S'^1 = & {}_2[s_0^0 \cdots s_{n-1}^0 \quad s'^1_0 \cdots s'^1_n]; \\ \textit{Si} & s'^0_n = 0 & \textit{alors} & S' = & S'^0 + 2^n S^1 = & {}_2[s'^0_0 \cdots s'^0_{n-1} \quad s_0^1 \cdots s_n^1]; \\ \textit{Si} & s'^0_n = 1 & \textit{alors} & S' = & S'^0 + 2^n S'^1 = & {}_2[s'^0_0 \cdots s'^0_{n-1} \quad s'^1_0 \cdots s'^1_n]. \end{array}$$

Les invariants de cet algorithme sont :

$$\begin{array}{rcccl} \sum_{i<n} a[i]2^i & + & \sum_{i<n} b[i]2^i & = & \sum_{i\le n} s[i]2^i \\ 1 \; + \; \sum_{i<n} a[i]2^i & + & \sum_{i<n} b[i]2^i & = & \sum_{i\le n} s'[i]2^i \end{array}$$

Effectuons le calcul $19 + 23 = 42$ par l'algorithme 6, en écrivant les poids faibles à gauche.

	19	1	1	0	0	1	0	0	0
+	23	1	1	1	0	1	0	0	0
=	S_0	01	01	10	00	01	00	00	00
	S'_0	11	11	01	10	11	10	10	10
=	S_1	011		100		010		000	
	S'_1	111		010		110		100	
=	S_2	01010				01000			
	S'_2	11010				11000			
=	42	010101000							
	43	110101000							

Transcrivons ce calcul en décimal, en n'écrivant que la partie S des sommes partielles :

	19	1	1	0	0	1	0	0	0
+	23	1	1	1	0	1	0	0	0
=	S_0	2	2	1	0	2	0	0	0
=	S_1	6		1		2		0	
=	S_2	10				2			
=	S	42							.

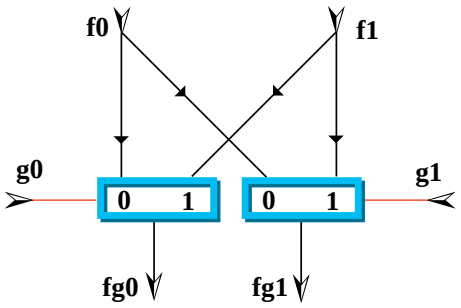


Figure 4.7 – Composition de deux fonctions

Circuit On résout explicitement l'équation (a) par les formules :

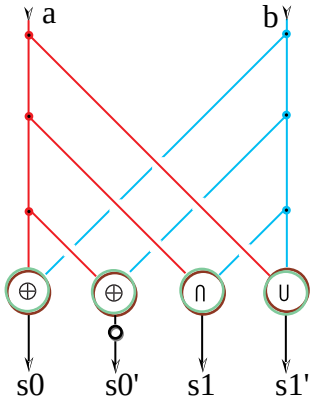
$$s_0 = a_0 \oplus b_0, \quad s_1 = a_0 \cap b_0,$$
$$s'_0 = \neg s_0, \quad s'_1 = a_0 \cup b_0.$$

Ceci nous conduit à définir le circuit *Recode* - en haut de la figure 4.6 par le code $_2$**Z** suivant.

```
Recode(a,b)=(s:[2],s':[2])
```

```
where
  s[0]=a^b; s'[0]=~s[0];
  s[1]=a&b; s'[1]=a|b
end where
```

Les quatre conditions de l'équation (b) sont équivalentes - pour chaque tranche de 1 bit - au deux multiplexeurs de la figure 4.7 - entrées communes, contrôles séparés.



Ce circuit complète la description des composants de la figure 4.6. Le code ${}_2\mathbf{Z}$ correspondant - hors de la couche finale de multiplexeurs - suit.

```
fAdd(n)(a:[n],b:[n])=(s:[n+1],s':[n+1])
where
  if n=1 then                                          // Recode
     (s,s')=Recode(a,b)
  else                                                 //n > 1
     m=n div 2; n'' = n - n';
                                                       // Lower half
     (s[0..m-1],s'[0..m-1])=fAdd(m)(a,b);
                                                       // Upper half
     a'=a[m..n-1]; b'=b[m..n-1];
     (h,h')=fAdd(n-m)(a', b');
                                                       // Combine outputs
     for k<m-n do
       s[k+m] =mux(s[m], h'[k],h[k]);
       s'[k+m]=mux(s'[m],h'[k],h[k])
     end for
  end if
end where
```

L'analyse de ce circuit montre que sa profondeur combinatoire est minimale - à trois **mux** près !

Proposition 4 *La profondeur combinatoire du circuit $fAdd(n)$ est :*

$$\tau(fAdd(n)) = |n|_2 + 3.$$

Tout circuit d'addition $Add(n)$ sur n bits a une profondeur combinatoire au moins égale à :

$$\tau(fAdd(n)) \geq |n|_2.$$

Preuve : La profondeur de $Recode$ est 2, celle de l'arbre de $Comp$ est $|n|_2$, et le multiplexeur ajoute un au total : $|n|_2 + 3$. Soit $Add(n)$ un circuit arbitraire, qui calcule la représentation binaire sur $n+1$ bits de la somme $S = A + B$ de deux nombres de n bits. Dans le cas particulier $A = {}_2[a_0 \cdots a_{n-1}]$ et $B = 1$, la valeur du bit le plus significatif de S est donnée par

$$s_n = \bigcap_{0 \leq i < n} a_i.$$

Dans le circuit $Add(n)$ remontons de s_n vers les deux entrées des multiplexeurs dont il est sortie, et continuons jusqu'à arriver sur une entrée. On doit nécessairement pouvoir suivre ainsi un chemin vers chacune des entrées a_i, pour $0 \leq i < n$, car s_n dépend de *tous* les a_i. Le nombre de points accessibles depuis s_n en passant par p multiplexeurs est au plus 2^p. Il existe donc un chemin qui va de l'un des a_i vers s_n, et qui traverse au moins $|n|_2$ multiplexeurs. Ceci s'applique pour l'addition, et pour toute fonction dont un bit de sortie dépend effectivement des n autres bits d'entrées ($Et(n)$, $Xor(n)$, soustraction, multiplication, ...). **Q.E.D.**

4.3 Soustraction

4.3.1 Complément à deux

La formule (2.8) du complément à deux permet de réduire la soustraction $d = a - b$ à une addition $d = 1 + a + \neg b$. C'est ainsi qu'en ajoutant trois inverseurs à l'additionneur binaire série de la figure 4.2, on obtient le soustracteur série de la figure 4.8. Invariant :

$$\sum_{t \in \mathbf{N}} a_t 2^t \; - \; \sum_{t \in \mathbf{N}} b_t 2^t \; = \; \sum_{t \in \mathbf{N}} d_t 2^t$$

On peut - de la même façon - reprendre toutes les structures d'additionneur de la section qui précède, et les transformer en soustracteurs de mêmes performances.

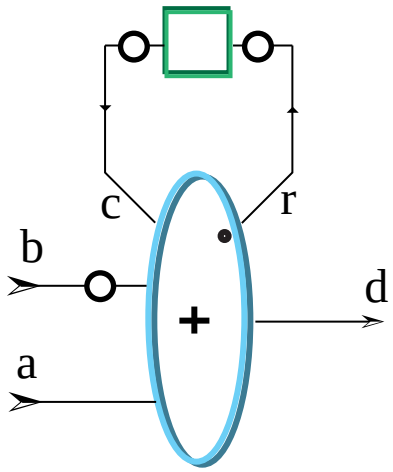


Figure 4.8 – Soustracteur binaire en série

4.3.2 Unité arithmétique et logique

Reprenons le schéma de l'addition série (figure 4.2), et remplaçons la fonction **abc** de l'additionneur binaire complet par une porte universelle $UAL \in \mathbf{B}^3 \mapsto \mathbf{B}^2$; chaque sortie $s = som(a,b,c)$, $r = ret(a,b,c)$ de l'UAL - unité arithmétique et logique - est une fonction booléenne $\mathbf{B}^3 \mapsto \mathbf{B}$ arbitraire, réalisée au moyen d'un multiplexeur à 8 voies, dont les entrées forment la table de vérité de la fonction à réaliser. Par un choix approprié des fonctions ret et som, l'unité arithmétique et logique ainsi construite permet de réaliser *tout* automate fini à deux entrées, une sortie et deux états internes. Les 16 bits $rO \cdots r7$ et $s0 \cdots s7$ qui spécifient les tables de vérité de ret et som sont le *micro-code* de l'*unité arithmétique et logique* UAL - en anglais ALU, pour *Arithmetic and Logic Unit*. Montrons quelques exemples d'opérations utiles que notre UAL réalise, pour divers choix des fonctions ret, som et de la retenue initiale r_0.

1. Addition binaire $S = A + B \pmod{2^t}$:

$$\begin{aligned} som(a,b,c) &\Rightarrow a \oplus b \oplus c, \\ ret(a,b,c) &\Rightarrow (a \cap b) \cup (b \cap c) \cup (c \cap a), \\ r_0 &= 0. \end{aligned}$$

2. Soustraction binaire $S = A - B \pmod{2^t}$ en complément à deux :

$$\begin{aligned} som(a,b,c) &\Rightarrow a \oplus \neg b \oplus c, \\ ret(a,b,c) &\Rightarrow (a \cap \neg b) \cup (\neg b \cap c) \cup (c \cap a), \\ r_0 &= 1. \end{aligned}$$

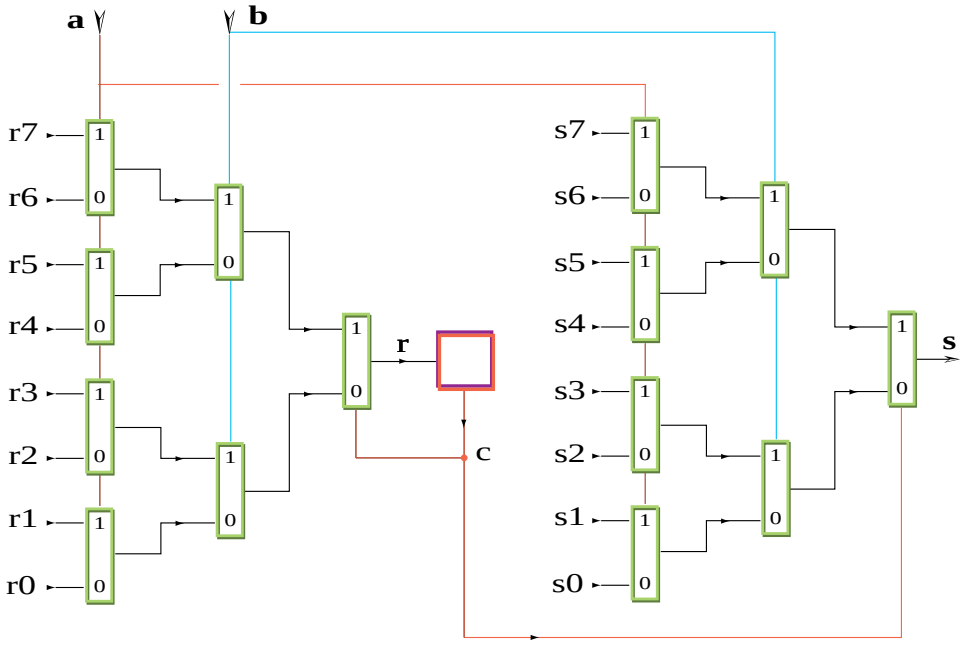


Figure 4.9 – Unité arithmétique et logique en série

3. Duplication $S = r_0 + 2 \times A \pmod{2^t}$:

$$\begin{aligned} som(a,b,c) &\Rightarrow c, \\ ret(a,b,c) &\Rightarrow a. \end{aligned}$$

4. Egalité $s_t = (A = B \bmod 2^t)$:

$$\begin{aligned} som(a,b,c) &\Rightarrow c, \\ ret(a,b,c) &\Rightarrow (a \cap b \cap c) \cup (\neg a \cap \neg b \cap c), \\ r_0 &= 1. \end{aligned}$$

5. Comparaison $s_t = (A \bmod 2^t) > (B \bmod 2^t)$:

$$\begin{aligned} som(a,b,c) &\Rightarrow c, \\ ret(a,b,c) &\Rightarrow (a \cap \neg b) \cup (a \cap b \cap c) \cup (\neg a \cap \neg b \cap c), \\ r_0 &= 0. \end{aligned}$$

6. Opérations logiques $s_t = a_t \odot b_t$, pour $\odot \in \mathbf{B}^2 \mapsto \mathbf{B} = \{\cup, \cap, \oplus, \cdots\}$:

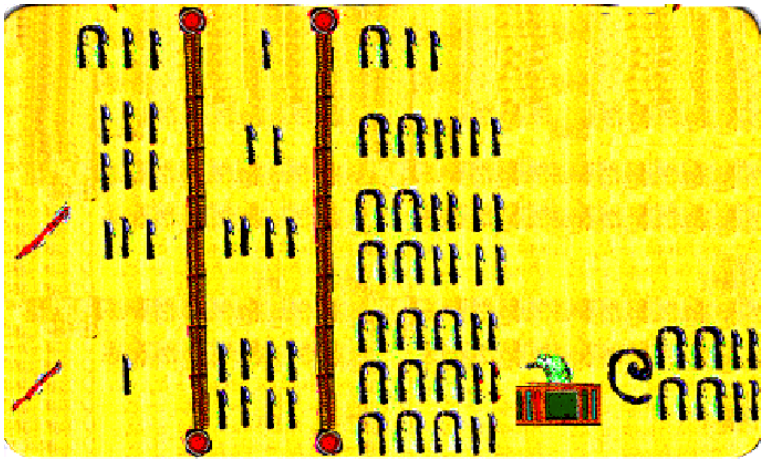
$$\begin{aligned} som(a,b,c) &\Rightarrow a \odot b, \\ ret(a,b,c) &\Rightarrow 0, \\ r_0 &= 0. \end{aligned}$$

7. Parité $s_t = a_0 \oplus a_1 \oplus \cdots \oplus a_t$:

$$\begin{aligned} som(a,b,c) &\Rightarrow a \oplus c, \\ ret(a,b,c) &\Rightarrow a \oplus c, \\ r_0 &= 0. \end{aligned}$$

On étendra la liste à sa guise : notre UAL peut calculer 2^{17} fonctions ! Ici encore, on peut transposer toutes les structures d'additionneur de la section 4.2 en des UAL de performances comparables. L'exercice en vaut la peine, car l'UAL est au cœur du microprocesseur - chapitre 5.

4.4 Multiplication



Malgré son âge vénérable (près de quatre millénaires), l'algorithme de multiplication décrit ici est très actuel. Il est utilisé systématiquement dans toutes les machines dont l'unité arithmétique et logique ne permet que le calcul direct d'additions et de décalages. C'est le cas de beaucoup de microprocesseurs modernes, qui ne disposent pas encore - en 1997 - de multiplicateur câblé.

Planche 4.2 – D'après le papyrus de Rhind, vers -1700

4.4.1 Multiplication égyptienne

Contrairement aux Babyloniens (base 60), les Egyptiens ne se servaient pas de tables pour calculer. En numération hiéroglyphe - alphabétique de base 10, dont on trouve des exemples dans les planches 2 et 4.2 - l'addition se calcule en fusionnant les représentations des nombres à ajouter, puis en propageant les retenues ; c'est l'algorithme du boulier - planches 6 et 4.3 - qui ajoute les chiffres en unaire, et les nombres (i.e. suites de chiffres) en décimal. Cet algorithme est particulièrement simple quand les deux nombres à additionner sont égaux, ce qui correspond à une multiplication par 2, dite duplication. L'opération inverse de division entière par deux, dite médiation, se réalise en partageant la représentation en deux. Le reste de la division par deux d'un nombre impair est symbolisé dans le papyrus de Rhind - planche 4.2 - par une marque oblique /. La multiplication égyptienne est clairement expliquée dans le papyrus, attribué au prêtre Ahmés. La multiplication y est réduite à une suite d'additions, duplications et médiations. Transcrivons l'exemple choisi

- planche 4.2 - de multiplication 12×12 en notation décimale :

n_2	d	n	m	p
0	1	12	12	0
0	2	6	24	0
1	4	3	48	48
1	8	1	96	144

$(R10)$

Dans cet algorithme, la colonne n_2 contient la représentation binaire 1100_2 du nombre $n = 12$, poids faibles en haut, forts en bas. Les chiffres 1 sont repérés par une marque oblique, les 0 par l'absence de marque. La colonne d contient les puissances de deux inférieures à douze. La colonne n contient les quotients entiers (12 6 3 1) de n par d. La colonne m =(12 24 48 96) contient les produits $m \times d$. La colonne p =(0 0 48 144) accumule les sommes des valeurs (0 0 $4m$ $12m$) de m correspondant aux bits à 1 de n_2. Seule figure sur le papyrus 4.2 la somme finale 144 de cette colonne.

Algorithme 7 (Multiplication égyptienne) *On obtient le produit $n \times m + p$ en calculant $\mathcal{M}(n, m, p)$ par les règles suivantes.*

$$\mathcal{M}(0, m, p) \quad \Rightarrow \quad p$$

C'est la condition de terminaison de l'algorithme. Pour $n > 0$, on distingue deux cas, suivant la parité de n :

$$\begin{aligned} \mathcal{M}(2n+1, m, p) &\Rightarrow \mathcal{M}(n, 2m, p+m) \\ \mathcal{M}(2n, m, p) &\Rightarrow \mathcal{M}(n, 2m, p) \end{aligned}$$

La quantité suivante est invariante au cours de la multiplication égyptienne.

$$n \times m + p \quad = \quad \mathcal{M}(n, m, p)$$

Suivons le déroulement de la multiplication égyptienne dans le calcul de :

$$\begin{aligned} 12 \times 12 \Rightarrow \mathcal{M}(12, &\quad 12 \quad , 0) \\ \Rightarrow \mathcal{M}(6, &\quad 24 \quad , 0) \\ \Rightarrow \mathcal{M}(3, &\quad 48 \quad , 0) \\ \Rightarrow \mathcal{M}(1, &\quad 96 \quad , 48) \\ \Rightarrow \mathcal{M}(0, &\quad 192 \quad , 144) \Rightarrow 144. \end{aligned}$$

On retrouve les trois colonnes du papyrus décimal (R10).

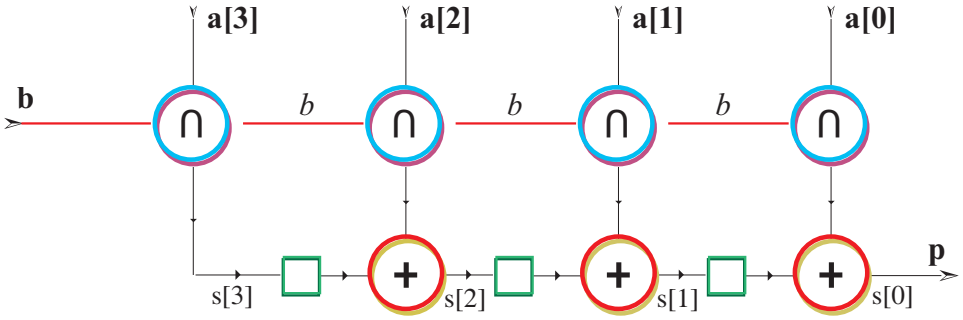


Figure 4.10 – Multiplicateur série/parallèle 4 bits

4.4.2 Multiplication série

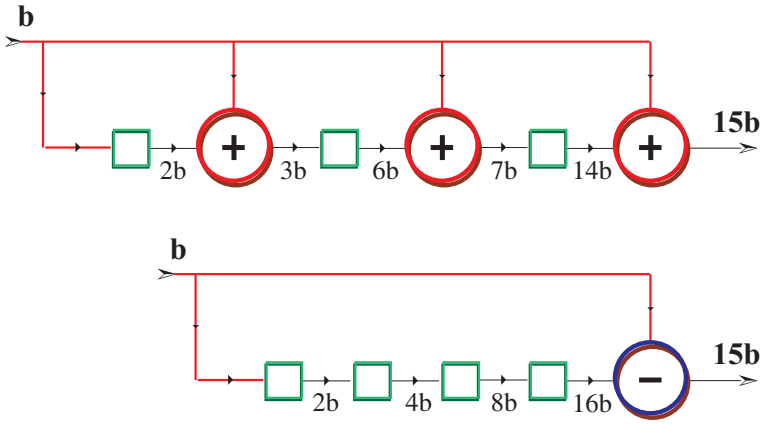
Multiplicateur série/parallèle

```
spMul(n)(a:[n],b) = p
where
  for k<n-1 do
    s[k]=(b[k]&a)+2*s[k+1]
  end for;
  s[n-1]=b[n-1]&a;
  p=s[0];
end where;
```

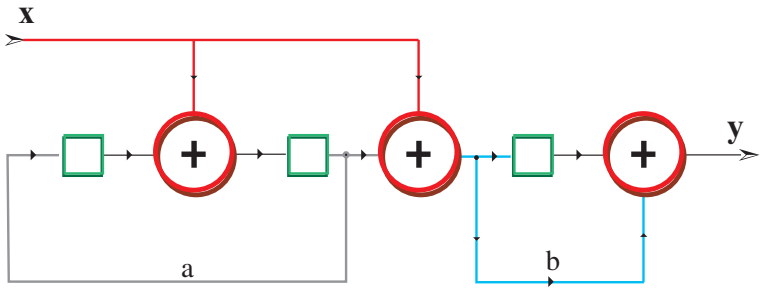
Invariant :

$$\sum_{t\in\mathbf{N}} b_t 2^t \times \sum_{k<n} a[k]2^k \quad = \quad \sum_{t\in\mathbf{N}} p_t 2^t$$

Multiplicateur par une constante Chaînes d'additions. Exemples de multiplicateurs par 15, en série.



Multiplication par 1/3 - soit $b = x/3$ - suivie d'une multiplication par 3 - soit $y = 3b$. Le résultat est l'identité $y = x$.



On peut décoder ce circuit en écrivant ses *équations*, sachant que +additionne et **reg** multiplie par deux. On trouve :

$$\begin{aligned} a &= 2 \times (x + 2 \times a), \\ b &= x + a, \\ y &= b + 2 \times b. \end{aligned}$$

La solution est bien : $a = -2x/3$, $b = x/3$ et $y = x$.

Multiplicateur série-série

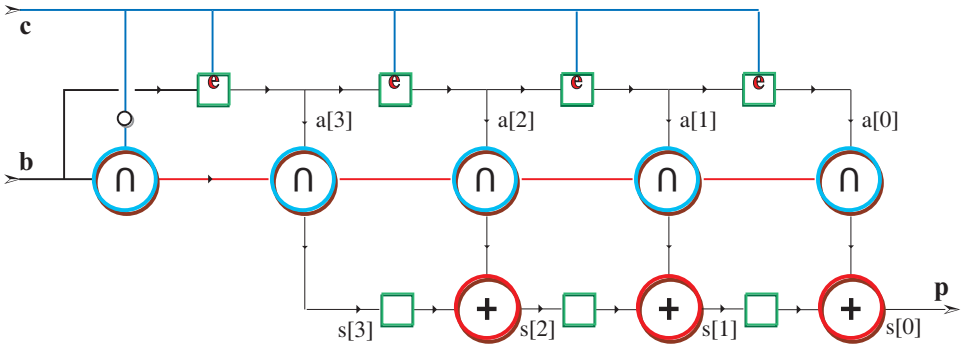


Figure 4.11 – Multiplicateur série-série 4 bits

Un multiplicateur *série/série* - figure 4.11 - est un multiplicateur série/parallèle - figure 4.10 - plus un registre à décalage, et un signal de contrôle c. Le contrôle valide le décalage du registre pour $c = 1$ et met le multiplicande $b \cap \neg c$ à zéro ; pour $c = 1$, le registre est fixe, et le multiplicande connecté à l'entrée b. Soient $A = [a_{n-1} \cdots a_0]_2$, $B = [b_{m-1} \cdots b_0]_2$, et $S = [s_{n+m-1} \cdots s_0]_2$, trois entiers binaires de n, m et $n+m$ bits respectivement. On calcule le produit $P = A \times B + S = [p_{n+m} \cdots p_0]_2$, sur $n+m+1$ bits en $2n+m+1$ cycles d'horloge, en opérant comme suit.

1. Entrée série, pour $t = 0, \ldots, n-1$, des n bits du multiplicande A et des n bits de poids faibles de S :

$$\begin{array}{l} s_{n-1} \cdots s_t \quad \rightarrow \\ a_{n-1} \cdots a_t \quad {}^{p}\rightarrow \end{array} \boxed{\begin{array}{|l|} \hline 0000 \cdots 0 * * * * \\ \hline s_{t-1} \cdots s_0^t * * * * \\ \hline a_{t-1} \cdots a_0^t * * * * \\ \hline \end{array}} \rightarrow$$

   Pendant ces n phases d'horloge, le contrôle c vaut 1.

2. Le contrôle c passe à 0 pour les m cycles suivants $t = n, \cdots, n+m-1$. Les m bits du multiplieur B arrivent en série par l'entrée a, poids faibles en tête. Dans le même temps, les m bits restant de S arrivent en série par l'entrée s, et les m bits de poids faibles du produit P apparaissent en série sur la sortie p :

$$\begin{array}{l} s_{m-1} \cdots s_{t+1} \quad \rightarrow \\ b_{m-1} \cdots b_{t+1} \quad \rightarrow \end{array} \boxed{\begin{array}{|l|} \hline r_n^t \cdots r_0^t \\ \hline s_n^t \cdots s_0^t \\ \hline a_{n-1} \cdots a_0 \\ \hline \end{array}} \rightarrow \quad p_{t-1} \cdots p_0$$

3. Après épuisement des m bits de B, on continue, de la même manière, la multiplication pendant $n+1$ cycles, avec $b_{m+t} = 0$. Au temps $t = 2n + m + 1$, les $n + m + 1$ derniers bits observés sur la sortie p représentent le produit $A \times B + S$ en binaire.

Remarquons que l'on peut faire chevaucher la phase 1 de la multiplication suivante avec la phase 3 de la multiplication courante, à partir du temps $t = n + m + 1$. On fait rentrer les n bits d'un nouvel opérande A', S' pendant que l'on vide les n bits de poids forts du produit courant. Ceci permet de réduire à $n + m + 1$ le nombre de cycles nécessaires au calcul d'un produit P de $n + m + 1$ bits, dans un traitement de multiplications à la chaîne. Pour le multiplicateur série/série, la relation :

$$(R^t + S^t) \times 2^{t-n} + \sum_{0 \leq i \leq t-n} p_i 2^i \quad = \quad A \times (\sum_{0 \leq i \leq t-n} b_i 2^i) + \sum_{0 \leq i \leq t} s_i 2^i$$

est invariante pour $n \leq t < n + m$. A l'instant $t = n + m - 1$, il vient :

$$A \times B + S = (R + S)2^{m-1} + \sum_{0 \leq i < m} p^i 2^i.$$

Exemple 1 *Calculons le produit* $6 \times 11 = 66$ *- en binaire :* $A = 110_2 \times B = 1011_2 = P = 1000010_2$ *- au moyen du multiplicateur série/série, dont on trace l'état interne après chargement (en 3 cycles) du multiplicande* A *:*

t	B	A	R	S	P
2	1011	110	000	000	
3	101	110	000	011	0
4	10	110	010	010	10
5	1	110	010	000	010
6	0	110	010	010	0010
7	0	110	010	000	00010
8	0	110	000	001	000010
9	0	110	000	000	100010

4.4.3 Multiplicateur parallèle

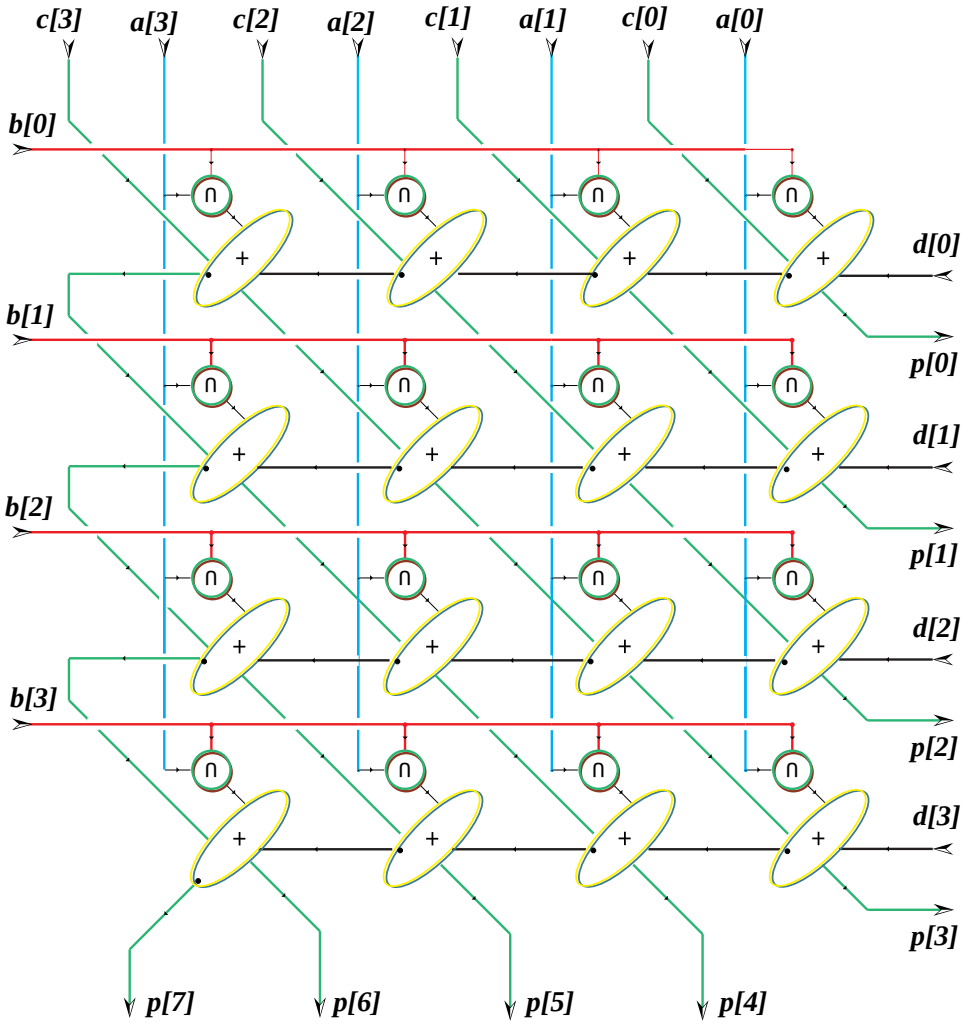
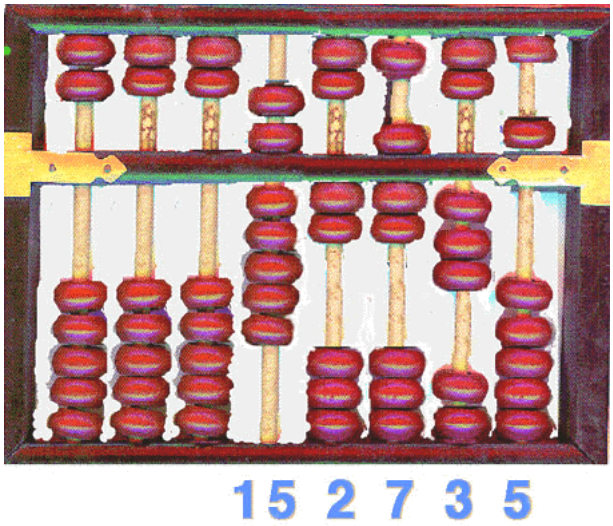


Figure 4.12 – Multiplicateur en matrice

Un multiplicateur série déroule l'algorithme 7 dans le temps, un multiplicateur parallèle le déroule dans l'espace. Le *multiplicateur en matrice* $n \times n$ est la structure régulière de **and** et **abc** de la figure 4.12. Sa fonction est de calculer

$$P = A \times B + C + D,$$

expression dans laquelle A, B, C, D sont des entiers présentés en entrée sur n bits, et P est la sortie produit sur $2n$ bits.



Le boulier est un système décimal de position, où la valeur des chiffres va de 0 à 15, pour faciliter les reports de retenues. Quand elles sont proches de la barre centrale, les billes du haut valent 5, celles du bas 1. Sinon, elles valent 0. La configuration de cette image représente le nombre

$$[0 \cdots 0\,15\,2\,7\,3\,5]_{10} = 152735_{10}.$$

Le boulier permet (entre autres) une addition avec accumulation, des poids forts vers les poids faibles, dans laquelle les reports de retenues sont limités à une seule colonne. Manié avec toute la dextérité humaine, cet outil de calcul, rustique mais ergonomique reste souvent plus rapide que ses concurrents électroniques, pour diverses tâches utiles.

Planche 4.3 – Le boulier chinois.

4.4.4 Addition sans retenue

L'introduction des numérations redondantes est nécessaire pour deux raisons, en apparence d'origines fort différentes :

1. Les représentations redondantes sont les seuls systèmes connus conduisant à une addition en *temps parallèle constant*. Nous montrons dans la proposition 4 qu'aucun système non redondant ne possède cette propriété. L'addition en temps constant est la clé des multiplicateurs rapides qui suivent.
2. Nous verrons au chapitre 6 que l'écriture d'un réel calculable dans une représentation non redondante *n'est pas calculable* ! Le recours à des systèmes redondants est, là aussi, nécessaire pour contourner cette difficulté fondamentale.

Nous ne traitons ici que le cas de la base $b = 2$, la plus simple et la plus utile.

Base $b = 2$, chiffres 0,1,2

Définition 10 *Les chiffres de la représentation* binaire molle *sont* $\{0, 1, 2\}$ *et la base 2.*

Représentations en binaire mou des sept premiers entiers naturels :

$$\begin{array}{rclrcl} 0 & = & {}_2 0 & 1 & = & {}_2 1 \\ 2 & = & {}_2 01 & 2 & = & {}_2 2 \\ 3 & = & {}_2 11 & 4 & = & {}_2 001 \\ 4 & = & {}_2 02 & 5 & = & {}_2 101 \\ 5 & = & {}_2 12 & 6 & = & {}_2 011 \\ 6 & = & {}_2 22 & 6 & = & {}_2 201 \end{array}$$

Ce système de représentation est *redondant* : certains entiers $(2, 4, 5, 6, 8, \cdots)$ possèdent plusieurs représentations après élimination des zéros non significatifs. Choisissons de coder un nombre A de n chiffres binaires mous par une paire d'entiers binaires $R, S \in \mathbf{B}^n$ dont A est la somme bit à bit :

$$A = {}_2[a_0 \cdots a_{n-1}] = {}_2[r_0 \cdots r_{n-1}] + {}_2[s_0 \cdots s_{n-1}] = R + S,$$

avec $a_k = r_k + s_k \in \{0, 1, 2\}$ pour $0 \leq k < n$.

Addition en délai constant

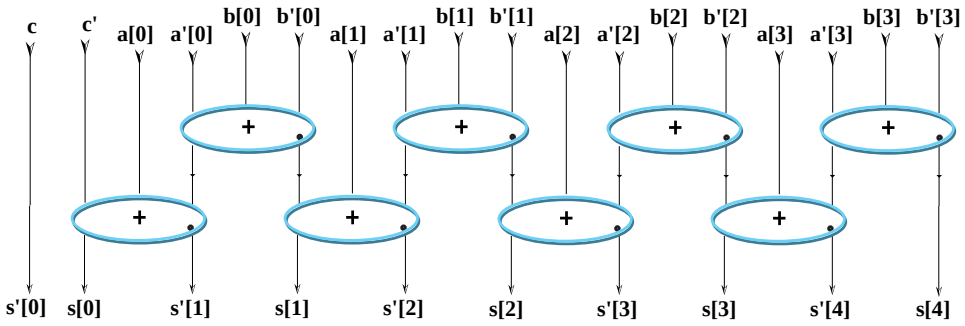


Figure 4.13 – Addition sans retenue

La proposition 4, établissant l'optimalité de l'algorithme 6 d'addition en temps logarithmique, semblait marquer la fin de notre étude de l'addition. Pourtant, l'algorithme d'addition qui suit est *en délai constant* ! Pour obtenir cela, il suffit d'accepter de représenter les sommes calculées en *binaire mou*.

Algorithme 8 *Soient $B \in \mathbf{B}^n$ un entier n bits, et A un nombre représenté par n chiffres binaires mous : $A = A' + A''$ avec $A', A'' \in \mathbf{B}^n$. On calcule la somme $C = A + B$ en binaire mou par $C = C' + C''$ avec $C', C'' \in \mathbf{B}^{n+1}$ en résolvant,*

au moyen de n additionneurs binaires complets abc, qui opèrent en parallèle pour $0 \leq k < n$, les équations :

$$a'_k + a''_k + b_k \Rightarrow c'_k + 2c''_{k+1},$$

dont on calcule l'unique solution booléenne $c'_k, c''_k \in \mathbf{B}$. On pose alors $c_k = c'_k + c''_k$ pour $0 \leq k < n$ et $c''_0 = c'_n = 0$.

