Notes de cours : Représentations en binaire

L1 Portail Mathémathiques - Informatique

Basé sur un document de Lhouari Nourine
Modifié par Aurélie LAGOUTTE
Université Clermont Auvergne

28 octobre 2021

Chapitre 0

Histoire de l'informatique

Sommaire

I Intérêt du module

Avoir une idée de la manière dont sont stockées les données dans un ordinateur est une première étape dans votre formation en Informatique, et qui vous permettra d'appréhender ensuite les cours d'algorithmique et de programmation. Vous savez probablement que les composants élémentaires des machines sont des composants électroniques tels que des fils électriques et des transistors, et que l'information qui y circule est *binaire* : soit le courant passe (au-dessus d'un certain seuil), soit il ne passe pas.

Le but de ce cours de Représentations en binaire est de comprendre comment un tel fonctionnement binaire permet de stocker une multitudes d'informations tels que des nombres, des caractères, etc...

L'ordinateur est né du besoin de calculer, de calculer toujours plus et toujours plus vite. Dès que les moyens techniques l'ont permis, l'homme a essayé d'automatiser le calcul. Mais avant d'en arriver à la première machine à calculer, puis à l'ordinateur, il a fallu que certaines bases, certains concepts soient développés. Faisons un petit tour d'horizon de l'histoire de l'informatique, pour alimenter votre culture générale scientifique.

II Fondations de l'informatique

Depuis longtemps, l'homme sait compter par exemple sur ses doigts (*digitus* en latin) et, dès avant JC, il s'est servi de différents objets pour l'aider à compter, notamment des cailloux. Très vite, la nécessité de calcul a donné lieu à la création d'*abaques* (tables à compter) grâce à l'emploi de baguettes, jetons ou boules (bouliers). Mais les méthodes utilisées (base 5, 10,

20 pour ceux comptant sur leurs doigts, base 12 pour ceux qui utilisaient aussi les phalanges, base 60 qui persiste pour compter les secondes et les minutes, système sans base tels que les chiffres romains...) étaient loin de permettre une avancée vers l'automatisation.

Un premier pas est fait avec l'introduction du zéro en tant que chiffre à part entière, par les Indiens quelques siècles après Jésus-Christ. Le zéro fera ensuite son apparition en Europe vers le VIIIème siècle, à la suite de traduction de travaux mathématiques arabes. Les chiffres arabes (dont l'origine doit en fait être attribuée aux Indiens, les Arabes ayant joué un rôle déterminant dans leur transmission en Occident) se sont répandus en Europe entre le Xème et le XVème siècle.

Ces derniers permettent, accompagnés de l'essor des sciences et du commerce lors de la Renaissance, d'ouvrir la voie à des systèmes de calcul plus rapides et plus automatiques.

Vers 1614, le mathématicien écossais *Neper* présente sa théorie des logarithmes permettant de transformer des multiplications en additions. Cette théorie amènera à la réalisation de la *règle à calcul*, outil de calcul pratique et efficace et qui resta longtemps utilisé.

En 1645, *Blaise Pascal* présente la Pascaline, première machine à calculer mécanique basée sur un principe de roues dentées. Cette machine pouvait additionner et soustraire des nombres de six chiffres et prenait en compte les retenues !

En 1694, *Leibniz*, célèbre mathématicien, élabore une machine plus sophistiquée que la machine de Pascal en y ajoutant un mécanisme permettant de faire une multiplication (les sources divergent néanmoins sur la fiabilité du système de retenue). Il découvre également le système binaire, utilisé par les ordinateurs actuels, en s'inspirant d'un vieil écrit chinois datant du Ier millénaire avant JC. Bien que d'autres mathématiciens du XVIIème siècle l'aient découvert avant lui (Thomas Harriot est considéré par certaines sources comme l'inventeur du système binaire moderne), l'effort de diffusion qu'a mené Leibniz lui vaut souvent l'attribution de la paternité du système binaire.

III XIX^e siècle : apparition de calculateurs

Au cours du XVIII^e siècle, le métier à tisser est inventé, puis perfectionné par *Jacquard* en 1801 par l'apport du système de cartes perforées (système des orgues de certains manèges).

Au milieu du XIX^e siècle, *Charles Babbage* fait le rapprochement entre les machines à calculer et le système de commandes automatique de Jacquard. Le concept de la première machine à calculer programmable est né. Il est d'ailleurs considéré comme le père de l'ordinateur. Mais il se heurtera à des problèmes techniques dans la réalisation de sa Machine Analytique et malgré l'aide d'*Ada* Augusta, fille du célèbre poète anglais Lord Byron, qui écrira les premiers programmes de l'histoire de l'informatique, il ne put jamais réaliser son projet.

En 1854, *George Boole*, conçoit un système de logique symbolique, appelée algèbre booléenne. Ce système appliqué à l'arithmétique binaire permettra la conception de l'ordinateur électronique.

En 1887, *Hollerith* construit un calculateur statistique fonctionnant avec des cartes perforées. Ce calculateur servira au recensement de la population américaine et, bien que la population ait augmenté de près de 25% par rapport au dernier recensement, les résultats

seront complets et publiés "seulement" deux ans et demi après ; soit trois fois plus rapidement que le recensement précédent. Il fonda la "Tabulating Machine Company", chargée de fabriquer des systèmes de cartes perforées.

IV XX^e siècle : naissance de l'ordinateur

Les machines à cartes perforées continuent à se développer et en 1924, la TMC de Hollerith devient l'International Business Machines, autrement dit IBM.

Au début du vingtième siècle, des avancées fondamentales importantes vont influencer le développement de l'ordinateur. L'américain *Shannon* fit le parallèle entre les chiffres binaires 0 et 1 et l'état (ouvert ou fermé) d'un relais dans un circuit électrique. Il réussit ainsi à faire une implémentation de l'algèbre de Boole en utilisant des circuits électriques et des relais. *Turing* énonce le principe d'une machine universelle, la *Machine de Turing* qui est à la base de sa théorie des nombres calculables (1936).

Vers la fin des années trente, suite aux travaux de Shannon et Turing, les chercheurs arrivent à la conclusion que la logique booléenne pouvait être employée efficacement dans la conception des calculateurs.

En 1941, *Zuse* fabrique en Allemagne le Z3, un calculateur binaire universel utilisant la représentation en virgule flottante, avec environ 2600 relais. Une partie de son financement est par ailleurs refusée par le gouvernement allemand, jugeant le projet non indispensable à l'effort de guerre. L'invention du Z3 a été peu diffusée à l'époque.

En 1944, IBM construit une machine électromécanique (ASCC Harvard Mark I) capable de multiplier deux nombres de 23 chiffres décimaux en 6 secondes. Cette machine, basée sur le système décimal et non pas binaire, est composée de 760 000 pièces électromécaniques et fonctionne avec des cartes perforées.

En 1945, l'ENIAC (Electronical Numerical Integrator And Calculator), machine universelle, programmable, numérique, basée sur le système décimal et entièrement électronique est construite. Cette machine de 30 tonnes et 18000 tubes à vide, multipliant deux nombres de 10 chiffres en 3 millisecondes sonne le glas des machines électromécaniques. Le défaut de l'ENIAC est que les programmes étaient câblés donc difficilement modifiables.

Également en 1945, *John von Neumann* conçoit une abstraction du système de commande. Son modèle, la *Machine de von Neumann* est le premier concept d'ordinateur. Elle possède les caractéristiques suivantes :

- machine *universelle* contrôlée par *programme* ;
- instructions du programme codées sous forme numérique binaire et *enregistrées en mémoire* ;
- instructions exécutées normalement en séquence mais pouvant être modifiées par le programme lui-même ;
- existence d'instructions permettant les ruptures de séquences.

Il décrit également les cinq composants essentiels de sa machine. Cette architecture, connue sous le nom d'*Architecture de von Neumann* (voir Fig. 1) est toujours celle sur laquelle est basée la conception des ordinateurs modernes. Ces 5 composants sont :

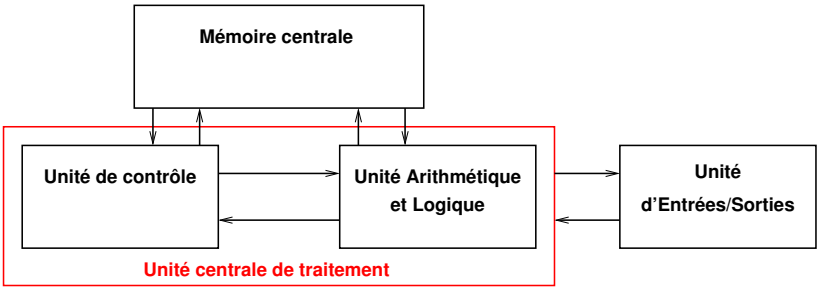


FIGURE 1 – Représentation schématisée de l'architecture de von Neumann

- l'unité arithmétique et logique (UAL) ;
- l'unité de commande ;
- la mémoire centrale ;
- l'unité d'entrée ;
- l'unité de sortie.

En 1949, l'EDSAC, machine basée sur les principes de von Neumann est construite et constitue donc le premier calculateur électronique à haute performance stockant données et programme dans sa mémoire centrale.

En 1947, la découverte par les laboratoires Bell du transistor permet l'apparition à partir de la fin des années 1950 d'une deuxième génération d'ordinateurs où les tubes sont remplacés par des transistors, augmentant ainsi considérablement la fiabilité tout en réduisant l'encombrement et la consommation électrique.