Exemple 2 *Calculons la somme $19 + 23 + 13 = 55$ par l'algorithme 8 :*

	19	010011_2
+	*23*	010111_2
+	*13*	001101_2
=	*R*	101110_2
+	*S*	001001_2
=	*55*	102111_2

// Combinational CarrySave Adder A+B+C+D+s0+r0+t0 = S+R

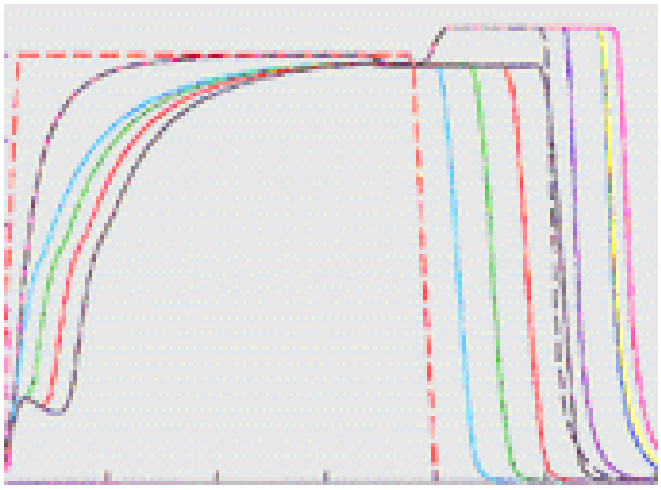
// $A = sum_{k<n} a[k]2^k$, $B = sum_{k<n} b[k]2^k$

// $C = sum_{k<n} a[k]2^k$, $D = sum_{k<n} b[k]2^k$

// $S = sum_{k<n+1} s[k]2^k$, $R = sum_{k<n+1} r[k]2^k$

```
csAdd(n)({a,a',b,b'}:[n],r0,s0)
      =(s,s'):([n+1],[n+1])
where
  f[0]=s0;
  r[0]=r0;
  for k<n do
    (e[k],f[k+1])=abc(a[k],a'[k],b[k]);
    (s[k],s'[k+1])=abc(b'[k],e[k],f[k]);
  end for;
  s[n]=f[n]
end where;
```

4.4.5 Multiplication par dichotomie



Sur ces traces d'oscilloscope, on mesure la vitesse d'établissement de divers signaux lors du calcul d'un produit par le multiplicateur logarithmique 32 bits de la planche 4.5. Le véritable calcul débute à l'arrivée de l'horloge, une fois tous les signaux pré-chargés à 3 Volt. Les signaux mesurés sont situés aux divers niveaux de la structure en arbre binaire de l'algorithme 9. L'écart mesuré entre les traces est bien constant (de l'ordre de 3ns), ce qui correspond à la théorie : quand on double la taille du multiplicateur par dichotomie, son délai augmente de cette même constante - ici, 3ns de plus pour le produit 64b que pour celui sur 32b ; la surface du multiplicateur 64b est (un peu) plus de quatre fois celle du multiplicateur 32b.

Planche 4.4 – Mesures de vitesse

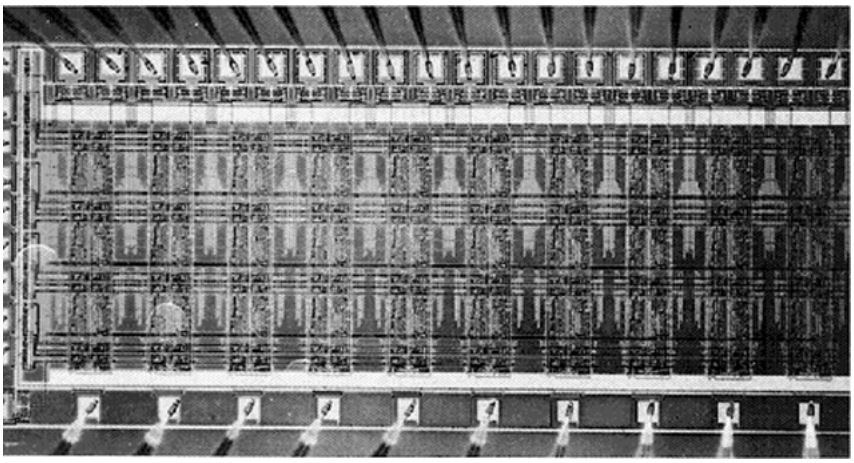
Décrivons maintenant un multiplicateur en *temps logarithmique.* Pour cela, on rappelle qu'il est possible d'additionner en *délai constant* dans le format *binaire mou* avec l'additionneur $csAdd$ de la section 4.13.

Algorithme 9 (Multiplicateur logarithmique)

Soient à multiplier deux entiers binaires $A = {}_2[a_0 \cdots a_{n-1}]$ *et*

$B = {}_2[b_0 \cdots b_{n-1}]$ *de longueurs égales à* $n = 2^{p+1}$ *bits :*

1. *Calculons, en parallèle, les* n^2 *produits bit à bit de* A *par* B*. Disposons, pour* $0 \leq i, j < n$*, chaque produit* $a_i \times b_j$ *dans la colonne* $c = i + j$ *et la ligne* $l = j \div 2$ *d'une matrice* $P^0[l, c]$ *de taille* $2n \times \frac{n}{2}$*. En convenant que les* a_i, b_j *sont nuls pour des indices hors de l'intervalle* $[0 \cdot\cdot n-1]$*, nous pouvons écrire :*



Circuit expérimental, pour tester la vitesse d'un multiplicateur par dichotomie dérivé de l'algorithme 9. La spécification logique de ce circuit est le code ${}_2\mathbf{Z}$ du dernier paragraphe. Sa géométrie réside dans cette image, ou l'on voit la structure d'arbre binaire dans les pistes métal entre les cellules. Les entrées sont en haut et à gauche, les sorties en bas. Les signaux testés figurent sur la planche 4.4.

Planche 4.5 – Multiplicateur par dichotomie

$$P^0[l,c] = a_{c-2l} \times b_{2l} + a_{c-1-2l} \times b_{1+2l},$$

de façon que $P = A \times B = \sum_{0 \leq l < \frac{n}{2}} \sum_{0 \leq c < 2n} P^0[l,c]2^c$. *Chaque ligne* l *de la matrice* P *représente ainsi, en binaire mou, le nombre :*

$$P^0_l \;=\; \sum_{0 \leq c < 2n} P^0[l,c]2^c \;=\; 2^l(A \times b_{2l} + 2A \times b_{2l+1}).$$

2. *La matrice suivante* P^1 *s'obtient en sommant, deux à deux et en parallèle, les lignes* $2l, 2l+1$ *de* P^0 *au moyen de* $n/4$ *additionneurs* $csAdd(2n)$. *On a toujours :*

$$P = A \times B = \sum_{0 \leq l < \frac{n}{4}} \sum_{0 \leq c < 2n} P^1[l,c]2^c.$$

En général, la matrice P^{k+1} *s'obtient, pour* $0 \leq k < n-1$, *de la même façon, en sommant deux à deux et en parallèle, les lignes* $2l, 2l+1$ *de* P^k *à l'aide de* $n/2^{k+2}$ *additionneurs* $csAdd(2n)$. *Cette construction termine en* $p = |n|_2 - 1$ *étages sur un vecteur* P^p *de* $2n$ *chiffres mous tel que :*

$$P = A \times B = \sum_{0 \leq c < 2n} P^p[0,c]2^c.$$

3. *Il ne reste plus qu'à convertir notre produit P^p, sur $2n$ chiffres en binaire mou, dans le résultat final $P = {}_2[p_0 \cdots p_{2n-1}]$. On utilise pour cela l'additionneur par dichotomie, qui effectue ce calcul en délai $|n|_2\tau + 3$.*

Proposition 5 *Un produit sur $2n$ bits se calcule, par l'algorithme 9, en délai :*

$$T_\times(n) \leq (2\tau(\mathbf{abc}) + \tau(\mathbf{mux}) + 2)|n|_2.$$

Exemple 3 *Calculons, au moyen de l'algorithme 9, le produit $A \times B = P$ avec $A = 35 = 100011_2$ et $B = 183 = 10110111_2$:*

$A \times b_i$	P^0	P^1	P^2
0000000100011_2			
	0000001100121_2		
0000001000110_2			
		0000011110101_2	
0000010001100_2			
	0000010001100_2		
0000000000000_2			
			1011211200101_2
0001000110000_2			
	0011001210000_2		
0010001100000_2			
		1011120010000_2	
0000000000000_2			
	1000110000000_2		
1000110000000_2			

On trouve bel et bien $P = 1011211200101_2 = 1100100000101_2 = 6405$.

Chapitre 5

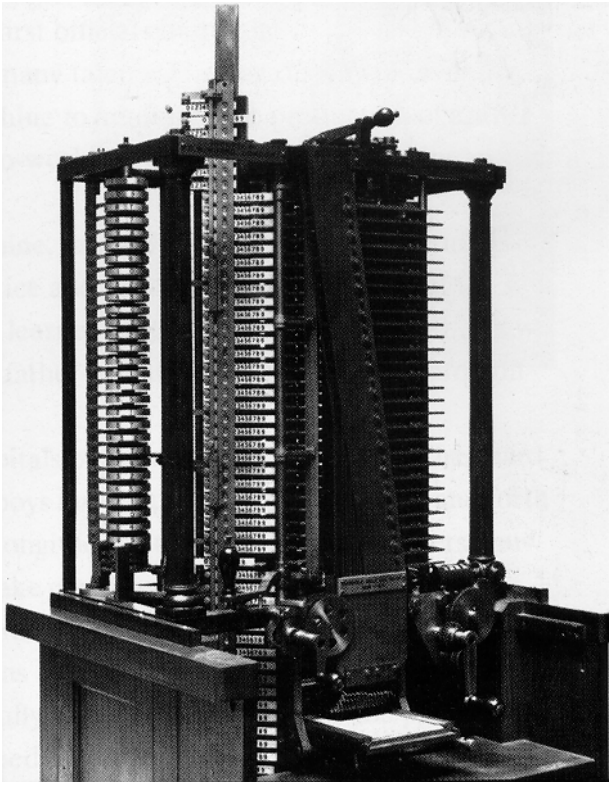
Machines universelles

Contents

The Analytical Engine consists of two parts :

1. The *store* in which all the variables to be operated upon, as well as all those quantities which ave arisen from the result of other operations, are placed.
2. The *mill* into wich the quantities about to be operated upon are always brought.

Charles Babbage



Ce modèle partiel de l'*Analytical Engine* de Babbage est réalisé par son fils Henry en 1910, trente neuf ans après la mort du père. Le modèle calcule en démonstration 20 décimales du nombre $\pi = 3 \cdot 14159265358979323846 \cdots$; on trouvera par la suite des erreurs dans son fonctionnement.

Planche 5.1 – L'*Analytical Engine*

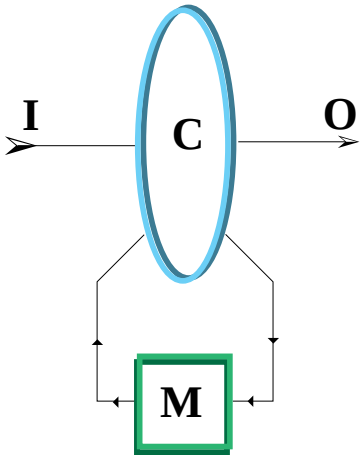
Charles Babbage

Alan Turing

Planche 5.2 – Deux précurseurs anglais : Babbage et Turing

5.1 Machine de Turing

Les machines que Turing étudie dans son papier de 1936 sont très générales. On peut leur donner le schéma de principe suivant.



On y retrouve les trois composants de tout système d'information : *communication* (les entrées I et les sorties O), *mémoire* (les registres M) et *calcul* (la logique combinatoire C).

A ce titre, la classe des machines de Turing comprend tous les circuits CDS. Elle est donc au moins aussi générale que tout ce qui est traité ici. La thèse de Church et Turing - chapitre 6 - nous justifie d'en rester là ; l'étude des circuits

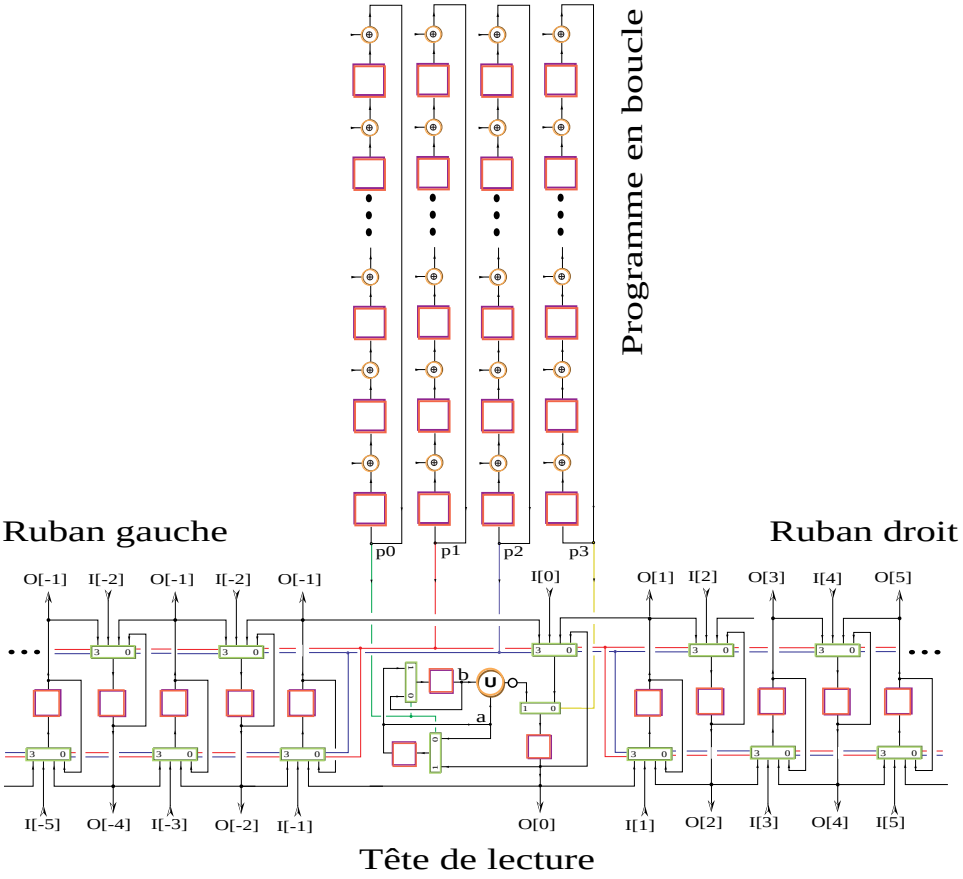


Figure 5.1 – Machine de Turing universelle

CDS (finis et infinis) se révèle à son tour au moins aussi générale que celle de tout modèle mathématique du calcul.

5.1.1 Circuit universel en série

Reprenons ici notre étude des circuits mathématiques CDS, en suivant le plan et les motivations de Turing.

Réduisons chacun des composants de la machine de principe ci-dessus à sa plus simple expression, sans pour autant perdre en généralité. L'objectif est de définir un circuit CDS particulier $MTU \in \mathcal{C}_{ds}$ - figure 5.1 - ayant la propriété qui suit.

Théorème 3 (Machine de Turing universelle)

Il existe un circuit (infini) $MTU \in \mathcal{C}_{ds}$ *qui est capable,* à lui seul *et après une programmation initiale finie, de reproduire - cycle par cycle en entrée, avec un retard d'un cycle en sortie - le comportement de* tout *circuit CDS fini donné.*

Preuve : Soit $g \in \mathcal{C}_{ds}$ le circuit CDS arbitraire de taille *finie*, que l'on veut simuler au moyen de la machine MTU définie dans la section suivante. Résumons d'abord la méthode de simulation.

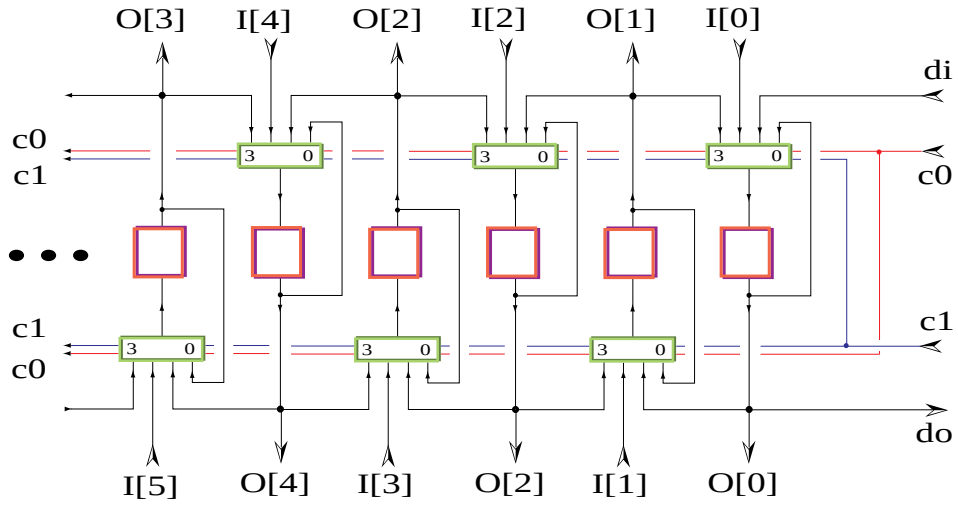
1. On commence par *compiler* le circuit g en un *programme* fini, représenté par le nombre entier $p = p(g) \in \mathbf{N}$. Cette traduction est calculable en temps fini par tout ordinateur doté de suffisamment de mémoire, ou par MTU elle-même, préalablement programmée pour cela.
2. Partant de la représentation binaire du nombre $p = {}_2p_0p_1 \cdots p_{k-1}$ sur $k = |p|_2$ bits, on construit - toujours de manière automatique - le circuit $\mathcal{P} = \mathcal{P}(p) \in \mathcal{C}_{ds}$ de la *boucle de programme*, suivant les recettes du paragraphe 5.1.2.
3. On branche alors les 4 bits de sortie de $\mathcal{P}$ sur les 4 bits $p[0,3]$ de l'entrée programme de MTU.

   On branche les fils d'entrées du circuit à émuler sur autant de pattes I dans le ruban de MTU, et on connecte les sorties aux pattes O. Soit $MTU(p) \in \mathcal{C}_{ds}$ le circuit résultant. Il contient à ce point un nombre *fini* de bits non nuls dans son programme $\mathcal{P}$, et aucun sur son ruban de données. Seul un nombre fini de pattes d'entrée ou sortie est utilisé.
4. La machine $MTU(p)$ émule le circuit f, où plutôt **reg**(f) pour tenir compte du délai d'un cycle introduit par la simulation. Pour réaliser cette performance, il faut opérer $MTU(p)$ avec une fréquence de temps $t \in \mathbf{N}/f$ qui soit plus rapide que celle de la machine à simuler dans un facteur f supérieur ou égal au nombre d'instructions du programme p. Avec 4 bits par instruction, on a $f \geq |p|_{16}$. **Q.E.D.**

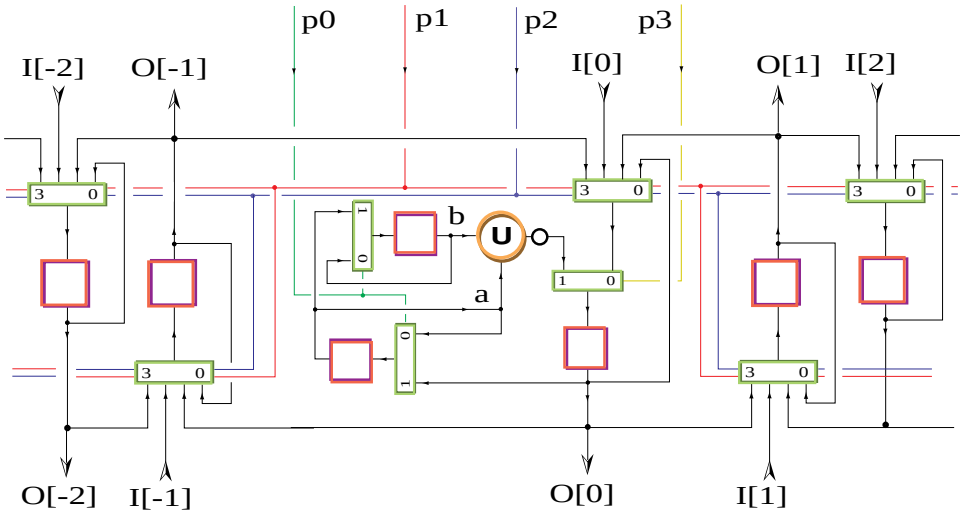
Tout ceci est facile à concevoir pour le mathématicien. Le physicien risque de rencontrer quelques difficultés à réaliser strictement le programme de Turing. Nous y reviendrons.

5.1.2 Une machine universelle

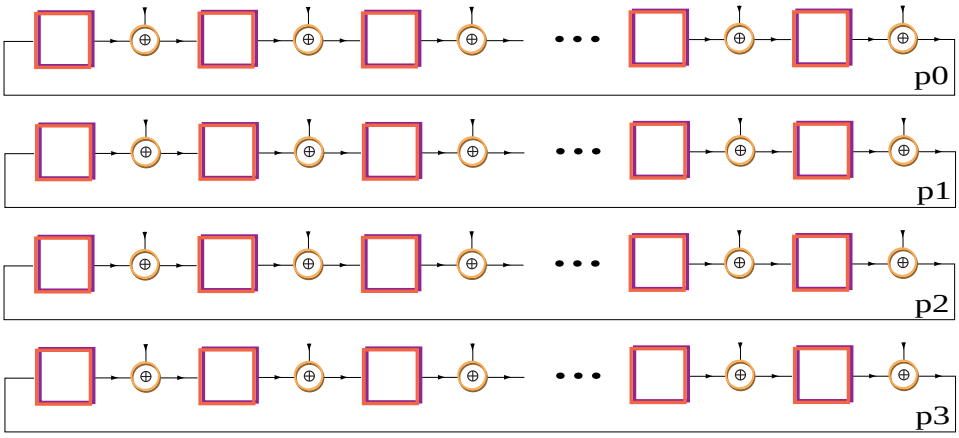
Ruban des données



Tête de lecture



Boucle du programme



Instructions de la machine de Turing de la figure 5.1.

Push Décale les deux registres $a, b : a \Rightarrow b$; $m[0] \Rightarrow a$. Code : $p[0,3] = {}_2 1000$.

Nor Opération : **nor**$(a, b) \Rightarrow m[0]$.
Code : $p[0,3] = {}_2 0001$.

Left Décale la *tête de lecture* d'une case à gauche : $m[k-1] \Rightarrow m[k]$, pour tout $k \in \mathbf{Z}$.
Code : $p[0,3] = {}_2 0110$.

Right Décale la *tête de lecture* d'une case à droite : $m[k+1] \Rightarrow m[k]$, pour tout $k \in \mathbf{Z}$.
Code : $p[0,3] = {}_2 0100$.

In Lit les entrées et les range en mémoire : $I[k] \Rightarrow m[k]$ pour tout k dans l'ensemble fini des pattes I initialement connectées. Une mémoire dont la patte d'entrée n'est pas connectée à l'extérieur garde sa valeur.
Code : $p[0,3] = {}_2 0010$.

John von Neumann (1903-1957) et son ordinateur, à l'*Institute for Advanced Studies* de Princeton, en 1949.

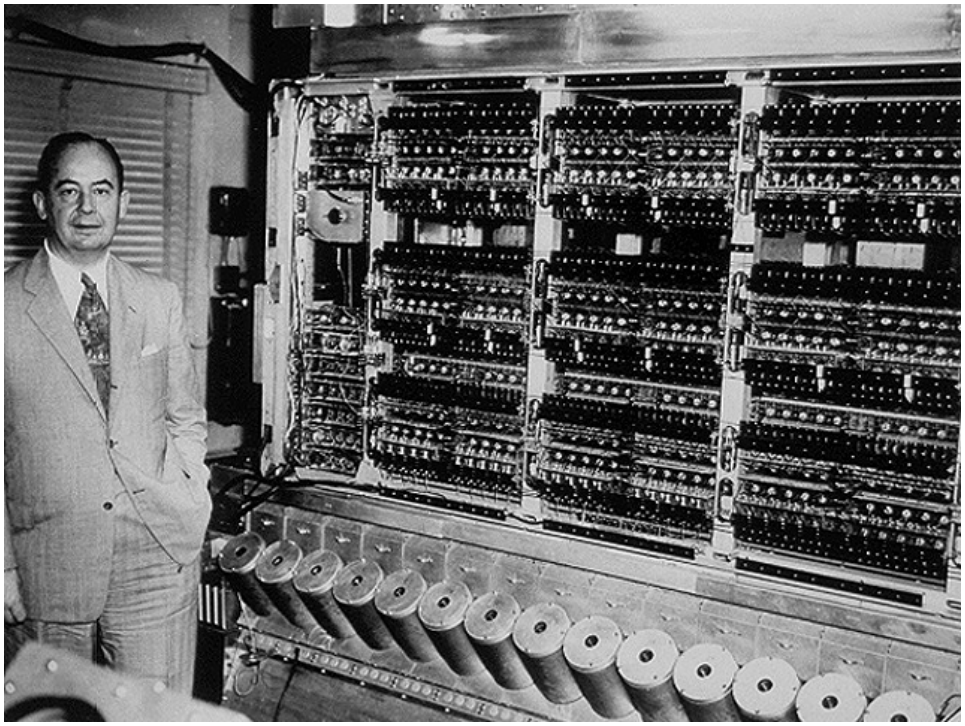


Planche 5.3 – von Neumann

5.2 Microprocesseur

On concevra un microprocesseur, aussi simple que possible, capable d'effectuer le calcul de la montre, étendu au calendrier.

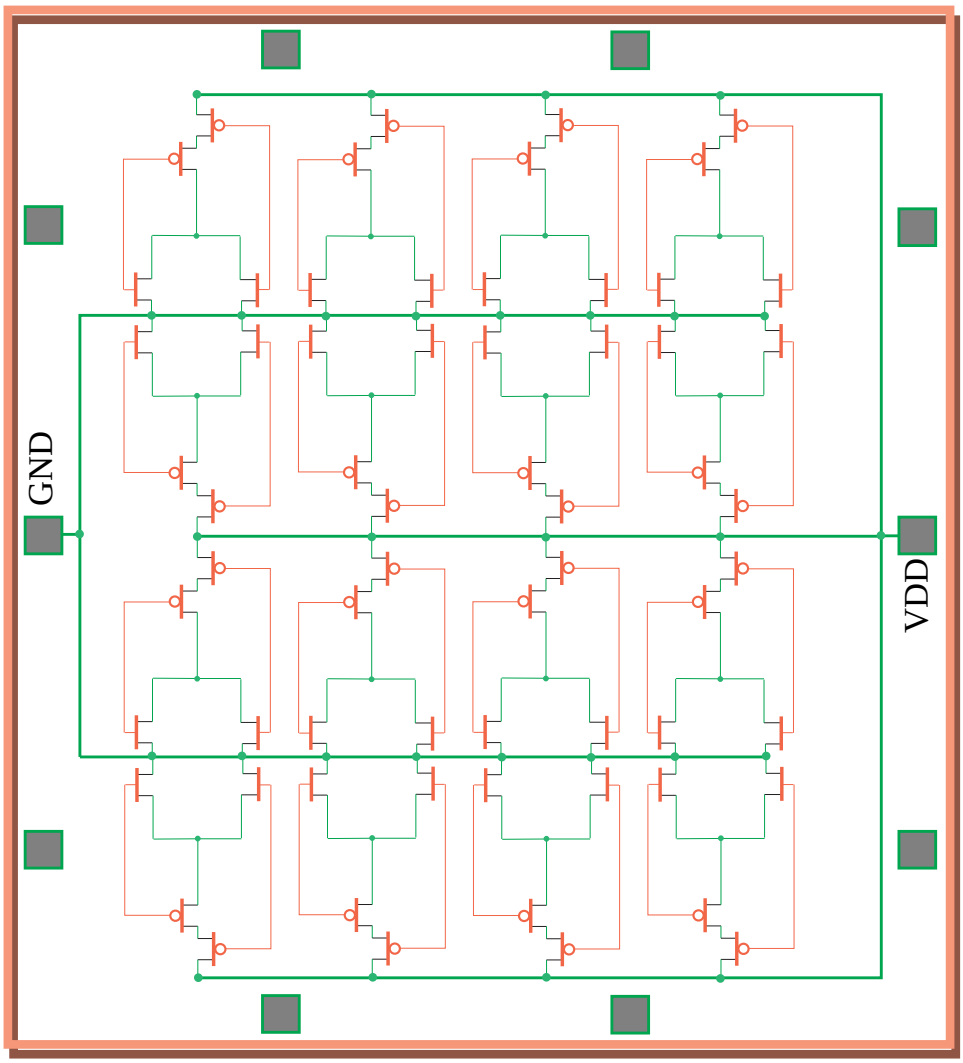
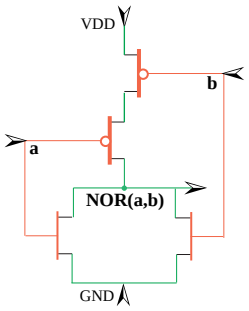


Figure 5.2 – Circuit pré-diffusé de 4×4 **nor**

5.3 Machines parallèles

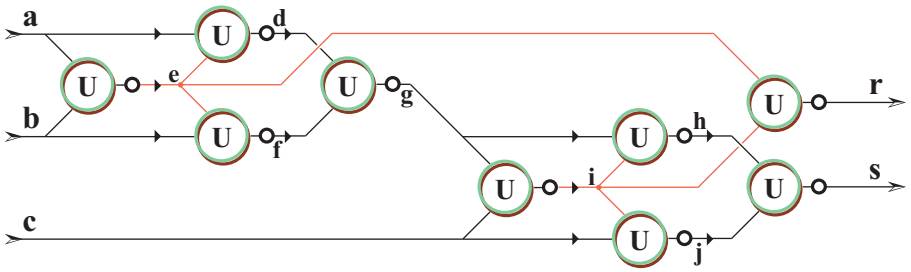
5.3.1 Circuit pré-diffusé



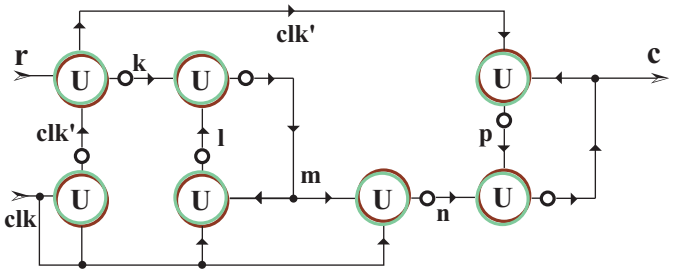
La base **nor** est pénible à manipuler, surtout par les humains. Elle reste pourtant importante, car il suffit de 4 transistors cMOS pour sa réalisation digitale. La porte **nor** est *petite*, donc *rapide*. Ensuite, elle forme - à elle *seule* - une base de l'algèbre de Boole.

Certains circuits *pré-diffusés* sont des matrices denses de portes **nor** (figure 5.2). Les couches de fabrication - jusqu'au métal non compris - sont préalablement diffusées sur des composants identiques tirés en très grande série. La métallisation est ensuite traitées sur des lots spécifiques à chaque client ; c'est la structure des connexions métalliques (spécifiée par l'utilisateur final) qui transforme le pré-diffusé en circuit spécifique - ASIC, pour *Application Specific Integrated Circuit*.

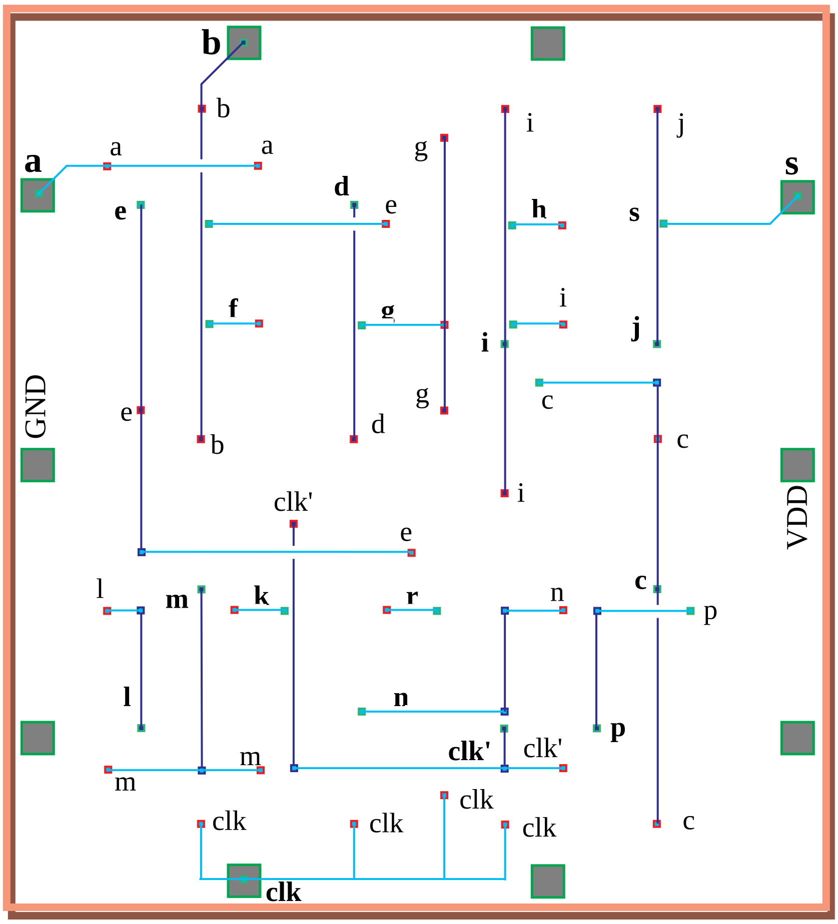
Par exemple, il suffit de 9 **nor** pour réaliser l'additionneur binaire complet **abc** suivant.



De la même façon, on réalise un registre **reg** avec 7 portes **nor** .



En combinant ces deux éléments - **abc** et **reg** - on réalise l'additionneur binaire en série de la figure 4.2. Après placement des variables dans la matrice de **nor** , puis routage (en métal) des connexions, on trouve les plans de la figure 5.3 pour cet additionneur binaire en série - circuit complet. On trouve avant une vue de la couche de métal qu'il convient d'ajouter au pré-diffusé de la figure 5.2 pour en faire l'ASIC de la figure 5.3.



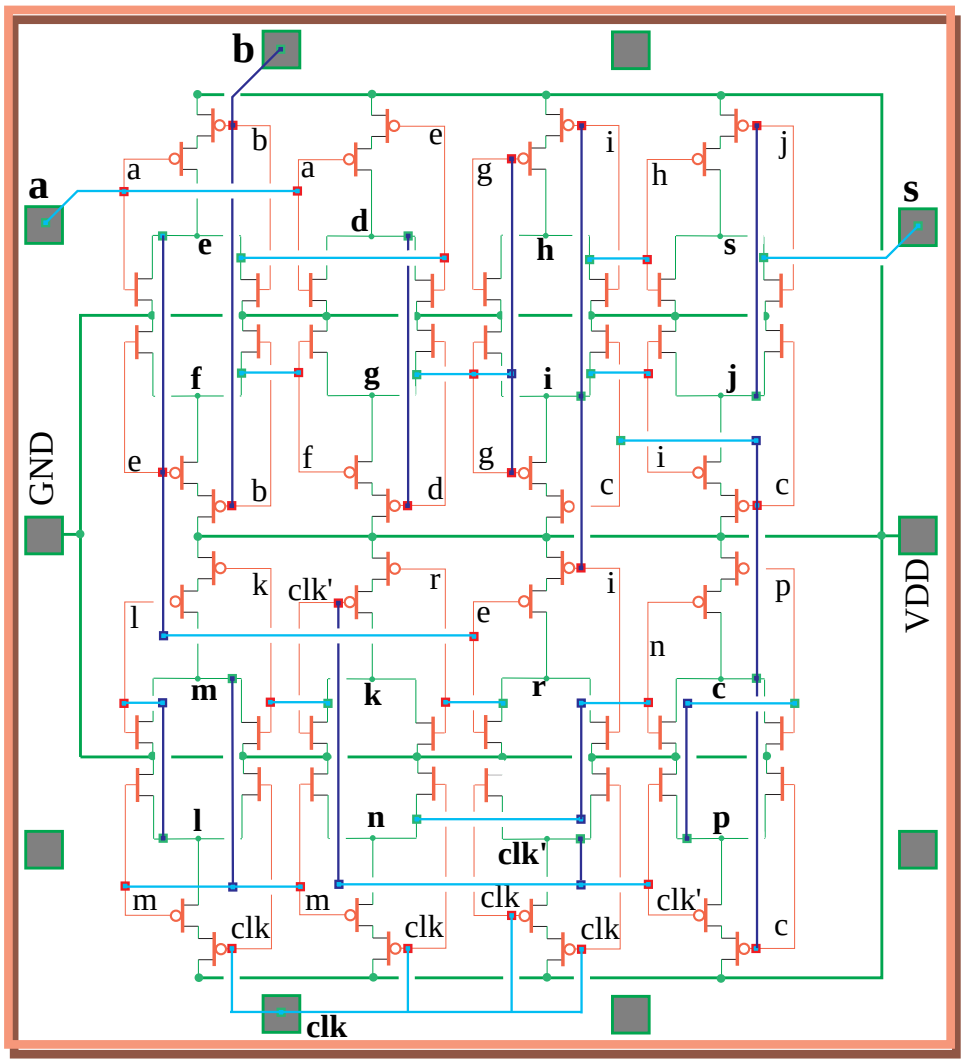


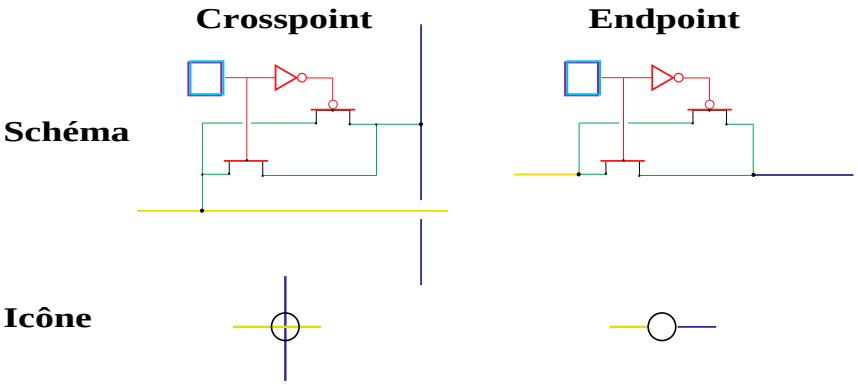
Figure 5.3 – Additionneur série, en pré-diffusé

5.3.2 Logique programmable

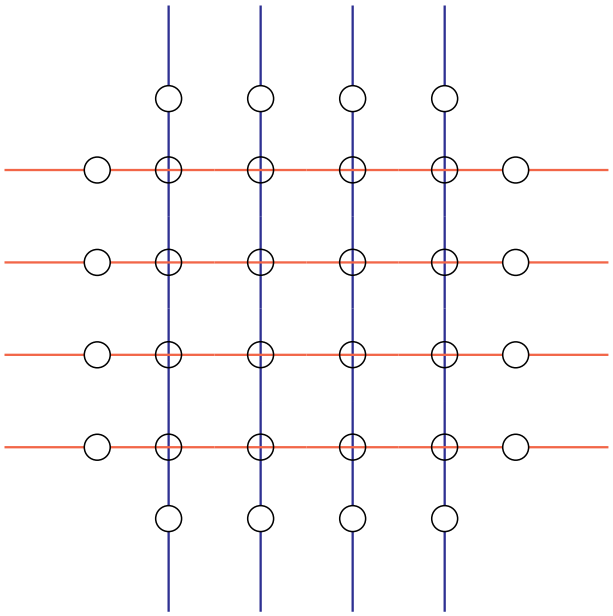
Machine universelle parallèle.

Connexion programmable

Deux points d'interconnexion programmable.



Matrice d'interconnexion programmable, avec 32 bits de configurations.

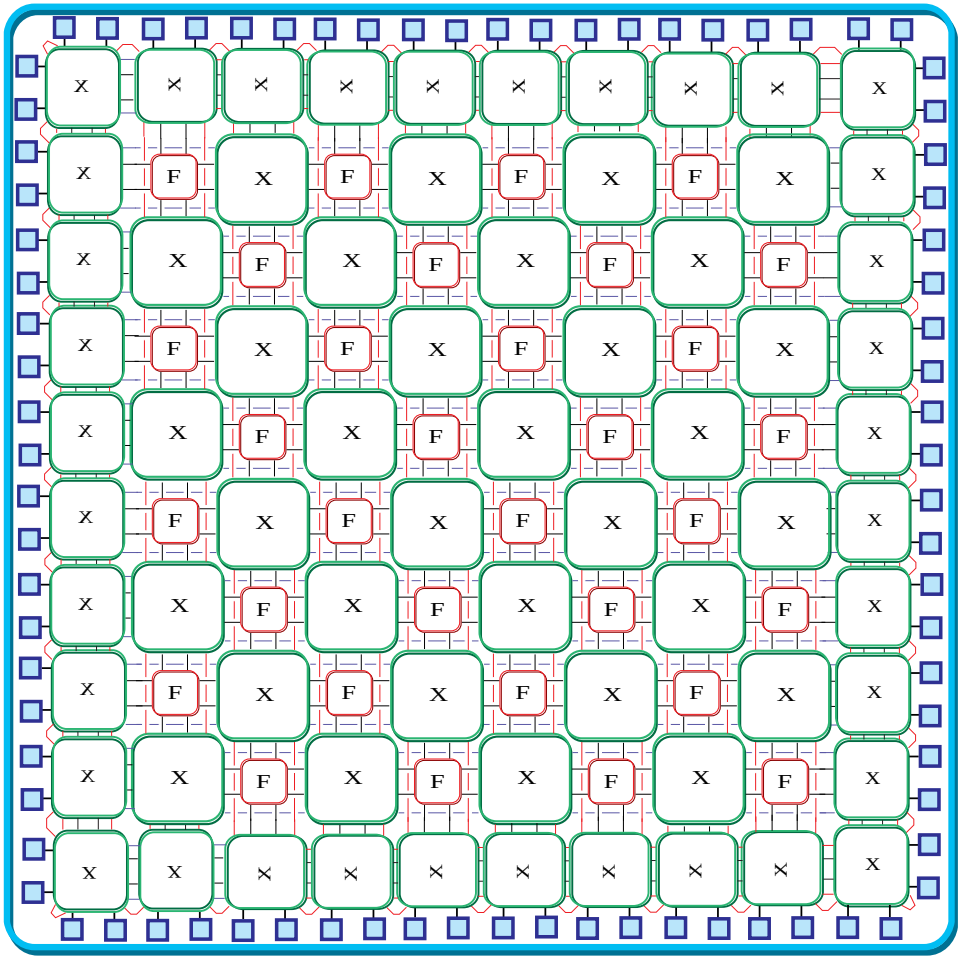


Boite $X(4,4)$ de routage - *switchbox* - sur 4×4=16 fils.

Fonction programmable

Porte universelle $\mathbf{B}^4 \mapsto \mathbf{B}^4$ - dite *look-up table* $4 \times LUT(4)$ - réalisée par une mémoire RAM de 16×4 bits. Sur chaque sortie, on place un registre optionnel, avec un multiplexeur en sortie qui permet de choisir entre la sortie directe et celle avec registre, sous contrôle d'un bit de configuration par sortie. Au total, on a 64+4=68 bits de configuration par fonction F.

5.3.3 *Field Programmable Gate Array* : FPGA



5.3.4 Machine parallèle universelle

Un FPGA infini est une machine *universelle* parallèle $MPU \in \mathcal{C}_{ds}$; le circuit MPU est capable, au même titre que la machine de Turing universelle MTU, de simuler tout circuit CDS fini donné. L'émulation se fait cette fois ci en *temps réel*, et la machine MTU opère à la fréquence du circuit simulé ; il n'y a pas ici de retard sur les sorties.

5.4 Programmation

5.4.1 Programme séquentiel

Emulateur de circuits

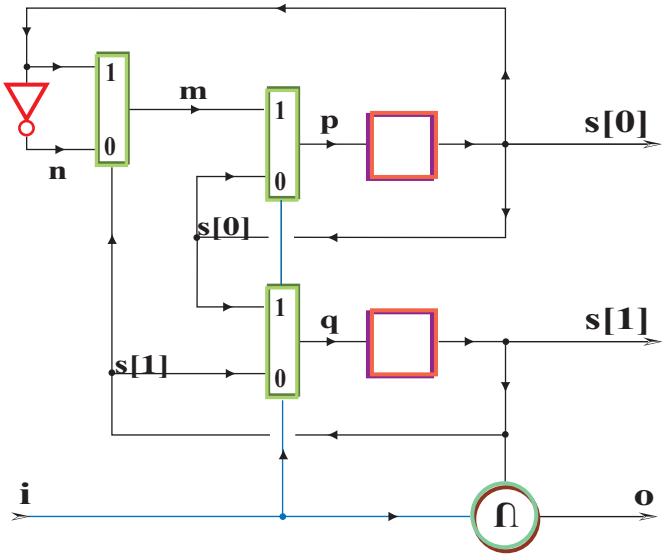


Figure 5.4 – Compteur modulo 3

Pour se convaincre de l'universalité de la machine MTU - figure 5.1 - montrons qu'elle est capable de simuler le compteur modulo 3 de la figure 5.4, et dont voici le code ${}_2\mathbf{Z}$.

```
CM3(i) = (s:[2],o)
where
  s[0]=reg p;
  s[1]=reg q;
  n=not s[0];
  q=mux(i,s[0],s[1]);
  m=mux(s[1],s[0],n);
  p=mux(i,m,s[0]);
  o=mux(i,s[1],i);
end where;
```

Pour éviter d'avoir à traduire ce circuit de l'algèbre (**not**, **mux**, **reg**) dans l'algèbre (**nor**, **reg**), changeons la structure de l'unité opératoire afin de la rendre capable des opérations **mux** et **not** , plutôt que **nor** - figure 5.5. Les mnémoniques des instructions de cette variante de MTU sont :

L,R,I,P,M,N pour *left, right, input, push, mux, not*.

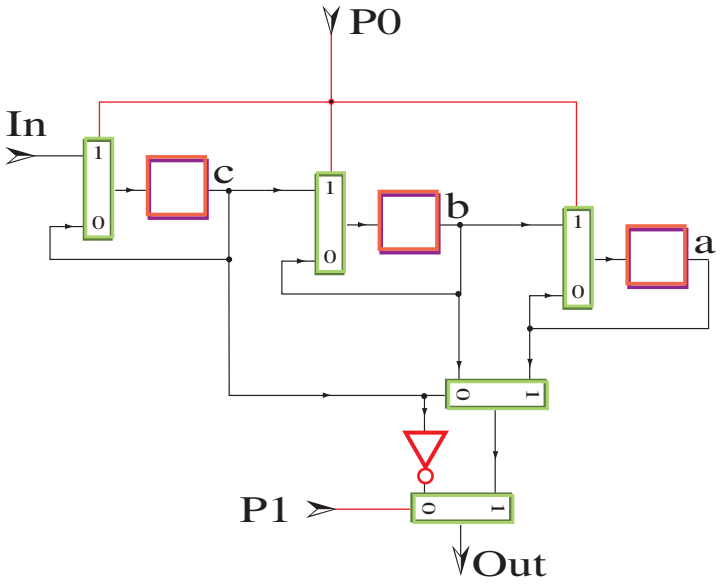


Figure 5.5 – Unité opératoire **mux** et **not** pour MTU

Leur signification est détaillée au paragraphe 5.1.2. On trouve alors le programme en boucle suivant, pour simuler le compteur $CM3$ modulo 3 sur la machine que Turing MTU.

$$(PRPRNRIPRMLLPLPLPRRMLPRPRPRRMLPRRRPORPNPLLLLN).$$

L'exécution de ce programme donne est présentée en figure 5.6.

Exercice 3 *Dresser la table de vérité du décodeur d'instructions de la machine de Turing - figure 5.1 - pour les 7 instructions, à l'exclusion de Load.*

Exercice 4 *Trouver une structure de registre à décalage cyclique à longueur variable pour implanter la partie instructions de la machine de Turing - figure 5.1. Donner une implantation complète de l'instruction Load de chargement du programme - par exemple celui de $CM3$ - compatible avec votre représentation du registre à décalage d'instructions.*

De la montre au calendrier

On programmera - sur le microprocesseur conçu pour cela - la montre du chapitre 1, avec calcul de la date et affichage du calendrier français, y compris les fêtes catholiques dérivées de la date de Pâques.

	Pile	Mémoire
	***	[s1] s0**
Push	s1 **	[s1] s0**
Right	s1 **	s1 [s0] **
Push	s0 s1 *	s1 [s0] **
Right	s0 s1 *	s1s0 [*] *
Not	s0 s1 *	s1s0 [n] *
Right	s0 s1 *	s1s0n [*] *
In	s0 s1 *	s1s0n [i] *
Push	i s0 s1	s1s0n [i] *
Right	i s0 s1	s1s0ni [*]
Mux	i s0 s1	s1s0ni [q]
Left	i s0 s1	s1s0n [i] q
Left	i s0 s1	s1s0 [n] iq
Push	n i s0	s1s0 [n] iq
Left	n i s0	s1 [s0] niq
Push	s0 n i	s1 [s0] niq
Left	s0 n i	[s1] s0niq
Push	s1 s0 n	[s1] s0niq
Right	s1 s0 n	s1 [s0] niq
Right	s1 s0 n	s1s0 [n] iq
Mux	s1 s0 n	s1s0 [m] iq
Left	s1 s0 n	s1 [s0] miq
Push	s0 s1 s0	s1 [s0] miq
Right	s0 s1 s0	s1s0 [m] iq

	Pile	Mémoire
	s0 s1 s0	s1s0 [m] iq
Push	m s0 s1	s1s0 [m] iq
Right	m s0 s1	s1s0m [i] q
Push	i m s0	s1s0m [i] q
Right	i m s0	s1s0 [m] iq
Right	i m s0	s1 [s0] miq
Mux	i m s0	s1 [p] miq
Left	i m s0	[s1] pmiq
Push	s1 i m	[s1] pmiq
Right	s1 i m	s1 [p] miq
Right	s1 i m	s1p [m] iq
Right	s1 i m	s1pm [i] q
Push	i s1 i	s1pm [i] q
Out	i s1 i	s1pm [o] q
Right	i s1 i	s1pmo [q]
Push	q i s1	s1pmo [q]
Not	q i s1	s1pmo [q']
Push	q' q i	s1pmo [q']
Left	q' q i	s1pm [o] q'
Left	q' q i	s1p [m] oq'
Left	q' q i	s1 [p] moq'
Left	q' q i	[s1] pmoq'
Not	q' q i	[q] pmoq'
		[q] p ***

Figure 5.6 – Un cycle de simulation de $CM3$ par MTU

5.4.2 Programmation parallèle

Configuration d'un FPGA qui émule la partie digitale de la montre du ch. 1.

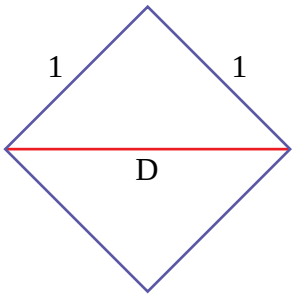
Chapitre 6

Nombres calculables

Contents

Les *pythagoriciens* étaient une des premières *Ecole de Pensée* de la Grèce antique (-700). Ils avaient compris l'universalité des entiers naturels, indépendamment de leurs représentations écrites, sur des abaques ou dans l'expression de rapports d'unités physiques. Leur philosophie, dont on n'a conservé qu'une tradition orale, visait à expliquer le *monde* en termes des entiers naturels et de leurs rapports, on dirait aujourd'hui les nombres rationnels.

Considérons le carré de coté unité ; soit D la longueur de sa diagonale.



Par le théorème de Pythagore, on a $D^2 = 1^2 + 1^2 = 2$.
Supposons maintenant le nombre D rationnel, soit $D = \frac{n}{d}$. Comme $D^2 = 2$, il vient :

$$n^2 = 2d^2 \qquad (\pi)$$

Comme la fraction $\frac{n}{d}$ est réduite, n et d ne peuvent être tous les deux pairs. Par (π), le numérateur n doit être pair, et donc d est impair. Le membre de gauche de (π) est alors un multiple de 4, puisque n est pair. Le membre droit de (π) est multiple de 2, mais pas de 4, puisque d est impair. C'est une contradiction ! Notre supposition - $D \in \mathbf{Q}$ - est donc fausse. Ce raisonnement par l'absurde démontre l'irrationalité de $D = \sqrt{2}$.

Figure 6.1 – Le secret des Pythagoriciens

La découverte de l'irrationalité de racine de deux $\sqrt{2}$ - figure 6.1 - et du nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$, fût célébré comme il se doit : en égorgeant plusieurs centaines de taureaux. Elle resta pendant deux siècles l'un des secrets de l'école.

Ce n'est pas tant l'irrationalité de $\sqrt{2}$ qui gêne les Pythagoriciens ; après tout, les Grecs connaissaient le développement en fractions continues périodiques du nombre d'or :

$$\phi = \sqrt{\phi - 1} = 1 + \frac{1}{\phi} = /1\,1\,1\,\cdots/ = /(1)/,$$

et de racine de 2 :

$$\sqrt{2} = 1 + \frac{1}{1+\sqrt{2}} = /1\,2\,2\,2\,\cdots/ = /1\,(2)/.$$

C'est la définition même de l'égalité entre deux nombres qui pose problème ! Comment, en effet, peut-on décider de l'égalité entre deux nombres définis par des expressions infinies ? En particulier, le paradoxe d'Achille et de la tortue, proposé par Zénon d'Ellée, pose la question de l'égalité :

$$\sum_{n\geq 0} \frac{1}{2^n} \stackrel{?}{=} 2. \tag{6.1}$$

Remarquons que si (6.1) était fausse, on pourrait s'en apercevoir par un calcul fini. Dans sa rédaction des Eléments d'Euclide, Eudoxe propose donc la définition suivante de l'égalité entre deux nombres x et y, présentés par des expressions arbitraires, finies ou non :

$$x = y \text{ si et seulement si } x \neq z \text{ implique } y \neq z, \text{ pour tout nombre } z.$$

Cette définition, acceptée pratiquement par tout mathématicien moderne, permet en effet d'établir l'égalité (6.1) de Zénon, et les questions de fondement finissent par passer de mode. Au point que le grand Euler sera accusé de manquer de rigueur pour son usage de séries totalement divergentes, comme $\sum_{n\geq 0} n! z^n$.

Il faudra attendre le XIX^e siècle pour que les mathématiciens perdent leurs réticences vis-à-vis de la construction des nombres réels. Les informaticiens, quant à eux, n'ont toujours pas surmonté cette crise. Ce qui, pour les Grecs, n'était qu'une position philosophique, est une nécessité pratique pour l'utilisateur de l'ordinateur, dans lequel toute structure n'est en dernier recours qu'un assemblage fini de zéros et de uns. Comment y représenter les nombres réels, tels que ϕ et $\sqrt{2}$?



Joseph Liouville Charles Hermite Ferdinand von Lindemann

Planche 6.1 – Liouville, Hermite et Lindemann

Nombres transcendants Ce n'est qu'au XIX^e siècle que la construction des nombres réels, telle que nous la connaissons, se dégage ; citons à ce propos les cours donnés par Cauchy à l'Ecole Polytechnique. Se développe alors l'analyse, qui permet de raisonner sur des nombres arbitrairement grands ou petits, tout en évitant l'introduction d'objets infinis. L'importance de l'analyse dans les applications, en particulier en physique, force les mathématiciens à accepter de manipuler des infinis potentiels, tout en restant très méfiants vis-à-vis de la signification philosophique de tels calculs.

Une des grandes questions de l'époque est celle de la transcendance des nombres. Soit **A** l'ensemble des nombres *algébriques*, c'est à dire racine d'un polynôme à coefficients entiers ; un nombre $x \in \mathbf{R}$ est dit *transcendant*, s'il n'est pas algébrique $x \notin \mathbf{A}$. L'existence des nombres transcendants est loin d'aller de soi. Ce n'est qu'en 1851 que Liouville fournit le premier exemple "naturel" en montrant que le nombre suivant est *transcendant* :

$$\sum_{n\geq 0} \frac{1}{2^{n!}}.$$

Il faut attendre 1873 pour que Hermite (polytechnicien comme Liouville), établisse la transcendance du nombre $e = 2.7183\cdots$. Neuf ans plus tard, Lindemann [1] montre enfin que la quadrature du cercle est *impossible*, en établissant la transcendance de $\pi = 3.14159\cdots$.

En utilisant son fameux *argument diagonal*, Cantor démontre pourtant - à la même époque - la chose suivante.

Proposition 6 (Cantor 1870) *Les nombres algébriques* **A** *sont dénombrables. Les réels* **R** *ne le sont pas.*

1. Suite à un exposé de Lindemann sur sa preuve, Kronecker lui dit : "A quoi sert votre merveilleux travail sur π ? Les irrationnels n'existent pas !"

Il en résulte qu'il y a une *infinité* de réels transcendants ; qui plus est, il y a *infiniment* plus de nombres transcendants que de nombres algébriques ! La preuve de Cantor, pour élémentaire qu'elle soit (exercice 7), est dure à avaler : elle montre en effet qu'il existe une infinité (non dénombrable) de nombres transcendants, *sans en construire un seul* explicitement. On la regarde pendant longtemps avec suspicion ; certains n'hésitent pas à qualifier ces résultats de *sorcellerie !* Entre les mains expertes de Hilbert, qui résout en 1889 le problème arithmétique de Gordon, les techniques de Cantor aboutissent pourtant à trop de résultats importants pour être ignorées ; elles entrent alors dans le bagage mathématique commun. Divers paradoxes menacent néanmoins de chasser les mathématiciens du *paradis* que Cantor leur avait trouvé. C'est vers la fin de la rédaction des *Principia Mathematica*, ouvrage commun avec Whitehead visant à fonder les mathématiques sur la théorie des ensembles, que Bertrand Russel découvre son fameux paradoxe : soit R l'ensemble de tous les ensembles qui ne se contiennent pas eux-mêmes, c'est à dire $R = \{E \ : E \notin E\}$; on voit alors que $R \in R$ si et seulement si $R \notin R$. Une contradiction [2] !

Georg Cantor (1845-1918) David Hilbert (1862-1943) Kurt Gödel (1906-1978)

Planche 6.2 – Trois précurseurs : Cantor, Hilbert et Gödel

Le programme de Hilbert Pour échapper à ces paradoxes, Hilbert propose de considérer les mathématiques comme un jeu formel sur des symboles, dont l'intérêt n'est pas dans le sens à donner aux objets manipulés, mais dans les règles de manipulation et les relations induites sur les objets. Dans ce *programme formaliste*, les mathématiques sont réduites à un jeu de règles d'inférence, permettant de déduire de nouveaux *théorèmes* à partir d'axiomes ou de théorèmes déjà établis. L'espoir était que ce jeu formel sur les symboles mette à jour les modes de raisonnements

2. Quand il apprend les déboires de son collègue Anglais, le mathématicien Henri Poincaré jubile. Il débute son amphithéâtre à la Sorbonne sur la phrase : "Messieurs, la théorie des ensembles n'est *plus* stérile. Elle vient d'accoucher d'une contradiction."

illégaux conduisant à des paradoxes. Pour cela, Hilbert développe une théorie de la preuve ou *métamathématique*, qui permet d'établir par des arguments constructifs, finis et donc indiscutables, la non-contradiction d'un système formel. Un tel système est dit :

1. *cohérent* si tous les théorèmes sont vrais ;
2. *complet* si toutes les formules vraies sont des théorèmes ;
3. *décidable* s'il existe une procédure automatique qui calcule en un temps fini si une expression donnée arbitraire est vraie ou non.

Gödel porte un coup *fatal* au programme formaliste de Hilbert en démontrant ce qui suit, à l'âge de 25 ans.

Théorème 4 (Gödel 1931) *Toute axiomatisation* cohérente *de l'arithmétique est nécessairement* incomplète.

L'essence de cette preuve est de construire une formule dont le sens est : Je suis vrai si et seulement si je ne suis pas démontrable. Le résultat de Gödel implique que, dans tout système formel assez riche pour exprimer les entiers naturels, il existe des propositions *vraies* qui ne sont pas *démontrables*.

6.1 Limite théorique du calcul

Planche 6.3 – Alonzo Church

6.1.1 La thèse de Church et Turing

Si le concept d'*algorithme* a, de tout temps, été fondamental en mathématique, il faudra attendre le programme formaliste pour qu'on ait besoin de donner une définition mathématiquement rigoureuse des termes : *algorithme, méthode effective, fonction calculable.*

De nombreux chercheurs proposent, au début de ce siècle, des modèles de la calculabilité : ce sont les ancêtres directs de nos structures de machines comme de nos langages de programmation. On s'aperçoit vite que tous ces modèles de la calculabilité sont équivalents, et deux propositions particulièrement simples sont émises simultanément et indépendamment par Turing et Church :

Définition 11 (Church 1936) *Une fonction est calculable si elle est représentable par une expression du λ-calcul.*

Le λ-calcul est l'ancêtre direct des langages de programmation. En particulier du langage Lisp. On peut écrire, en deux pages de Lisp, un compilateur du λ-calcul en Lisp. Inversement, et en se limitant à un noyau simple du langage Lisp, on peut écrire un traducteur inverse `Lisp->Church` de Lisp vers le λ-calcul. Ceci montre que, vis-à-vis de la définition de Church, Lisp et le λ-calcul jouent le même rôle : est définissable en Lisp ce qui l'est dans le λ-calcul et inversement.

Définition 12 (Turing 1936) *Une fonction est calculable si elle est définissable par le calcul d'une machine de Turing.*

Les machines de Turing sont présentées au chapitre 5.

Définition 13 *Une machine de Turing est* universelle *si elle est capable de* simuler *toute autre machine de Turing.*

Tout microprocesseur programmable est aussi une machine universelle, à condition de le doter d'une mémoire séquentielle *non bornée*.

On sait simuler en Lisp toute machine - chapitre 5 - en particulier une machine universelle. Inversement, on sait *compiler* le langage Lisp sur tout ordinateur. Il faudrait seulement, pour obtenir une simulation et une compilation exactes, être capables de gérer un nombre d'objets (paires et atomes) réellement *arbitraire*, et non pas simplement très grand comme c'est le cas dans notre monde physique. De fait, pour tout langage universel L comme Pascal, Fortran, Lisp, C, ..., une adaptation relativement simple des *compilateurs* existants permet de les transformer en traducteurs de L vers T_n aussi bien que vers le λ-calcul. Church a trouvé une façon élégante de résumer la (longue) histoire ci-dessus :

Proposition 7 (Thèse de Church et Turing, 1936) *Toute notion de calculabilité qui soit à la fois générale et concevable physiquement est nécessairement équivalente à celle des définitions 12 ou 11.*

Le point de vue exprimé ici est simple : aux (importantes) questions de temps et de mémoire près, tous les ordinateurs, du plus petit au plus gros, et tous les langages de programmation (universels), du plus simple au plus complexe, du plus moderne au plus ancien, se valent ! Ce qui est calculable sur l'un l'est sur l'autre, et

réciproquement. Je peux donc en théorie, avec mon micro-ordinateur, assez de disquettes et de jours ouvrables, simuler tout calcul que vous faites avec votre supercalculateur. Qu'une fonction soit calculable ou non est une propriété intrinsèque, qui ne dépend nullement du modèle de calculabilité.

Définition 14 *Un* langage *informatique permet de codifier les algorithmes sous forme de programmes. Un* programme *représente un algorithme par une suite finie de symboles, expression du langage de programmation. Traduit en machine sous forme d'un nombre binaire, notre programme devient directement exécutable.*

L'ensemble des programmes est dénombrable : ce sont des suites finies de symboles dans un alphabet fini. Il en va donc de même pour l'ensemble des algorithmes, objets abstraits dont les programmes constituent les représentations concrètes. Nous avons utilisé plusieurs langages de description d'algorithmes : Les règles de réécriture. Le langage Lisp et son ancêtre, le λ-calcul. Divers langages machines. Tous ces langages sont universels :

Définition 15 *Un langage est* universel *s'il permet de représenter tout algorithme par un programme fini.*

Ce qui différencie une machine universelle d'une machine physique, c'est le caractère non borné de sa mémoire. Toute machine physique est représentable par un *automate fini,* c'est-à-dire qu'elle ne peut exécuter qu'un nombre fini et fixe d'algorithmes différents, calculant tous sur des données de taille bornée. Contrairement à notre ordinateur favori, la *véritable* machine de Turing ne termine jamais son exécution sur le message de panique : `ERREUR~: PLUS DE MEMOIRE.` De par sa mémoire, la machine universelle dispose d'une réserve illimitée de calculs ! Ceci rend *chaque* machine universelle capable de simuler *toute* autre machine, universelle ou non. Elle peut exécuter, à sa propre vitesse, tous les algorithmes possibles, c'est-à-dire calculer toutes les *fonctions calculables*.

Plus que de simples automates finis, les ordinateurs successifs sont des approximations finies, de plus en plus grosses, d'une machine universelle.

6.1.2 Le problème de l'arrêt

La question de l'arrêt des programmes, nommé *entscheidungsproblem* par Hilbert, est la suivante : existe-t-il ou non un algorithme permettant de décider si un calcul donné va terminer ou boucler ?

Exemple 4 *Considérons la fonction d'Ackermann :*

$$\begin{aligned} A(n+1,r,a) &\Rightarrow A(n,r,A(1,r,a)),\\ A(1,r+1,a) &\Rightarrow A(a,r,a),\\ A(1,0,a) &\Rightarrow 2a,\\ A(0,r,a) &\Rightarrow a. \end{aligned}$$

Dans chaque appel de A les arguments (n, r) décroissent dans l'ordre lexicographique, soit :

$$(n+1, r) > (n, r),\ (n+1, r) > (1, r),\ (1, r+1) > (a, r).$$

Comme l'ensemble $(\mathbf{N}, \mathbf{N})$ des paires d'entiers muni de l'ordre lexicographique ne contient pas de suite infinie décroissante, il en résulte que le calcul la fonction d'Ackermann $A(n, m, p)$ - qui donne vite des nombres tellement gigantesques qu'aucun ordinateur ne sait les calculer - termine pourtant toujours, pour $n, r, a \in \mathbf{N}$.

Exemple 5 *Considérons la fonction F, définie par les trois règles de calcul :*

$$\begin{aligned} F(2n) &\Rightarrow F(n), \\ F(2n+1) &\Rightarrow F(3n+2), \\ F(1) &\Rightarrow 0. \end{aligned}$$

Quand le calcul de $F(n)$ termine, sa valeur, pour $n \in \mathbf{N}$ entier, est forcément $F(n) \stackrel{*}{\Rightarrow} 0$. La suite des valeurs de n dans le calcul de $F(n)$ est cependant très particulière, comme en attestent les deux calculs suivants :

$$F(21) \Rightarrow F(32) \Rightarrow F(16) \Rightarrow F(8) \Rightarrow F(4) \Rightarrow F(2) \Rightarrow F(1) \Rightarrow 0.$$

$$\begin{aligned} F(27) &\Rightarrow F(41) \Rightarrow F(62) \Rightarrow F(31) \Rightarrow F(47) \Rightarrow F(71) \Rightarrow F(107) \\ &\Rightarrow F(161) \Rightarrow F(242) \Rightarrow F(121) \Rightarrow F(182) \Rightarrow F(91) \\ &\Rightarrow F(137) \Rightarrow F(206) \Rightarrow F(103) \Rightarrow F(155) \Rightarrow F(233) \\ &\Rightarrow F(350) \Rightarrow F(175) \Rightarrow F(263) \Rightarrow F(395) \Rightarrow F(593) \\ &\Rightarrow F(890) \Rightarrow F(445) \Rightarrow F(668) \Rightarrow F(334) \Rightarrow F(167) \\ &\Rightarrow F(251) \Rightarrow F(377) \Rightarrow F(566) \Rightarrow F(283) \Rightarrow F(425) \\ &\Rightarrow F(638) \Rightarrow F(319) \Rightarrow F(479) \Rightarrow F(719) \Rightarrow F(1079) \\ &\Rightarrow F(1619) \Rightarrow F(2429) \Rightarrow F(3644) \Rightarrow F(1822) \Rightarrow F(911) \\ &\Rightarrow F(1367) \Rightarrow F(2051) \Rightarrow F(3077) \Rightarrow F(4616) \Rightarrow F(2308) \\ &\Rightarrow F(1154) \Rightarrow F(577) \Rightarrow F(866) \Rightarrow F(433) \Rightarrow F(650) \\ &\Rightarrow F(325) \Rightarrow F(488) \Rightarrow F(244) \Rightarrow F(122) \Rightarrow F(61) \\ &\Rightarrow F(92) \Rightarrow F(46) \Rightarrow F(23) \Rightarrow F(35) \Rightarrow F(53) \\ &\Rightarrow F(80) \Rightarrow F(40) \Rightarrow F(20) \Rightarrow F(10) \Rightarrow F(5) \\ &\Rightarrow F(8) \Rightarrow F(4) \Rightarrow F(2) \Rightarrow F(1) \Rightarrow 0. \end{aligned}$$

On a vérifié sur ordinateur que la fonction F termine pour tous les entiers de moins de 6 chiffres, et que F termine pour un ensemble infini de nombres, de

densité 1 sur **N**. *Cependant, nul ne sait aujourd'hui si* $F(n)$ *termine pour tout* $n \in \mathbf{N}$.

Planche 6.4 – Pierre de Fermat (1601-65)

Exemple 6 *Fermat - planche 6.4 - avait pour habitude de noter ses observations en marge de sa copie des œuvres de Diophante. Il y écrit en 1637 que l'équation :*

$$x^n + y^n \quad = \quad z^n \tag{6.2}$$

n'a pas de solution en nombres entiers positifs, pour $n > 2$. *Il ajoute avoir trouvé "une merveilleuse démonstration" de ce* théorème, *trop longue cependant pour tenir dans cette marge. A ce jour, nul n'a retrouvé la preuve de Fermat. Nul n'a été capable non plus de produire un contre-exemple, malgré l'aide des ordinateurs. La conjecture de Fermat est devenue un théorème, démontré en 1994 par le mathématicien anglais Wiles.*

La recherche exhaustive de contre-exemples est simple dans son principe. Un argument élémentaire montre que, s'ils existent, les contre-exemples à (6.2) sont nécessairement de la forme : $1 < x < y < z$ *et* $2 < n < z$. *Ceci nous permet d'écrire un tel programme, ici en* ${}_2\mathbf{Z}$.

```
for z>2 do
  for 2<n<z do
    for 1<y<z do
      for 1<x<y do
     assert (x^n + y^n ≠  z^n)
    end.
```

Le calcul de cette expression $_2\mathbf{Z}$ *particulière* termine*, si et seulement si la conjecture de Fermat est* fausse.

Ces exemples montrent tout l'intérêt et la difficulté de la question de Hilbert. C'est en 1936 que le mathématicien anglais Turing répond par la négative.

Théorème 5 (Turing) *Le problème de l'arrêt est* indécidable*.*

Preuve : Supposons que M. Bogue, programmeur professionnel, prétend avoir codé une fonction Lisp[3] `(Termine? f x)`, dont les arguments sont `f` et `x`, et qui s'évalue à *vrai* si le calcul de l'expression Lisp `(f x)` termine, et *faux* s'il ne termine pas. Bien entendu, M. Bogue affirme que son calcul termine en temps fini pour toute fonction `f` et argument `x`. Afin de protéger ses droits d'auteur, M. Bogue ne donne pas lecture de son code source, mais autorise, moyennant finance, l'exécution de son programme sur des données de notre choix. Turing pose alors les définitions :

```
(de Boucle () (Boucle))  ; Ne termine pas~!
(de Contredit (f)        ;
    (if (Termine? f f)   ; Si (f f) termine,
        (Boucle)         ; on boucle,
        T))              ; sinon, on termine~!
```

Turing soumet à la machine de M. Bogue le calcul de l'expression :

```
(Termine Contredit Contredit)
```

Si `(Contredit Contredit)` termine, alors l'expression diagonale de Turing vaut 1 ; mais alors, `(Contredit Contredit)` est égal à `(Boucle)`, qui ne termine pas …. Cette contradiction expose l'imposture de M. Bogue, qui fait faillite. **Q.E.D.**

L'existence de fonctions non calculables, dont l'hypothétique `Termine?` est notre premier exemple, résulte directement du résultat de Cantor :

Proposition 8 *L'ensemble* $\mathbf{N} \mapsto \mathbf{N}$ *des applications des entiers dans eux-mêmes n'est pas dénombrable. L'ensemble des applications* calculables *de* $\mathbf{N} \mapsto \mathbf{N}$ *est dénombrable.*

3. L'argument de Turing est codé ici dans le langage Lisp, dont la syntaxe facilite la manipulation d'expressions symboliques et donne la clé de la concision de cette preuve. On peut bien entendu transcrire l'argument de Turing dans n'importe quel autre langage de programmation universel. Suivant le langage, l'effort de codage est plus ou moins important ; dans tous les cas classiques, le code résultant est plus long qu'en Lisp.

L'argument de Cantor, qui montre l'existence d'une *infinité non dénombrable* de fonctions non calculables, semble plus fort que celui de Turing, qui ne montre l'existence que d'une seule fonction non calculable, à savoir : `Termine?`. C'est pourtant l'argument de Turing qui se révèle, de très loin, le plus utile : il est en effet explicite (effectif, constructif), alors que celui de Cantor ne nous fournit aucun exemple de fonction non calculable. Toutes les fonctions non calculables connues, et il en est beaucoup (cf. exercice 6), se *réduisent* en effet à celle de Turing, le problème de l'arrêt.