Ensuite, apparaissent les circuits intégrés qui amènent, dans les années 1960, la troisième génération d'ordinateur. Dans les années 70, la miniaturisation amène à la création de microprocesseur, c'est la quatrième génération. Apparaissent aussi dans les décennies suivantes des unités de périphériques de grande capacité (CD/DVD), les imprimantes laser, les écrans couleurs et surtout la naissance des communications inter-ordinateurs et donc des réseaux.

L'importance majeure de l'ordinateur dans notre société et les progrès considérables qu'il a permis ont donné naissance à une nouvelle discipline : l'*informatique*. Cette discipline contient à la fois la conception, l'exploitation et l'évolution de l'ordinateur et des algorithmes. L'académie française en a donné en 1965 la définition suivante :

Important 1 *L'informatique est la science du traitement rationnel de l'information, considérée comme le support des connaissances dans les domaines scientifiques, économiques et sociaux, notamment à l'aide de machines automatiques.*

Chapitre 1

Représentations des entiers

Sommaire

Dans ce chapitre, nous allons essayer de répondre aux questions suivantes :

- Comment les nombres positifs sont-ils représentés ?
- Quel est le plus grand nombre qui peut être représenté par un mot machine ?
- Comment les nombres relatifs sont-ils représentés ?

I Représentation interne des informations

Dans un ordinateur, toute information est stockée sous forme de **BIT** (binary digit, c'est-à-dire un chiffre binaire, qui est soit 0 soit 1). Ce choix est dû aux limitations de l'électronique (le courant passe ou ne passe pas, condensateur chargé ou non, ouvert ou fermé...). On parle ainsi de représentation *binaire* de l'information (il n'y a que **deux** états/chiffres différents).

On regroupe généralement les bits par paquets de 8 bits, 16 bits, 32 bits ou 64 bits. Un paquet de 8 bits est appelé un *octet*. Par exemple, la séquence de bits suivante forme un octet : 10100110. Au lieu de dire un *paquet* de 16 bits (ou 32 bits, etc...), on parle généralement d'un *mot* de 16 bits (ou 32 bits, etc...).

La mémoire des ordinateurs n'étant pas illimitée, le nombre de bits sur lequel on travaille est généralement limité. On verra plus tard qu'il est parfois essentiel de savoir sur combien de bits l'information va être stockée. De nos jours, un nombre est généralement stocké sur 64 bits dans un ordinateur, mais cela peut varier d'un ordinateur à un autre, voire même sur le même ordinateur en fonction de ce que l'on est en train de faire (vous verrez cela lorsque l'on parlera du *type* d'une variable dans le cours d'Introduction à l'Algorithmique et dans celui d'Introduction à la Programmation en C). Dans ce cours, pour simplifier les calculs, on se placera souvent sur un octet (mais pas toujours !).

Définition 1 *On appellera* **taille du mot mémoire** *le nombre de bits sur lequel on travaille. Dans un ordinateur, cela correspond au nombre de bits que peut contenir un emplacement mémoire "élémentaire" appelé registre général du processeur. La taille du mot mémoire d'un ordinateur est spécifiée dans ses caractéristiques.*

La mémoire des ordinateurs est généralement constituée d'un alignement d'octets que l'on compte selon les unités suivantes :

Avec des puissances de 10 :

- 1 Kilo-octet (Ko) : 10^3 octets = 1 000 octets
- 1 Méga-octet (Mo) : 1 000 Ko = 10^6 octets = 1 000 000 octets
- 1 Giga-octet (Go) : 1 000 Mo = 10^9 octets
- 1 Téra-octet (To) : 1 000 Go = 10^{12} octets

Ou avec des puissances de 2 :

- 1 Kibi-octet (Kio) : 2^{10} octets = 1024 octets
- 1 Mébi-octet (Mio) : 2^{10} Kio = 1024$\times$ 1024 octets = 2^{20} octets$\approx$ 1,05 Mo
- 1 Gibi-octet (Gio) : 2^{10} Mio = 2^{30} octets $\approx$ 1,07 Go
- 1 Tébi-octet (Tio) : 2^{10} Gio = 2^{40} octets $\approx$ 1,1 To

Remarque 1 *À l'origine, les préfixes spécifiquement dédiés aux puissances de deux (Kibi, Mébi, ...) n'existaient pas, mais l'architecture des ordinateurs rendaient plus naturelle en pratique l'utilisation de tailles en puissances de 2. Les préfixes traditionnels décimaux (Kilo, Méga, ...) ont donc été utilisés par abus de langage. Ainsi, on avait à l'époque " 1 Ko = 1024 octets". Cependant, cela entretenait une confusion avec l'usage traditionnel des préfixes décimaux dans les autres mesures (1 Km = 1 000 m). Pour tenter de dissiper la confusion, les préfixes de puissance de deux (Kibi, Mébi, ...) ont été créés en 1998. Malheureusement, la confusion demeure car d'une part, certains professionnels continuent à utiliser l'ancien système, et d'autre part, le comptage en puissance de 10 persiste, notamment chez les fabricants de disques durs pour des raisons commerciales. En effet, il est plus vendeur d'afficher un disque dur de 100 Go que de 93,13 Gio, à capacité égale.*

II Représentation des entiers positifs en binaire non-signé

II.1 Introduction

On considère une machine dont la taille du mot mémoire est $n = 8$. Avec ce mot, on peut engendrer $2^n = 256$ combinaisons possibles. Ceci nous permet de représenter $2^n = 256$ nombres. Nous allons voir comment.

Dans la vie courante, les entiers sont représentés en base 10, utilisant ainsi les dix chiffres décimaux $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Pour obtenir la valeur représentée par la suite de chiffres 2 405 par exemple, on multiplie chaque chiffre par la puissance de 10 qui correspond à son emplacement (chiffre des unités, des dizaines, des centaines, des milliers, etc....). Ainsi, $2\ 405 = 2 \times 1000 + 4 \times 100 + 0 \times 10 + 5 \times 1 = 2 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$. La représentation *binaire*, utilisée par les ordinateurs, fait de même, mais cette fois-ci avec les puissances de 2 : on n'utilise plus que deux chiffres différents (0 ou 1) au lieu de dix, et la position du chiffre correspond à une puissance de 2. Ainsi, l'écriture binaire 1101 correspond au nombre $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13$. Pour différencier l'écriture binaire 1101 de l'écriture décimale 1101, qui ne représentent pas le même nombre, on écrira en indice (en bas à droite) la *base* utilisée : en décimal, la base est 10 et en binaire, la base est 2. En reprenant nos exemples : $1101_2 = 13_{10}$ et $1101_2 \neq 1101_{10}$.

En fait, n'importe quel nombre peut être utilisée comme une base : on pourrait imaginer utiliser la base 5, alors dans ce cas on ne peut utiliser que les cinq premiers chiffres $\{0, 1, 2, 3, 4\}$ et il faut compter en puissance de 5 : $132_5 = 1 \times 5^2 + 3 \times 5^1 + 2 \times 5^0 = 25 + 15 + 2 = 42_{10}$. Dans ce cours d'informatique, on utilisera beaucoup la base 2, ainsi que la base 8 et la base 16 pour des raisons que l'on verra un peu plus loin dans ce chapitre. Et bien sûr, on utilisera la base 10 pour se rapporter à des nombres que l'on comprend bien.

De manière générale, si l'on se place en base b sur n chiffres, tous les nombres dont on dispose s'écrivent $a_{n-1} \dots a_2 a_1 a_0$ où tous les a_i doivent être des entiers entre 0 et $b-1$ inclus, et l'on obtient la valeur représentée x en calculant :

$$x = a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + ... + a_1 \times b^1 + a_0 \times b^0$$

Exemple 1
1. **b=10 :** $5402_{10} = 5 \times 10^3 + 4 \times 10^2 + 0 \times 10^1 + 2 \times 10^0 = 5000 + 400 + 2$ *(unités, dizaines, ...).*
2. **b=2 :** $1001\ 0111_2 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 128 + 16 + 4 + 2 + 1 = 151_{10}$.

La représentation en base 2 est dite *Représentation Binaire Non Signée* (RBNS). Le bit le plus à gauche est appelé *bit de poids fort* et celui le plus à droite *bit de poids faible*. Comme pour les nombres en base 10 :

- on peut généralement omettre les zéros placés à gauche du premier chiffre non-nul : ils sont sous-entendus sauf mention explicite du contraire. Ainsi, $001101_2 = 1101_2$, de la même façon que $00321_{10} = 321_{10}$ *(attention à cette règle lorsque l'on verra la représentations des nombres relatifs).*

- pour faciliter la lecture, on insère parfois des espaces entre les chiffres : on écrit souvent $32\ 455_{10}$ à la place de 32455_{10}. En base 10, on a l'habitude de regrouper les chiffres trois par trois en partant toujours de la droite (par exemple $324\ 55_{10}$ ne serait pas très compréhensible) ; en binaire, on a l'habitude de regrouper les chiffres 4 par 4 en partant toujours de la droite : $10\ 1001\ 0111_2 = 1010010111_2$.

Remarque 2 *Avec n bits en RBNS, le plus petit nombre est celui qui s'écrit avec uniquement des 0, et il vaut toujours zéro ; le plus grand s'écrit avec uniquement des 1, et il vaut* $2^n - 1$. *Par exemple sur un octet (*$n = 8$*) :* $0000\ 0000_2 = 0_{10}$ *et* $1111\ 1111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255 = 2^8 - 1$.

Une machine de mot mémoire de n bits permet de représenter en RBNS tous les entiers compris entre 0 et $2^n - 1$ (voire Figure 1.1). Tout nombre supérieur à $2^n - 1$ ne peut être représenté sur cette machine.