Exercice 5 *Construire explicitement une machine de Turing universelle avec un ruban* unique *pour le programme et les données. Réduire, autant que possible, son nombre d'états internes - registres hors ruban.*

Exercice 6 (Le programme le plus lent) *Parmi les expressions E_n que l'on peut écrire avec n caractères dans un langage de programmation, il en est dont l'évaluation* termine*, et d'autres dont l'évaluation* ne termine pas*. Soit $T(n)$ le nombre maximal de pas de calcul nécessaire à l'évaluation d'une expression E_n qui termine. Montrer que l'application $T \in \mathbf{N} \mapsto \mathbf{N}$ n'est pas calculable. Pour cela, on montrera que : si T est calculable, alors, le problème de l'arrêt est décidable.*

Exercice 7 *Démontrer la proposition 6 de Cantor.*

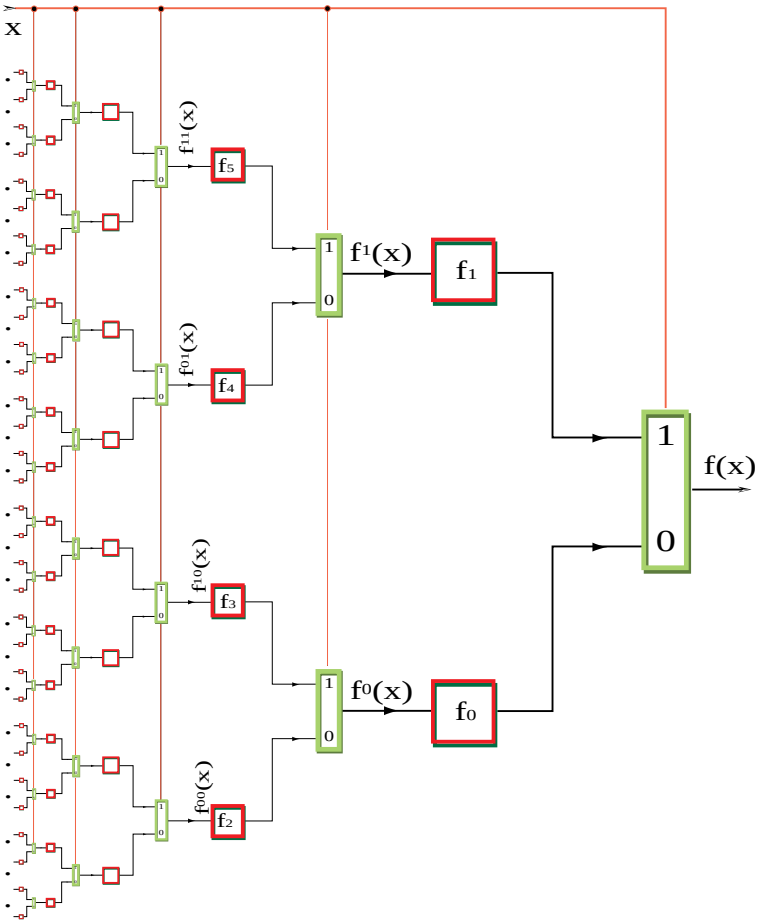


Figure 6.2 – Circuit causal universel

6.2 Fonctions calculables

Le texte [7] *On circuits and numbers*, distribué en documentation du cours sert de référence pour cette section.

Une fonction 2-adique est une application $\mathbf{f} \in {}_2\mathbf{Z}^{\,i} \mapsto {}_2\mathbf{Z}^{\,o}$ avec i entrées et o sorties. En identifiant la valeur au temps t du vecteur d'entrée $x[0..i-1]$ avec un nombre en base $b = 2^i$, et les sorties avec un nombre en base $b' = 2^o$, on peut écrire, plus simplement : $\mathbf{f} \in {}_b\mathbf{Z} \mapsto {}_{b'}\mathbf{Z}$. L'égalité $y = \mathbf{f}(x)$ signifie donc, en faisant intervenir explicitement le temps synchrone $t \in \mathbf{N}$:

$$\mathbf{f}(\sum_{t\in\mathbf{N}} x_t b^t) \;=\; \sum_{t\in\mathbf{N}} y_t b'^t.$$

6.2.1 Automate fini

Circuit fini.

6.2.2 Fonction causale

Définition 16 *Une fonction 2-adique* $\mathbf{f} \in {}_2\mathbf{Z}^{\,i} \mapsto {}_2\mathbf{Z}^{\,o}$ *est dite* causale *si elle répond aux conditions équivalentes qui suivent.*

- *La valeur* y_n *des sorties au temps* $t = n \in \mathbf{N}$ *est uniquement déterminée par celle des entrées* $x_0, \cdots x_n$*, du temps* $t = 0$ *au temps* $t = n$*.*
- *Il existe une famille de fonctions combinatoires* $f_t \in \mathbf{B}^{i(t+1)} \mapsto \mathbf{B}^o$ *telle que :*

$$\mathbf{f}(\sum_{t\in\mathbf{N}} y_t b'^t) = \sum_{t\in\mathbf{N}} b^t f(x_0, \cdots x_t).$$

- *On a :* $${}_b|f(a) - f(b)| \;\leq\; {}_{b'}|a - b|.$$

Théorème 6 *Une fonction 2-adique est causale si et seulement si elle est réalisable par un circuit* $\mathcal{C}_{ds}$*, fini ou infini.*

Décomposition synchrone :

$$f(x) \;=\; \mathbf{mux}(x, b_1 + 2f_1(x \div 2), b_0 + 2f_0(x \div 2)).$$

On a $b_0 = f(0) \cdot|\cdot 2$, $b_1 = f(1) \cdot|\cdot 2$, $f_1(x) = f(2x+1) \div 2$ et $f_0(x) = f(2x) \div 2$

6.2.3 Fonction à délai borné

L'*antiflop* $f(x) = x \div 2$ n'est pas une fonction causale.

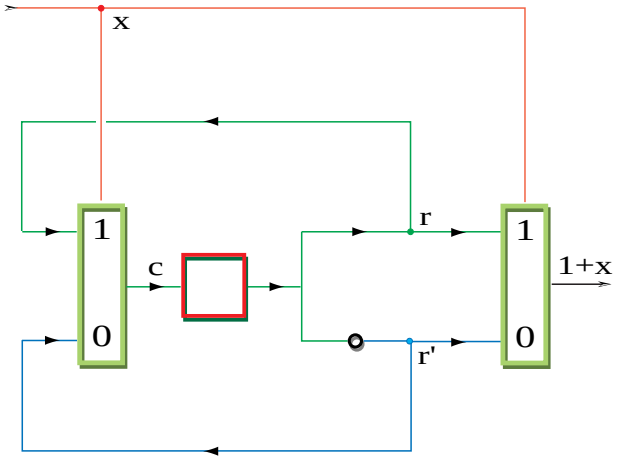


Figure 6.3 – Synthèse SDD d'un incrémenteur série

6.2.4 Fonction continue

Définition 17 *Une fonction 2-adique* $\mathbf{f} \in {}_2\mathbf{Z}^{\,i} \mapsto {}_2\mathbf{Z}^{\,o}$ *est dite* continue *si elle répond aux conditions équivalentes qui suivent.*

- *La valeur* y_n *des sorties au temps* $t = n \in \mathbf{N}$ *est uniquement déterminée par celle d'un nombre fini d'entrées* $x_0, \cdots x_{m(n)}$*, du temps* $t = 0$ *au temps* $t = m(n) \in \mathbf{N}$*.*
- *Il existe une famille de fonctions combinatoires* $f \in \mathbf{B}^{i(m(t)+1)} \mapsto \mathbf{B}^o$ *telle que :*

$$\mathbf{f}(\sum_{t\in\mathbf{N}} y_t b'^t) = \sum_{t\in\mathbf{N}} b^t f_t(x_0, \cdots x_{m(t)}).$$

- *Pour tout nombre réel positif* $\epsilon \in \mathbf{R}$*, il existe un nombre réel positif* $\eta \in \mathbf{R}$ *tel que :* ${}'_b|a - b| < \epsilon$ *implique* ${}'_b|\mathbf{f}(a) - \mathbf{f}(b)| < \epsilon$*.*

Théorème 7 *Une fonction 2-adique est continue si et seulement si elle est réalisable par :*

- *un circuit combinatoire infini ;*
- *une machine de Turing universelle, munie d'un programme de taille infinie ;*
- *un circuit* $\mathcal{C}_{ds}$*avec validation des sorties.*

Exercice 8 *Montrer que les trois clauses de la définition 16 sont équivalentes.*
Montrer que les trois clauses de la définition 17 sont équivalentes.

Exercice 9 (Escalier du Diable de Gosper) *La fonction* f *qui suit est continue sur* $[0\cdot\cdot1]$ *; on a* $f(0) = 0, f(1) = 1$ *; pourtant,* f *est décroissante presque partout.*

$$\begin{aligned}
\lfloor x \rfloor = 0 \quad f(x) &= \frac{2}{3} f(3x); \\
\lfloor x \rfloor = 1 \quad f(x) &= \frac{3}{2} - \frac{x}{2}; \\
\lfloor x \rfloor = 2 \quad f(x) &= \frac{2}{3} f(3x-6) + \frac{1}{2}; \\
x > 3 \quad f(x) &= \frac{3}{2} f(x/3).
\end{aligned}$$

Est-elle calculable sur $\mathbf{Q}$ *? sur* $\mathcal{R}$ *?*

Programmer votre ordinateur favori pour tracer le graphe de f*. Choisir plusieurs résolutions afin de bien visualiser le caractère* fractal *de la fonction* f*.*

6.3 Réel calculable $\mathcal{R}$

Il a 26 ans quand il devient professeur de Mathématiques à l'Ecole Polytechnique.

Planche 6.5 – Augustin Louis Cauchy (1789-1857).

Nous introduisons ici les réels calculables $\mathcal{R}$, comme limites effectives d'intervalles rationnels. Ceci permet de séparer rapidement les opérations calculables, de celles qui ne le sont pas.

6.3.1 Construction par intervalles

Définition 18 *Un nombre réel $r \in \mathcal{R}$ calculable est défini par une suite d'intervalles*

$$r(0), r(1), \cdots, r(n), \cdots$$

d'extrémités $r(n) = [i(n), s(n)]$ rationnelles $i(n), s(n) \in \mathbf{Q}$ telle que :
(i) Chaque intervalle $r(n)$ contient le suivant $r(n+1)$:

$$r(0) \supseteq r(1) \supseteq \cdots \supseteq r(n) \supseteq r(n+1) \supseteq \cdots.$$

(ii) Pour $n \in \mathbf{N}$ assez grand, la longueur de $r(n)$ devient arbitrairement petite

$$\lim_{n\to\infty} \Delta(s(n), i(n)) = 0.$$

Le nombre r est donc l'unique réel appartenant à tous les intervalles :

$$r = r(\infty) = \bigcap_{n\geq 0} r(n).$$

(iii) L'application $r : \mathbf{N} \mapsto [\mathbf{Q}, \mathbf{Q}]$ des entiers $\mathbf{N}$ dans les intervalles rationnels $[\mathbf{Q}, \mathbf{Q}]$ est une fonction calculable.

Exemple 7 *Le nombre d'or* $\phi = (1+\sqrt{5})/2$ *peut se représenter par l'application calculable* $\phi \in \mathbf{N} \mapsto [\mathbf{Q}, \mathbf{Q}]$ *donnée par :*

$$\phi(n) \;=\; [\frac{F_{2n+2}}{F_{2n+1}}, \frac{F_{2n+1}}{F_{2n}}],$$

où $F_0 = 0, F_1 = 1, \ldots, F_{n+2} = F_{n+1} + F_n, \ldots$ *sont les nombres de Fibonacci.*

Exemple 8 *Pour tout entier* $z \in \mathbf{N}$*, désignons par* $\mathcal{F}_z$ *le nombre de solutions à l'équation de Fermat :* $x^n + y^n = z^n$*. Définissons l'application* $\mathcal{F}_{ermat} \in \mathbf{N} \mapsto [\mathbf{Q}, \mathbf{Q}]$ *par :*

$$\mathcal{F}_{ermat}(k) \;=\; [\sum_{i \leq k} \frac{\mathcal{F}_i}{i!}, \sum_{i \leq k} \frac{\mathcal{F}_i}{i!} + 1/k].$$

Elle est calculable car le nombre entier $\mathcal{F}_n$ *est calculable, pour tout* $n \in \mathbf{N}$*. Le nombre* $\mathcal{F}_{ermat} = \mathcal{F}_{ermat}(\infty) \in \mathcal{R}$ *est un réel calculable, et on montre facilement que* $0 \leq \mathcal{F}_{ermat} < 2$*; bien entendu,* $\mathcal{F}_{ermat} = 0$*, si et seulement si la conjecture de Fermat est vraie.*

On connaît maintenant une façon beaucoup plus rapide de calculer ce nombre.

Théorème 8 (Wiles, 1994) $\mathcal{F}_{ermat} = 0$.

6.3.2 Opérations non calculables sur $\mathcal{R}$

Proposition 9 *L'égalité à deux* $x \stackrel{?}{=} 2$ *est* indécidable *pour un réel calculable* $x \in \mathcal{R}$ *arbitraire.*

Preuve : Soit T une machine de Turing arbitraire. Attachons à T un *compteur de Zénon* C que l'on définit ainsi :

- initialement, $C(0) = 1$;
- à chaque transition $t \mapsto t+1$ de la machine T, on calcule :

$$C(t+1) = 1 + \frac{1}{2}C(t).$$

Le nombre $C(\infty) = lim_{t\to\infty} C(t)$ est un réel calculable $C(\infty) \in \mathcal{R}$, puisque nous venons d'en donner l'algorithme de calcul. Sa valeur est telle que :

$$\begin{aligned} C(\infty) &= 2 && \text{si le calcul de } T \text{ ne termine pas ;} \\ C(\infty) &= 2 - \frac{1}{2^n} && \text{si le calcul de } T \text{termine en } n \text{ étapes.} \end{aligned}$$

Si l'on pouvait décider de l'égalité de C_∞ à deux, on pourrait aussi décider de l'arrêt de T ; c'est impossible par la proposition 5. **Q.E.D.**

Nombres réels et programmes

Sans la condition (iii), les nombres de la Définition 18 seraient isomorphes à ceux de Cauchy, Dedekind et Cantor, les *réels classiques,* non dénombrables. Cette condition (iii) impose que la suite des intervalles définissant un nombre réel soit donnée de manière effective. Représentons ici un nombre réel par un programme *fini* (dans un langage $\mathcal{L}$), qui définit l'algorithme de calcul de l'intervalle rationel $r(n) = [i(n), s(n)]$, à partir de l'entrée $n \in \mathbf{N}$.

Proposition 10 (Cantor,Turing) *Soit $\mathcal{R}$ le sous-ensemble des programmes de $\mathcal{L}$ qui représentent un nombre réel : (i) l'ensemble $\mathcal{R}$ est dénombrable ;*
(ii) aucun algorithme ne peut énumérer exhaustivement les éléments de $\mathcal{R}$.

De façon équivalente, nous voyons qu'il est indécidable de savoir si un programme arbitraire de $\mathcal{L}$ représente ou non un nombre $\mathcal{R}$.

L'écriture des réels calculables n'est pas calculable !

Proposition 11 (Turing) *Aucun algorithme ne peut déterminer, en temps fini pour tout réel calculable $r \in \mathcal{R}$:*

1. *si le nombre r est nul : $r = 0$;*
2. *si $r > r'$, pour $r' \in \mathcal{R}$ donné arbitraire ;*
3. *si le nombre r est entier, i.e. $r \in \mathbf{N}$;*
4. *si le nombre r est rationnel, i.e. $r \in \mathbf{Q}$;*
5. *le signe $sgn(r) \in \{-1, 0, 1\}$ de r ;*
6. *le premier bit de l'écriture binaire de r ;*
7. *le premier chiffre de l'écriture décimale de r ;*
8. *le premier terme $\lfloor r \rfloor \in \mathbf{Z}$ de l'écriture en fraction continue de r.*

De fait, la Proposition 11 résulte de l'énoncé plus général :

Proposition 12 (Rice) *Toute fonction* calculable $f \in \mathcal{R} \mapsto \mathcal{R}$ *est continue et indéfiniment dérivable.*

Ainsi, tout prédicat calculable est trivial :

$$\forall p \in \mathcal{R} \mapsto \mathbf{B},\ \forall x, y \in \mathcal{R}\ :\ p(x) = p(y).$$

De même, toute application calculable $f \in \mathcal{R} \mapsto \mathbf{N}$, à valeur entière, est constante !

6.3.3 Opérations calculables sur $\mathcal{R}$

Montrons maintenant que les quatre opérations sont calculables sur $\mathcal{R}$, et que $\mathcal{R}$ est un *corps*. Il importe, avant de définir une arithmétique sur les intervalles, puis de l'étendre aux suites d'intervalles, de bien comprendre la topologie des intervalles.

Calculs à l'infini

$+$	0	1	∞	$\perp$
0	0	1	∞	$\perp$
1	1	2	∞	$\perp$
∞	∞	∞	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\times$	0	1	∞	$\perp$
0	0	0	$\perp$	$\perp$
1	0	1	∞	$\perp$
∞	$\perp$	∞	∞	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$-$	0	1	∞	$\perp$
0	0	-1	∞	$\perp$
1	1	0	∞	$\perp$
∞	∞	∞	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$/$	0	1	∞	$\perp$
0	$\perp$	0	0	$\perp$
1	∞	1	0	$\perp$
∞	∞	∞	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Figure 6.4 – Opérations sur l'infini et l'indéfini

Comme (proposition 11) on ne peut décider si $r \neq 0$, il est, en général, impossible d'éviter la division par 0 dans le calcul de l'inverse $1/r$. Nous devons donc nous résigner à incorporer le nombre

$$\infty = \frac{1}{0}$$

à notre collection de réels calculables. Pour la même raison, on ne peut éviter le nombre indéfini

$$\perp = \frac{0}{0} = 0 \times \infty = 0^0 = \infty - \infty.$$

L'indéfini $\perp$ peut être considéré comme une expression qui dirait : "je suis un nombre". Au vu de la Proposition 6(ii), c'est une information appréciable ; à part ça, le nombre $\perp$ ne donne aucune information sur sa valeur, que nous identifions donc à l'intervalle de tous les nombres définis, y compris l'infini ∞ :

$$\perp = [-\infty, +\infty].$$

Le résultat de toute opération avec $\perp$ est indéfini ; on trouvera, en figure 6.4, les tables d'opérations de $\perp$ et ∞. Pour distinguer $\perp$ et ∞ des autres nombres, nous disons que $r \in \mathbf{R}$ est *défini* quand $r \neq \perp$, et que r est *fini* quand $r \neq \perp$ et $r \neq \infty$.

Représentation stéréographique de R

Pour pouvoir traiter l'infini ∞ comme tout autre nombre, il est commode d'utiliser la projection stéréographique de la droite réelle, sur le cercle de rayon unité centré à l'origine, comme dans la figure 6.5. L'image

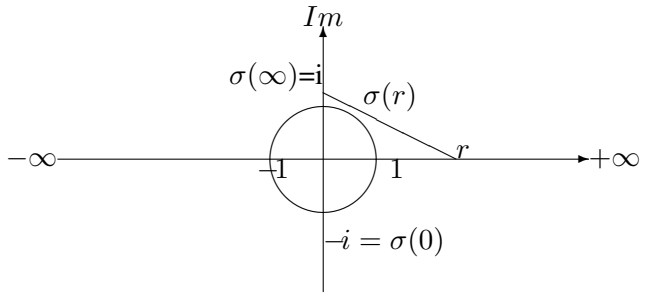


Figure 6.5 – Projection stéréographique de **R**

$$\sigma(r) = \frac{ir+1}{r+i} = \frac{2r}{r^2+1} + i\frac{r^2-1}{r^2+1}$$

du point $r \in (-\infty, +\infty]$ se trouve à l'intersection du cercle unitaire avec la droite joignant r au point $i = (0,1)$. Inversement, tout point $\sigma = e^{i\theta}$ du cercle est l'image stéréographique de $r = \frac{\cos\theta}{1-\sin\theta}$. La longueur Δ de la corde joignant $\sigma(x)$ à $\sigma(y)$ est donnée par :

$$\Delta(x,y) \quad = \quad \frac{2|x-y|}{\sqrt{(1+x^2)(1+y^2)}}. \tag{6.3}$$

En particulier :

$$\Delta(0,r) = \frac{2|r|}{\sqrt{1+r^2}}, \Delta(r,\infty) = \frac{2}{\sqrt{1+r^2}}, \Delta(0,1) = \Delta(1,\infty) = \sqrt{2}.$$

Par sa construction géométrique, il est clair que $0 \leq \Delta(x,y) \leq 2$. La relation (6.3) définit une distance *finie* entre deux nombres arbitraires x et y :

$$\begin{aligned} \Delta(x,y) &= \Delta(y,x); \\ \forall z \ : \Delta(x,y) &\leq \Delta(x,z) + \Delta(z,y). \end{aligned}$$

Nous utilisons Δ pour argumenter de la convergence à l'infini, comme en tout point fini.

Dans la correspondance stéréographique, l'intervalle $[r, r']$ est envoyé sur l'arc $\widehat{\theta, \theta'}$ du cercle joignant $\sigma(r) = e^{i\theta}$ à $\sigma(r') = e^{i\theta'}$, en tournant dans un sens opposé à celui des aiguilles d'une montre. Inversement, tout arc $\widehat{\theta, \theta'}$ est l'image d'un intervalle $[r, r']$. La nouveauté réside ici dans les intervalles ouverts, comme $(2, -2)$, qui représente l'ensemble $\{r \in \mathbf{R} \ : \ r > 2 \text{ or } r < -2\}$ des points dont la valeur absolue est plus grande que deux ; c'est le complémentaire de l'intervalle $[-2, 2] = \{r \ : \ -2 \leq r \leq 2\}$.

Définition 19 *On peut classifier les intervalles comme suit :*

1. *Un intervalle fini $[i, s]$, avec $i < s$, représente $\{r \in \mathbf{R} \ : \ i \leq r \leq s\}$. On utilise les parenthèses pour exclure la borne correspondante, comme dans $(i, s] = \{r \in \mathbf{R} \ : \ i < r \leq s\}$; de même pour $[i, s)$ et (i, s). Un intervalle fini $[i, s]$ est positif si $i > 0$, négatif si $s < 0$; il contient zéro dans tous les autres cas.*
2. *Un intervalle infini $[i, s]$, avec $i > s$, représente le complémentaire $\{r \in \mathbf{R} \ : \ r \notin (s, i)\}$ de l'intervalle fini (s, i) ; de même pour $(i, s]$ et $[i, s)$, ainsi que pour $(i, s) = \{r \in \mathbf{R} \ : \ i < r, \text{ ou } r < s\}$.*
3. *Un intervalle de longueur nulle est : réduit à un point $[i, i] = i$; le complémentaire $(i, i) = \{r \in \mathbf{R} \ : r \neq i\}$ d'un point ; l'intervalle vide $\emptyset = [i, i)$;*
4. *le complémentaire $\perp = (0+, 0-] = (-\infty, +\infty]$ de l'intervalle vide, c'est-à-dire l'ensemble de tous les nombres.*

Arithmétique d'intervalles

Etendons aux intervalles les quatre opérations arithmétiques :

Addition : La somme de deux intervalles $[i, s]$ et $[i', s']$ se calcule par

$$[i, s] + [i', s'] \quad = \quad [i + i', s + s'],$$

quand $i \leq s$, $i' \leq s'$, ou $i \leq s, i' > s', i + i' > s + s'$, ou $i > s, i' \leq s'$ et $i + i' > s + s'$. Dans tous les autres cas : $[i, s] + [i', s'] = \perp$.

Soustraction : L'opposé de l'intervalle $[i, s]$ se calcule par :

$$-[i, s] \quad = \quad [-s, -i].$$

Multiplication : Le produit de deux intervalles $[i, s]$ et $[i', s']$ se calcule par :

$$[i, s] \times [i', s'] \quad = \quad [min(ii', is', si', ss'), max(ii', is', si', ss')],$$

si $\infty \notin [i, s]$ et $\infty \notin [i', s']$. Si $0 \notin [i, s], 0 \notin [i', s']$ et $\infty \in [i, s]$ ou $\infty \in [i', s']$, on a alors :

$$[i, s] \times [i', s'] \quad = \quad [min(|ii'|,|is'|,|si'|,|ss'|), -min(|ii'|,|is'|,|si'|,|ss'|)].$$

Dans tous les autres cas : $[i, s] \times [i', s'] = \perp$.

Division : L'inverse de l'intervalle $[i, s]$ se calcule par :

$$1/[i, s] \quad = \quad [1/s, 1/i].$$

6.3.4 Ecriture binaire classique des réels

Point fractionnaire

Le *point fractionnaire* permet d'étendre aux nombres *réels* les numérations de position, comme dans $\sqrt{2} = 1 \cdot 414213 \cdots$. En français, on met traditionnellement une virgule en place du point fractionnaire pour écrire $\sqrt{2} = 1,414213 \cdots$. Nous utilisons le *point* dans ce texte, conformément à l'usage international.

Soit b une base entière $b \in \mathbf{N} + 2$, et $a_k \in \mathbf{N} \cdot|\cdot b$ une suite de chiffres, définie maintenant pour $k \in \mathbf{Z}$. La notation

$$[a_{n-1} \cdots a_1 a_0 \cdot a_{-1} a_{-2} \cdots a_{-k} \cdots]_b$$

représente le nombre réel $a \in \mathbf{R}$ positif $a > 0$ donné par la formule :

$$a \ = \ \sum_{k \in \mathbf{Z}} a_k 2^k,$$

en convenant que $a_k = 0$ pour $k > n$. Cette formule fixe l'algorithme de lecture des nombres réels représentés en base b, avec point fractionnaire.

Algorithme 10 (Ecriture par les poids forts)

L'écriture en base $b \in \mathbf{N} + 2$ du nombre réel positif $a \in \mathbf{R}$, $a > 0$ est la suite infinie *de* chiffres $0 \leq a_k < b$ *:*

$$\mathcal{E}'_b(a) = [a_{n-1} \cdots a_1 a_0 \cdot a_{-1} a_{-2} \cdots a_{-k} \cdots]_b \, .$$

On la calcule en deux temps.

1. *Les chiffres a_k - pour $k \in \mathbf{N}$ - situés à* gauche *du point fractionnaire se calculent en appliquant l'algorithme 2 à la partie entière $e = a \div 1 = \lfloor a \rfloor \in \mathbf{N}$ de a, soit $\mathcal{E}_b(e) = [a_{n-1} \cdots a_1 a_0]_b$.*
2. *les chiffres a_{-k}, pour $k \in \mathbf{N}$, situés à* droite *du point fractionnaire s'obtiennent en partant du nombre $A_0 = a \cdot|\cdot 1 = a - e$, réel positif inférieur à un, et en calculant de proche en proche, pour $k \in -\mathbf{N}$:*

$$\begin{aligned} a_{k-1} &= (b \times A_k) \div 1, \\ A_{k-1} &= (b \times A_k) \cdot|\cdot 1. \end{aligned}$$

Pour mener ce calcul, on observe d'abord que la partie entière $q \div 1$ du nombre rationnel $q = \frac{n}{d} \in \mathbf{Q}$ n'est autre que le quotient de la division entière du numérateur n par le dénominateur d, soit

$$\lfloor \frac{n}{d} \rfloor = \frac{n}{d} \div 1 = n \div d.$$

Quant à la partie fractionnaire $q \cdot|\cdot 1 \in \mathbf{Q}$ de q, son numérateur est le reste $n \cdot|\cdot d$ de la division, et son dénominateur d est le même que celui de q, soit $q \cdot|\cdot 1 = \frac{n \cdot|\cdot d}{d}$; ainsi, $\frac{4}{3} \div 1 = 1$ et $\frac{4}{3} \cdot|\cdot 1 = \frac{1}{3}$.

Ecrivons par exemple le nombre rationnel $1/3$ en binaire par les poids forts au moyen de l'algorithme 10.

k	0	-1	-2	-3	-4	-5	$\cdots$
A_k	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{2}{3}$	$\cdots$
a_k	$\cdot$	0	1	0	1	0	$\cdots$

Comme le motif se répète, on constate que l'écriture binaire *infinie* de 1/3 est *périodique* :

$$\frac{1}{3} \;=\; 0.0101010101\cdots$$

Pour donner une représentation *finie* à de telles suites finalement périodiques, convenons d'écrire la partie périodique de la suite entre deux parenthèses. Dans le cas d'espèce, cette convention donne :

$$\frac{1}{3} \;=\; 0.(01)_2, \tag{6.4}$$

avec l'indice 2 à droite pour nous montrer que la base de cette écriture est $b = 2$, et que les poids forts sont en tête à gauche. La sémantique de cette écriture, c'est à dire la valeur de cette expression est, bien entendu :

$$\frac{1}{3} \;=\; 0.(01)_2 \;=\; 1/4 + 1/16 + 1/64 + \cdots \;=\; \sum_{k \in \mathbf{N}+1} 2^{-2k}.$$

La somme d'une série géométrique de raison $|x| < 1$ est :

$$1 + x + x^2 + x^3 + \cdots \;=\; \sum_{k \in \mathbf{N}} x^k \quad = \quad \frac{1}{1-x}. \tag{6.5}$$

On vérifie ainsi que l'on a bien : $1/4 + 1/16 + 1/64 + \cdots = \frac{1}{4} \cdot \frac{1}{1-1/4} = \frac{1}{3}$. Considérons un second exemple, celui de l'écriture de $22/7$ en binaire. La partie entière est $22 \div 7 = 3$ qui s'écrit en binaire $3 = 11_2$, par l'algorithme 2. La partie fractionnaire est $1/7$. Pour l'écrire au moyen de l'algorithme 10, observons que tous les dénominateurs des nombres A_k sont égaux à 7, et simplifions le calcul en conséquence.

$-k$	0	1	2	3	4	5	6	7	$\cdots$
$7A_k$	1	2	4	1	2	4	1	2	$\cdots$
a_k	$\cdot$	0	0	1	0	0	1	0	$\cdots$

Le calcul se répète, comme précédemment, et on trouve donc :

$$\frac{22}{7} \;=\; 11 \cdot (001)_2.$$

La pratique des calculettes nous apprend que le nombre 1 apparaît aussi sous la forme $0 \cdot 99999\cdots$. En effet, il y a deux représentations par point fractionnaire des nombres entiers en base b ; par exemple :

$$1 = 1 \cdot (0)_2 = 0 \cdot (1)_2 = 1 \cdot (0)_{10} = 0 \cdot (9)_{10}.$$

Cette double représentation est une anomalie des mathématiques classiques ; en informatique, c'est un problème théorique majeur ! Nous montrons au chap. 6 que la représentation en base b d'un nombre réel *calculable* n'est *pas*, en général, calculable.

Théorème 9 *Tout nombre réel positif $a \in \mathbf{R} > 0$ admet une écriture* infinie *en base $b \in \mathbf{N} + 2$, avec point fractionnaire, telle que :*

$$a = a_{n-1}\cdots a_0 \cdot a_{-1}\cdots a_{-k}\cdots = \sum_{n>k} a_k b^k.$$

1. *Les nombres rationnels de la forme $a = \frac{n}{b^i}$, avec $n, i \in \mathbf{N}$, ont* exactement deux *écritures équivalentes en base b ; l'une est finalement périodique, de période (0) ; l'autre est finalement périodique, de période $(b-1)$.*
2. *Pour tout nombre réel qui n'est pas de cette forme, l'écriture en base b est* unique.
3. *L'écriture du nombre a est finalement périodique si et seulement si $a \in \mathbf{Q}$ est rationnel.*

Preuve : L'algorithme 10 permet d'obtenir une représentation en base b de tout réel positif a.

1. Partant de (6.5), soit $1 \cdot (1)_b = \sum_{n\in\mathbf{N}} b^{-n} = \frac{1}{1-b^-} = \frac{b}{b-1}$, on vérifie que le nombre $a = a_{n-1}\cdots a_0 \cdot a_{-1}\cdots a_{-k}(0)_b$, avec $a_{-k} > 0$, est égal au nombre $a' = a_{n-1}\cdots a_0 \cdot a_{-1}\cdots a'_{-k}(b-1)_b$, avec $a'_{-k} = a_{-k} - 1$.
2. Soient $a = a_{n-1}\cdots a_0 \cdot a_{-1}\cdots a_{-k}\cdots$ et $a' = a'_{n-1}\cdots a'_0 \cdot a'_{-1}\cdots a'_{-k}\cdots$ deux nombres dont la représentation en base b est différente à partir du chiffre d'indice $-k$, soit $a_j = a'_j$ pour $j > -k$, et $a_{-k} < a'_{-k}$. La différence $d = a' - a$ entre ces deux valeurs s'exprime par la quantité $d = (a'_{-k} - a_{-k})b^{-k} + r$, avec $r = \sum_{j>k}(a'_{-j} - a_{-j})b^{-j}$. Il vient $d \geq b^{-k} + r$. Comme les a_{-j} et a'_{-j} sont des chiffres en base b, on a $0 \leq a_{-j}, a'_{-j} \leq b-1$, pour $j > k$, et donc $-b+1 \leq a'_{-j} - a_{-j}$. En sommant par (6.5), on encadre r au moyen de $r \geq -b^{-k}$, l'égalité n'étant possible que dans le cas ou $a'_{-j} = 0$ et $a_{-j} = b-1$, pour tout $j > k$. On trouve donc $a' > a$, sauf dans le cas 1 qui précède, où $a' = a$.

3. Considérons tout d'abord un nombre a dont l'écriture en base b est finalement périodique, soit :

$$\mathcal{E'}_b(a) = a_{n-1} \cdots a_1 a_0 \cdot a_{-1} a_{-2} \cdots a_{-i} (a_{-i-1} \cdots a_{-i-p})_b.$$

La valeur de ce nombre est $a = b^{-i}(e + x)$, expression dans laquelle e est l'entier dont l'écriture en base b est $\mathcal{E}_b(e) = a_{n-1} \cdots a_0 a_{-1} \cdots a_{-i}$, et x est le nombre d'écriture périodique $\mathcal{E'}_b(x) = 0 \cdot (a_{-i-1} \cdots a_{-i-p})_b$. Soit y le nombre entier dont l'écriture en base b est $\mathcal{E}_b(e) = a_{-i-1} \cdots a_{-i-p}$. Par périodicité, on a $x = 0 \cdot a_{-i-1} \cdots a_{-i-p} (a_{-i-1} \cdots a_{-i-p})_b$. La valeur de cette expression donne $x = b^{-i-p}(y + b^{-i-p-1}x)$. Ceci se résout explicitement par $x = \frac{n}{d}$, soit $x \in \mathbf{Q}$ rationnel, avec $n = yb^{-i-p}$ et $d = 1 - yb^{-2i-2p-1}$. En substituant dans l'égalité $a = b^{-i}(e + x)$, on prouve ainsi que $a \in \mathbf{Q}$ est aussi rationnel.

Inversement, partons d'un nombre rationnel $a = \frac{n}{d}$ que l'on écrit en base b au moyen de l'algorithme 10. Une fois écrite la partie entière $n \div d$ de a, il reste à écrire la partie fractionnaire $a' = a \cdot|\cdot 1 = \frac{n'}{d}$, avec $n' < d$. On pose $d \times A_0 = n'$ et, par construction, tous les nombres entiers $d \times A_k$ rencontrés dans l'algorithme 10 pour $k < 0$ sont tels que $0 \leq d \times A_k < d$. Après au plus d étapes de l'algorithme, on doit nécessairement rencontrer deux fois le même nombre, soit $A_{k'} = A_k$, avec $k < k'$. L'écriture devient alors périodique à partir de ce point, et $p = k' - k$ est la longueur de cette période. **Q.E.D.**

6.4 Limites pratiques du calcul

6.4.1 Mesures du calcul

1. Nombre de transistors.
2. Surface.
3. Période.
4. Profondeur combinatoire.
5. Bits par seconde.
6. Opérations par seconde.
7. Opérations par nano-seconde et nano-acre.
8. Opérations par τ et μ^2.

6.4.2 Complexité combinatoire

Face à la pléthore des formules possibles pour calculer chaque fonction combinatoire, que faire ?

Définition 20 *La* complexité *d'une fonction combinatoire f est le nombre* minimal *d'équations* **mux** *, parmi tous les circuits $\mathcal{C}_{ds}$ qui calculent f.*

Cette notion semble satisfaisante mathématiquement, car elle est relativement indépendante de la base combinatoire choisie. En effet, les fonctions $\neg, \cap, \cup$ sont toutes réalisables avec un seul **mux** ; cependant, la réalisation du ou-exclusif **xor** , du **nor** et du **nand** requièrent *deux* Mux. On en tiendra compte. Suit une bonne nouvelle.

Proposition 13 *Toute fonction combinatoire $f \in \mathbf{B}^n \to \mathbf{B}$ se réalise - par synthèse BDD - avec au plus $2^n/n$ multiplexeurs. La construction BDD est* statistiquement *(quasi) optimale.*

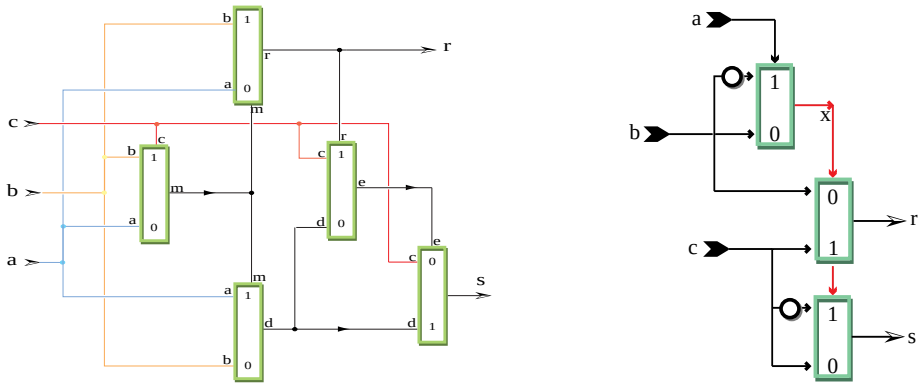
Preuve : ... **Q.E.D.**

Ce résultat indique que la synthèse BDD est *souvent* optimale, du moins en théorie. Par exemple, la synthèse $BDD(abc)$ comme celle $BDD'(abc)$ donne 7 portes. Ceci n'est pas optimal, au vu du circuit de Madre, en 5 **mux** - figure 1.3. Le circuit qui suit - en 3 portes - semble encore plus petit.

```
abc(a,b,c)   =   (s,r)                    // Full Adder en 3 équations
where
 x = a ^ b;                                          // x = a ⊕ b
 s = x ^ c;                                      // s = a ⊕ b ⊕ c
 r = mux(x, c, a);                          // r = ab ⊕ bc ⊕ ca
end where;
```

Cependant, une fois compilé en équations de base, ce circuit comporte aussi 5 **mux** car chaque **xor** vaut 2 multiplexeurs.

Proposition 14 (Madre) *La complexité de* **abc** *est 5 multiplexeurs ou inverseurs. A l'équivalence syntaxique près, Il y a exactement* deux *circuits qui calculent* **abc** *en 5* **mux** *:*



Preuve : La preuve de J.C. Madre est obtenue par une énumération exhaustive qui a demande plus d'une semaine de calculs sur ordinateurs en 94. Les deux solutions trouvées sont le circuit de Madre (découvert par ordinateur et présenté au chapitre 1) et le circuit ci-dessus (bien connu en électronique). **Q.E.D.**

Ce résultat est l'un des rares dans ce genre difficile. De tels résultats n'ont pas une incidence significative pour l'ingénieur dont le métier est de concevoir et réaliser des circuits petits (en nano-acre) et rapides (en nano-secondes).

Enfin, si la proposition 13 indique que la synthèse BDD est une *bonne* méthode pour synthétiser des fonctions sans structure évidente - comme $Dec7$, **abc** et $Dec24$ - elle se comporte très mal dans les cas ou une telle structure existe, par exemple en arithmétique.

Proposition 15 (Bryant) *La synthèse BDD d'un multiplicateur conduit à un circuit exponentiellement plus complexe que l'optimal - dans tous les cas.*

Preuve : Voir [8]. **Q.E.D.**

6.4.3 Complexité arithmétique

$$\begin{array}{rr} & 12345679 \\ \times & 81 \\ \hline = & 999999999 \end{array}$$

Figure 6.6 – Un produit décimal

Depuis l'école primaire, la pratique du calcul nous apprend que les produits sont plus longs à calculer que les sommes. Qu'en est-il pour les ordinateurs ? Formuler précisément cette question demande beaucoup de soin. Nous aimerions savoir quelle est la complexité intrinsèque de la multiplication c'est-à-dire :

parmi tous les algorithmes de multiplication, quel est le meilleur ?

Observons tout d'abord que la réponse à ces questions dépend étroitement du système choisi pour représenter les nombres entiers : Dans la représentation *unaire*, le produit $P = N \times M$, comme la somme $S = N + M$, se calculent en un nombre d'opérations proportionnel à la taille P ou S du résultat. Dans le système unaire, addition et multiplication, ont toutes deux une complexité linéaire. Cette observation manque quelque peu d'intérêt puisque nous savons, dans le système binaire, calculer les sommes en $ln_2(S)$ opérations, et les produits en $ln_2^2(P)$, soit exponentiellement plus vite qu'en unaire.

Limitons donc notre étude aux systèmes logarithmiques, dans lesquels tout entier n est représenté par $O(log(n))$ bits comme en binaire ou en représentation *factorisée* - exercice 10. Dans la représentation *factorisée* des nombres entiers les produits se calculent en un temps $O(log(P))$, proportionnel à la taille de la représentation du produit $P = A \times B$; les sommes semblent demander un temps de calcul $O(\sqrt{S})$ exponentiellement plus long ! Limitons nous donc aux calculs dans le système binaire. Nous comptons ici le nombre total d'opérations binaires effectuées ; que ces opérations aient lieu en parallèle, ou en série n'importe pas ici. Avec ces hypothèses, il résulte du chapitre 4 que la somme sur n bits se calcule en temps linéaire :

$$\mathcal{T}_+(n) = O(n),$$

alors que le produit nécessite en temps quadratique :

$$\mathcal{T}_\times(n) = O(n^2).$$

Même si l'on impose la représentation binaire, nous verrons dans ce chapitre plusieurs algorithmes de multiplication entière, dont le nombre d'opérations (temps de calcul séquentiel d'un produit $n \times n$ bits) est inférieur et donné par la table :

Algorithme :	Egypte	Karatsuba	K_k	Schönhague-Straßen
Temps :	n^2	$n^{log_2 3}$	$n^{log_k(2k-1)}$	$nlog(n)loglog(n)$

Ceci soulève des questions importantes, et difficiles : Comment peut-on analyser le temps d'exécution d'un calcul ? Quelles structures de données doit-on utiliser pour représenter les nombres, dans une machine donnée ? Quels sont les algorithmes optimaux pour une structure de machine donnée ? Quelles architectures de machines sont optimales, pour effectuer un calcul donné ?

Algorithme de Karatsuba

Soient à calculer, les $2k$ bits $P = A \times B = {}_2[p_0 \cdots p_{2k-1}]$ du produit de deux entiers de k bits, $A = {}_2[a_0 \cdots a_{k-1}]$ et $B = {}_2[b_0 \cdots b_{k-1}]$. Supposons $k = 2d$ pair, et découpons A et B en deux moitiés de d bits chacune :

$$A = A_0 + xA_1,\ B = B_0 + xB_1,\ \text{avec } x = 2^d. \tag{6.6}$$

Dans l'équation (6.6), A_0 et A_1 (respectivement B_0 et B_1) représentent le reste et le quotient de la division entière de A (respectivement B) par $x = 2^d$:

$$\begin{aligned} A_0 = A \dot{\vert} 2^d = {}_2[a_0 \cdots a_{d-1}], \qquad & B_0 = B \dot{\vert} 2^d = {}_2[b_0 \cdots b_{d-1}], \\ A_1 = A \div 2^d = {}_2[a_d \cdots a_{2d-1}], \qquad & B_1 = B \div 2^d = {}_2[b_d \cdots b_{2d-1}]. \end{aligned}$$

Quand A et B sont initialement présentés en binaire, le calcul de A_0, A_1 et B_0, B_1 se fait en temps proportionnel à k. Ecrivons le produit

$$\begin{aligned} P(x) &= (A_0 + xA_1) \times (B_0 + xB_1) \\ &= A_0B_0 + x(A_0B_1 + A_1B_0) + x^2A_1B_1 \\ &= P_0 + xP_1 + x^2P_2 \end{aligned}$$

sous forme d'un polynôme du deuxième degré en la variable x, que l'on évalue en $x = 2^d$ pour trouver $A \times B = P = P(2^d)$. Etant donnés P_0 et P_2, chacun présenté en binaire sur k bits, P_1 sur $k + 1$ bits, le calcul de

$$P = P_0 + 2^d(P_1 + 2^dP_2)$$

s'effectue, par additions et décalages, en un temps proportionnel à k.

Les nombres P_0, P_1, P_2 peuvent s'obtenir en calculant les quatre produits :

$$\begin{array}{rr} A_0 \times B_0 \Rightarrow \Pi_1 & \Pi_1 \Rightarrow P_0 \\ A_0 \times B_1 \Rightarrow \Pi_2 & \\ A_1 \times B_0 \Rightarrow \Pi_3 & \Pi_2 + \Pi_3 \Rightarrow P_1 \\ A_1 \times B_1 \Rightarrow \Pi_4 & \Pi_4 \Rightarrow P_2 \end{array} . \tag{6.7}$$

On peut aussi reconstruire P_0, P_1 et P_2 à partir des trois produits :

$$\begin{array}{rr} A_0 \times B_0 \Rightarrow \Pi_1 & \Pi_1 \Rightarrow P_0 \\ (A_0 + A_1) \times (B_0 + B_1) \Rightarrow \Pi_2 & \Pi_1 - \Pi_2 - \Pi_3 \Rightarrow P_1 \\ A_1 \times B_1 \Rightarrow \Pi_3 & \Pi_3 \Rightarrow P_2 \end{array} . \tag{6.8}$$

Le calcul (6.7) comprend 4 multiplications et une addition ; le calcul (6.8) comprend 3 multiplications, 2 additions et 2 soustractions. Ainsi, (6.8) échange avec (6.7), une multiplication en temps quadratique, contre une addition et deux soustractions, en temps linéaire. Pour k assez grand, on gagne au total à utiliser (6.8) plutôt que (6.7).

Ce gain devient significatif lorsqu'on utilise la méthode de manière *récursive*.

Algorithme 11 (Multiplication de Karatsuba)

Pour multiplier $A = {}_2[a_0 \cdots a_{k-1}]$ *par* $B = {}_2[b_0 \cdots b_{k-1}]$ *:*

1. *Si* $k \leq k_0$*, on utilise l'algorithme 7 de multiplication égyptienne, en* $O(k^2)$ *opérations. La valeur de* k_0 *dépend de la machine et du codage précis de l'algorithme ; le plus simple est de déterminer expérimentalement la valeur de cette constante pour une implantation donnée (exercice 11).*
2. *Si* $k > k_0$*, on pose* $d = k \div 2, x = 2^d$ *et on calcule*

$$A_0 = A \cdot|\cdot 2^d,\ B_0 = B \cdot|\cdot 2^d,\ A_1 = A \div 2^d,\ B_1 = B \div 2^d.$$

En utilisant de façon récursive *l'algorithme 11 que nous sommes en train de décrire, on calcule les trois produits :*

$$\begin{aligned} A_0 \times B_0 &\Rightarrow \Pi_0, \\ (A_0 + A_1) \times (B_0 + B_1) &\Rightarrow \Pi_1, \\ A_1 \times B_1 &\Rightarrow \Pi_\infty. \end{aligned}$$

Le produit final $P = A \times B$ *s'obtient par :*

$$\Pi_0 + x(\Pi_1 - \Pi_0 - \Pi_\infty + x\Pi_\infty) \Rightarrow P.$$

Les exercices 12 et 13 montrent que :

Proposition 16 *L'algorithme 11 de Karatsuba permet le calcul de produits sur* k *bits en*

$$ck^{log_2(3)} + c'k$$

opérations ; ici, $log_2(3) = 1.585\cdots$ *et* c, c' *sont des constantes dépendantes de la machine et de l'implantation.*

L'algorithme 11 de Karatsuba calcule le produit de polynômes

$$P(x) = (A_0 + xA_1)(B_0 + xB_1) = (P_0 + xP_1 + x^2P_2)$$

de degré 2, par évaluation en trois points

$$\Pi_0 = P(0), \Pi_1 = P(1), \Pi_\infty = P(\infty),$$

et reconstruction du résultat par interpolation.

Si on partage maintenant nos nombres en 3, on peut calculer le produit des polynômes

$$P(x) = (A_0+xA_1+x^2A_2)(B_0+xB_1+x^2B_2) = (P_0+xP_1+x^2P_2+x^3P_3+x^4P_4)$$

par évaluation en 5 points simples, par exemple :

$$\Pi_0 = P(0), \Pi_{\overline{1}} = P(-1), \Pi_1 = P(1), \Pi_2 = P(2) \Pi_{\overline{2}} = P(-2).$$

Ceci permet de reconstruire P en temps linéaire, par interpolation. Le temps de calcul $K_3(N)$ de cet algorithme satisfait la relation :

$$K_3(N) = 5K_2(\frac{N}{3}) + aN,$$

dont la solution (exercice 12) est :

$$K_3(N) = O(N^{log_3 5}) = O(N^{1.465\cdots}) \leq O(N\sqrt{N}).$$

Plus généralement :

Algorithme 12 (Algorithme K_k) *Pour calculer le produit de deux entiers A, B de longueur $n = km$ bits :*

1. *On forme les deux polynômes $A(x)$ et $B(x)$ tels que $A(2^m) = A$, et $B(2^m) = B$. On évalue ensuite $A(x)$ et $B(x)$ en $2k-1$ points :*

$$\begin{aligned} A(x_1) \Rightarrow A_1, \quad \cdots, \quad A(x_{2k-1}) \Rightarrow A_{2k-1}, \\ B(x_1) \Rightarrow B_1, \quad \cdots, \quad B(x_{2k-1}) \Rightarrow B_{2k-1}. \end{aligned}$$

2. *En utilisant récursivement (pour $N > N_0$ assez grand) l'algorithme 12 K_k, on calcule les $2k-1$ produits :*

$$A_1 \times B_1 \Rightarrow P(x_1), \quad \cdots \quad A_{2k-1} \times B_{2k-1} \Rightarrow P(x_{2k-1}).$$

3. *On reconstruit le polynôme $P(x) = {}_x[P_0\, P_1 \cdots P_{2k-1}]$ par des formules d'interpolation propres aux points $\{x_1, \cdots x_{2k-1}\}$ choisis.*
4. *On évalue*

$$P(2^k) \Rightarrow P$$

pour trouver le produit final entier $P = A \times B$.

Soit $K_k(N)$ le temps de calcul de l'algorithme K_k sur N bits. On a :

$$K_k(N) = (2k-1)K_k(\frac{N}{k}) + akN,$$

d'où il résulte (exercice 12) que : $K_k(N) = N^{\alpha_k(N)}$, avec

$$\alpha_k(N) = log_k(2k-1) = \frac{log(2k-1)}{log(k)} \simeq 1 + \frac{1}{log(k)}.$$

Ainsi, $K_3(N) \simeq N^{1.404}, K_4(N) \simeq N^{1.365}, \cdots, K_9(N) \simeq N^{1.238}, \cdots$.

Comme $\alpha_k(N)$ devient arbitrairement proche de 1 pour k assez grand, nous venons de montrer que :

Proposition 17 *Pour tout $\epsilon > 0$, la complexité du calcul du produit entier est, au plus :*

$$\mathcal{T}_\times(N) = O(N^{1+\epsilon}).$$

Division de Newton

Montrons maintenant que l'opération de division a même complexité, à une constante multiplicative près, que la multiplication.

Définition 21 *L'* inverse à n bits *du nombre D, pour $\frac{1}{2} \leq D < 1$, est un nombre $I_n(D)$ tel que :*

$$|D \times I_n(D) - 1| < \frac{1}{2^n}.$$

Exemple 9 *L'inverse à 10 bits du nombre $N = \frac{153}{128} = {}_{\frac{1}{2}}1.0011001$ est le nombre : $I_{10}(N) = \frac{107}{128} = {}_{\frac{1}{2}}0.1101011$. En effet,*

$$N \times I_{10}(N) = \frac{13371}{16384} = 1 - \frac{13}{16384} = 1 - {}_{\frac{1}{2}}0.00000000001101.$$

Le calcul de la division entière $(N, D) \mapsto (N \div D, N \cdot|\cdot D)$ se réduit à celui de l'inverse à $n = ln_2(N)$ bits ; en effet, le nombre $q = N \times I_n(D)$ est proche du quotient, car :

$$|q - \frac{N}{D}| = |\frac{N}{D}(1 - D \times I_n(D))| < 1.$$

Partant de q, on obtient le quotient en calculant $N - qD \Rightarrow r$, puis en posant :

$$N \div D = \begin{cases} q-1 & \text{si } r < 0, \\ q & \text{si } 0 \leq r < D, \\ q+1 & \text{si } r \geq D. \end{cases}$$

Pour calculer l'inverse d'un nombre D tel que $\frac{1}{2} \leq D < 1$, on pose $x = 1 - D \in]0 \cdots \frac{1}{2}]$ et on utilise la formule :

$$\mathcal{I}(x) = \frac{1}{1-x} = (1+x)\frac{1}{1-x^2} = \prod_{0 \leq i < n} (1 + x^{2^i}) \frac{1}{1 - x^{2^n}}. \qquad (6.9)$$

Algorithme 13 (Inverse par la méthode de Newton)
On obtient l'inverse à N bits $I_N(D)$ d'un nombre D tel que $\frac{1}{2} \leq D \leq 1$ en calculant $\mathcal{I}(1-D, N)$ par les règles :

$$\begin{aligned} \mathcal{I}(X,1) &\Rightarrow 1, \\ \mathcal{I}(X,2N) &\Rightarrow (1+X) \times \mathcal{I}(X^2, N). \end{aligned}$$

On a, bien entendu, $\mathcal{I}(x, 2^p) \stackrel{p}{\Rightarrow} \prod_{0 \leq i < p}(1 + x^{2^i})\mathcal{I}(x^{2^p}, 1)$, et l'algorithme 13 termine en $n = ln_2(N)$ étapes. Soit $\mathcal{T}_\times(N)$ la complexité du produit N bits, et $I(N)$ le nombre d'opérations effectuées dans le calcul de l'inverse par l'algorithme 13. On a :

$$I(N) = 2\mathcal{T}_\times(N) + I(\frac{N}{2}) = 2\sum_{i \geq 0} \mathcal{T}_\times(\frac{N}{2^i}).$$

Comme $\mathcal{T}_\times(2N) > 2\mathcal{T}_\times(N)$, il vient :

Proposition 18 *La complexité $\mathcal{T}_/(N)$ du calcul de l'inverse à N bits est, au plus, 4 fois celle de la multiplication :*

$$\mathcal{T}_/(N) \leq 4\mathcal{T}_\times(N).$$

Remarquons enfin que l'algorithme 13 peut aussi servir à construire un circuit de division. Un tel diviseur n bits est constitué de $2n$ multiplicateurs, connectés suivant les plans de l'algorithme 13. En utilisant des multiplicateurs en temps logarithmique (algorithme 9), ceci conduit à une division sur n bits, en profondeur combinatoire $O(log^2(n))$.

Exercices

Exercice 10 (Décomposition Première) *Soit*

$$P = \{p_n \in \mathbf{N} \ : p_n \textit{ est le n-ième nombre premier}\}$$

la suite des nombres premiers $P = 2, 3, 5, 7, 11, 13, 17, \cdots$.
La correspondance de Gödel représente tout entier $N > 0$ par la suite finie

$$N = (e_0, e_1, \cdots, e_k),$$

de ses exposants, jusqu'au dernier non nul $e_k \neq 0$, dans la décomposition

$$N = \prod_n p_n^{e_n},$$

de N en un produit de puissances premières. En déduire une bijection calculable :

$$\mathbf{N} \leftrightarrow \mathbf{N}^* = \{\textit{ensemble des suites finies d'entiers}\}.$$

1. *Montrer que la taille de la représentation factorisée de N est, au plus, $K \times log(N)$, pour une constante K que l'on déterminera explicitement..*
2. *Ecrire un algorithme, utilisant les cinq opérations de l'arithmétique comme primitives, qui calcule le décodage $\mathbf{N}^* \mapsto \mathbf{N}$ en $O(log^2(N))$ opérations.*
3. *Ecrire un algorithme de codage $\mathbf{N} \mapsto \mathbf{N}^*$ en temps $O(\sqrt{N})$; notez que ce temps de calcul est exponentiel en la taille $log(N)$ du résultat calculé.*
4. *Ecrire un algorithme de calcul du produit de deux nombres en représentation factorisée ; son temps de calcul doit être linéaire en la taille $log(N)$ du produit N calculé.*
5. *Ecrire un algorithme de calcul de la somme de deux nombres en représentation factorisée, en temps $O(\sqrt{N})$.*

Exercice 11 *Coder l'algorithme 11 de Karatsuba sur votre machine favorite, en utilisant votre langage de prédilection. Déterminer, par des mesures de temps d'exécution, la valeur de la constante k_0.*

Exercice 12 *Soient a, b, c et d des constantes positives. Résoudre explicitement la récurrence*

$$T(n) = \begin{cases} d & \textit{pour} \quad n = 1, \\ aT(n/c) + bn & \textit{pour} \quad n > 1, \end{cases}$$

dans le cas particulier ou $n = c^m$ est une puissance de c. En déduire une formule pour le cas général $n \in \mathbf{N}$ du type

$$T(n) = \begin{cases} O(n), & \textit{si} \quad a < c, \\ O(nlogn), & \textit{si} \quad a = c, \\ O(n^{log_c a}), & \textit{si} \quad a > c, \end{cases}$$

dans laquelle on rendra explicites au premier ordre toutes les constantes qui se cachent sous la notation grand oh $O(n)$ *de Landau.*

Exercice 13 *Utiliser l'exercice 12 pour démontrer la proposition 16.*

Exercice 14 *Programmer l'algorithme K_3*

$$_x[a_0a_1a_2] \times {}_x[b_0b_1b_2] = {}_x[p_0p_1p_2p_3p_4].$$

à partir des formules d'interpolation :

$$\begin{array}{rcl} \pi(0) = & a_0b_0 & = p_0, \\ \pi(1) = & (a_0 + a_1 + a_2)(b_0 + b_1 + b_2) & = p_0 + p_1 + p_2 + p_3 + p_4, \\ \pi(-1) = & (a_0 - a_1 + a_2)(b_0 - b_1 + b_2 & = p_0 - p_1 + p_2 - p_3 + p_4, \\ \pi(2) = & (a_0 + 2a_1 + 4a_2)(b_0 + 2b_1 + 4b_2) & = {}_{p_0+2p_1+4p_2+8p_3+16p_4}, \\ \pi(\infty) = & a_2b_2 & = p_4, \end{array}$$

que l'on devra d'abord inverser.

6.4.4 Complexité exponentielle

Problèmes NP-complets. Voir le cours de Jacques Stern.

III

Applications

Chapitre 7

Physique digitale

Contents

Visitons trois endroits où la physique expérimentale rencontre l'informatique.

– Dans le modèle du monde que nous livrent Newton et Laplace, la position des astres est représentée par des nombres réels qui varient continûment avec le temps. Pourtant, toute mesure physique se fait avec une précision finie. Qui plus est, toute mesure prend forcement un temps non nul. Il en résulte que la position mesurée des astres est une suite *discrète* de nombres *entiers* que l'on peut représenter - sans perdre en généralité - par une *suite binaire*. Ce point de vue *digital* - où le temps et les valeurs sont discrets - vaut pour toute la physique expérimentale. Il rejoint d'ailleurs le point de vue théorique de la mécanique quantique ; on consultera avec bonheur [17] et [15] à ce sujet.

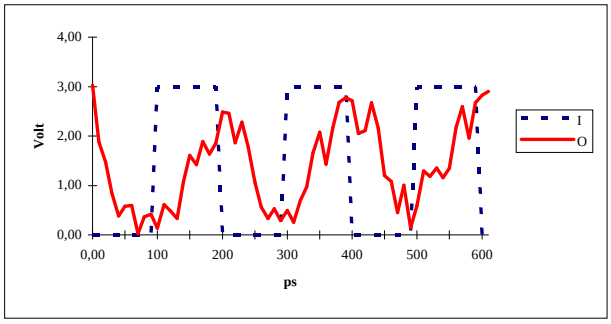
  Notre propos se limite à présenter certains convertisseurs entre les mondes analogiques et digitaux, pour les circuits électroniques.

– Une fois acquise la mesure digitale des paramètres, il reste à l'exploiter. La fonction de l'appareil de mesure est alors un *algorithme* qui traite les données numériques captées en *temps réel*, c'est à dire au fil de leur arrivée. Nous illustrons ce propos d'un appareil qui calcule à la volée les trajectoires de certaines particules en physique des hautes énergies.

– Souvent, la mesure directe de certains paramètres physiques n'est pas possible. Soit il y en a trop ; soit ils sont inaccessibles ; soit l'objet à mesurer n'existe pas encore hors du cerveau de son concepteur, qui cherche à vérifier si oui ou non la chose peut marcher. La solution est alors de fabriquer une maquette informatique du système. Dans ce *monde virtuel*, on fait alors des suites mesures, par simulation numérique.

  Nous traitons un cas où il faut calculer les températures atteintes à l'intérieur d'un circuit en fonctionnement - en tout point et à tout instant. Ceci pour vérifier si le circuit ne va pas griller dès que l'on branchera alimentations et horloge.

  Nous suivons ici des lois connues de la physique - équation de la chaleur pour l'exemple. Rien n'empêche le lecteur de suivre son imagination : construire un *monde virtuel*, où les lois de la physique seraient différentes des nôtres ; le programmer et en visualiser les conséquences.



Signal émis I et reçu O

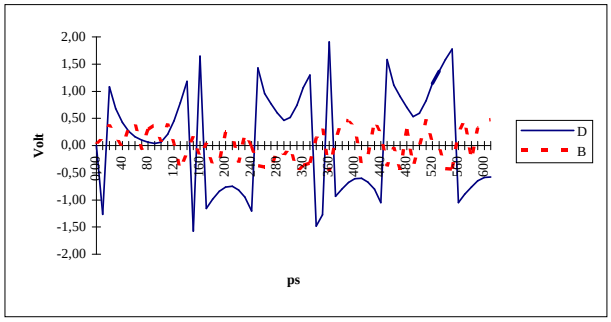


Figure 7.1 – Bruit B et distorsion D

7.1 Mesure numérique

Toute mesure physique a une précision *finie* et demande un temps *non nul*. La pratique expérimentale nous apprend aussi qu'il n'est pas possible de reproduire une mesure numérique à l'identique : les derniers chiffres décimaux lus varient, au-delà d'un certain rang ; le rang du dernier chiffre stable que l'on arrive à lire est la précision de la mesure.

7.1.1 Bruit, et limites de la mesure

On explique ces perturbations des mesures par l'existence de *bruit*, processus aléatoire qui vient inévitablement s'ajouter au phénomène que l'on tente de quantifier expérimentalement. On entend ce bruit dans une radio, et on le voit sur une télévision : quand la station cesse d'émettre. Ce sont deux exemples de *bruit blanc*. A ces bruits internes viennent s'ajouter de multiples bruits externes. Tous les appareils du voisinage induisent un bruit sur notre alimentation électrique. Même si on découple soigneusement cette alimentation, des particules cosmiques viennent heurter en permanence les composants. La charge qui mémorise un bit de mémoire

dans les circuits dRAM est si faible que diverses particules ont assez d'énergie pour l'inverser - et contribuer, à l'occasion, aux déboires de l'ordinateur d'une station spatiale. Il est bien entendu de multiples autres sources de bruits, propres à chaque appareil de mesure et à l'environnement dans lequel il opère.

Cherchons par exemple à mesurer la différence de tension U aux bornes d'une résistance, avec une précision de l'ordre du μV - soit 10^{-6} volt. On voit apparaître des fluctuations aléatoires de la tension mesurée. Ce phénomène a été mis en évidence expérimentale par Johnson en 1928. On peut représenter la tension mesurée par la somme $U + B$, où B est une variable aléatoire : de moyenne nulle, et dont la densité spectrale de puissance est *uniforme*. Comme cette puissance est proportionnelle à la température du système (et qu'elle ne dépend de rien d'autre, en particulier pas de la tension U) on parle de *bruit thermique*. Il résulte inexorablement du mouvement brownien des électrons dans les conducteurs - voir la simulation de ce phénomène en section 7.4.

Le *bruit thermique* ne perturbe pas encore les circuits digitaux, dont la tension utile reste encore 10^4 fois supérieure. Il devient un facteur limitatif pour certains circuits analogiques, comme le voltmètre de haute précision, ou le récepteur de transmissions spatiales. La théorie de Shannon montre en effet que le nombre n de bits exacts dans la mesure numérique d'un signal est intrinsèquement borné par la fonction suivante du rapport entre la puissance P du signal et celle N du bruit :

$$n < \frac{1}{2} log_2(1 + \frac{P}{N}).$$

Notre exposé du chap. 8 se limite au cas discret ; renvoyons donc à [25] pour une démonstration de cette limite fondamentale à la précision des *mesures en continu*.

Notre dernière observation est que l'on peut améliorer la précision d'une mesure, au détriment du temps nécessaire à effectuer cette mesure. Ce compromis fondamental entre la précision Δx et la fréquence Δf de la mesure est donnée par la *relation d'incertitude* :

$$\Delta x \Delta f \geq \frac{1}{4\pi}.$$

On trouve la preuve de cette relation mathématique dans les ouvrages sur la *Transformée de Fourier* [23]. En changeant le quantum d'information en quantum d'énergie, on fera l'analogie avec le principe d'incertitude de Heisenberg en mécanique quantique [15].

Tout ceci limite en définitive le *débit* de la mesure en continu de tout paramètre du monde physique ; ce débit s'exprime en bits par seconde, et on rejoint ici le point de vue de la théorie de l'information - chap. 8.

7.1.2 Conversion digitale/analogique

Etudions les techniques de mesure par circuits électroniques : elles sont aujourd'hui partie prenante dans *tout* appareil de haute précision.

Amplificateur différentiel

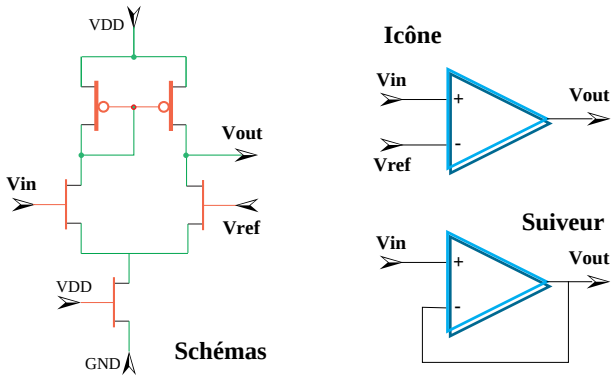


Figure 7.2 – Amplificateur différentiel

L'*amplificateur différentiel* de la fig. 7.2 est la brique de base avec laquelle nous allons réaliser divers circuits *analogiques*.

Dans un circuit digital à l'état stable, les tensions ont une valeur logique définie - soit $V < V_{th}$ pour le zéro logique, soit $V >$ **vdd** $- V_{th}$ pour le un - et tous les transistors sont alors, soit *bloqués*, soit *saturés*.

Dans un circuit analogique comme l'amplificateur différentiel, on opère dans la plage de tension $V_{th} < V <$ **vdd** $- V_{th}$ où les transistors sont dans un état intermédiaire - dit *résistif*. Pour Vin et $Vref$ dans cette plage, on observe alors un courant résiduel dans le montage, qui est la somme $I = I_g + I_d$ de ceux qui passent respectivement dans les deux chemins - gauche pour I_g et droit pour I_d - entre l'alimentation **vdd** et la terre **gnd** dans les schémas de la fig. 7.2.

Par le montage - dit en *diode* - de ces schémas, la grille des deux transistors pMOS est commune. Par construction, ils ont donc toujours la même résistance. Quand $Vin > Vref$, la conductance du transistor nMOS de décharge à gauche est supérieure à celle de droite et on a $I_g > I_d$; inversement, $I_g < I_d$ quand $Vin < Vref$. On peut vérifier que la réponse en courant de l'amplificateur différentiel est donnée par la formule $Ig - Id = I \times tanh(k(Vin - Vref))$, où $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ est la *tangente hyperbolique*.

En mode *suiveur*, on boucle la sortie sur l'entrée $Vref = Vout$ - voir fig. 7.2. La tension de sortie $Vout$ de l'amplificateur différentiel est proportionnelle à la différence $Vin - Vref$ entre les tensions d'entrée, soit $Vout = g(Vin - Vref)$; ici le nombre $g > 1$ est le *gain* de l'amplificateur. En substituant $Vref = Vout$, il vient :

$$Vout = \frac{g}{1+g} Vin \approx (1 - \frac{1}{g}) Vin.$$

La tension de sortie du circuit suiveur est proportionnelle à la tension d'entrée. Le circuit suiveur permet donc de *copier* une tension analogique d'entrée, en une sortie de tension proportionnelle, dans un rapport $1 - \frac{1}{g}$ proche de l'unité.

Convertisseur analogique/digital

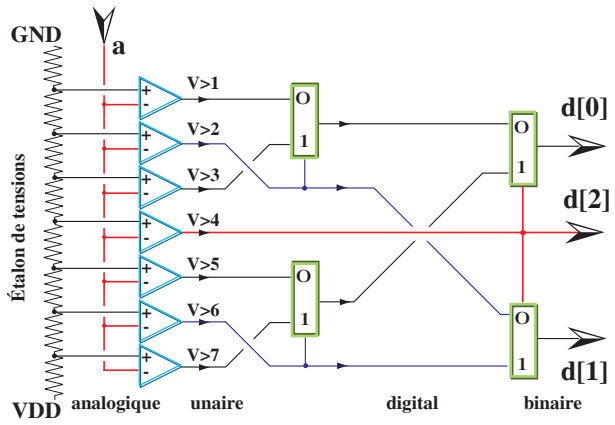


Figure 7.3 – Convertisseur A/D parallèle sur 3 bits

Si l'on fixe $Vref = \mathbf{vdd}/2$, la sortie $Vout$ de l'amplificateur différentiel devient digitale - $Vout < V_{th}$ pour $Vref < \mathbf{vdd}/2 - V$ et $Vout > \mathbf{vdd} - V_{th}$ pour $Vref > \mathbf{vdd}/2 + V$ - sauf quand Vin est dans la plage entre $\mathbf{vdd}/2 - V$ et $\mathbf{vdd}/2 + V$, dont la largeur $2V$ est typiquement petite, quelques dixièmes de Volt. Hors de cette fine plage, le montage se comporte comme un convertisseur analogique/digital sur un bit.

En combinant $7 = 2^3 - 1$ convertisseurs sur un bit suivant les schémas de la fig. 7.3, on obtient un convertisseur A/D - analogique vers digital - parallèle sur 3 bits.

L'entrée analogique a du convertisseur A/D est une tension électrique. Sa sortie digitale $d[0, 2]$ de D/A se compose des trois bits $d[0], d[1], d[2]$, qui représentent en binaire la valeur $D = d[0] + 2d[1] + 4d[2]$ du potentiel électrique de l'entrée analogique, rapporté à une tension maximale de référence.

Convertisseur digital/analogique

Le convertisseur D/A - digital vers analogique, fig. 7.4 sur 3 bits a une fonction inverse de celle du convertisseur A/D.

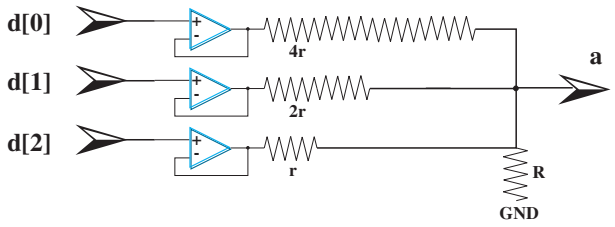


Figure 7.4 – Convertisseur D/A sur 3 bits

L'entrée digitale $d[0,2]$ de D/A se compose des trois bits $d[0], d[1], d[2]$, qui codent en binaire l'entier $D = d[0] + 2d[1] + 4d[2]$, et donc $0 \leq D \leq 7$. La sortie analogique a de D/A est une tension électrique proportionnelle à D fois la tension maximum en sortie. On choisit la résistance à la terre R grande devant r.

7.1.3 Voltmètre numérique

En généralisant de 3 à N le montage A/D en parallèle de la fig. 7.3, on voit que la taille de la partie analogique d'un tel circuit croît exponentiellement avec n. Il y a en effet 2^N résistances et amplificateurs différentiels. En dépit de la précision des procédés de gravure sur Silicium évoqués au chapitre 3, chaque élément a des caractéristiques électriques *analogiques* légèrement différentes de ses copies voisines. Chaque micron carré de circuit apporte ainsi sa contribution au *bruit* ambiant. Avec un nombre exponentiel de composants analogiques dans le montage, on passe vite le seuil ou le bruit propre à l'instrument de mesure dépasse celui du phénomène à mesurer. Les convertisseurs A/D en parallèle du type de la fig. 7.3 sont donc *rapides*, mais *limités* en précision.