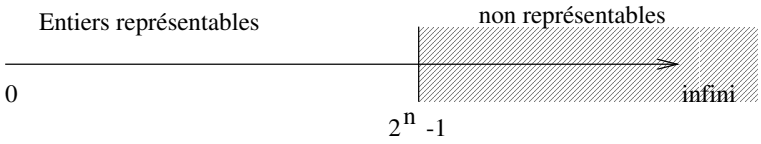


FIGURE 1.1 – L'ensemble des entiers naturels représentables sur une machine de mot mémoire de n bits

Remarque 3 *Par abus de langage, on dit souvent qu'un nombre est écrit "en binaire" au lieu de dire en RBNS (Représentation Binaire Non-Signée). Cependant, on verra que cette dénomination n'est pas assez précise, car "en binaire" signifie seulement "écrit avec uniquement des 0 et des 1". Or, on verra lorsque l'on parlera des nombres négatifs qu'il existe plusieurs conventions différentes pour lire/écrire un nombre en binaire.*

II.2 Conversion d'une base à l'autre

Convertir de la base 2 vers la base 10 On utilise la méthode donnée précédemment lors de la définition de la Représentation en Binaire Non-Signée : on multiplie chaque chiffre par la puissance de 2 correspondante. On peut s'aider d'une représentation à l'aide d'un tableau comme montré ci-dessous, où les deux premières lignes montrent la puissance de 2 correspondant à chaque position. Comme les deux seuls chiffres binaires (0 et 1) se comportent de manière particulièrement facile avec la multiplication, il suffit de reporter en-dessous de chaque bit non-nul la puissance de 2 correspondante, puis d'en faire la somme. Par exemple, pour convertir $1000\ 1010_2$ en décimal :

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	**0**	**0**	**0**	**1**	**0**	**1**	**0**

$= 128 \quad +8 \quad +2 \quad = \mathbf{138_{10}}$

Convertir de la base 10 vers la base 2 La conversion d'un nombre décimal en base 2 se fait par des divisions euclidiennes successives par 2 : on commence par diviser notre nombre par deux, celui nous donne un reste entre 0 et 1, et un quotient plus petit que le nombre dont on est partis ; ensuite on divise ce quotient par deux, ce qui nous donne un nouveau reste entre 0 et 1, et un nouveau quotient, que l'on divise à son tour par 2, etc... On s'arrête lorsque le quotient est égal à 0 (puisque diviser à nouveau par 2 nous donnerait indéfiniment un reste et un quotient tous les deux égaux à 0). Les restes donnent des nombres de 0 à 1 qui forment l'écriture en binaire du nombre. **Attention :** le reste de la **première division** donne le bit de poids faible, c'est-à-dire celui qu'on écrit à **droite** ; inversement, le bit de la dernière division euclidienne donne le bit de poids fort, c'est-à-dire celui qu'on écrit à gauche.

Dans ce cours, on utilisera une forme plus concise pour écrire nos divisions euclidiennes, comme sur l'exemple suivant :

$$335 = 2 \times 167 + 1 \quad \text{au lieu de} \quad \begin{array}{r|l} 335 & 2 \\ \hline 1 & 167 \end{array}$$

c'est-à-dire, de manière plus générale :

$$\text{Dividende} = 2 \times \text{quotient} + \text{reste} \quad \text{au lieu de} \quad \begin{array}{r|l} \text{Dividende} & 2 \\ \hline \text{reste} & \text{quotient} \end{array}$$

Nous sommes prêts pour notre premier exemple : convertissons le nombre 335_{10} en base 2.

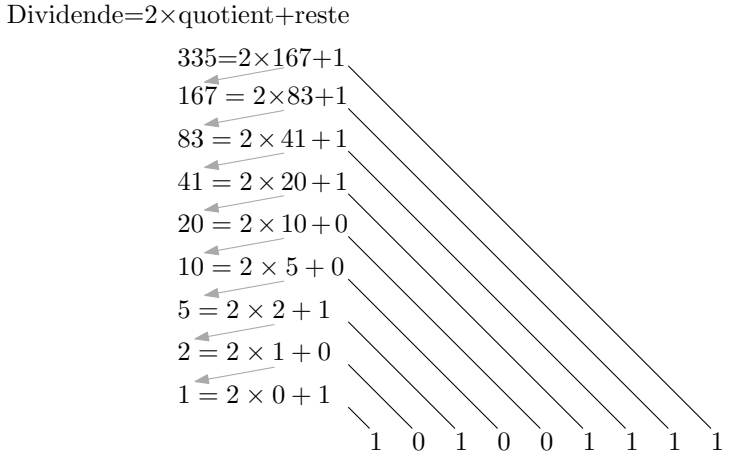


FIGURE 1.2 – Conversion de décimal à binaire du nombre 335

Remarque 4 *Pour convertir un nombre écrit en base 10 vers la base b où b est n'importe quel nombre, on applique la même méthode : on réalise des divisions euclidiennes successives par b, ce qui nous donne à chaque fois un reste entre* 0 *et* $b - 1$*, c'est-à-dire un chiffre en base b. De même que pour le binaire, le reste de la première division nous donne le chiffre à écrire le plus à droite, et le dernier nous donne celui à écrire le plus à gauche.*

Remarque 5 *Dans le cas spécifique du passage de l'écriture en base 10 vers la base 2, on dispose d'une technique plus "astucieuse" qui permet de gagner du temps dans la plupart des cas : il suffit de réussir à écrire notre entier comme une somme de puissances de 2. Par exemple, le nombre* 166_{10} *vaut* $128 + 38$*, où* 128 *est la plus grande puissance de 2 qui soit inférieure ou égal à 166. On continue à décomposer ainsi la partie restante, c'est-à-dire le 38 :*

$$166 = 128 + 38 = 128 + 32 + 6 = 128 + 32 + 4 + 2$$

On en déduit que les bits à 1 dans l'écriture de 166_{10} *sont précisément les bits de poids* $128, 32, 4$ *et* 2 *:*

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	**0**	**1**	**0**	**0**	**1**	**1**	**0**

II.3 Cas des bases puissance de 2 (2^p)

En plus de la base 2, nous allons utiliser occasionnellement la base 8 et la base 16. Ces bases étant des puissances de 2, on peut très facilement passer de la base 2 vers la base 8 ou 16, et inversement, sans repasser par la base 10 comme étape intermédiaire. On dit qu'un nombre écrit en base 8 est écrit en *octal* ; les 8 chiffres utilisés sont $0, 1, 2, \ldots, 7$. On dit qu'un nombre écrit en base 16 est écrit en *hexadécimal* ; les 16 "chiffres" utilisés sont $0, 1, 2 \ldots, A, B, C, D, E, F$ (les lettres servent à représenter les valeurs $10, 11, 12, 13, 14, 15$, dans cet ordre, mais en n'utilisant qu'un seul caractère). Pour indiquer que l'on utilise la représentation hexadécimal, on pourra utiliser soit l'indice 16, soit l'indice h.

Exemple 2
- $305_8 = 3 \times 8^2 + 0 \times 8^1 + 5 \times 8^0 = 3 \times 64 + 0 + 5 = 197_{10}$
- $A3_h = A \times 16^1 + 3 \times 16^0 = 10 \times 16^1 + 3 \times 16^0 = 163_{10}$.

Les bases 8 et 16 (et plus généralement, toutes les bases de la forme 2^p) sont importantes car elle permettent une représentation plus compacte de la représentation des nombres en binaire, comme nous allons le voir.

Conversion de la base 8 ou 16 (ou 2^p) vers la base 2 Partant de la représentation du nombre en base 8, on remplace chaque chiffre de cette représentation ($0, 1, 2, \ldots$ ou 7) par l'écriture en binaire de ce chiffre sur 3 bits. De même, partant de la représentation du nombre en base 16, on remplace chaque chiffre de cette représentation ($0, 1, 2, \ldots 9, A, B, \ldots$ ou F) par l'écriture en binaire de ce chiffre sur 4 bits. La même règle s'applique pour la base 2^p : on remplace chaque chiffre par l'écriture binaire de celui-ci sur p bits.

Exemple 3
- $305_8 = 011\ 000\ 101_2 = 0\ 1100\ 0101_2$ *car* $3_8 = 011_2$, $0_8 = 000_2$ *et* $5_8 = 101_2$
- $A3_h = 1010\ 0011_2$ *car* A_8 *correspond à la valeur* $10_{10} = 1010_2$ *et* $3_h = 0011_2$

Conversion de la base 2 vers la base 8 ou 16 (ou 2^p) Pour convertir en base 8 en partant de la représentation en base 2, il suffit de regrouper les bits par paquet de 3 en partant **de la droite**. Ensuite on remplace chaque paquet de 3 bits par le chiffre correspondant en base 8, qui sera toujours entre 0 et 7. Pour convertir en base 16, on procède de même en regroupant les bits par paquet de 4, toujours en partant de la droite, et chaque paquet nous donnera une valeur entre 0 et F. Pour convertir en base 2^p, on regroupe les bits par paquets de p, toujours en partant de la droite.