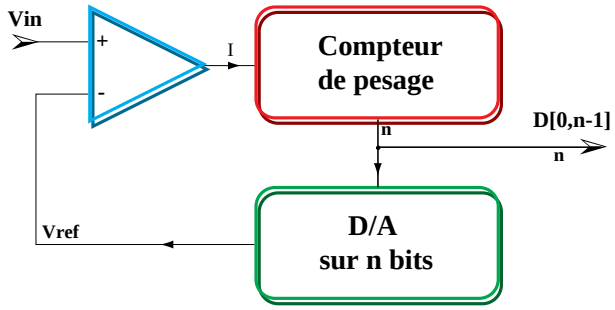


Figure 7.5 – Convertisseur A/D série sur n bits

Le principe d'un convertisseur A/D en série, plus précis mais plus lent, est donné dans la fig. 7.5. Un *compteur de pesage* propose en permanence une valeur digitale de sortie $D[0, N-1]$ sur N bits. Cette valeur $D = \sum D[k]2^k$ est transformée en une tension analogique proportionnelle $Vref = kD$ par un convertisseur D/A comme celui de la fig. 7.4. La tension sert de référence pour l'*unique* amplificateur

différentiel du montage, qui la compare avec l'entrée analogique Vin. La valeur digitale I indique au compteur de pesage que $k^{-}Vin < D$ pour $I = 0$ et $k^{-}Vin > D$ pour $I = 1$.

On peut réaliser le pesage au moyen d'un compteur synchrone, que l'on incrémente systématiquement d'une unité quand $I = 1$, et que l'on décrémente quand $I = 0$ - voir l'exercice 1 au chap. 4. Cette solution simple est satisfaisante si le signal d'entrée varie de moins d'une unité pendant une période du compteur. C'est par exemple le cas pour un numériseur de son *HiFi*. Si la valeur à numériser change brutalement, comme quand on change de pixel dans la caméra CCD de la fig. 7.7, il faut utiliser le compteur par dichotomie de l'exercice 15, dont la valeur finale est acquise en N cycles pour N bits. Dans ce contexte, il faut 2^{N} cycles au compteur $up - down$ pour arriver au même résultat.

Exercice 15 *Concevoir un compteur de pesage par dichotomie. A chaque cycle du compteur, on acquiert un bit du compte final, en allant des poids forts vers les poids faibles ; il faut donc* N *cycles pour* N *bits.*

7.2 Caméra digitale

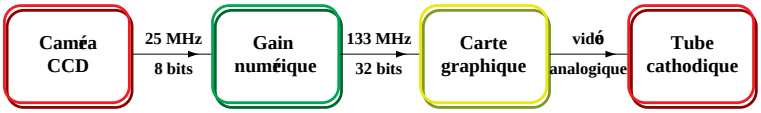


Figure 7.6 – Chaîne d'acquisition numérique d'images

Présentons un système complet, issu d'une réalisation en imagerie médicale par fibroscopie - fig. 7.6. Le système est une chaîne de traitements - *pipe-line* : sa source est une caméra CCD, qui communique les images acquises au circuit de *gain statique* ; après traitement, celui ci les communique au circuit de *gain dynamique*, qui les traite et les communique à la mémoire d'un *ordinateur* ; les images sont alors reprises par un *contrôleur graphique*, qui les communique au destinataire final : l'écran cathodique que consulte le médecin.

A chaque stade du pipe-line, on trouve une combinaison des trois éléments : *communication, mémorisation* et *traitement*. Les images sont *modulées*, au fil du pipe-line, par diverses représentations des pixels, dans l'espace et le temps.

L'*invariant* le plus important de ce pipe-line *temps réel* est sa *bande passante* : elle reste identique de sa source à la destination. Ceci est caractéristique des pipe-lines *réversibles* : on peut reconstruire l'entrée à partir de la sortie. C'est par exemple le cas de tout système de télécommunication sans erreur. C'est aussi le cas en imagerie médicale : l'éthique veut qu'on archive toujours le document original, pas le résultat d'un traitement numérique. La fonction des deux étapes de *gain* dans notre pipe-line est donc d'améliorer la lisibilité de l'image par le médecin, tout en se limitant à des transformations réversibles.

La caméra est une matrice carrée de 1Kx1K= 1M capteurs CCD, soit 2^{20} pixels. La numérisation se fait avec une précision de 8 bits par pixel, et à la fréquence de 25 images par secondes. La bande passante de cette *source* vaut : $25 \times 8 \times 1M = 200$ Mb/s - méga-bit par seconde ; soit aussi 25 MB/s - méga-octet (byte) par seconde.

7.2.1 Convertisseur de photons en électrons : CCD

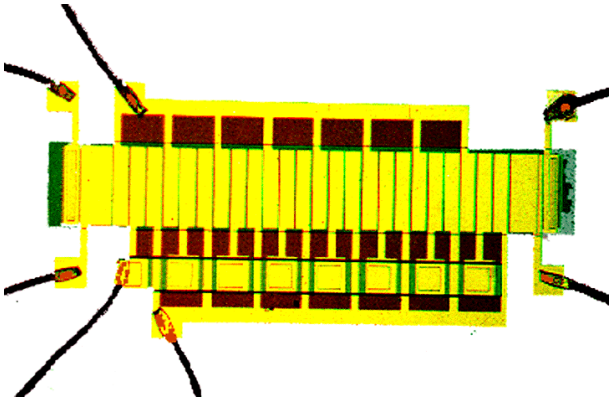


Planche 7.1 – Premier circuit CCD, Bell Laboratories, 1970

Un CCD *charge-coupled device* est un condensateur MOS. Il récupère, sous forme de charge électrique, l'énergie que dissipent les photons de lumière en tombant dessus. En *numérisant* à intervalles réguliers la valeur d'un tableau de CCD - fig. 7.1 - on obtient une caméra numérique à *état solide*, sans film ni développement photographique. A quelque $ l'unité, la caméra CCD - de surveillance, observation, pilotage, analyse et contrôle - fait partie de notre futur.

On dispose les capteurs CCD en matrice sur une *puce*, suivant les plans de la fig. 7.7. Dans ce montage, un *bus commun* relie tous les pixels (un seul à la fois, comme l'assure le contrôle du transistor) à un convertisseur A/D *unique*. Dans les technologies à *transfert de charge*, la valeur des pixels est communiquée au travers d'un *registre à décalage analogique*, qui permet de communiquer la mesure de chaque pixel sans perte de précision. Le montage que nous proposons est bien plus simple, mais il est moins précis : la charge du pixel est diluée par la (forte) capacité du bus, d'où une perte d'information lors du transfert. Ignorons la ici, pour simplifier.

Les contrôles du bus assurent que les pixels sont connectés au convertisseur *chacun à son tour*, dans l'ordre du *balayage télévision*. Après *numérisation*, le pixel lu est remis à zéro, et on passe au suivant. Chaque récepteur CCD de la matrice intègre donc la lumière reçue pendant une période de rafraîchissement, soit 1/25 ième de seconde.

Exercice 16 *Concevoir les détails de la logique de contrôle de la caméra numérique de la fig. 7.7.*

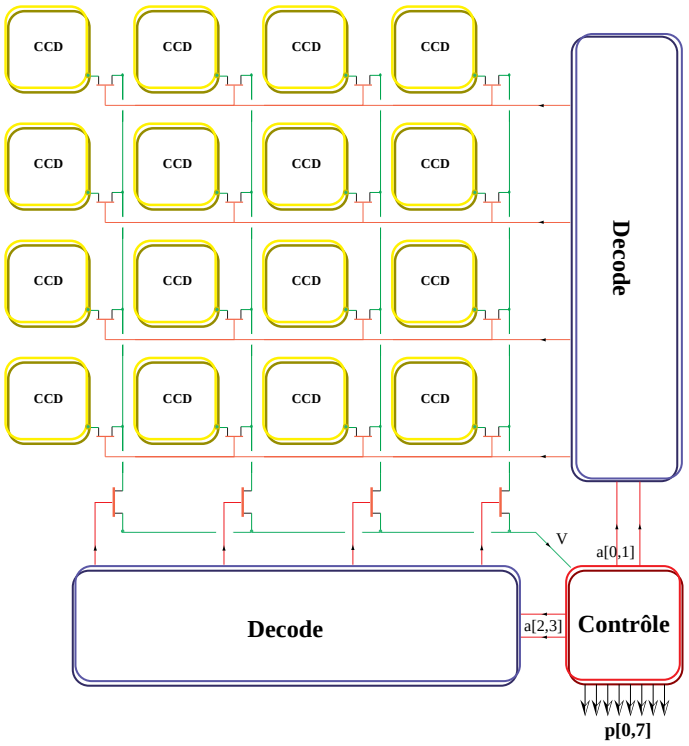


Figure 7.7 – Schéma de principe d'une caméra CCD 16 pixels

7.2.2 Réception asynchrone

Métastabilité du registre

Fenêtre d'acquisition

7.2.3 Amélioration numérique de l'image

Gain statique

Gain dynamique

7.3 Détecteur de particules

Le texte de référence pour cette section est [20], distribué en support de cours.

7.3.1 Problème physique

7.3.2 Droite digitale

7.3.3 Transformée d'Hough rapide

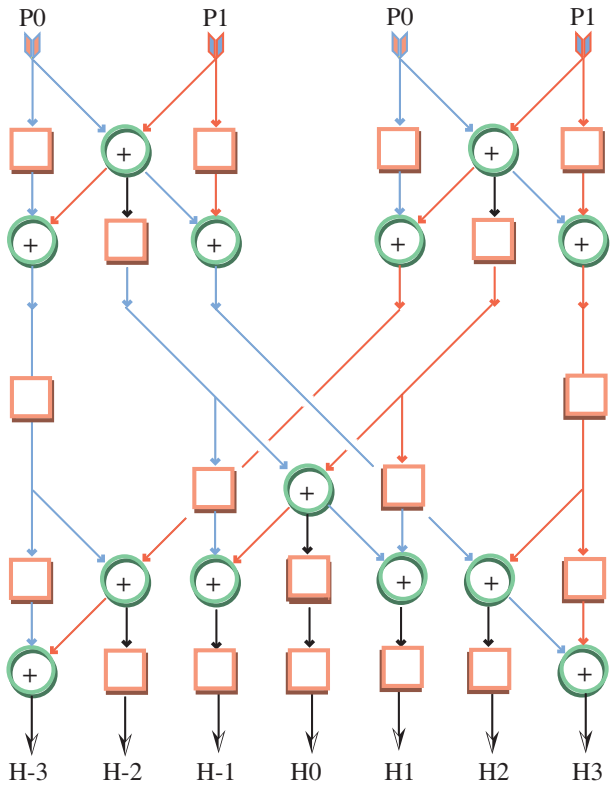


Figure 7.8 – Transformée d'Hough rapide, sur 4 points

7.4 Equation de la chaleur

Le texte de référence pour cette section est [20], distribué en support.

7.4.1 Du continu au discret

7.4.2 Méthode des images

7.4.3 Solution en trois dimensions

Chapitre 8

Théorie de la communication

Contents

Le schéma d'un système *abstrait* de télécommunication est donné par la figure 8.1 : une *source* d'information engendre une suite de *messages*, transmis au *destinataire* par un *canal* de communication, qui est soumis à des *perturbations*. La fonction du système est de *reproduire* l'information de la source, à l'identique chez le destinataire.

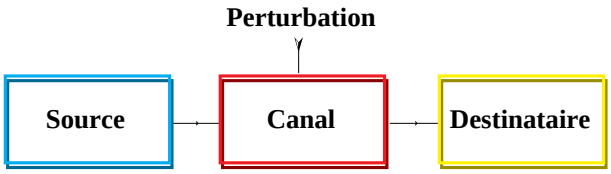


Figure 8.1 – Le schéma de Shannon.

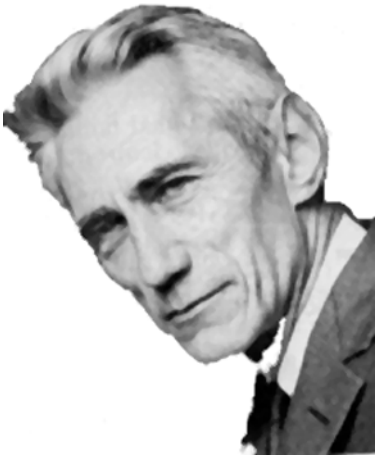
Dans un circuit électronique *digital* et *synchrone*, chaque porte logique est source d'une information binaire, à chaque cycle de l'horloge. Ce *bit d'information* est communiqué aux portes qui en dépendent par le canal d'un *réseau conducteur*, déposé à cet effet sur le support de silicium. En mode normal de fonctionnement, les erreurs de transmission sont négligeables *par construction*.

Ce canal - *binaire sans bruit* - peut transmettre un *bit d'information* par cycle. Du moins, tant que la période d'horloge reste supérieure au *délai RC équivalent* du réseau de distribution. Aux fréquences $f > 1/\tau$ supérieures à cette *valeur critique*, des erreurs apparaissent. Même le canal de transmission le plus performant connu - *conducteurs métalliques micrométriques* - ne peut transmettre qu'une quantité *finie* d'information par unité de temps. La *capacité* d'un canal est le maximum de la quantité d'information transmise. Pour notre cas : $C = 1/f$.

Notre second exemple de télécommunication passe d'une longueur de canal *micrométrique* dans les circuits digitaux, aux distances *cosmiques* dans la transmission d'images par sonde planétaire. Soit P_e la puissance du signal émit par la sonde. La puissance P_r reçue à terre par le canal de l'*éther* électromagnétique est proportionnelle $P_r \approx P_e/R^2$ à la puissance émise, divisée par le carré R^2 de la distance entre sonde et terre. Un émetteur directionnel diminue d'autant la constante de proportionnalité, mais la dépendance reste en $1/R^2$. A l'inverse du canal qui précède, le rapport P_r/N entre signal et bruit est ici *quasiment* nul.

Dans ce contexte, il semble intuitivement *impossible* de reconstruire une image venant de Saturne, ou même de la Lune. La *théorie de la communication* montre l'inverse. Pour une sonde spatiale, le décodage exact des images reste possible tant que la puissance reçue est supérieure à la sensibilité du détecteur *A/D* terrestre - cf. chap. 7. La capacité $\frac{1}{2}log_2(1 + P_r/N)$ de ce canal est très faible, mais elle n'est pas nulle. A l'aide de *codes correcteurs adaptées* à un tel bruit, il est possible de recevoir et de restituer l'information transmise - *lentement*, mais (quasiment) *sans erreur*. Pour s'en convaincre, consulter la dernière image de planète reçue sur Internet.

La théorie de Shannon s'applique à ces deux échelles extrêmes, ainsi qu'à



Claude E. Shannon (1916-)

David A. Huffman (1925-).

Planche 8.1 – Shannon et Huffman

tout autre système connu de télécommunication : analogique à temps continu - téléphone, radio, télévision - comme digital à temps discret - télégraphe, fax, câble et satellite numérique. Nous montrons au chap. 9 qu'on peut même mesurer (indirectement) la *capacité* de l'œil et de l'oreille humaine, au sens de Shannon.

La théorie de Shannon propose une mesure quantitative de l'information dont l'origine vient de la statistique *mathématique*. Ceci la rend indépendante de la réalisation *physique* du codage - modulation et support de propagation.

L'*information* de Shannon est ainsi une *abstraction mathématique* : elle reste invariante au travers des multiples *modulations* de l'information, que l'on utilise pour réaliser la communication dans le monde physique.

Le mot *information* n'a pas ici la même signification que dans la langue usuelle. Elle dépend exclusivement de propriétés statistiques de la source et du canal, qui ne correspondent pas à la classification humaine usuelle l'information. Lors d'un enregistrement binaire sur disque magnétique, c'est ce sens purement technique du mot information qui importe, et non l'effet que son contenu aura lors d'une lecture par un être humain. Beaucoup des informations que l'on trouve sur Internet sont d'ailleurs *fausses* ; ceci ne gène en rien pour les compresser et les communiquer sans erreur, conformément à la *théorie de Shannon*.

8.1 Théorie de Shannon

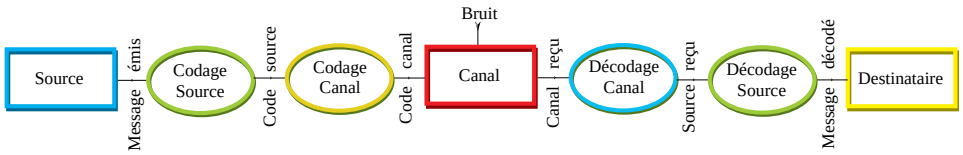


Figure 8.2 – Schéma optimal théorique.

Théorème 10 (Shannon) *Dans le schéma 8.1, on peut quantifier l'*entropie *de la source par un nombre réel $H > 0$, et la* capacité *du canal par un nombre $C > 0$.*

– *La transmission* quasiment *sans erreur est possible quand $H < C$.*

– *Elle est* impossible *quand $H > C$.*

La théorie de Shannon décompose la communication en deux problèmes : code de la source, et code du canal - fig. 8.2. Ces deux problèmes sont indépendants, bien qu'ils semblent antagonistes en pratique. Pour le code source, on cherche à *comprimer les données*, en *enlevant* de la redondance inutile, pour gagner en temps de transmission et/ou en espace de stockage. Pour le code canal, on fait l'inverse : on ajoute la redondance maximale compatible avec la capacité du canal, au moyen d'un *code correcteur d'erreur*.

L'*antagonisme* se produit quand un texte fortement comprimé est transmis, avec un bit d'erreur ; il est alors probable que la totalité du texte soit perdue pour le récepteur. En échange, un bit d'erreur dans la transmission d'un texte non-comprimé n'a probablement pas plus d'effet qu'une faute typographique banale. La redondance *naturelle* de notre langue autorise un seuil de tolérance aux erreurs. La redondance *artificielle* introduite dans canal par un *code correcteur* obtient le même effet, d'une façon systématique et mieux adaptée à la nature des erreurs propres au canal.

Le problème du codage *effectif* de la source est résolu, du moins en théorie. Considérons en effet une source d'entropie H et un nombre réel $\epsilon > 0$; on *sait* construire explicitement des codes dont le nombre moyen m de bit par cycle est tel que : $H \leq m < H + \epsilon$. La section 8.2 donne deux méthodes pour y arriver : l'*algorithme de Huffman*, et le *codage arithmétique*.

Pour le canal, la preuve de Shannon [25] montre l'existence de *codes optimaux* (adaptés à la capacité du canal) par un argument de *moyenne statistique*. Il ne livre pas de technique effective pour construire de tels codes. L'argument de Shannon montre même que *presque tous* les codes sont optimaux : toujours sans en livrer explicitement *un seul*. Pour gagner, il semble alors nécessaire d'avoir à tirer les codes *optimaux* au hasard ! Ce commentaire de *Wozencraft* et *Reiffen*, en 1961 résume bien la frustration de certains chercheurs :

Any code of which we cannot think is good !

En dépit de progrès spectaculaires depuis 1948, le problème de trouver une méthode *effective* pour le codage *optimal* d'un canal de capacité C, avec un nombre moyen m de bit par cycle tel que $C \leq m < C + \epsilon$, ne semble toujours pas résolu en 1998, pour le cas général. Ceci étant, les recherches contemporaines sont tout aussi actives et fructueuses dans les deux domaines : compression des données et contrôle des erreurs.

Contentons-nous dans ce chapitre introductif d'examiner quelques cas particuliers du résultat de Shannon : significatifs, mais simples. Pour en savoir plus, le lecteur consultera [26].

8.1.1 Code et entropie

Code préfixe

Le code $c = 0\ 10\ 11$ est réduit ; le code $c' = 00\ 01\ 10$ ne l'est pas.

Figure 8.3 – Deux codes préfixes, avec arbre de décodage

Une source *discrète* d'information de base $b \in \mathbf{N}+2$ produit à chaque cycle $t \in \mathbf{N}$ un événement $e(t)$, à choisir parmi $\{e_0, e_1 \cdots e_{b-1}\}$. Le message m de la source est la suite $m = s_0 s_1 \cdots s_t \cdots$ des *symboles*, qui représentent les événements à chaque cycle par un *chiffre* entier $0 \leq s_t < b$.

Dans un codage *par symbole*, on associe à chaque chiffre $k < b$ un *code binaire* $c(k)$ unique, de longueur $l(k)$. Le code binaire d'un message est obtenu, par concaténation

$$c(m) = c(s_0 s_1 \cdots s_{n-1}) = c(s_0)c(s_1) \cdots c(s_{n-1}),$$

des codes binaires correspondants aux symboles successifs du mot m.

Ainsi le mot $m = 0201100$ de 7 chiffres ternaires $\{0, 1, 2\}$ se code par les dix bits du mot $c(m) = 0110101000$ obtenu par concaténation des codes de chaque chiffre : $c(0) = 0$, $c(1) = 10$, $c(2) = 11$. Ce code particulier possède deux propriétés désirables, en général.

1. On peut le *décoder* : le récepteur du mot m peut clairement en séparer les *symboles* $c(m) = 0\ 11\ 0\ 10\ 10\ 0\ 0$, et retrouver ainsi les sept chiffres

du message initial en procédant simplement de gauche à droite. On dit que le code c est *préfixe*. Le code $c''(0) = 0$, $c''(1) = 1$, $c''(2) = 10$ n'est pas préfixe, puisque le mot 10 peut provenir de deux messages différents : 2 et 10.

2. Le code préfixe c est aussi tel que tout mot binaire se décompose de façon unique par concaténation du code d'un chiffre ternaire $c(d_m)$ avec un mot binaire m' : $m = c(d_m)m'$. On dit de ce code qu'il est *réduit*. Le code préfixe $c'(0) = 00$, $c'(1) = 01$, $c'(2) = 10$ n'est pas *réduit*, puisque le mot 11 n'est préfixe d'aucun code valide.

Définition 22 (Code préfixe) *Un code $c(0), c(1) \cdots c(b-1)$ est dit* préfixe *s'il n'existe pas d'égalité $c(i) = c(j)m$ où le code de i est la concaténation du code de j avec un mot binaire m, et le code de j serait un préfixe du code de i, pour $i \neq j$.*

Le code est dit réduit *si tout mot binaire m de longueur supérieure au plus long mot de code se décompose de façon* unique *$m = c(d_m)m'$ par concaténation du code d'un chiffre d_m et d'un mot binaire m'.*

La condition préfixe est clairement nécessaire pour garantir que le décodage est unique. On vérifie qu'elle est suffisante en associant à tout code préfixe un arbre binaire de décodage, comme c'est fait pour les codes c et c' dans la fig. 8.3.

Le décodage d'un mot binaire $m = m_0 m_1 \cdots m_{k-1}$ part de la racine de l'arbre, et descend à gauche si $m_0 = 0$, à droite sinon. La décision suivante est commandée par m_1 et ainsi de suite jusqu'à trouver une feuille de l'arbre.

Dans un code réduit, chaque feuille identifie un chiffre $k < b$ dont le code $c(k)$ indique l'unique chemin de décision qui amène de la racine de l'arbre à cette feuille. Si le code n'est pas réduit - voir c' - les mots inutilisés correspondent à des feuilles vides. Le code est donc réduit si et seulement si l'arbre binaire est *complet*, c'est à dire qu'il possède b *feuilles* (une par chiffre) et $b-1$ *nœuds internes* (un par décision binaire).

Etudions maintenant les longueurs $l(k)$ des codes préfixes $c(k)$, pour $k < b$. Pour cela, associons des probabilités aux points de l'arbre binaire du code comme suit : la racine a probabilité 1 ; la probabilité de chaque autre point de l'arbre est moitié de celle de son père.

La probabilité p d'un point est égale à $p = 2^{-l}$, où l est sa distance à la racine, c'est à dire la *profondeur* du point dans l'arbre. On voit aussi par récurrence que la somme des probabilités des décisions de profondeur j et des feuilles de profondeur au plus j, est égale à *un*, pour tout j entier. Pour j assez grand, tous les points sont des feuilles, et on trouve l'inégalité de Kraft :

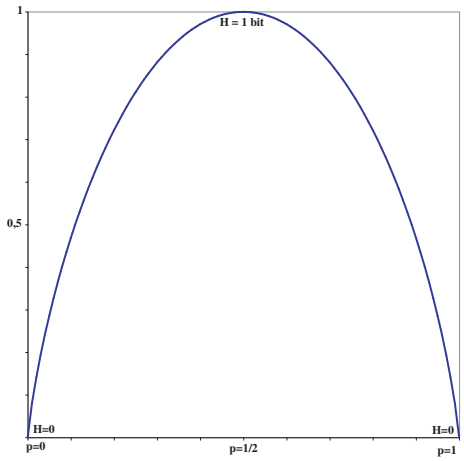
$$\sum_{0 \leq k < b} 2^{-l(k)} \leq 1, \tag{8.1}$$

expression dans laquelle on a égalité à un si et seulement si le code est *réduit*.

Réciproquement, à partir de b nombres entiers $l(k)$ satisfaisant à l'inégalité (8.1), nous pouvons construire un code préfixe c dans l'arbre duquel le chiffre k

est situé à profondeur $l(k)$, pour tout $k < b$. Nous le calculons explicitement en section 8.2.1, en appliquant l'algorithme 14 de Huffman $c = Hu(p_0 \cdots p_{b-1})$ à la distribution de probabilité : $p_k = 2^{-l(k)}$.

Entropie



Aux valeurs extrêmes : $H(0,1) = H(1,0) = 0$; ceci veut dire que les deux sources binaires constantes - soit $0 = (0)$ et $-1 = (1)$ - n'apportent *aucune* information, au sens de Shannon.
Echanger p avec $1-p$ revient à échanger 0 et 1 : $H(p, 1-p) = H(1-p, p)$.
Par symétrie, le maximum de l'entropie est au centre : $H(1/2, 1/2) = 1$.

Figure 8.4 – Entropie $H(p, 1-p)$

Définition 23 (Entropie d'une source stationnaire)
Une source de base b est stationnaire *s'il existe une probabilité p_k d'occurrence moyenne de chaque chiffre $k < b$, avec $\sum_k p_k = 1$. L'*entropie *d'une source stationnaire est donnée par :*

$$H(p_0 \cdots p_{b-1}) = -\sum_{0 \leq k < b} p_k log_2(p_k). \tag{8.2}$$

Par la définition 8.2, l'entropie d'une source binaire stationnaire de probabilité $p_0 = p$ et $p_1 = 1 - p = \bar{p}$ est donnée par la fonction (fig. 8.4) :

$$H(p, \bar{p}) = -p\, log_2(p) - \bar{p}\, log_2(\bar{p}).$$

Proposition 19 *L'entropie $H(p_0 \cdots p_{b-1})$ d'une source stationnaire de base b est telle que :*

$$0 \leq H \leq log_2(b).$$

Le cas $H = 0$ correspond à une source constante. Toutes les autres sources ont une entropie $H > 0$ strictement positive.

Le maximum $log_2(b) = H(1/b \cdots 1/b)$ de l'entropie est atteint pour la distribution équiprobable, et pour elle seule.

Preuve : Soient $p_0 \cdots p_{b-1}$ et $q_0 \cdots q_{b-1}$ deux distributions de probabilités sur les chiffres en base b. L'inégalité de Gibbs nous vient de la thermodynamique :

$$\sum_{1 \leq k < b} p_k log(q_k/p_k) \leq 0, \tag{8.3}$$

avec égalité si et seulement si $p_k = q_k$ pour tout $k < b$.

Le logarithme *népérien* est tel que $log_e(x) \leq x - 1$ pour tout $x > 0$, avec égalité si et seulement si $x = 1$. En substituant dans (8.3), on trouve :

$$\sum_k p_k log_e(q_k/p_k) \leq \sum_k p_k(q_k/p_k - 1) = \sum_k q_k - \sum_k p_k = 0.$$

Comme (8.3) reste invariant en multipliant par $log_e(2) > 0$, l'inégalité de Gibbs vaut aussi pour la base 2 des logarithmes. En choisissant $q_k = 1/b$ dans (8.3), on trouve bien $H \leq log_2(b)$, l'égalité n'ayant lieu que pour une source équiprobable. **Q.E.D.**

Bit b et shannon Sh

Le choix de la base 2 pour les logarithmes de (8.2) revient à prendre le *bit* comme unité de mesure de l'information.

Le standard ISO (*International Standards Organization*) propose le nom de *shannon* pour l'unité binaire d'information, et le note Sh. Ceci distingue *à juste titre* le *shannon* - qui mesure le contenu *statistique* de l'information - du *bit* - qui mesure la taille de sa représentation *physique*. Le *shannon* ne correspond directement au bit physique que dans le codage d'une source binaire équiprobable, et de ses extensions. Dans *tous* les autres cas, on arrive à la correspondance *1 Sh = 1 b* qu'après un codage de la source, qui la réalise en moyenne, et à la limite pour des messages de longueur infinie : en pratique, on a donc *1 Sh < 1 b*.

Ceci étant, la mesure quantitative par le *shannon* à laquelle mène la théorie ne dépend pas du choix de la base 2. Nous le montrons en section 8.2 pour la base 3. Vérifions ici qu'il en va de même pour la base 4, et les autres puissances de deux.

Extensions de la source

Si S est une source de base b, son *extension* d'ordre deux S^2 est la source de base b^2 obtenue en groupant les chiffres consécutifs de rang pair et impair dans S. Pour laisser invariant le *débit d'information*, la fréquence de la source S^2 doit être moitié de celle de S. On définit de même l'extension S^n d'ordre n de S.

Proposition 20 *Une extension S^{N} d'ordre N d'une source S est* stationnaire *si et seulement si S l'est. Les deux sources ont alors la même entropie.*

Preuve : Montrons ce résultat pour le cas particulier d'une source sans mémoire.

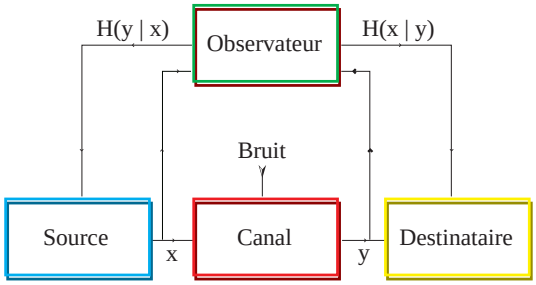
Une source est dite *sans mémoire* si deux chiffres successifs sont *indépendants*. Si les probabilités de S sont $p_0 \cdots p_{b-1}$, celles de S^2 sont les b^2 produits croisés $p_0p_0 \cdots p_ip_j \cdots p_{b-1}p_{b-1}$, dont on calcule l'entropie :

$$\sum_{i,j} p_ip_j log_2(p_ip_j) = \sum_i p_i(\sum_j p_j log_2(p_j)) + \sum_j p_j(\sum_i p_i log_2(p_i)),$$

soit $2H(S)$. Comme la fréquence des chiffres de S^2 est la moitié de celle de S, l'entropie de S^2 est bien égale à celle de S : $H(S^2) = H(S)$. La généralisation à une extension S^{N} d'ordre N (base b^{N}) de la source S est sans difficulté. **Q.E.D.**

8.1.2 Code optimal pour la source

Considerons une source $S(p_0 \cdots p_{b-1})$ stationnaire de base b. Nous allons construire un codage binaire dont la longueur moyenne m est arbitrairement proche de l'entropie $H(p_0 \cdots p_{b-1})$, ce qui montre la partie du théorème 10 de Shannon relative à la source.



Pour éliminer les erreurs de communication, installons dans la figure ci-dessus un *espion de Shannon*. Il observe en permanence l'entrée x et la sortie y. Il peut ainsi calculer l'information minimale $H(x|y)$ nécessaire pour reconstruire x connaissant y ; cette information mesure l'*ambiguïté* sur x, connaissant y. L'espion vend $H(x|y)$ au récepteur. Comme profit fait seule loi, il calcule aussi l'*ambiguïté* $H(y|x)$ sur y, connaissant x, et la vend à l'émetteur. Disposant de y et $H(x|y)$, le récepteur calcule x. L'émetteur calcule y à partir de x et de $H(y|x)$. A ce point, émetteur, récepteur et espion disposent donc de la même information : tous trois connaissent la valeur de x comme la valeur de y ; le nombre minimum de bits pour coder cette information est l'entropie conjointe $H(x, y)$.

Pour pratiquer son double jeu, l'espion dispose de deux canaux de communication - sans erreur, et top secret : un vers le destinataire, l'autre vers l'émetteur.

Notre espion connaît la proposition 23 ; les capacités respectives de ces canaux sont donc *légèrement supérieures* aux entropies conditionnelles : $H(x|y)$ et $H(y|x)$.

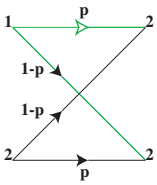
L'*information mutuelle* $I(x,y)$ est ce qui subsiste quand on *tue l'espion* ! Cette quantité d'information $I(x,y)$ est donnée par l'une des formes équivalentes :

$$\begin{aligned} I(x,y) &= \sum_x \sum_y Pr(x,y) log_2 \frac{Pr(x,y)}{Pr(x)Pr(y)} \\ &= H(x) - H(x|y) = H(y) - H(y|x) \\ &= H(x) + H(y) - H(x,y) = H(x,y) - H(x|y) - H(y|x). \end{aligned}$$

Il reste à observer que $I(x,y)$ est une fonction *strictement convexe* (nous l'admettons ici) des $Pr(x)$. Ceci implique l'existence d'un maximum unique. La *capacité* du canal bruité est la valeur de ce maximum de l'*information mutuelle*, pour toutes les distributions de probabilité $\sum Pr(x) = 1$ sur les entrées :

$$C(x,y) = max\{I(x,y)\}. \tag{8.4}$$

Exemple 10 (Canal boiteux) *Reprenons de [25] l'exemple du canal ternaire dont le diagramme de transitions est donné par :*



Le symbole 0 est immune au bruit. Les symboles 1 et 2 forment, entre eux, un canal binaire symétrique sans mémoire, d'entropie : $H = H(p, \bar{p})$.

Soient $P = Pr(0)$ *et* $Q = Pr(1) = Pr(2)$ *les probabilités des symboles d'entrée, l'égalité* $Pr(1) = Pr(2)$ *venant par symétrie. Comme* $H(x|y) = 2QH$ *et* $H(x) = -Plog_2P - 2Qlog_2Q$*, nous devons maximiser*

$$I(x,y) = -Plog_2P - 2Qlog_2Q - 2QH,$$

sujet à la contrainte : $P + 2Q = 1$. *Introduisons le* multiplicateur de Lagrange λ *dans :*

$$U = I(x,y) + \lambda(P + 2Q).$$

Le maximum de U *annule les dérivées partielles :*

$$\begin{aligned} \frac{\delta U}{\delta P} &= -1/ln2 - log_2P + \lambda = 0, \\ \frac{\delta U}{\delta Q} &= -2/ln2 - 2log_2Q - 2H + 2\lambda = 0. \end{aligned}$$

Eliminons λ, pour trouver $log_2 P = log_2 Q + H$, soit $P = Q2^H$ et :

$$C = log_2(1 + 2^{1-H}).$$

Vérifions les cas simples.

- *Pour $H = 0$, soit $p = 1$ ou $p = 0$,on retrouve la capacité $C = log_2 3$ du canal ternaire sans bruit. Il ne boite plus.*
- *Pour $H = 1$, soit $p = 1/2$, on retrouve la capacité $C = 1$ du canal binaire sans bruit. Le bit 0 est communiqué par le symbole 0, avec $Pr(0) = 1/2$. L'autre bit est communiqué aussi avec probabilité $Pr(1) = 1/2$, par la conjonction des symboles 1 et 2. Le code* optimal *du canal boiteux, pour ce cas, est : $[0, 1] \rightleftharpoons [100, 011]$.*
- *Pour les valeurs intermédiaires de H (et donc de p), on constate que le taux d'utilisation P du symbole 0 est supérieur à celui Q des autres, moins fiables !*

Codage du Canal Nous disposons de tous les éléments pour conclure la preuve du théorème 10, de Shannon.

Proposition 21 *Soit $\mathcal{S}$ une source d'entropie $S = H(\mathcal{S})$, et $\mathcal{C}$ un canal bruité de capacité : $C = max\{I(x, y)\}$. Il existe un codage de la source pour le canal, dont on maîtrise le taux d'erreur, si et seulement si $S < C$.*

Preuve : Reprenons l'argument des *codes aléatoires*, déjà utilisé pour le *canal binaire sans mémoire symétrique* - proposition 25.

Dans un codage sur $\aleph$ cycles du canal, nous disposons de $2^{\aleph H(s)}$ messages à la source, qu'il faut affecter, de toutes les manières, aux $2^{\aleph H(c)}$ codes du canal. En moyenne sur ces affectations source $\rightleftharpoons$ canal, la probabilité qu'un message soit un code de la source vaut : $2^{\aleph(H(s)-H(c))}$.

Le nombre de messages r reçus et différents de c est $2^{\aleph H(c|r)}$: en moyenne sur un ensemble statistiquement significatif de transmissions au travers du canal, pour chaque sortie r fixée.

La probabilité $Pr(erreur) = \epsilon(\aleph)$ qu'une erreur ne soit pas corrigée par un décodage à *vraisemblance maximale* (sur $\aleph$ cycles) est bornée par le produit :

$$\epsilon(\aleph) = 2^{\aleph(S-H(c))} 2^{\aleph H(c|r)} \leq 2^{-\aleph(C-S)},$$

dont la limite vaut $\lim_{\aleph \mapsto \infty} \epsilon(\aleph) = 0$ quand $S < C$.

Supposer la possibilité d'une communication fiable à taux $S > C$ impliquerait une l'information mutuelle $I(c, r)$ négative, par (8.4) ; c'est absurde : on ne peut pas tuer l'espion, et en même temps, continuer à recevoir ses informations. **Q.E.D.**

8.2 Compression des données

Le code de la source vise à minimiser le nombre de bits qui en repésentent les événements. On *enlève* de la redondance, afin de gagner en temps de transmission et/ou en espace de stockage.

Dans cette section, nous présentons les deux méthodes les plus populaires pour le codage de la source : l'algorithme de Huffman, et le codage arithmétique. Ces méthodes de *codage entropique* construisent des codes dont la longueur moyenne tend vers l'entropie. Les constructions se font à partir de la donnée exacte des probabilités de la source.

Dans beaucoup de cas pratiques, ces probabilités ne sont pas connues à priori. Un *codage adaptatif* doit donc les estimer, pour pouvoir procéder au codage entropique.

8.2.1 Algorithme de Huffman

Algorithme 14 (Huffman) *On construit un code préfixe de base b $H_u = H_u(p_0 \cdots p_b - 1)$ adapté à la distribution de probabilité $1 = \sum_k p_k$ par la méthode suivante.*

1. *Extraire de la liste les deux probabilités p_i et p_j les plus faibles, soit : $p_i, p_j < p_k$ pour $k \neq i$ et $k \neq j$. Associer p_i et p_j dans un arbre de décodage à deux feuilles.*
2. *Remettre une seule probabilité $p_i + p_j$ dans la liste en place de p_i et p_j, pour représenter l'arbre qui joint i et j. Continuer l'algorithme avec cette nouvelle distribution de probabilité sur la base $b - 1$.*
3. *Quand il ne reste plus* qu'une seule probabilité*, sa valeur $1 = \sum_k p_k$ est associée à la racine de l'arbre binaire de décision qui vient de se construire ; il regroupe alors tous les sous-arbres vus en route. Le code préfixe H_u d'ordre b est définit par cet arbre de décision.*

Si on applique l'algorithme de Huffman aux probabilités 1/2,1/4 et 1/4, on retrouve le code $c = H_u(1/2, 1/4, 1/4)$ de l'introduction. Sa longueur moyenne $m = 1/2 + 2/4 + 2/4 = 3/2$ est égale à son entropie

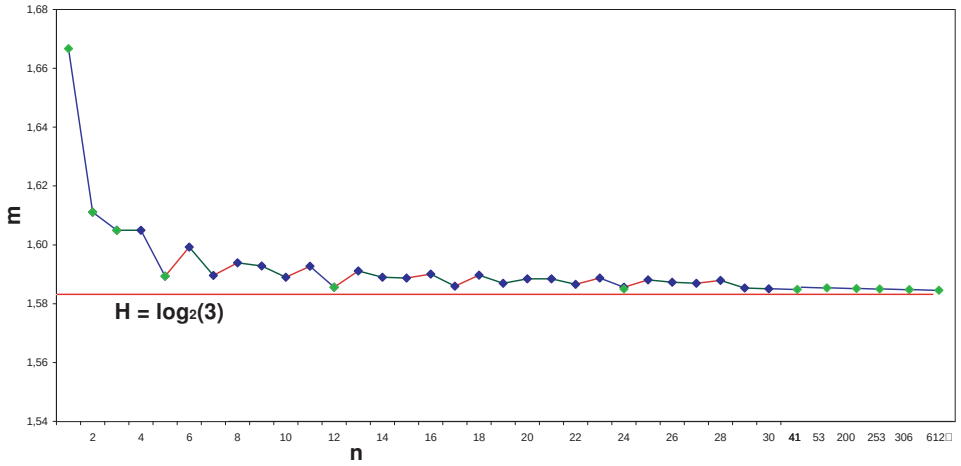
$$H = 1/2log_2(2) + 1/4log_2(4) + 1/4log_2(4) = 3/2.$$

Il est donc *optimal* vis à vis de cette distribution de probabilité. Sa longueur est 1 bit et demi (on devrait dire *1.5 sh*), en moyenne par chiffre en base 3.

Si l'on calcule maintenant $H_u(1/3, 1/3, 1/3)$, on retrouve encore le code c. Sa longueur moyenne $m = 5/3$ est dans ce cas supérieure à l'entropie

$$H = log_2(3) = 1.584962500721\cdots$$

La différence vaut $m - H$ = *0.08 Sh* par chiffre.



Pour $n > 30$, seules sont indiquées les valeurs de n pour lesquelles $H_u(1/n)$ est un nouveau minimum. La différence $H_u - H$ est inférieure à 10^{-5} pour $n = 306$. Pour $n = 612$, on trouve $H_u = 1.5849639$ alors que $H = 1.5849625$.

Figure 8.5 – Longueur moyenne des codes de Huffman pour une source équiprobable d'ordre 3, en fonction du nombre de bit par bloc

Considérons ensuite l'extension d'ordre 2 de la source. Elle est équiprobable, $p(k) = 1/9$ pour $k < 9$, d'entropie $H = log_2(3)$.

Le code de Huffman $H_u(1/9 \cdots 1/9)$ place 7 chiffres à profondeur 3, et 2 à 4 bits. Sa longueur moyenne est : $(7 \times \frac{3}{9} + 2 \times \frac{4}{9})/2 = 1 \cdot 6(1)_{10} = H + 0 \cdot 026 \cdots$ Sh.

Proposition 22 *Soit $\mathcal{S}$ une source d'entropie $H(S)$, et soit $H_u(S)$ la longueur moyenne d'un code associé à $\mathcal{S}$ par l'algorithme 14 de Huffman.*

1. *Le code est* optimal *$H_u(S) = H(S)$ si et seulement si la distribution de probabilité vérifie l'égalité de Kraft :*
   $1 = \sum Pr(s) = \sum 2^{-l(s)}$, *avec $l(s) \in \mathbf{N}$ entier.*
2. *Pour tout autre source : $H_u(S) > H(S)$.*
3. *Pour toute extension d'ordre N de la source : $H_u(S^{\mathrm{N}}) < H(S) + 1/\mathrm{N}$.*

La figure 8.5 donne la suite des longueurs moyennes des codes de Huffman pour n symboles d'une source ternaire équiprobable. On voit que la convergence vers l'entropie n'est pas une phénomène *monotone* : le code pour $n = 6$ est de longueur moyenne supérieure à celui pour $n = 5$. Les valeurs de n pour lesquelles la construction améliore la longueur moyenne des codes déjà vus (pour $k < n$) sont rares ; du moins pour le codage binaire d'une source ternaire équiprobable. Les *minima relatifs* dans ce cas sont donnés par la figure 8.5, pour $n < 1000$.

8.2.2 Algorithme LZW

Soit $S = (s_{\mathbf{N}})$ une source binaire stochastique dont on ne connait pas la distribution de probabilité. L'algorithme de LZW, présenté ici pour le cas d'une source binaire, permet de comprimer de multiples types de fichiers : c'est l'algorithme utilisée par la commande $compress$ du système Unix, et par d'autres logiciels de *compression des données*. Ces applications utilisent une version de LZW en base 256, soit un alphabet d'octets. La base 2 semble en effet particulièrement mal adaptée pour des textes dont les caractères sont en ASCII 7/8 bits. Pourtant, la nature même de l'algorithme LZW fait que le taux de compression obtenu en base 2 devient arbitrairement proche de celui obtenu dans d'autres bases, à condition de l'appliquer sur un document suffisament long. Revenons donc au cas plus simple de la source binaire.

Par une analyse séquencielle de la source S, LZW construit une *suite* $M(k) = m_1 \cdots m_k$ de mots binaire finis, extraits d'un préfixe $S_{\mathbf{N}} = s_1 \cdots s_{\mathbf{N}}$ de S :

$$m_1 m_2 \cdots m_k = s_1 \cdots s_{\mathbf{N}}.$$

Pour construire le mot suivant, on considère les bits suivants de la source $w(j) = s_{\mathbf{N}+1} \cdots s_{\mathbf{N}+j}$, et on choisit le plus petit entier j pour lequel la suite de bits est *différente* de tous les mots $w(j) \notin M(k)$. On pose alors $m_{k+1} = w(j)$, et le processus reprend, à partir de :

$$m_1 m_2 \cdots m_k m_{k+1} = s_1 \cdots s_{\mathbf{N}} \cdots s_{\mathbf{N}+j}.$$

Par le choix de j, le mot m_{k+1} est nécessairement de la forme :

$$m_{k+1} = m_{c(k+1)} s_{\mathbf{N}+j},$$

avec $m_{c(k+1)} \in M(k)$.

Pour initialiser la construction, on convient que m_0 est le mot vide.

Exemple 11 (Code LZW(010)) *Pour* comprendre *comment LZW reconnait effectivement les motifs des mots du message, à toutes les échelles pour un codage assez long, considérons la source périodique* $S = (010) = 0100100100\cdots$. *Rien n'est plus loin du hasard, mais comment LZW s'y prennent-ils ?*

Partant du mot vide m_0, *le premier préfixe de* (010) *est 0 :* $m_1 = m_0 0 = 0$. *Il reste* (100), *et on trouve ensuite* $m_2 = m_0 1 = 1$. *Le reste de s vaut maintenant* (001). *Le mot suivant est* $m_3 = m_1 0 = 00 = 0^2$, *reste* (100). *Suivent* $m_4 = m_2 0 = 10$ *et* $m_5 = m_1 1 = 01$, *reste* (010). *Résumons l'état courant par un arbre de décisions binaires, dont les mots* m_k *étiquettent les noeuds : fig. 8.6.*

Les k *mots de* $M(k)$ *sont tous situés sur 3 branches de l'arbre ; il est facile de voir qu'elles sont équiprobables. La profondeur des trois feuilles vaut donc* $k/3 \pm 2$. *La longueur* $N(k)$ *des symboles sources codés par* k *messages est alors :* $N(k) = 3 \sum_{j<k/3} j = \frac{k^2}{6} + O(k)$.

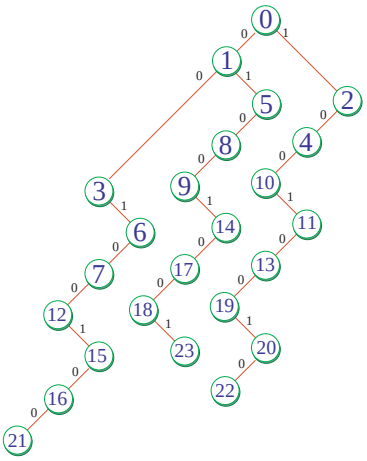


Figure 8.6 – Arbre des décisions binaires, dans le codage LZW des 23 premiers bits de la suite périodique (010)

Comme $N(k) = \frac{k^2}{6} + O(k)$ *et que l'on code avec* $klog_2(k)$ *bits, le taux de compression est donné par* $N(k) = \frac{k}{6log_2(k)} + O(1)$. *C'est énorme : comme prévu. Suite à l'exemple 12 pour faire encore mieux.*

Revenons au cas général, avec $M(k)$ messages rangés dans un arbre $A = A(k)$ de décisions. Observons que l'arbre de décisions contient toute l'information nécessaire à la reconstruction du message initial (s'en assurer). Le message suivant m_{k+1} peut se ranger dans A, en exactement $k+1$ positions : les feuilles de l'arbre binaire complété. Il faut et il suffit de $l_2(k) = \lceil log_2(k+1) \rceil$ pour réaliser un codage des mots $M(k)$, avec : $\sum_{j<k} l_2(j) = klog_2(k) + O(k)$.

Il ne reste plus, pour comprendre le facteur de *compression*, qu'à évaluer le nombre $N(k)$ de symboles à la source S, pour les messages de $M(k)$. Comme il y a autant de bits de part et d'autre, $N(k) = |M(k)|$ est la somme des longueurs des messages de $M(k)$. On peut démontrer, sous des hypothèses stochastiques très générales, que

$$\lim_{k \mapsto \infty} \frac{klog_2(k)}{N(k)} = H,$$

où $H = H(S)$ est l'entropie de la source. Pour en savoir plus sur les propriétés mathématiques et les implantations efficaces de LZW, consulter [ZL] et [W].

Peut-on faire mieux ?

Dans le cas général, le résultat mentionné ci-dessus montre que non. Après un codage *entropique* comme LZW, la statistique des accès dans l'arbre de décisions est *uniforme* : c'est un *bruit blanc artificiel* ; en géneral, il est tout aussi incompressible que la version *naturelle* de ce phénomène.

Exemple 12 (LZW(010), suite) *Essayons quand même, pour comprendre d'autres méthodes que LZW, pour le* codage adaptatif. *Les techniques les plus avancées analysent en permanence la statistique des accés dans l'arbre, et utilisent le codage arithmétique dynamique pour tirer le parti* entropique *possible du moindre écart à la moyenne statistique.*

Contentons nous, pour cet exemple, d'un codage adaptatif très simple. C'est la queue à l'Italienne *: le dernier arrivé sera le premier servi ! Elle est aussi connue sous le nom :* move to front. *Les* $2k+1$ *événements (feuilles de l'arbre* $A(k)$*) sont rangés par ordre de préférence, et les têtes de liste recoivent des codes plus courts que la moyenne, au détriment des mauvais, qui recoivent des codes plus longs. Le dernier arrivé se met systèmatiquement en tête, et repousse tous les autres d'un cran. Si la chance ne lui sourie plus, il va à son tour percoller doucement vers les bas. S'il est un être statistiquement à part, il sera souvent aux premières loges, et il mérite (peut-être) un code à part. Dans le doute, il est parfois utile de reserver un petit tampon pour les codes privilégiés. Pour rester général, il faut que cet aspect du codage soit toujours mis en compétition avec le codage SANS, et se débranche automatiquement quand il perd.*

Pour en revenir à (010), je proposerais bien un tampon de 4 bits, mais on va encore m'accuser de truquer pour l'exemple. Alors, disons 8 bits, dernier prix ici - et comme ca, je pourrais faire passer aussi le LZW de (01010110).

La proposition courante est donc de gérer les $2k+1$ *événements possibles par une* queue à l'Italienne. *Les 8 premiers de la queue recoivent un code, disons sur 4bits ; ceci, au passage, pénalise* tous *les autres codes d'un bit - oops : 1 Sh.*

La chute est alors que, pour (010), l'heuristique move to front *marche à merveille ; que notre taux de compression remonte à* $N(k) = \frac{3k}{4} + O(1)$*. A partir d'un* KB *de données (010), LZW donne moins d'un bit de sortie pour 5000 bits d'entrée. Et voila qui nous était vanté, il y a peu, comme du bruit blanc incompressible ! D'autant plus que (010) se code avec 3 bits* en tout*, non ? Fin ...*

8.3 Contrôle des erreurs

8.3.1 Code d'un disque compact

8.3.2 Code d'un téléphone mobile

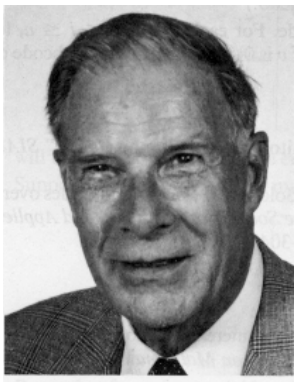
Norme AMPS - *American Mobile Phone System*.

8.3.3 Code d'une sonde spatiale

Proposition 23 *1. L'entropie* $H = H(p_0 \cdots p_{b-1})$ *d'une source stationnaire de base* b *est inférieure à la longueur moyenne* $m = \sum_k p_k l(k)$ *de tout code préfixe :* $H \leq m$.

2. Pour toute extension d'ordre N *de la source, il existe un code préfixe de longueur moyenne :* $m < H + 1/\mathrm{N}$.

Evariste Galois (1811 - 32)



Richard Hamming (1915-)

Planche 8.2 – Galois et Hamming

Preuve :

1. Soit $q_k = 2^{-l(k)}$ la distribution de probabilité associée à un code préfixe arbitraire de la source, pour $k < b$. Ecrivons l'inégalité (8.3) de Gibbs :

$$\sum_k p_k log(1/p_k) \leq \sum_k p_k log(1/q_k).$$

   En substituant pour la valeur de $q_k = 2^{-l(k)}$, avec $m = \sum_{0 \leq k < b} p_k l(k)$, on trouve $H \leq m$ avec égalité si et seulement si $p_k = q_k$ pour tout $k < b$.

2. Pour montrer que $m < H + \epsilon$, traitons d'abord le cas où tous les nombres $l_k = -log_2(p_k)$ sont des entiers, pour $k < b$. Comme $\sum_k 2^{-l_k} = 1$ (égalité de Kraft) il existe un code préfixe réduit pour lequel $l(k) = l_k$ est la profondeur de la feuille associée au chiffre $k < b$. La longueur moyenne de ce code est égale à l'entropie :

$$m = \sum_k l_k 2^{-l_k} = \sum_k p_k log_2(p_k) = H.$$

   On voit ainsi qu'il existe un code préfixe *optimal* - soit $m = H$ - si et seulement si tous les p_k sont de la forme $p_k = 2^{-l(k)}$), avec $l(k)$ entier pour $k < b$.

   Dans le cas général où les nombres $l_k = log_2(p_k)$ sont des réels arbitraires, on choisit les entiers $l(k) = \lceil l_k \rceil$ de façon que $l_k \leq l(k) < l_k + 1$, et un code préfixe pour la distribution de probabilité : $q_k = 2^{-l(k)}$. La longueur moyenne de ce code est $m = \sum_k p_k l(k) < \sum_k p_k l_k + \sum_k p_k = H + 1$, soit $m < H + 1$. Appliquons alors cette construction à une extension S^{N} d'ordre N de la source. On trouve une longueur moyenne $m' < H_{\mathrm{N}} + 1$ par chiffre de S^{N}. Un chiffre de S^{N} code N chiffres de S, donc $m' = \mathrm{N} \times m$; nous savons

que $H_{\mathrm{N}} = \mathrm{N} \times H$, par la proposition 20. En substituant, et en divisant par N, on trouve : $m < H + 1/\mathrm{N}$. **Q.E.D.**

8.3.4 Code optimal pour le canal

Quand on tente de transmettre avec précision la valeur d'une variable $v(t) \in \mathbf{R}$ au travers d'un dispositif *physique* de communication, on se heurte à deux difficultés.

1. Le signal transmis est *distordu* : retardé, limité en fréquence, ...
2. Au signal transmis - distordu mais déterministe - s'ajoute un *bruit aléatoire*. Ce bruit est à peu près indépendant du contenu de la communication. Il est impossible de le reproduire à l'exact expérimental, d'une transmission sur l'autre : *le bruit n'est pas déterministe*.

Traitons une difficulté après l'autre. Définissons d'abord la capacité C du canal discret *sans erreur*, et montrons que toute source d'entropie $H < C$ peut être codée pour ce canal, et transmettre H bit par cycle, en moyenne et dans la durée. Il n'existe pas de code de cette nature quand $H > C$.

Définissons ensuite la capacité C du canal discret *avec erreur*, et montrons que toute source d'entropie $H < C$ peut être codée pour ce canal, afin de transmettre H bit par cycle en moyenne, avec une probabilité d'erreur inférieure à ϵ, pour $\epsilon > 0$ fixé à l'avance, et arbitrairement petit. C'est impossible quand $H > C$.

Capacité du canal discret sans erreur

Le Télétype et le télégraphe sont les deux exemples choisis en 1948 par Shannon [25] pour motiver la définition de la *capacité* du canal de communication, discret et sans erreur. Pour être technologiquement désuets, ils n'en restent pas moins instructifs.

Exemple 13 *En pressant l'une des 32 touches du* Télétype*, l'opérateur choisit un caractère et le communique en l'imprimant à distance. La* capacité *maximale du Télétype à transmettre l'information est clairement : 5 bits par cycles.*

Exemple 14 *L'opérateur du télégraphe dispose de quatre* symboles *: le point P et le trait D, et deux espaces : un pour les séparer les lettres L ; l'autre pour séparer les mot M. Le code donné (en binaire moderne) à notre télégraphiste est :* $[P, D, L, M] = [10, 1110, 000, 000000]$*; le 1 représente un cycle d'émission en circuit fermé, et le 0 un cycle en circuit ouvert. Ajoutons qu'un espace L ou M doit* toujours *être suivi d'un point P ou d'un trait D.*

Avec ces contraintes, quelle est la capacité (en bit par cycle) du télégraphe à transmettre l'information ?

Définition 24 (Capacité du canal discret, sans bruit) *Soit un canal de communication discret et sans erreur, capable de transmettre un symbole s_k parmi $0 \leq k < b$ en t_k cycles. Sa capacité maximale est donnée par la limite :*

$$C = \lim_{t \to \infty} \frac{log_2 N(t)}{t},$$

où $N(t)$ est le nombre de codes valides pour le canal, de longueur $t \in \mathbf{N}$ cycles.

Vérifions que la définition 24 permet de calculer la *capacité* du télégraphe. Tout code se termine forcèment par l'un des mots : $\{P, D, LP, LD, OP, OD\}$. Le nombre de codes en t cycles est donc donné par $N(t) = N(t-2) + N(t-4) + N(t-5) + N(t-7) + N(t-8) + N(t-10)$, en convenant que $N(0) = 1$ et $N(t) = 0$ pour $t < 0$. La théorie des *récurrences linéaires* nous apprend que le nombre de solutions $N(t)$ est assymptotiquement équivalent à ρ^t, où ρ est la plus grande racine réelle de :

$$1 = x^2 + x^4 + x^5 + x^7 + x^8 + x^{10}.$$

Tous comptes faits, la capacité du télégraphe est : $C = -log_2 \rho = 0.539 \cdots$ Sh par cycle.

Proposition 24 *Soit $\mathcal{S}$ une source d'entropie H et $\mathcal{C}$ un canal discret sans erreur, de capacité $C = \lim_{t \to \infty} log_2 N(t)/t$.*

1. *Si $H < C$, il existe une extension finie d'ordre $\mathbf{N}$ de la source et un code préfixe qui affecte à chaque événement $s \in \mathcal{S}^{\mathbf{N}}$ de la source un code unique sur n cycles pour le canal.*
2. *C'est impossible quand $H > C$.*

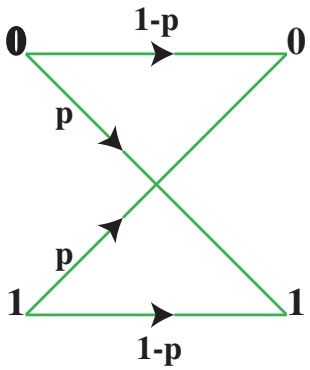
Preuve :

1. Par la proposition 23, nous savons construire un code préfixe pour une extension d'ordre $\mathbf{N}$ de la source, dont la longueur moyenne m est telle que : $\mathbf{N}m < \mathbf{N}H + 1$. En choisissant $\mathbf{N} \geq 1/(C-H)$, on assure que $2^{\mathbf{N}H+1} < N(\mathbf{N})$: il y a plus de codes de durée $\mathbf{N}$ disponibles sur le canal, que de messages sources. Il est donc possible d'affecter un code canal différent à chaque événement d'ordre $\mathbf{N}$ de la source.
2. Tout codage de la source a une longueur moyenne m telle que : $m \geq H$ - proposition 23. Quand $H > C$, la longueur totale $2^{\mathbf{N}m}$ des codesde la source devient supérieure au nombre $N(\mathbf{N})$ de codes disponibles sur le canal, et le codage est impossible : $2^{\mathbf{N}m} \geq 2^{\mathbf{N}H} > 2^{\mathbf{N}C} \geq N(\mathbf{N})$. **Q.E.D.**

Capacité du canal discret, en présence de bruit

Le codage du canal est un passage *délicat*. Décomposons la preuve en deux temps : illustrons d'abord le résultat pour le modèle de canal bruité le plus simple. Généralisons ensuite, pour conclure enfin la preuve du théorème 10.

Canal binaire symétrique sans mémoire Le système de communication le plus simple est le canal binaire sans mémoire, de capacité 1 bit par cycle, en l'absence d'erreur. Introduisons du *bruit* dans ce canal. On représente la situation par le diagramme :



Ici, $p = Pr(0,1) = Pr(1,0)$ est la probabilité d'une erreur de transmission, et $\bar{p} = 1 - p = Pr(0,0) = Pr(1,1)$ est celle d'une communication sans erreur.

On peut représenter la transmission par l'équation :

$$Y = X \oplus E,$$

dans laquelle le bruit est donné par le vecteur binaire $E = (e_{\mathbf{N}})$ des erreurs. La suite E est *aléatoire*, de distribution : $Pr(e_{\mathbf{N}} = 0) = \bar{p}$ - pas d'erreur sur $y_{\mathbf{N}}$ - et $Pr(e_{\mathbf{N}} = 1) = p$ - erreur sur $y_{\mathbf{N}}$. Les erreurs sont indépendantes de l'entrée X transmise, ce qui facilite leur analyse. Reconnaissons l'entropie $H_p = H(E) = H(p, \bar{p})$ de la figure 8.4. Nous allons montrer que la capacité C_p du canal binaire symétrique sans mémoire est donnée par :

$$C_p = 1 - H_p = 1 + p\, log_2(p) + \bar{p}\, log_2(\bar{p}).$$

Pour le canal sans erreur $p = 0$, on retrouve bien : $C_0 = 1$. Il en va de même pour $p = 1$, qui correspond à une erreur systématique : on la corrige par un simple inverseur, et donc $C_1 = C_0 = 1$. Plus généralement, on retrouve la symétrie : $C_p = C_{\bar{p}}$, modulo un inverseur dans le décodeur. La capacité du canal devient nulle $C_{1/2} = 0$ pour la distribution équiprobable $p = \bar{p} = 1/2$: dans ce cas extrême, autant remplacer tout le dispositif de communication par un robot qui tire les bits reçus au hasard - équiprobable.

Plus précisément, nous allons considérons *tous* les codes possibles pour le canal, et calculer le taux d'erreur qui résulte en moyenne, sur tous les codes possibles. Il existe alors forcément des codes spécifiques dont le taux d'erreur est inférieur au taux moyen.

Le schéma de la figure 8.7 décompose la communication en quatre étapes : $s \mapsto c \mapsto r \mapsto d$. A chaque message source $s \in 2^{\mathbf{N}S}$, on associe un code $c \in 2^{\mathbf{N}}$

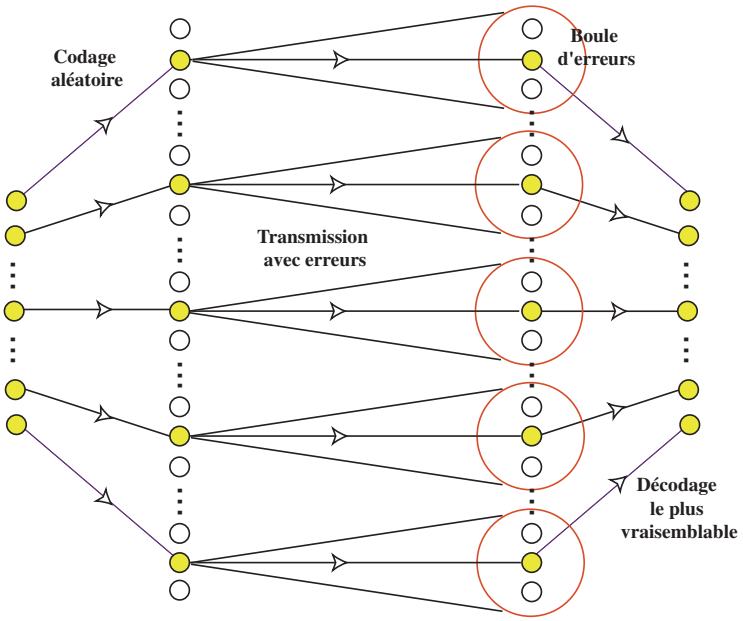


Figure 8.7 – Source, code canal, message reçu, décodage.

pour le canal. Ce code est *aléatoire* : le choix de $2^{\mathrm{N}S}$ mots de code parmi 2^{N} possibilités est *uniforme*. Soit r la suite des N bits reçus, et $e = c \oplus r$ le vecteur des erreurs du canal. Le décodage $d \in 2^{\mathrm{N}S}$ se fait à *vraisemblance maximale* : on recherche le mot de code d le plus proche de r, c'est à dire celui qui minimise le nombre de bits non nuls dans l'erreur présumée e (distance de Hamming). Quand $d = s$, le décodage est exact et toutes les erreurs $e = c \oplus r$ du canal sont *corrigées*. Nous proposons de montrer que la probabilité d'un décodage *faux* $d \neq s$ devient arbitrairement faible, quand N devient arbitrairement grand. Si d'aventure un décodage faux $d \neq s$ se produit quand même, c'est tout l'ensemble des N bits reçus qui devient corrompu.

En moyenne sur N bits, on trouve $moy(E) = \sum_{k<\mathrm{N}} e_k = \mathrm{N}p$ *erreurs* du canal, pour $\mathrm{N}\bar{p}$ bits correctement transmis. La *variance* $var(E) = \sqrt{\mathrm{N}p\bar{p}}$ mesure l'*écart type* avec la moyenne : $var^2(E) = moy((E - moy(E))^2)$. La probabilité qu'une erreur particulière s'écarte de ce comportement typique est faible, en vertu de l'inégalité de Tchebishev (que l'on démontre en cours de probabilité) :

$$\forall \epsilon > 0 \ : \ Pr(|E - moy(E)| > \frac{var(E)}{\sqrt{\epsilon}}) < \epsilon.$$

Définition 25 (Boule d'erreur) *Lors d'une transmission sur* N *cycles par le canal binaire symétrique sans mémoire, la boule des* erreurs vraisemblables *est :*

$$\mathcal{B}_\epsilon(\mathrm{N}) = \{e \in 2^{\mathrm{N}} \ : \ \sum_{k<\mathrm{N}} e_k - Np < \sqrt{\frac{\mathrm{N}p\bar{p}}{\epsilon}}\},$$

pour tout $\epsilon > 0$ *réel, et* $\mathbf{N} \in \mathbf{N}$ *entier.*

La probabilité qu'une erreur spécifique e sorte de la *boule* des erreurs vraisemblables est bornée par l'inégalité de Tchebishev :

$$Pr(e \notin \mathcal{B}_\epsilon(\mathbf{N})) < \epsilon. \tag{8.5}$$

Pour calculer la *taille* de la *boule d'erreur*, considérons une *erreur typique* : $e \in \mathcal{B}_\epsilon(\mathbf{N})$. Soit $\nu = \nu_2(e) = \sum_{k<\mathbf{N}} e_k$ le nombre d'erreurs (bits $e_k = 1$) dans e. Par la définition 25, on peut écrire $\nu = Np + \sigma$, avec $\sigma = \sigma(e) < \sqrt{\frac{\mathbf{N}p\bar{p}}{\epsilon}}$. La probabilité de e est alors donnée par : $Pr(e) = \prod_{k<\mathbf{N}} p^\nu \bar{p}^{\,\mathbf{N}-\nu} = 2^{\mathbf{N}H}\,\frac{\bar{p}^\sigma}{p^\sigma}$. Son entropie $H(e) = log_2(\frac{1}{Pr(e)})$ est bornée par : $log_2(\frac{1}{Pr(e)}) < H\mathbf{N} + \alpha\sqrt{\mathbf{N}}$, avec $\alpha = log_2(\frac{\bar{p}}{p})\sqrt{\frac{p\bar{p}}{\epsilon}}$. La *taille* de la *boule d'erreur* est donc, au plus :

$$|\mathcal{B}_\epsilon(\mathbf{N})| < 2^{H\mathbf{N}+\alpha\sqrt{\mathbf{N}}}. \tag{8.6}$$

Proposition 25 *Il existe des codages* aléatoires *du canal binaire symétrique sans mémoire avec* p *erreurs en moyenne par cycle, qui sont quasi* optimaux. *Pour toute source d'entropie* $S < C = 1 - H$*, la transmission se fait (suivant le schéma de communication de la figure 8.7) avec un taux d'erreur que l'on peut rendre arbitrairement faible, pour tout message assez long.*

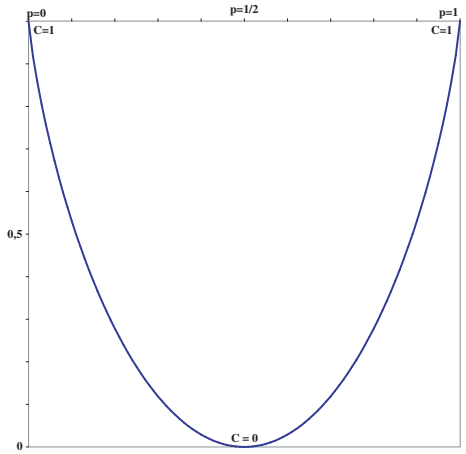


Figure 8.8 – Capacité du canal binaire symétrique sans mémoire

Reste le cas général : $0 < C_p < 1$. Le canal est représenté par l'équation $Y = X \oplus E$, dans laquelle le vecteur d'erreur est indépendant de X. Son entropie vaut $H_p = H(E)\ Sh$ par cycle. Pour toute source d'entropie $S < C_p = 1 - H_p$ Sh par cycle, nous devons montrer qu'il existe un codage *adapté* au canal : la

transmission est possible, avec un taux d'erreur que l'on peut rendre arbitrairement faible, pour un nombre de cycles de transmission arbitrairement grand.

Considérons donc $\mathbf{N}$ cycles de transmission. Il y a $2^{\mathbf{N}}$ codes possibles pour le canal, et $2^{\mathbf{N}S}$ messages source. Un codage de la source pour le canal associe une suite binaire différente à chaque message source. Comme $2^{\mathbf{N}S} < 2^{\mathbf{N}}$, nous avons l'embarras du choix.

Comment procèder ? La réponse de Shannon : tirer le code au hasard !

Preuve : Reprenons les quatre étapes d'une communication suivant le schéma de la figure 8.7 : $s \mapsto c \mapsto r \mapsto d$. Soit $e = c \oplus r \in 2^{\mathbf{N}}$ le vecteur des erreurs dans cette communication par le canal.

Quand $e \in \mathcal{B}_\epsilon(\mathbf{N})$ et que s est le mot de code le plus proche de r (distance de Hamming), toutes les erreurs de e sont *corrigées* par le décodage à vraisemblance maximale, et le décodage $d = s$ est *exact*.

Une erreur de transmission $d \neq s$ peut se produire dans deux cas.

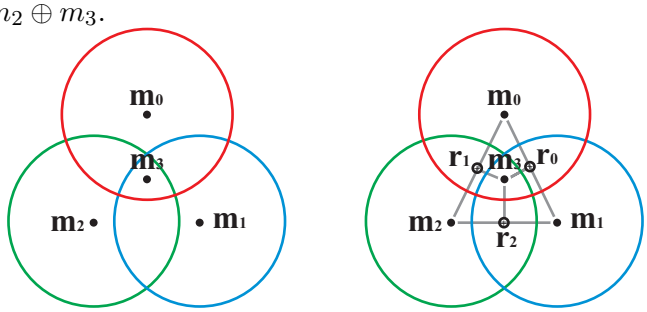
1. L'erreur est atypique : $e \notin \mathcal{B}_\epsilon(\mathbf{N})$. D'après 8.5, ceci arrive avec une probabilité inférieure à ϵ.
2. L'erreur $e \in \mathcal{B}_\epsilon(\mathbf{N})$ est typique, mais il existe un autre mot de code $s' \neq s$, dont la distance de Hamming à r est inférieure à celle de s.
   La probabilité qu'un message du canal soit un mot du code de la source est $2^{\mathbf{N}(S-1)}$: la moyenne est prise ici sur l'ensemble des $\begin{pmatrix} 2^{\mathbf{N}} \\ 2^{\mathbf{N}S} \end{pmatrix}$ codes possibles pour le canal, c'est à dire pour un *code aléatoire*.
   Par (8.6), la taille de la *boule de transmission probable* $\mathcal{B}(c) = c \oplus \mathcal{B}_\epsilon(\mathbf{N})$ du code c est au plus : $2^{H\mathbf{N}+\alpha\sqrt{\mathbf{N}}}$. La probabilité qu'un autre mot $s' \neq s$ du code appartienne à cette boule est bornée par :

$$Pr(erreur) < 2^{\mathbf{N}(S-1)} 2^{H\mathbf{N}+\alpha\sqrt{\mathbf{N}}} = 2^{-\mathbf{N}(C-S+\frac{\alpha}{\sqrt{\mathbf{N}}})}.$$

   Nous voyons que $\lim_{\mathbf{N}\mapsto\infty} Pr(erreur) = 0$ si et seulement si $C > S$.

Q.E.D.