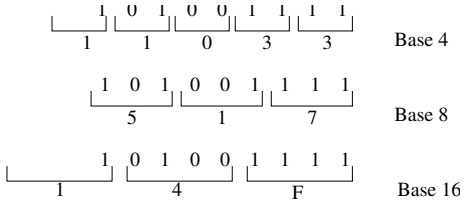


FIGURE 1.3 – Conversion d'un nombre binaire en base 4, 8 et 16

II.4 Addition de 2 nombres en base 2

On peut additionner deux nombres écrits en base 2 sans repasser par la base 10. Pour cela, on applique la même méthode que celle apprise en primaire pour faire des additions en décimal, à quelques détails près (voir Figure 1.4) :

- On écrit les deux nombres l'un en dessous de l'autre.
- On va parcourir les colonnes une par une, en commençant par celle la plus à droite.
- On additionne les deux chiffres de cette colonne, et l'on écrit le résultat en bas si ce résultat tient sur un seul chiffre (par exemple 4+3=7, ou encore 0+1=1) ; par contre s'il ne tient pas sur un seul chiffre, on écrit le chiffre des unités en bas et on marque une retenue pour la colonne suivante. Par exemple 9+6=15 donc on écrit le 5 en bas et on note une retenue. Attention en base 2 : $1_2 + 1_2 = 10_2$ donc on écrit le zéro (chiffre des unités) en bas, et on reporte la retenue.
- On répète l'opération pour toutes les colonnes, en allant de la droite vers la gauche.

Comme pour la base décimale, il arrive que le résultat de l'addition nécessite un chiffre/bit de plus à gauche pour être écrit, par rapport à la taille des nombres que l'on a additionné. Si l'on dispose d'autant de bits que l'on veut, cela ne pose pas de problème. Par contre, en pratique, un ordinateur dispose toujours d'un nombre de bits fixé, que l'on ne peut pas agrandir, par exemple $n = 32$. La dernière retenue est donc perdue, et le résultat est faux (quoiqu'il arrive, le résultat ne peut pas s'écrire sur seulement n bits). On dit alors qu'il y a *dépassement de capacité*.

Il faut donc être vigilant dans les exercices : si l'on vous indique dans l'énoncé que l'on travaille sur n bits pour un certain n fixé, vous devez écrire votre réponse sur n bits et indiquer s'il y a ou non dépassement de capacité.

Remarque 6 ***Attention !*** *On verra plus tard comment faire une addition entre deux nombres écrits en représentation Complément à 2, qui est une représentation différente de la RBNS. La règle du dépassement de capacité sera différente de celle donnée ici. Vérifiez bien dans l'énoncé s'il s'agit d'une addition en RBNS ou d'une addition dans l'autre représentation.*

$$\begin{array}{cccccccccc} & {\scriptstyle 1} & {\scriptstyle 1} & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\ & & 1 & 1 & 0 & 1 & \mathbf{1} & 1 & 1 & 0 \\ + & & & & & & & & & \\ & & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ \hline & \mathbf{(1)} & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}$$

FIGURE 1.4 – Addition de deux nombres binaires pour $n = 8$

III Les entiers relatifs

Pour l'instant, nous ne savons représenter en binaire que des entiers positifs. Or, un ordinateur a besoin de manipuler également des entiers négatifs, il faut donc trouver une nouvelle convention qui nous permettrait d'interpréter différemment les mots binaires, de manière à pouvoir représenter tous les entiers relatifs. Notre objectif est de représenter autant de nombres négatifs que positifs et un seul 0. Comme nous allons le voir, ceci n'est pas possible. Par conséquent, il existe plusieurs façons de représenter un nombre relatif en machine. Nous allons en voir quatre en montrant leurs avantages et leurs inconvénients.

III.1 Valeur absolue signée (VAS)

En se plaçant sur n bits, on réserve le bit de poids le plus fort (à gauche) pour encoder le signe de l'entier relatif ($0 \leftarrow$ positif, $1 \leftarrow$ négatif), et on utilise les $n-1$ autres bits pour écrire la valeur absolue du nombre en RBNS. Pour indiquer qu'un mot binaire est encodé en Valeur Absolue Signée, on indiquera VAS en indice. En Valeur Absolue Signée sur n bits, on peut écrire tous les entiers de $-(2^{n-1}-1) = 11\cdots11_{VAS}$ à $2^{n-1}-1 = 011\cdots11_{VAS}$. En particulier sur un octet, on peut représenter tous les entiers de -127 à 127.

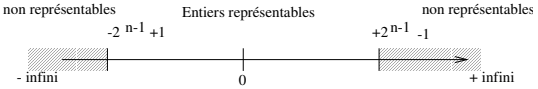


FIGURE 1.5 – Valeur absolue

Exemple 4 *On travaille sur un octet :*

- $89_{10} = 101\ 1001_2 = \mathbf{0}101\ 1001_{VAS}$
- $-55_{10} = \mathbf{1}011\ 0111_{VAS}$ *car* $55 = 011\ 0111_2$

- -203_{10} *et* 203_{10} *ne sont pas représentables sur un octet en VAS car il faut strictement plus de 7 bits pour écrire* 203*, leur valeur absolue.*

Les inconvénients de cette représentation sont :
- 2 représentations de 0 ($+0 = 0000\ 0000_{VAS}$ et $-0 = 1000\ 0000_{VAS}$).
- Opérations arithmétiques peu aisées : l'addition en binaire comme vue pour la RBNS donne souvent des résultats faux : $3_{10} + (-3)_{10} = 0_{10}$ mais l'addition binaire de $0000\ 0011_{VAS}$ et $1000\ 0011_{VAS}$ donne $1000\ 0110_{VAS} = -6_{10}$.

III.2 Complément à 1, noté aussi C1 ou $\overline{1}$

On se place sur n bits et on ne pourra représenter que les nombres de $-(2^{n-1}-1)$ à $2^{n-1}-1$, c'est-à-dire ceux dont la valeur absolue s'écrit en RBNS sur $n-1$ bits (comme pour la VAS). On écrit les entiers positifs comme en RBNS (le bit de poids fort sera donc forcément égal à 0). Pour écrire un entier négatif, on commence par écrire sa valeur absolue en RBNS (le bit de poids fort sera donc forcément égal à 0), puis inverse chaque bit : un zéro devient 1 et inversement. Le bit de poids fort deviendra donc forcément égal à 1 pour un nombre négatif. Pour indiquer qu'un mot binaire utilise la représentation Complément à 1, on mettre un $\overline{1}$ en indice.

Pour retrouver la valeur décimale d'un nombre à partir de sa représentation C1, on commence donc par regarder le bit de poids le plus fort : s'il s'agit d'un zéro, on lit le mot binaire comme en RBNS ; sinon, on inverse tous les bits du mot, on interprète ce nouveau mot comme en RBNS puis on ajoute le signe moins devant cette valeur absolue.

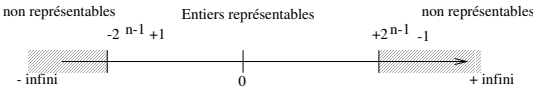


FIGURE 1.6 – Complément à 1

Exemple 5 *On travaille sur un octet :*

- $89_{10} = 0101\ 1001_2 = 0101\ 1001_{\overline{1}}$
- -55_{10} *: on commence par écrire* 55 *en RBNS sur un octet* $55_{10} = 0011\ 0111_2$*, puis on échange tous les bits :* $-55_{10} = 11001000_{\overline{1}}$.
- $1001\ 1100_{\overline{1}}$ *: le bit de poids fort est égal à 1, donc il s'agit d'un nombre négatif. On inverse donc tous les bits, on obtient* $0110\ 0011_2 = 99_{10}$ *donc* $1001\ 1100_{\overline{1}} = -99_{10}$.
- $0100\ 1000_{\overline{1}}$ *: le bit de poids fort est égal à 0, donc il s'agit d'un nombre positif. Il suffit de le lire comme en RBNS :* $0100\ 1000_{\overline{1}} = 0100\ 1000_2 = 72_{10}$.

Les inconvénients de cette représentation sont :
- 2 représentations de 0 : $000\cdots00_{\overline{1}}$ et $111\cdots11_{\overline{1}}$.
- Opérations arithmétiques peu aisées $3 + -3 = 00000011 + 11111100 = 0(11111111)$, mais $4 + -3 = 00000100 + 11111100 = 0(00000000)$.

III.3 Complément à 2, noté aussi C2 ou $\overline{2}$

On se place sur n bits. Pour écrire un entier positif entre 0 et $2^{n-1}-1$, on écrit comme en RBNS sur n bits. Pour écrire un entier négatif, on commence par écrire sa valeur absolue en RBNS, puis on inverse chaque bit, puis on fait une addition binaire entre ce mot et le chiffre 1.

Exemple 6 *Pour écrire* -55 *en Complément à 2 sur un octet :*

- *On écrit 55 en RBNS sur un octet :* $0011\ 0111_2$.
- *On prend le complément à 1 :* 1100 1000
- *On additionne 1 :* 1100 1001
- *On obtient donc* $-55_{10} = 1100\ 1001_{\overline{2}}$.

Un nombre positif ou nul doit toujours avoir son bit de poids fort égal à 0, et un nombre strictement négatif doit toujours avoir son bit de poids fort égal à 1. Sur un octet en complément à 2, on peut ainsi écrire tous les entiers relatifs de $-128_{10} = 1000\ 0000_{\overline{2}}$ à $127_{10} = 0111\ 1111_{\overline{2}}$. Plus généralement sur n bits, on peut écrire en complément à 2 tous les entiers relatifs de -2^{n-1} à $2^{n-1}-1$.

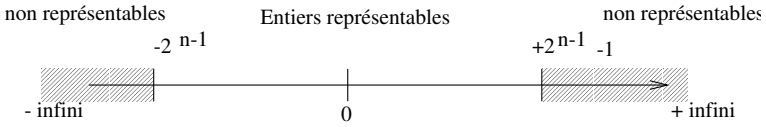


FIGURE 1.7 – Complément à 2

Bien que cette représentation puisse paraître artificielle au premier abord, le complément à 2 est le codage le plus utilisé dans les machines actuelles car elle permet de faire facilement des additions en binaire entre deux entiers relatifs, comme on le verra.