Exemple 15 (Code de Hamming) *Le message source $m[0..3]$ comprend 4 bits. On ajoute 3 bits de redondance : $r_0 = m_0 \oplus m_1 \oplus m_3$, $r_1 = m_0 \oplus m_2 \oplus m_3$ et $r_2 = m_1 \oplus m_2 \oplus m_3$.*



Capacité du canal bruité *Le comportement du canal* bruité *est fonction de* deux *variables aléatoires : la sortie y et l'entrée x. La probabilité* conjointe $Pr(x, y)$ *de l'événement x en entrée et de l'événement y en sortie est liée à la probabilité* conditionnelle $Pr(x|y)$ *de l'entrée x, connaissant la sortie y et à la probabilité $Pr(y|x)$ de la sortie, connaissant l'entrée. C'est la règle de Bayes :*

$$Pr(x, y) = Pr(x|y)Pr(y) = Pr(y|x)Pr(x).$$

L'entropie conjointe $H(x, y)$ *et l'entropie* conditionnelle $H(x|y)$, $H(y|x)$ *sont définies de la même façon ; la règle de Bayes se traduit par l'égalité :*

$$H(x, y) = H(x|y) + H(y) = H(y|x) + H(x).$$

Le code de 7 bits $c[0..6] = [r_0r_1m_0r_2m_1m_2m_3]$ est envoyé par un canal bruité. Le message de 7 bits reçu est tel que $r[0..6] = c[0..6] \oplus e[0..6]$, où e est le vecteur d'erreur. Montrons que le décodage exact est possible s'il y a au plus un seul bit d'erreur dans le vecteur e.

On calcule les 3 bits du syndrome *: $s_0 = c_0 \oplus c_2 \oplus c_4 \oplus c_6$, $s_1 = c_1 \oplus c_2 \oplus c_5 \oplus c_6$ et $s_2 = c_3 \oplus c_4 \oplus c_5 \oplus c_6$. L'entier $s = s_0 + 2s_1 + 4s_2$ vaut 0 si la transmission est sans erreur ; si $s \neq 0$, le nombre $s - 1$ donne la position du bit d'erreur dans le vecteur $e[0..6]$. Connaissant l'erreur $e = 2^{s-1}$, on reconstruit le code canal, et donc le message source, par $c = r \oplus 2^{s-1}$.*

Si on suppose que les 8 cas de comportement du canal (pas d'erreur, une erreur en 7 positions possibles) équiprobables, nous pouvons calculer l'entropie $H = \sum_{i<8} \frac{1}{8} log_2 \frac{1}{8}$ de l'erreur : $H = 3$. Comme cette erreur est indépendante du message, la capacité par cycle du canal est donnée par $C = 7 - H$, soit 4 bits par cycle. Le code de Hamming est donc optimal, pour cette distribution bien particulière des erreurs. Son taux *est : 4/7 Sh.*

Considérons le canal binaire symétrique sans mémoire, avec probabilité ϵ d'erreur par cycle. La probabilité qu'un message de 4 bits soit correctement transmis est : $Pr(e^4 = 0) = (1 - \epsilon)^4 = 1 - 4\epsilon + 6\epsilon^2 + O(\epsilon^3)$.

Si on applique maintenant le code de Hamming, la probabilité de reception correcte des 4 bits du message, après correction d'une erreur éventuelle sur 7 bits, devient : $Pr(|e^7| \leq 1) = (1 - \epsilon)^7 + 7\epsilon(1 - \epsilon)^6 = 1 - 21\epsilon^2 + O(\epsilon^3)$.

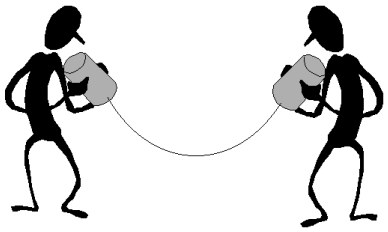
Observons que, pour $\epsilon > 1/7$, le taux d'erreur avec codage de Hamming devient supérieur à celui obtenu sans codage. En effet, une erreur double *sur 7 bits invalide* tous à la fois *les 4 quatres bits décodés. Ce code particulier n'est donc pas adapté à des erreurs aussi fréquentes.*

Pour $\epsilon < 1/7$, le bénéfice de l'utilisation du codage de Hamming est d'autant plus significatif que ϵ est petit : pour $\epsilon = 1/100$, le taux d'erreur tombe vers 1/2000 après codage de Hamming.

Chapitre 9

Codage et transmission : audio et vidéo

Contents



La particularité de la communication numérique du signal audio et vidéo, est que le destinataire final est un être humain [1]. Pour être de *haute fidélité*, la communication multimédia - audio et/ou vidéo - doit se rapprocher du pouvoir de discernement physiologique de l'oreille et de l' œil humain. L'idéal est un codage numérique assez précis pour tromper nos sens, tout en restant réalisable au plus bas coût possible dans la technologie du moment. Cette contrainte *anthropomorphe* impose, d'un coté, une borne sur la précision numérique et la fréquence d'échantillonnage de tout appareil multimédia. De l'autre coté, cette borne mesure aussi la qualité de notre perception auditive ou visuelle par un nombre rationnel, qui exprime - en bits par seconde - le débit d'information nécessaire à tromper nos sens, soit leur *capacité*, au sens de Shannon.

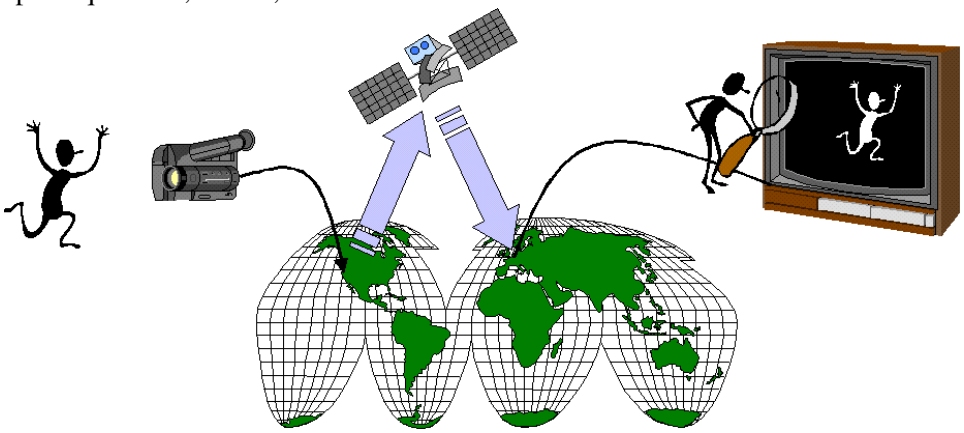
Par exemple, le signal de la parole en téléphonie est échantillonné sur 8 bits à 8 kHz - soit 8000 fois par seconde - ce qui donne un débit de 64 kbit/s. Pour le radiotéléphone cellulaire GSM, le débit de cette parole numérique est ramené à 13 kbit/s. Un signal vidéo correspondant à une cadence de 25 images par secondes est échantillonné sur 8 bits à 13,5 MHz pour la luminance, et 6.75 MHz pour chacune des deux chrominances. En ajoutant 27 Mbit/s pour la synchronisation - retour de ligne, fin de trame - ceci fait un débit total de $8 \times (13.5 + 2 \times 6.75) + 27 = 243$ Mbit/s. Dans le visiophone, le signal vidéo est codé au débit téléphonique de 64 kbit/s, au prix d'une dégradation notable de la taille et qualité des images transmise.

On sait d'expérience que ni le téléphone, ni la télévision ne savent tromper nos sens. Il faudrait, pour y arriver, multiplier par au moins 1000 les débits indiqués. On voit aussi que, si l'oreille est environ 1000 fois plus rapide que l' œil, le pouvoir de discernement de ce dernier est tel que les débits vidéo sont de 10 à 100 fois plus importants que les débits audio. Ceci étant, les principes qui régissent le traitement du son ne dépendent pas de son origine : de la parole humaine au chant des oiseaux, en passant par la musique et le bruit d'une chute d'eau. Il en va globalement de même pour les images, les capteurs médicaux, et toute mesure du monde physique. Soulignons qu'un signal vidéo numérique est représenté, pour la transmission et le stockage, par une fonction d'une seule variable - intensité variant avec un temps dont l'échelle dépend de la fréquence comme de la précision des échantillons - alors que le signal de départ est fonction de beaucoup de variables - coordonnées en x, en y, en couleur RVB [2], voire plus en stéréo.

1. Odorat, goût et toucher sont bien moins compris que l'ouïe et la vue. Il se passera du temps avant que le concept de télécommunication numérique des odeurs - *TeleSmell* - ne devienne pratique courante.

2. RVB = rouge+vert+bleu

Considérons ici quelques aspects de *chaîne de communication* des sons et des images, en tentant de distinguer : ce qui est illustration de la théorie générale de l'information ; ce qui est propre au type particulier de signal et de récepteur spécifique - œil, oreille, ...



9.1 Signal analogique et signal digital

Partons d'un signal analogique, par exemple l'intensité du courant qui sort d'un microphone en cours d'enregistrement. Par un choix convenable des unités physiques, on peut représenter ce signal par une fonction $a \in \mathbf{R} \mapsto [-1, 1[$ qui varie continûment avec le temps réel $t \in \mathbf{R}$, et dont la valeur reste à tout instant bornée : $-1 \leq a(t) < 1$ - voir fig. 9.1. Le signal *digital* s'obtient en deux temps :

1. on échantillonne les valeurs $a(nT)$ du signal analogique a aux instants multiples entiers $n \in \mathbf{Z}$ de la période d'échantillonnage T ;
2. on représente chaque échantillon réel $a(nT) \in \mathbf{R}$ par le nombre entier $a_n = \lfloor 2^{m-1} a(nT) \rceil$, que l'on code sur m bits en complément à deux.

La suite binaire résultante est le signal digital. Sa *bande passante* vaut mf bits par seconde, avec $f = 1/T$ la fréquence d'échantillonnage du signal analogique.

Un exemple vidéo du procédé est donné par la caméra *CCD* de la fig. 7.7.

9.1.1 Fréquence de Nyquist

Il semble *a priori* que l'on perde de l'information, à ne garder de la fonction continue $a \in \mathbf{R} \mapsto [-1, 1[$ que ses échantillons $a(nT)$ aux instants discrets $n \in \mathbf{Z}$. Pour s'en convaincre, il suffit de considérer les trois fonctions *analogiques* $O(t) = 0$, $A(t) = sin(2\pi t)$ et $B(t) = -sin(2\pi t)$. Pour les trois, la suite des échantillons aux temps entiers $n \in \mathbf{Z}$ est identiquement nulle. Il n'est donc pas possible, en général, de reconstruire le signal analogique à partir du signal digital.

Pour que cette reconstruction du signal analogique $a(t), t \in \mathbf{R}$ à partir de ses échantillons discrets $a(nT), n \in \mathbf{Z}$ devienne possible et *exacte*, il ne suffit pourtant que de deux hypothèses mathématiques.

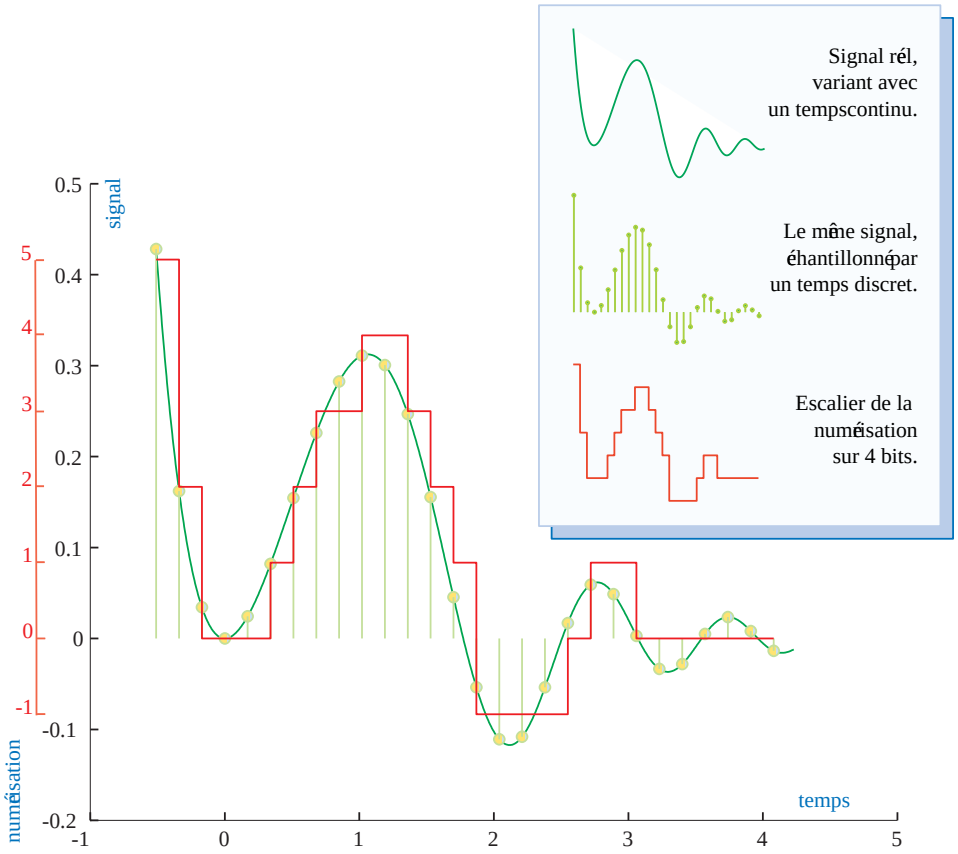


Figure 9.1 – Echantillonnage et escalier de la numérisation

1. La fonction a admet une transformée de Fourier $\hat{a}$; elles sont liées par :

$$\hat{a}(f) = \int_{-\infty}^{+\infty} a(t) e^{-2i\pi ft} dt, \qquad a(t) = \int_{-\infty}^{+\infty} \hat{a}(f) e^{+2i\pi ft} df. \tag{9.1}$$

2. La transformée de Fourier a un *spectre fini* : $\hat{a}(f) = 0$ pour $|f| > F$. Le *critère de Nyquist*

$$2F < f \tag{9.2}$$

indique que la fréquence $f = 1/T$ d'échantillonnage doit être *supérieure* à la *largeur* $2F$ du *spectre* en fréquences $\hat{a}$ du signal analogique a pour en permettre la reconstruction *exacte* à partir de ses échantillons discrets.

En reprenant les fonctions A et B de notre exemple initial, on voit qu'un échantillonage à deux fois la fréquence propre du signal analogique ne permet

Il est préfet de Grenoble - sous Napoléon - quand il écrit son *Mémoire sur la Chaleur* qui donne les bases de ce qui est maintenant l'analyse de Fourier. Il soumet son travail - disant que toute fonction périodique est développable en série de sinus et cosinus - à un prix de l'Académie des Sciences, dont les juges sont Lagrange, Laplace, et Legendre. Il ne l'obtient pas, car les juges ne sont pas convaincus par ses preuves : il faudra en effet plus d'un siècle pour développer les mathématiques - théorie de la mesure et des distributions - qui donnent finalement raison à Fourier, pour une classe de fonctions si large qu'elle englobe toutes celles du traitement moderne du signal.

Planche 9.1 – Jean-Baptiste Joseph Fourier, 1768-1830

pas, en général, de reconstruire sans ambiguïté la fonction de départ - on trouve $A(n) = B(n) = 0$, alors que $A(t) = -B(t)$. En théorie du signal, on parle d'*interférence entre symboles* - en anglais *aliasing* - pour indiquer que la fonction est sous-échantillonnée vis à vis du critère de Nyquist. Ceci montre que le critère est bien nécessaire. Il est moins intuitif que le critère de Nyquist soit aussi une condition suffisante, comme le montre la formule d'*interpolation* :

$$a(t) = \sum_{n \in \mathbf{Z}} a(nT) sinc(t/T - n). \tag{9.3}$$

Le *sinus cardinal* est donné par :

$$sinc(x) = \frac{sin(\pi x)}{\pi x}.$$

Nous énonçons ici sans preuve le *théorème interpolation* (9.3). Sa dérivation est une application classique de l'analyse de Fourier, dans laquelle le sinus cardinal

apparaît naturellement comme transformée de Fourier du *filtre passe-bande* idéal, $II(f) = 1$ pour $|f| < 1/2$, et $II(f) = 0$ sinon ; on a :

$$sinc(t) = \widehat{II}(f) = \int_{-\infty}^{+\infty} II(f) e^{-2i\pi ft} df = \int_{-\frac{1}{2}}^{+\frac{1}{2}} e^{-2i\pi ft} df.$$

Nous revenons plus loin sur la mise en œuvre pratique de la formule (9.3) en audio et vidéo.

Nous renvoyons le lecteur aux ouvrages sur la transformée de Fourier [22], [23] pour une dérivation du théorème interpolation, et la belle justification de cette dérivation à l'aide de la théorie des distributions [24], dans laquelle on intègre et on dérive des fonctions aussi discontinues que la distribution δ de Dirac, telle que $\delta(t) = 0$ pour $t \neq 0$, $\delta(0) = \infty$ et $\int_{-\infty}^{+\infty} \delta(t) dt = 1$.

Comme toute fonction mesurable physiquement est nécessairement - par la nature même du procédé de mesure - bornée en fréquence ; le théorème interpolation s'applique donc dans tous les cas pratiques. Ceci est vrai alors que sa justification mathématique fait intervenir des fonctions qui sont de toute évidence impossible à réaliser physiquement : à commencer par $sin(t)$, un signal *réellement* périodique, de durée et d'énergie *infinies* !

9.1.2 Erreurs de numérisation

Le *théorème interpolation* montre que l'on peut représenter de façon *exacte* une fonction analogique $a(t)$ bornée en fréquence par 1/2, avec la suite $a(n)$ de ses échantillons discrets aux temps entiers $n \in \mathbf{Z}$.

La perte d'information - voir fig. 9.2 et chap. 7 - arrive lors de la conversion de l'échantillon analogique $a(n)$ en échantillon numérique $a_n = \lfloor 2^m a(n) \rceil 2^m$ sur $m+1$ bits. En appliquant la formule interpolation (9.3) à la suite des échantillons numériques a_n, on trouve une fonction analogique $a'(t)$ qui est proche du signal initial $a(t)$. La différence $b(t) = a'(t) - a(t)$ entre les deux est le *bruit numérique* résultant du procédé.

Le nombre de bits m servant à représenter l'échantillon numérique $d = a_n$ en complément à deux,

$$d \;=\; {}_{\frac{1}{2}}d_0\dot{d}_1 \cdots d_{m-1} = -d_0 + \sum_{1 \le k < m} d_k 2^{-k},$$

résulte, en audio et vidéo, d'un compromis complexe entre le coût des équipements, et la qualité subjective de la restitution finale.

9.2 Chaîne de communication des images

9.2.1 Débit des données

Le débit de diverses sources multimédias :
- Voix PCM : 64 kb/s.
- Son CD : 1.5 Mb/s.
- Télévision HDTV : 580 Mb/s.

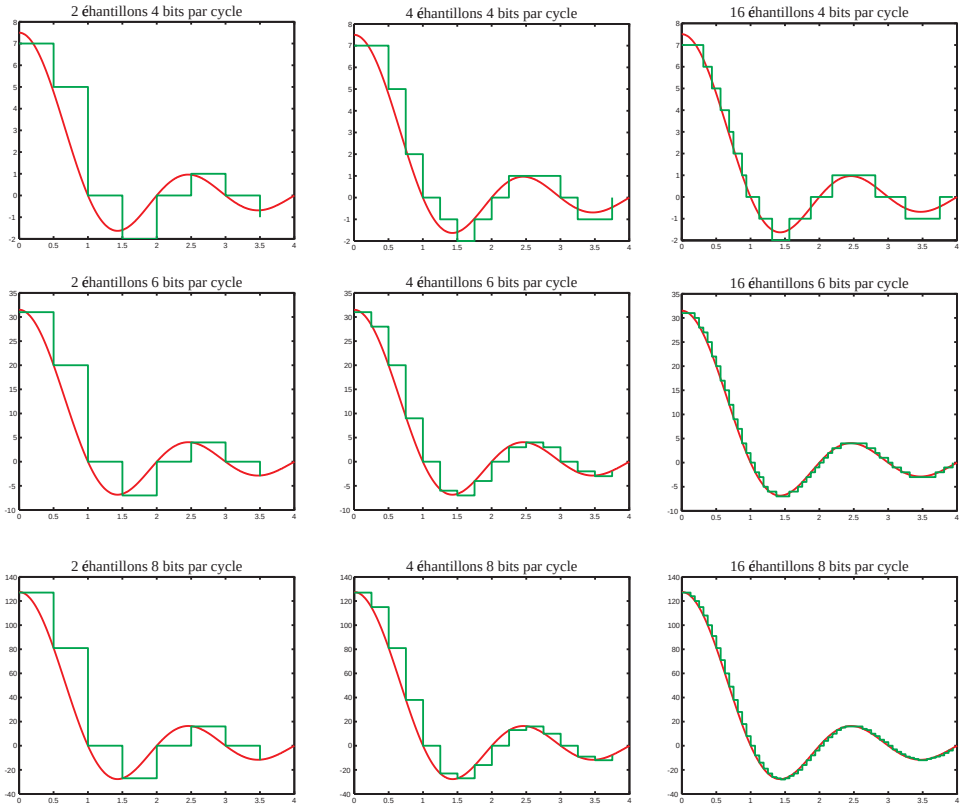
Le débit de divers canaux de transmission :
- Téléphone fixe : 64 kb/s.
- Téléphone mobile GSM : 6.5 ou 13 kb/s.
- Lien réseau T1 : 1.5 Mb/s, T3 : 45 Mb/s.
- Lien réseau Ethernet : 10, 100 , 1000 Mb/s.
- Lien réseau ATM : 155 , 650 Mb/s.

9.2.2 Codages de la source multimédia

Les formats audio les plus communs sont 8 bps - *bits per sample* - en téléphonie, 8 bps pour certains CD, et 32 bps en audio professionnelle - soit 16 bps par voie stéréo.

Les images digitales en noir et blanc (imprimante, fax) comportent 1 bppx - *bits per pixel* ; le format 8 bppx permet de représenter 256 niveaux de gris. Avec 24 bppx, on dispose de l'intensité sur 8 bits de chaque couleur fondamentale : RVB [3] - en anglais RGB. C'est le format de départ de la chaîne des images.

3. pour rouge, vert et bleu



Echantillonnage et numérisation en fonction de la fréquence et de la précision.

Figure 9.2 – Erreurs de numérisation

Images au format BMP et GIF Le format dit *256 couleurs* utilise 8 bppx pour indexer dans une *table des couleurs*, propre à chaque image, et y trouver les $3 \times 8 = 24$ bits de la représentation RVB du pixel.

Codage sur 256 couleurs par *quantification vectorielle*.

Codage par longueur de séquence : RLE.

Codage : LZW.

Images au format JPEG Objectif : de 0.2 à 2 bppx, pour des documents photographiques numérisés. Le taux de compression est modulable, au prix d'une *dégradation* de l'image.

Le coeur de l'algorithme JPEG, qui suit, est décrit en section 9.3.

Vidéo et audio au format MPEG1 Voir les sections 9.4.1 et 9.4.2.

9.2.3 Codages du canal multimédias

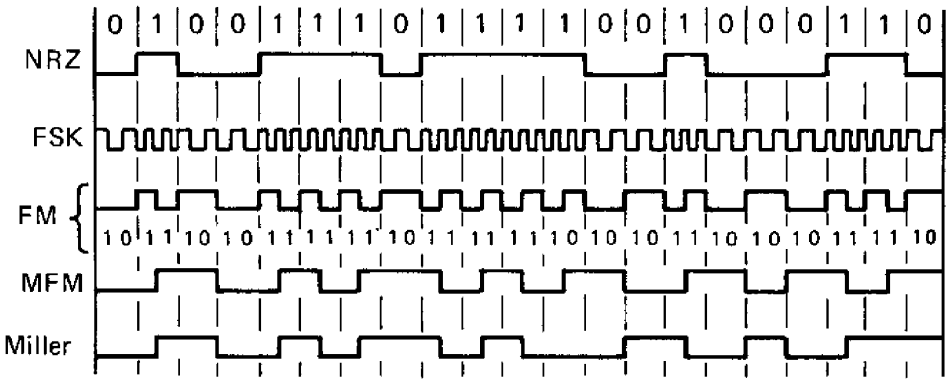
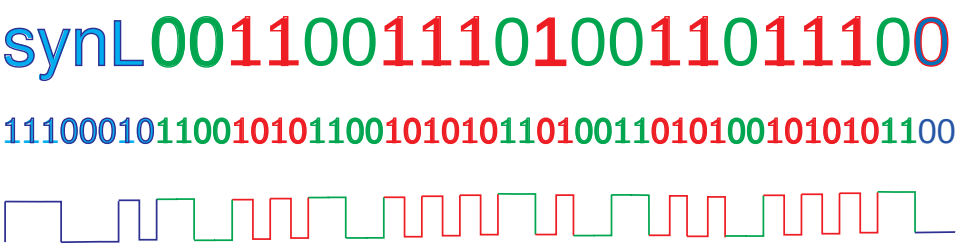


Figure 9.3 – Transmission en bande de base

Standard AESEBU [4].



9.3 Compression d'images photographiques fixes : JPEG

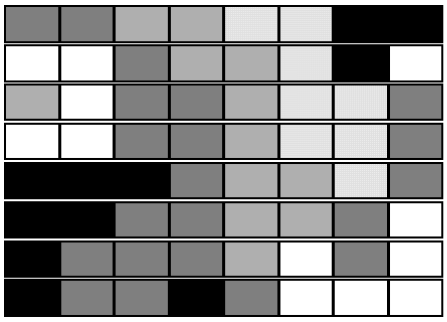
9.3.1 Codage des couleurs

Codage des couleurs par $YCbCr$, avec $8b$ pour Y, et 4b pour Cb et pour Cr, soit 16 bppx.

4. Audio Engineering Society, European Broadcasting Union

9.3.2 Découpage en blocs

9.3.3 Transformée en cosinus



619	-29	24	2	1	-3	0	1
22	-6	-5	0	2	1	-2	3
11	0	5	-4	0	7	0	0
47	-11	-1	-2	1	2	3	1
4	0	-3	1	0	3	0	-2
-8	-2	0	2	-1	-4	-2	-1
1	0	-2	1	3	1	-1	0
-3	-1	-1	-4	1	0	1	-3

9.3.4 Quantification

Luminance

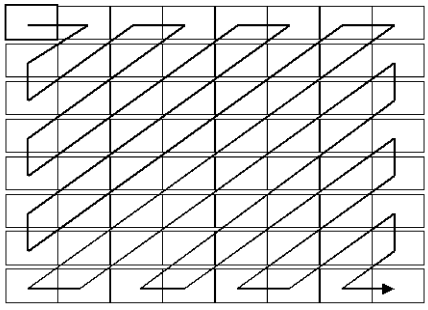
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	106

Chrominance

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

9.3.5 Codage entropique

39	-3	2	0	0	0	0	0
2	-1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
3	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0



(39, -3, 2, 1, -1, 2, 0, 0, 0, 3, 0, -1, EOB)

100101/0100/0110/001/000/0110/11111011111/11000/1010 = 44b = 0.7 b/pel

Zig zag, codage par longueur des suites nulles, enfin : Huffman, ou codage arithmétique.

9.4 Compression vidéo et audio : MPEG

Objectif : audio +vidéo au débit 1.5 Mb/s constant d'un CD.

9.4.1 Compression vidéo : MPEG1

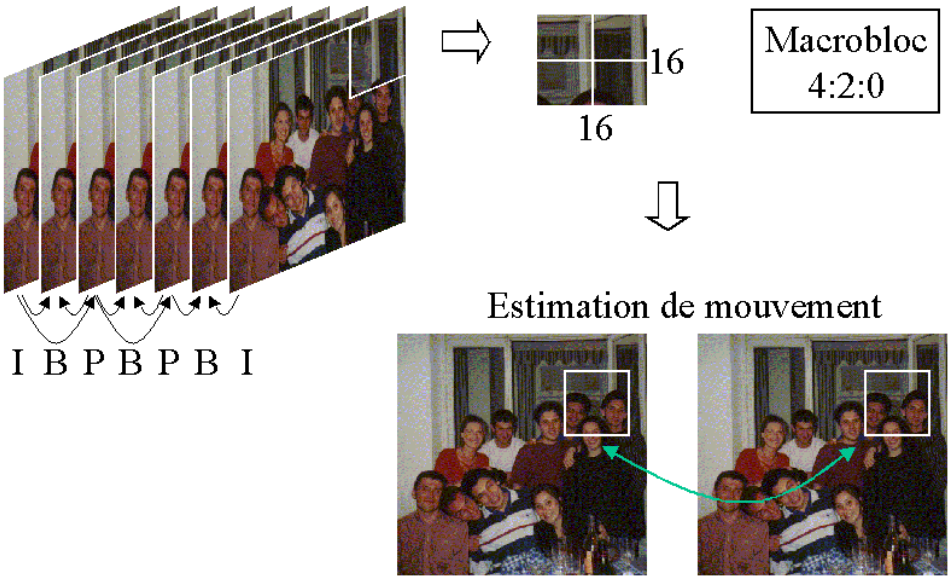
Techniques : interpollation de trames, et compensation de mouvement.
Trois types de trames :

I Codage indépendant des autres trames.

P Codage par prédiction unidirectionelle.

B Codage par prédiction bidirectionelle.

Options pour chaque macro-bloc : compensation de mouvement ou codage JPEG, tables de quantisation.

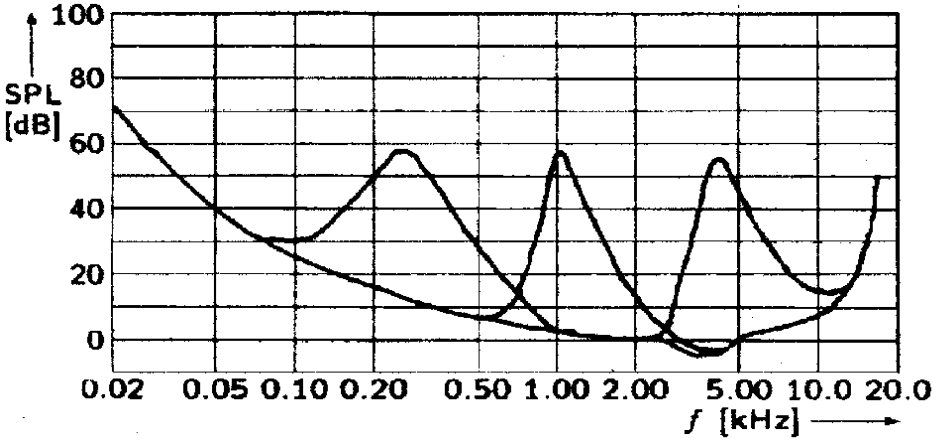


9.4.2 Compression audio : MPEG1

Objectif : taux de compression de 5 à 20, tout en maintenant une haute qualité auditive.

Les méthodes temporelles sont peu efficaces, et la compression se passe donc dans le domaine des fréquences. Transformée en cosinus sur une fenêtre glissante de 16 échantillons.

On tire alors parti de deux propriété de notre canal auditif : le seuil minimum d'audition, en dessous duquel on peut remplacer le signal par 0. Le phénomène de *masquage psycho-acoustique*.



IV

Appendices

Index

5 Sigles

b *Bit* : élément binaire d'information ; vaut 0 ou 1.

B *Byte* : vaut 8 bits.

KB *Kilo Byte* : vaut 1024 octets.

MB *Mega Byte* : vaut 1024 KB.

GB *Giga Byte* : vaut 1024 MB.

ALU *Arithmetic and Logic Unit* : unité arithmétique et logique UAL.

ASCII

cMOS *Complementary MOS* :

C++

CCD *Charge Coupled Device* : caméra numérique.

CDS Circuit digital synchrone :

COBOL

DEC *Digital Equipment Corporation*.

GPS *Geographic Positional System* :

GSM *? ?* : norme européenne pour la téléphonie mobile numérique.

IBM *International Business Machines*.

LCD *Liquid Cristal Display* :

LHC *Large Hadron Collider* :

MODEM *Modulator Demodulator* :

MOS *Metal Oxide Semi-conductor* :

OCR *Optical Character Recognition* :

PC *Personal Computer* :

RAM *Random Access Memory* :

ROM *Read Only Memory* :

UAL Unité arithmétique et logique : ALU. caméra numérique.

VAX *? ?* :

Bibliographie

[1] G. Guitel, *Histoire comparée des numérations écrites*, Flammarion, Paris, 1975.

[2] O. Negeubauer, *The Exact Sciences in Antiquity*, *Princeton University Press*, 1952.

[3] M. V. Wilkes, *The cMOS end-point and related topics in computing*, in *IEEE Trans. Computers*, Computing & Control Engineering Journal, pp.101–106, I.E.E. 1986.

[4] D. E. Knuth, *The Art of Computer Programming*, Volume 2 / *Seminumerical Algorithms*, *Addison-Wesley*, 1980.

[5] C. Mead, *Scaling of MOS Technology to Submicrometre Feature Sizes*, Journal of VLSI Signal Processing, V 8, N 1, pp. 9-26, 1994.

[6] W. J. Gilbert, *Modern Algebra with Applications*, A Wiley-Interscience publication, John Wiley & Sons, New York, 1976.

[7] J. Vuillemin, *On circuits and numbers*, in *IEEE Trans. on Computers*, 43(8) :868–879, 1994.

[8] R. E. Bryant, *Graph-based Algorithms for Boolean Function Manipulation*, in *IEEE Trans. Computers*, 35 :8 :677–691, 1986.

[9] Carver Mead, *ANALOG VLSI AND NEURAL SYSTEMS*, Addison-Wesley, 1989.

[10] R. F. Lyon, "Two's complement pipeline multipliers", *IEEE Trans. on Comm.*, vol. COM-24, pp. 418–425, 1976.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach*, Morgan Kaufmann, 1990.

[12] W. S. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo and S. L. Sze, "A user programmable reconfigurable logic array", *IEEE 1986 Custom Integrated Circuits Conference*, pp. 233–235, 1986.

[13] M. E. Louie and M. D. Ercegovac, "A variable precision multiplier for field-programmable gate arrays", *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, Berkeley, CA, USA, February 1994.

[14] J. E. Vuillemin, "A combinatorial limit to the computing power of VLSI circuits", IEEE Transactions on Computers, C-32 :3 :294-300, 1983.

[15] J. L. Basdevant, "Mécanique quantique", Ecole Polytechnique, 1995.

[16] J. E. Vuillemin, "On computing power", *Programming Languages and System Architectures*, J. Gutknecht, editor, Lecture Notes in Computer Science Nr. 782, Springer-Verlag, pp. 69–86, 1994.

[17] R. P. Feynman, R. B. Leighton and M. Sands, *The Feynman Lectures on Physics*, 3 volumes, Addison-Wesley, 1963.

[18] R. Dautray and J. L. Lions, *Mathematical Analysis and Numerical Methods for Sciences and Technology*, 9 volumes, Springer-Verlag, 1990.

[19] D. Belosloudtsev, P. Bertin, R. K. Bock, P. Boucard, V. Dörsing, P. Kammel, S. Khabarov, F. Klefenz, W. Krischer, A. Kugel, L. Lundheim, R. Männer, L. Moll, K. H. Noffz, A. Reinsch, M. Shand, J. Vuillemin and R. Zoz, "Programmable Active Memories in real-time tasks : implementing data-driven triggers for LHC experiments", to appear in the *Journal of Nuclear Instruments and Methods for Physics Research*, Elsevier Publishers, Amsterdam, NL, 1995.

[20] J. E. Vuillemin, "Fast linear Hough transform", *1994 International Conference on Application-Specific Array Processors*, pp. 1–9, IEEE Computer Society Press, 1994.

[21] J. E. Vuillemin, "Contribution à la résolution numérique des équations de Laplace et de la chaleur", *Mathematical Modelling and Numerical Analysis*, edited by AFCET Gauthier-Villars, RAIRO, vol. 27(5), pp. 591–611, 1993.

[22] J. M. Bony, *Cours d'analyse*, Ecole Polytechnique, 1994.

[23] R. N. Bracewell, *The Fourier Transform and Its Applications*, McGraw-Hill, New York, 1978.

[24] L. Swartz, *Théorie des distributions*, Vols. 1 et 2, Herman & Cie, Paris, 1950 et 1951.

[25] C. E. Shannon, W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, Urbana, 1949.

[26] G. Battail, *Théorie de l'information*, Masson, Paris, 1997.

[27] J. Vuillemin, P. Bertin , D. Roncin, M. Shand, H. Touati, P. Boucard *Programmable Active Memories : the Coming of Age*, IEEE Trans. on VLSI,Vol. 4, NO. 1, 56-69, March 1996.

[28] R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *CACM*, vol. 21(2), pp. 120–126, 1978.

[29] E. F. Brickell, "A survey of hardware implementations of RSA", *Crypto '89*, Lecture Notes in Computer Science Nr. 435, Springer-Verlag, pp. 368–370, 1990.

[30] M. Shand and J. Vuillemin, "Fast implementation of RSA cryptography", *11th IEEE Symposium on Computer Arithmetic*, Windsor, Ontario, Canada, pp. 252–259, 1993.

Liste des figures

Liste des planches

6 Index

Index