L'inconvénient de cette représentation est que l'intervalle des négatifs n'est pas le symétrique de celui des positifs. L'avantage fondamental est que l'addition fonctionne correctement.

Conversion dans l'autre sens Pour passer de la représentation d'un nombre en complément à 2 sur n bits à sa représentation décimale, on procède ainsi

- Lorsque le bit de poids fort est 0, on sait que le nombre est positif et on l'interprète comme en RBNS. Dans l'autre cas, on sait que le nombre est négatif, et il nous faut chercher sa valeur absolue. On pourrait être tenté de soustraire 1 puis d'inverser tous les bits, mais la soustraction n'est pas toujours aisée avec les seuls outils que l'on a vu jusqu'ici. Il se trouve que l'on peut ré-appliquer la même méthode que pour l'encodage : on prend le complément à 1 du mot binaire et on additionne 1 (on peut prouver que cela donne toujours la bonne valeur). On obtient un mot binaire que l'on interprète en RBNS, et il nous suffit de rajouter le signe négatif.

- Une autre méthode, plus rapide, ne nécessite pas de distinguer deux cas en fonction du bit de poids fort. Dans tous les cas, on interprète de la même manière qu'en RBNS à la différence près que le bit de poids fort a pour poids -2^{n-1} (au lieu de 2^{n-1} en RBNS).

Exemple 7 *On souhaite calculer la valeur décimale de* $(0010\ 1100)_{\overline{2}}$ *et de* $(1000\ 1010)_{\overline{2}}$*.*

- *Avec la première méthode : pour* $(0010\ 1100)_{\overline{2}}$*, on voit que le bit de poids fort est égal à 0, donc il s'agit d'un nombre positif. On le lit comme en RBNS :*

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	1	0	0

$= \quad 32 \quad +8 \quad +4 \quad = 44$

Donc $(0010\ 1100)_{\overline{2}} = 44_{10}$*.*
Pour $(1000\ 1010)_{\overline{2}}$*, on voit que le bit de poids fort est égal à 1, donc il s'agit d'un nombre négatif. On cherche sa valeur absolue en prenant le complément à 1* $(0111\ 0101)$ *puis on additionne 1 :* $0111\ 0110$*. On interprète en RBNS :*

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	0	1	1	0

$= \quad 64 \quad +32 \quad +16 \quad +4 \quad +2 \quad = 118$

Donc $(1000\ 1010)_{\overline{2}} = -118_{10}$*.*

- *Avec la deuxième méthode : on n'a pas besoin de distinguer deux cas entre positifs et négatifs.*

-128	64	32	16	8	4	2	1
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	1	0	0

$= \quad 32 \quad +8 \quad +4 \quad = 44$

Pour le second nombre $(1000\ 1010)_{\overline{2}}$ *:*

-128	64	32	16	8	4	2	1
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	1	0	1	0

$= \quad -128 \quad +8 \quad +2 \quad = -118$

III.4 Codage par Excédent

Le codage par excédent consiste à décaler toute la plage des entiers relatifs à représenter vers une plage d'entiers positifs ou nul, que l'on peut alors représenter en RBNS. Il est

essentiel de savoir de combien la plage a été décalée, c'est-à-dire savoir quel est l'excédent utilisé. Par exemple, en excédent 128 sur un octet, on va pouvoir représenter la plage d'entiers $[-128; 127]$ par la plage d'entiers $[0, 255]$ en décalant chaque entier de $+128$. Ainsi, le nombre -55_{10} sera représenté en Excédent 128 comme la RBNS de $-55 + 128 = 73_{10}$ soit 0100 $1001_{exc(128)}$. Le nombre 55_{10} sera représenté en Excédent 128 comme la RBNS de $55 + 128 = 183_{10}$ soit 1011 $0111_{exc(128)}$.

Sur n bits, si l'excédent utilisé n'est pas précisé, alors il est sous-entendu qu'il s'agit de 2^{n-1}. Avec cet excédent-là, un nombre positif aura toujours son bit de poids fort égal à 1, alors que celui d'un nombre négatif sera toujours égal à 0. L'écriture en excédent 2^{n-1} sur n bits et l'écriture en complément à 2 sur n bits ne diffèrent que par le bit de poids fort, qui est systématiquement inversé. La plage représentable sur n bits en excédent 2^{n-1} est l'intervalle allant de -2^{n-1} à $2^{n-1} - 1$.

Pour obtenir la valeur décimale d'un nombre écrit en excédent E, il suffit d'interpréter le mot binaire comme s'il l'on était en RBNS, puis de retrancher E au résultat. Par exemple pour 0101 $1001_{exc(128)}$: d'abord on calcule 0101 $1001_2 = 64 + 16 + 8 + 1 = 89_{10}$ puis on retranche 128 : $89 - 128 = -39$ donc 0101 $1001_{exc(128)} = -39$.

Les inconvénients de cette représentation sont :
- Représentation des positifs différente de la RBNS.
- Opérations arithmétiques peu aisées car l'addition binaire donne généralement des résultats faux.

L'avantage est qu'il n'y a qu'une seule écriture de la valeur 0.

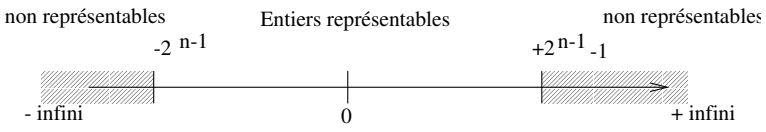


FIGURE 1.8 – Excédent 128

Remarque 7 *On verra plus tard que la représentation en excédent 127 est utile lorsque l'on cherche à représenter des flottants.*

III.5 Conclusion

Tout ensemble de nombres qui contient autant de nombres négatifs et positifs et un seul 0 a un nombre impair d'éléments, alors qu'avec un mot de n bits, on engendre un nombre pair de configurations binaires (2^n).

- En mathématique, l'ensemble $\mathbb{Z}$ des entiers relatifs contient un nombre infini d'éléments.
- En informatique numérique, l'ensemble des entiers avec signe est fini, et dépend de la machine et du langage de programmation utilisé.

IV Opérations arithmétiques en binaire

IV.1 Addition

L'addition en binaire non-signé a déjà été vue dans le début du chapitre. Nous allons voir maintenant comment faire une addition en Complément à 2. Le fait que cette dernière fonctionne bien explique pourquoi la plupart des ordinateurs utilisent effectivement le complément à 2 pour représenter les entiers relatifs.

On travaille sur n bits. Pour faire une addition en complément en deux, on procède pareillement que pour une addition en RBNS sauf que :

- Si l'addition mène à une retenue qui dépasse sur le $n+1$-ème bit (ce qui indique un dépassement de capacité en RBNS sur n bits), on jette systématiquement cette retenue.
- Pour détecter un dépassement de capacité, la règle est différente de celle pour la RBNS : il y a dépassement de capacité en addition Complément à 2 lorsque la retenue qui entre sur le n-ième bit est différente de celle qui dépasserait sur le $n+1$-ième bit.

Voyons des exemples d'addition en complément à deux sur un octet :

1 1 0 1 0 1 0 1	→ -43	1 0 0 0 1 0 1 1 → -117
+ 0 1 1 0 0 0 1 1	→ 99	+ 0 1 1 0 0 0 1 1 → 99
0 0 1 1 1 0 0 0	→ 56	1 1 1 0 1 1 1 0 → -18

Sur les deux exemples ci-dessus, il n'y a pas de dépassement de capacité dans l'addition en Complément à Deux car les deux retenues encadrées sont égales (dans le premier exemple, elles sont toutes les deux égales à 1 ; dans le deuxième exemple, elles sont toutes les deux égales à 0 car les cases des retenues sont vides). On peut vérifier que les deux résultats sont corrects, grâce au fait que l'on n'a pas gardé le 1 qui dépassait sur le 9ème bit dans le premier exemple. On peut même prouver en général qu'il y a dépassement de capacité dans une addition en complément à 2 sur n bits si et seulement si le résultat n'est pas représentable sur n bits en complément à 2.

Voyons maintenant deux exemples où il a dépassement de capacité car les deux retenues encadrées sont différentes l'une de l'autre :

0 0 1 0 0 0 1 0 → 34	1 1 0 1 0 1 0 1 → -43
+ 0 1 1 0 0 0 1 1 → 99	+ 1 0 0 1 0 0 0 0 → -112
1 0 0 0 0 1 0 1 → -123 ≠ 133	0 1 1 0 0 1 0 1 → 101 ≠ -155

Exemple 8 *Addition en complément à deux sur un octet :*

- $01111111_{\overline{2}} + 01111111_{\overline{2}} = (0)11111110_{\overline{2}}$ *ie.* $127 + 127 = -2$.
- $10000000_{\overline{2}} + 10000000_{\overline{2}} = (1)00000000_{\overline{2}}$ *ie.* $-128 + -128 = 0$.

Dans une addition en complément à deux, il faut bien penser à systématiquement jeter ce qui dépasse du nombre de bits dont on dispose. Cela ne signifie pas forcément qu'il y a dépassement de capacité.
La condition de dépassement de capacité est différente selon que l'on fait une addition en RBNS ou en Complément à 2.

La plupart des processeurs possèdent un indicateur de dépassement de capacité pour l'addition pour la RBNS d'une part, et pour l'addition en Complément en 2 d'autre part. Comme nous le verrons plus tard après définition des opérateurs booléens, ce dernier peut s'exprimer comme une fonction booléenne grâce à l'opération **xor** (ou exclusif : soit l'un, soit l'autre, mais pas les deux).

IV.2 Soustraction

La soustraction $x - y$ se fait tout simplement en faisant une addition entre x et $-y$. Il suffit donc de savoir écrire $-y$ en Complément à 2.

IV.3 Multiplication et Division

Remarque 8 *La multiplication et la division peuvent se réaliser par des additions et des soustractions successives. Donc, une machine disposant d'un additionneur peut réaliser les quatre opérations de base.*

Cas particulier des multiplications et divisions par une puissance de 2

Lorsqu'une multiplication ou une division a comme opérande 2^k, on peut la simuler par décalage (Shift) à gauche ou à droite de k positions de l'autre opérande. Cela est similaire à la multiplication et la division par 10^k lorsque l'on travaille en décimal (on ajoute ou on enlève des zéros).

Exemple 9 $13 \times 8 = 13 \times 2^3$ *: on écrit* $13_{10} = 1101_2$ *puis on ajoute 3 zéros à la fin* $110\ 1000_2 = 104$ *(comme si l'on multipliait par 1000 en base 10).*

$96/16 = 96/2^4$ *: on écrit* $96_{10} = 0110\ 0000_2$ *puis on enlève 4 zéros à la fin (comme si l'on divisait par* 10 000 *en base 10) :* $0110_2 = 6_{10}$.

Attention : En Complément à 2 sur un nombre fixé de bits (disons sur un octet), lors d'un décalage à droite, il faut rajouter des 0 à gauche si le nombre était positif, et des 1 à gauche si le nombre était négatif.

Exemple 10 $-24/4 = 1110\ 1000_{\overline{2}}/2^2$ *donc* $-24/4 = \mathbf{11}11\ 1010_{\overline{2}} = -6$ *par décalage à droite.*

Il existe des instructions de décalage sur la plupart des machines, appelée *SAL (Shift Arithmetic Left)* pour le décalage à gauche et *SAR (Shift Arithmetic Right)* pour le décalage à droite.

Chapitre 2

Représentation des nombres réels

Sommaire

En informatique, on utilise souvent l'expression "les flottants" pour parler des nombres réels. Ceci vient d'un abus de langage pour parler de la représentation des nombres réels en virgule flottante, par opposition à la représentation en virgule fixe. La représentation en virgule flottante suit globalement les mêmes principes de ceux de la notation scientifique, où l'on a besoin d'encoder deux informations différentes : l'une appelée *mantisse* et l'autre appelée *exposant*. Cela permet de représenter un intervalle de réels plus grand que la méthode à virgule fixe, pour le même nombre de bits.

I Écrire des réels en base quelconque avec une virgule

Tout comme la base 10, la base 2 ou même n'importe quelle base b permet d'écrire des nombres à virgule, et la valeur représentée par l'écriture suivante :

$$a_{n-1}a_{n-2}\cdots a_0 \ , \ a_{-1}a_{-2}\cdots a_{-m}$$

est calculée ainsi :

$$x = a_{n-1} \times b^{n-1} + ... + a_0 \times b^0 + a_{-1} \times b^{-1} + ... + a_{-m} \times b^{-m}$$

Exemple 11 *Quelques exemples :*

- $123,456_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$
- $101,011_2 = 1\times2^2+0\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2}+1\times2^{-3} = (4+1+0,25+0,125)_{10} = 5,375_{10}$

Conversion dans l'autre sens Pour passer de l'écriture d'un nombre à virgule écrit en décimal à son écriture à virgule en binaire, on effectue des multiplications successives par 2, on l'on découpe chaque résultat intermédiaire entre une partie entière valant 0 ou 1 (ce qui nous donne les bits de l'écriture) et une partie fractionnaire que l'on doit à son tour multiplier par 2, etc.... jusqu'à terminer sur 1,0 (dans l'idéal).

Exemple 12 *On considère le nombre* $5,375_{10}$. *Sa partie entière vaut 5 et s'écrit* 101_2. *Sa partie décimale vaut 0,375 donc on applique les multiplications successives par 2 :*

$$0,375 \times 2 = \boxed{0},75$$
$$0,75 \times 2 = \boxed{1},5$$
$$0,5 \times 2 = \boxed{1},0$$

On obtient donc que $5,375_{10} = 101,011_2$.

Dans certains cas, la méthode des multiplications par 2 tombe dans un cycle qui se répète indéfiniment, comme dans l'exemple suivant où l'on essaye d'écrire $0,45_{10}$ en binaire.

$$0,45 \times 2 = \boxed{0},9$$
$$0,9 \times 2 = \boxed{1},8$$
$$0,8 \times 2 = \boxed{1},6$$
$$0,6 \times 2 = \boxed{1},2$$
$$0,2 \times 2 = \boxed{0},4$$
$$0,4 \times 2 = \boxed{0},8$$

L'écriture binaire de 0,45 demande donc une infinité de chiffres :

$$0,45_{10} = 0,01\,1100\,1100\,1100\,1100\ldots_2$$

On utilise alors la convention suivante : on souligne la partie qui se répète. $0,45_{10}$ s'écrit alors $0,01\underline{1100}_2$. Notons que ce phénomène se produit également en base 10 : $\frac{1}{3} = 0,333333\ldots = 0,\underline{3}_{10}$ ou encore $\frac{39}{74} = 0,5270270270\ldots = 0,5\underline{270}_{10}$

Dans tous les cas, cette écriture des décimaux nécessite de pouvoir écrire le caractère "virgule" en plus des chiffres, ce qui pose problème en binaire si l'on ne dispose que de 2 caractères différents (0 ou 1). Des conventions "à virgule flottante" ont alors été développées pour écrire les nombres de telle sorte que l'on puisse retrouver la place de la virgule même si elle n'est pas écrite.

Remarque 9 *De la même manière que pour transformer l'écriture en base 10 d'un entier positif vers la RBNS, on dispose d'une technique "astucieuse" pour écrire plus rapidement la représentation en binaire avec une virgule d'un nombre décimal dans certains cas : on cherche à l'écrire comme une somme de puissances de 2, en s'autorisant cette fois des puissances dont l'exposant est négatif. Exemple :* $0,625_{10} = 0,5 + 0,125 = 2^{-1} + 2^{-3}$*. On sait donc que les bits de poids 1 dans l'écriture en binaire à virgule à* $0,625_{10}$ *sont précisément les bits de poids* 2^{-1} *et* 2^{-3}*, donc* $0,625_{10} = 0,101_2$*. Attention cependant : cette méthode "astucieuse" échoue pour les nombres dont l'écriture nécessite une infinité de bits.*

II Virgule flottante et forme normalisée

Écrire un nombre x en virgule flottante c'est le mettre sous la forme :

$$x = m * b^e$$

où m appelé *mantisse*, est un nombre comportant une partie entière et une partie décimale, b est la base et e appelé *exposant* est un entier.

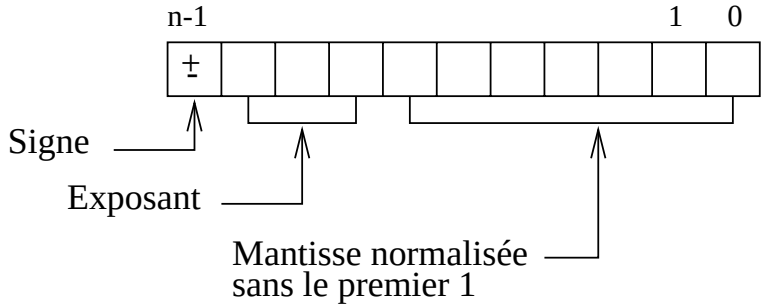
Ceci est une généralisation de la *notation scientifique*, qui requiert de plus que la base soit égale à 10 et que la mantisse soit un nombre de l'intervalle $[1, 10[$ (c'est-à-dire un nombre avec exactement un chiffre non-nul avant la virgule). Par exemple, le nombre $236,43$ s'écrit en notation scientifique ainsi : $2.3643 * 10^2$.

En base 2, nous suivrons la convention la plus courante où la mantisse est un nombre entre $[1, 2[$, ce qui nous assure d'avoir un seul chiffre non-nul avant la virgule, qui sera donc forcément un 1. On dit alors que la mantisse est **normalisée**.

Exemple 13 *En virgule flottante en base 2, le nombre* $100,01_2$ *peut s'écrire : ...* ; $10001_2 * 2^{-2}; 1000,1_2 * 2^{-1}; 100,01_2 * 2^0; 10,001_2 * 2^1;$ $\boxed{1,0001_2 * 2^2}$ *;* $0,10001_2 * 2^3$ *;* $0,010001_2 * 2^4; ....$

Seul le nombre encadré a une mantisse normalisée.

III Représentation en mémoire selon la norme IEEE754



- On travaille sur n bits, avec $n = 32$ pour le format en simple précision et $n = 64$ pour le format en double précision

- Le bit le plus à gauche contient le signe (positif $\rightarrow 0$, négatif $\rightarrow 1$).
- k bits sont réservés pour l'exposant codé **en excédent** 127 avec $k = 8$ pour la simple précision, et **en excédent** 1023 avec $k = 11$ pour la double précision.
- Le reste est réservé à la mantisse normalisée **dont on n'écrit pas le premier 1** (celui avant la virgule ; il est implicite)

Ainsi, un nombre flottant en IEEE 754 s'écrit :

$$x = \pm(1, m)_2 * 2^e$$

où $e \in [\![-126, 127]\!]$ est l'exposant codé en excéd. 127 pour la simple précision,
(ou $e \in [\![-1022, 1023]\!]$ codé en excédent 1023 pour la double précision).

Les valeurs extrêmes $-2^{k-1}+1$ et 2^{k-1} de e (codées comme $0\cdots0$ où $1\cdots1$) sont réservés pour des codages particuliers (voir tableau pour un résumé) :

- Exposant codé $0\cdots0$
  ($e = -127$ en simple précision et $e = -1023$ en double précision) :
  - Avec une mantisse ne contenant que des zéros : la valeur représentée est ±0 suivant le bit de signe.
  - Avec une mantisse contenant au moins un 1 : on représente des nombres **dénormalisés**. Leur mode de calcul n'est pas exigible dans le cadre de ce cours. Pour information : l'exposant à utiliser est $e = -126$ (et non pas -127) ; pour obtenir la mantisse, on prend les bits réservés à la mantisse et on rajoute un zéro avant la virgule (et non pas un 1).
- Exposant codé $1\cdots1$
  ($e = 128$ en simple précision et $e = 1024$ en double précision) :
  - Avec une mantisse ne contenant que des zéros : la valeur représentée est $\pm\infty$ suivant le bit de signe.
  - Avec une mantisse contenant au moins un 1 : aucune valeur représentée NaN (*Not a number*).

Bit de signe	*Exposant*	*Mantisse*	*Objet représenté*
0 ou 1	$000\cdots000$	$000\cdots000$	0
0 ou 1	$000\cdots000$	$\neq 0$	Nombre dénormalisé
0	$111\cdots111$	$000\cdots000$	$+\infty$
1	$111\cdots111$	$000\cdots000$	$-\infty$
0 ou 1	$111\cdots111$	$\neq 0$	*Not a Number*

En résumé :

	Simple précision	Double précision
Nombre de bits pour le signe	1	1
Nombre de bits pour l'exposant	8	11
Nombre de bits pour la mantisse	23	52
Nombre total de bits	32	64
Codage de l'exposant	En excédent 127	En excédent 1023
Variations de l'exposant (pour les nombres normalisés)	de -126 à +127	de -1022 à +1023
Plus petit nombre normalisé	2^{-126}	2^{-1022}
Plus grand nombre normalisé	environ 2^{+128}	environ 2^{+1024}
Echelle des nombres décimaux	environ de 10^{-38} à 10^{+38}	environ de 10^{-308} à 10^{+308}

Exemple 14 *La représentation IEEE 754 du nombre* $x = -\frac{5}{8}$ *en simple précision :* $x = -5 * 2^{-3} = -101_2 * 2^{-3}$ *donc en forme normalisée :* $-1,01_2 * 2^{-1}$

L'exposant est codé par excédent à 127 *donc on encode* $127 - 1 = 126$.

Signe	*Exposant en exc. 127*	*Mantisse sans le premier 1*
1	0111 1110	0100 0000 0000 0000 0000 000

Remarque 10 *Si le nombre de bits de la mantisse n'est pas suffisant pour représenter le nombre de manière exacte (par exemple en cas de partie décimale avec une infinité de chiffres), on arrondit au nombre représentable le plus proche, avec un cas particulier : si le nombre est exactement au milieu entre deux nombres représentables, on choisit celui dont la mantisse termine par un 0. La méthode de l'arrondi au flottant représentable le plus proche nécessite quelques détails techniques qui ne sont pas traités dans le cadre de ce cours. Par contre, il faut être conscient que la représentation binaire en virgule flottante, quel que soit le nombre de bits, ne fait qu'approcher la plupart des nombres décimaux.*

- En mathématique, l'ensemble $\mathbb{R}$ des réels contient un nombre infini non dénombrable d'éléments.
- En informatique numérique, l'ensemble des nombres en virgule flottante normalisée est un ensemble fini qui dépend de la machine et du langage de programmation utilisé.

IV Opérations en virgule flottante

Soient x et y deux nombres représentés en machine selon la convention du flottant normalisé (simple précision, pour simplifier, mais la règle est similaire pour la double précision). On note par

- S_x, S_y les signes de x et y.
- E_x, E_y leurs exposants en excédent 127.
- M_x, M_y leurs mantisses normalisées (avec le 1 avant la virgule).

IV.1 Addition (et Soustraction) flottante

Addition de deux flottants positifs

Soient x et y deux flottants positifs dont on souhaite calculer la somme. On travaille à partir de leur forme normalisée en base 2 (si x et y nous sont donnés en norme IEEE754, on commence donc par écrire leur forme normalisée). L'addition consiste à réaliser les étapes suivantes :

- L'exposant du plus petit est amené à la valeur de l'exposant du plus grand. Ceci se fait en introduisant des zéros non significatifs à la gauche de la mantisse.
- Les mantisses sont additionnées comme en RBNS sur $2 * n$ bits.
- Si nécessaire, réaliser une phase de normalisation de la mantisse résultante et vérification du débordement.
- La mantisse résultante ne peut avoir que n bits : si besoin, nous devons donc arrondir le dernier nombre.

Exemple 15 *Soient x et y les flottants représentés en norme IEE754 (simple précision) par :*

$$x = 0 \mid 1000\ 0010 \mid 01000000000000000000000$$

$$y = 0 \mid 0111\ 1011 \mid 11010000000000000000000$$

On a donc $x = 1,01_2 \times 2^3$ et $y = 1,1101_2 \times 2^{-4}$. Le nombre y est le plus petit, il faut donc écrire y sous la forme $y = \ldots \times 2^3$, c'est-à-dire $y = 0,00000011101 \times 2^3$. On additionne les mantisses en RBNS (en conservant la position de la virgule) :

$$\begin{array}{rl} & 1,01 \\ + & 0,00000011101 \\ \hline & 1,01000011101 \end{array}$$

On a donc $x + y = 1,01000011101_2 \times 2^3$. Il s'agit bien d'une forme normalisée, donc on n'a pas besoin de re-normaliser. On écrit maintenant le résultat en norme IEEE754 :

$$0 \mid 1000\ 0010 \mid 01000011101000000000000$$

Addition de deux flottants négatifs On traite d'abord le bit de signe : on sait que le résultat sera négatif donc le bit de signe sera 1. Pour le reste, on additionne $|x|$ et $|y|$, ce qui revient au cas précédent de somme de deux nombres positifs.

Exemple 16 *Soient x et y les flottants représentés en norme IEE754 (simple précision) par :*

$$x = 1 \mid 0111\ 1010 \mid 11000000000000000000000$$

$$y = 1 \mid 0111\ 1001 \mid 10010000000000000000000$$

On a donc $x = -1,11_2 \times 2^{-5}$ et $y = -1,1001_2 \times 2^{-6}$. On travaille avec $|x| = 1,11_2 \times 2^{-5}$ et $|y| = 1,1001_2 \times 2^{-6}$. Le nombre y est le plus petit en valeur absolue, il faut donc écrire $|y|$ sous la forme $|y| = \ldots \times 2^{-5}$, c'est-à-dire $|y| = 0,11001 \times 2^{-5}$. On additionne les mantisses en RBNS (en conservant la position de la virgule) :

$$\begin{array}{rr} & {\scriptstyle 1\ 1\ \ 1} \\ & 1,11 \\ + & 0,11001 \\ \hline & 10,10001 \end{array}$$

On a donc $|x| + |y| = 10,10001_2 \times 2^{-5}$*, ce que l'on renormalise en* $1,010001_2 \times 2^{-4}$*. On obtient donc :*

$$1 \mid 0111\ 1011 \mid 01000100000000000000000$$

Addition de deux flottants de signe différents Supposons que la valeur absolue du nombre positif soit plus grande que celle du nombre négatif, ce qui nous assure que le résultat sera positif. Appelons x le nombre positif et y le nombre négatif. L'exposant de $|y|$ est ramené à l'exposant de x, qui est plus grand en valeur absolue. Puis on transforme la mantisse de $|y|$ selon une règle similaire au complément à 2. Cette étape assez technique est hors programme pour ce cours de Représentations en binaire. Une fois ceci effectué, on fait alors une addition en complément à 2 entre la mantisse de x et la mantisse nouvellement écrite de $|y|$. Si besoin, on re-normalise le résultat.

Si la valeur absolue du nombre positif x est plus petite que celle du nombre négatif y, alors on calcule en fait $|y| - x$ et on met le bit de signe à 1.

IV.2 Multiplication flottante

Soit $z = x * y$ leur produit en machine.

- $S_z = S_x * S_y$ (règle des signes).
- $E_z = E_x + E_y - 127$ (puisque E_x et E_y sont représentés en excédent 127). Le test du débordement peut être vérifié à ce point.
- $M_z = M_x * M_y$ le produit se représente sur $2 * n$ bits.
- Ce produit peut être dénormalisé. Dans ce cas, il faut décaler d'un bit à droite et ajouter 1 à l'exposant.
- La mantisse résultante ne peut avoir que n bits : si besoin, nous devons donc arrondir le dernier nombre.

Cas particulier des multiplications et divisions par des puissances de 2 Comme pour les entiers, la multiplication et la division par une puissance de 2 est très facile. Remarquons que les puissances de 2 ont la particularité d'avoir tous leurs 23 bits de mantisse égaux à 0 dans la norme IEE754 (leur mantisse en forme normalisée comporte un seul 1, c'est celui qui est juste avant le zéro et que l'on omet dans la norme IEEE754).

Exemple 17 *Considérons la multiplication entre x et y avec :*

$$x = 0 \mid 0111\ 0010 \mid 01000011101000000000000$$

et

$$y = 1 \mid 1000\ 0010 \mid 00000000000000000000000$$

on a donc $x = 1,01000011101_2 \times 2^{-13}$ *et* $y = -1,0_2 \times 2^3$ *donc on obtient de manière directe que* $x \times y = -1,01000011101_2 \times 2^{-13} \times 2^3 = -1,01000011101_2 \times 2^{-10}$*, c'est-à-dire :*

$$1 \mid 0111\ 0101 \mid 01000011101000000000000$$

Remarque sur la précision informatique La machine permet de représenter de manière exacte tout entier relatif compris entre le plus petit nombre et le plus grand. Contrairement aux entiers naturels, les réels n'ont qu'une image approximative en machine, car il existe une infinité de nombres réels entre deux entiers (par exemple 0 et 1). En notation flottante double précision, on ne peut représenter de manière exacte que $\approx 2^{64}$ nombres.

Chapitre 3

Représentation des caractères et codes correcteurs

Sommaire

Pour stocker des caractères sur un ordinateur, on stocke en fait un numéro qui correspond au code de ce caractère selon une certaine convention. Il existe plusieurs conventions pour associer un numéro à chaque caractère, parfois non compatibles entre elles. On peut citer EBCDIC(Extended Binary Decimal Interchange Code), codage sur 8 bits spécifique à IBM et utilisé sur quelques gros ordinateurs ; et l'ASCII, qui est le plus influent de nos jours, notamment grâce à ses extensions Latin-1 (ISO/CEI 8859-1) et Unicode. Cette dernière, bien plus élaborée que l'ASCII de base, est encore en évolution et tend à être la plus utilisée car elle propose un standard international pour remplacer les différentes tables de codage nationales.

I ASCII(American Standard Code for Information Interchange)

L'ASCII est un code sur 7 bits, qui permet de coder 128 caractères en les numérotant de 0 à 127.

Il comprend :

- les lettres de l'alphabet en majuscules et minuscules ($41_h..5A_h$ et $61_h..7A_h$).
- les chiffres décimaux.($30_h..39_h$)
- quelques symboles mathématiques (<,>,...).
- 32 caractères de contrôle (0..31). Par exemple 7_h : BEL (pour *Bell* : bip sonore) ; $0D_h$: CR (pour *Carriage return* : retour chariot/saut à la ligne).

Voici une table ASCII complète (sur deux pages) :

Décimal	Octal	Hexadécimal	Binaire	Caractère	Commentaire
000	000	00	00000000	NUL	(Null char.)
001	001	01	00000001	SOH	(Start of Header)
002	002	02	00000010	STX	(Start of Text)
003	003	03	00000011	ETX	(End of Text)
004	004	04	00000100	EOT	(End of Transmission)
005	005	05	00000101	ENQ	(Enquiry)
006	006	06	00000110	ACK	(Acknowledgment)
007	007	07	00000111	BEL	(Bell)
008	010	08	00001000	BS	(Backspace)
009	011	09	00001001	HT	(Horizontal Tab)
010	012	0A	00001010	LF	(Line Feed)
011	013	0B	00001011	VT	(Vertical Tab)
012	014	0C	00001100	FF	(Form Feed)
013	015	0D	00001101	CR	(Carriage Return)
014	016	0E	00001110	SO	(Shift Out)
015	017	0F	00001111	SI	(Shift In)
016	020	10	00010000	DLE	(Data Link Escape)
017	021	11	00010001	DC1	(XON)(Device Control 1)
018	022	12	00010010	DC2	(Device Control 2)
019	023	13	00010011	DC3	(XOFF)(Device Control 3)
020	024	14	00010100	DC4	(Device Control 4)
021	025	15	00010101	NAK	(Negative Acknowledgement)
022	026	16	00010110	SYN	(Synchronous Idle)
023	027	17	00010111	ETB	(End of Trans. Block)
024	030	18	00011000	CAN	(Cancel)
025	031	19	00011001	EM	(End of Medium)
026	032	1A	00011010	SUB	(Substitute)
027	033	1B	00011011	ESC	(Escape)
028	034	1C	00011100	FS	(File Separator)
029	035	1D	00011101	GS	(Group Separator)
030	036	1E	00011110	RS	(Request to Send)(Record Separator)
031	037	1F	00011111	US	(Unit Separator)
032	040	20	00100000	SP	(Space)
033	041	21	00100001	!	(exclamation mark)
034	042	22	00100010	"	(double quote)
035	043	23	00100011	#	(number sign)
036	044	24	00100100	$	(dollar sign)
037	045	25	00100101	%	(percent)
038	046	26	00100110	&	(ampersand)
039	047	27	00100111	’	(single quote)
040	050	28	00101000	(	(left opening parenthesis)
041	051	29	00101001	)	(right closing parenthesis)
042	052	2A	00101010	*	(asterisk)
043	053	2B	00101011	+	(plus)
044	054	2C	00101100	,	(comma)
045	055	2D	00101101	-	(minus or dash)
046	056	2E	00101110	.	(dot)
047	057	2F	00101111	/	(forward slash)
048	060	30	00110000	0	
049	061	31	00110001	1	
050	062	32	00110010	2	
051	063	33	00110011	3	
052	064	34	00110100	4	
053	065	35	00110101	5	
054	066	36	00110110	6	
055	067	37	00110111	7	
056	070	38	00111000	8	
057	071	39	00111001	9	
058	072	3A	00111010	:	(colon)
059	073	3B	00111011	;	(semi-colon)
060	074	3C	00111100	<	(less than sign)
061	075	3D	00111101	=	(equal sign)

062	076	3E	00111110	>	(greater than sign)
063	077	3F	00111111	?	(question mark)
064	100	40	01000000	@	(AT symbol)
065	101	41	01000001	A	
066	102	42	01000010	B	
067	103	43	01000011	C	
068	104	44	01000100	D	
069	105	45	01000101	E	
070	106	46	01000110	F	
071	107	47	01000111	G	
072	110	48	01001000	H	
073	111	49	01001001	I	
074	112	4A	01001010	J	
075	113	4B	01001011	K	
076	114	4C	01001100	L	
077	115	4D	01001101	M	
078	116	4E	01001110	N	
079	117	4F	01001111	O	
080	120	50	01010000	P	
081	121	51	01010001	Q	
082	122	52	01010010	R	
083	123	53	01010011	S	
084	124	54	01010100	T	
085	125	55	01010101	U	
086	126	56	01010110	V	
087	127	57	01010111	W	
088	130	58	01011000	X	
089	131	59	01011001	Y	
090	132	5A	01011010	Z	
091	133	5B	01011011	[	(left opening bracket)
092	134	5C	01011100	\	(back slash)
093	135	5D	01011101	]	(right closing bracket)
094	136	5E	01011110	^	(caret cirumflex)
095	137	5F	01011111	_	(underscore)
096	140	60	01100000	‘	
097	141	61	01100001	a	
098	142	62	01100010	b	
099	143	63	01100011	c	
100	144	64	01100100	d	
101	145	65	01100101	e	
102	146	66	01100110	f	
103	147	67	01100111	g	
104	150	68	01101000	h	
105	151	69	01101001	i	
106	152	6A	01101010	j	
107	153	6B	01101011	k	
108	154	6C	01101100	l	
109	155	6D	01101101	m	
110	156	6E	01101110	n	
111	157	6F	01101111	o	
112	160	70	01110000	p	
113	161	71	01110001	q	
114	162	72	01110010	r	
115	163	73	01110011	s	
116	164	74	01110100	t	
117	165	75	01110101	u	
118	166	76	01110110	v	
119	167	77	01110111	w	
120	170	78	01111000	x	
121	171	79	01111001	y	
122	172	7A	01111010	z	
123	173	7B	01111011	{	(left opening brace)
124	174	7C	01111100	|	(vertical bar)
125	175	7D	01111101	}	(right closing brace)
126	176	7E	01111110	~	(tilde)
127	177	7F	01111111	DEL	(delete)

Remarque 11 *Sur un octet, le 8ème bit est souvent utilisé, soit pour :*

- *utiliser 128 caractères supplémentaires (numéros 128 à 255), pour encoder des symboles nationaux, mathématiques, grecs, etc... On parle alors d'ASCII "étendu". Il existe de nombreuses tables "ASCII étendu" différentes.*
- *ajouter un bit de parité, qui permet de détecter une erreur de transmission dans certains cas. Ceci sera expliqué dans la section suivante.*

II Erreurs de transmission et Codes correcteurs

Aucun système informatique n'est à l'abri des erreurs lors du transport de l'information via les réseaux de communication ou par corruption d'un support (exemple : CD/DVD rayé). Ainsi, si l'on envoie le mot binaire 1101 0001, il se peut que le destinataire reçoive le mot binaire 1101 0011, où l'avant-dernier bit a été modifié involontairement. On dit alors que ce bit a été *corrompu*. Plusieurs algorithmes de détection et éventuellement de correction d'erreurs ont été développés. Chacun a ses limites en terme de quantité d'erreurs détectables.

II.1 Bit de parité

Cette méthode consiste à rajouter un bit (**bit de parité**) aux n bits formant le mot binaire.

- Parité paire : le bit rajouté est calculé de façon à ce que le nombre total de bits à 1, y compris le bit de parité, soit pair.
- Parité impaire : le bit rajouté est calculé de façon à ce que le nombre total de bits à 1, y compris le bit de parité, soit impair.

Exemple 18 *En ASCII, la lettre "A" se code* 100 0001 *; si l'on rajoute un bit de parité comme 8ème bit à gauche, on obtient :*
$\rightarrow$ *en parité paire, l'octet* **0**100 0001 *;*
$\rightarrow$ *en parité impaire, l'octet* **1**100 0001*.*

Ainsi, lors de la réception d'un message, on peut vérifier que le bit de parité est conforme à la convention choisie (parité paire ou impaire). Si ce n'est pas le cas, on sait qu'au moins un bit a été corrompu. On ne sait pas lequel, donc on ne peut pas corriger l'erreur, mais on peut demander à l'expéditeur qu'il nous envoie à nouveau le message.

Propriétés de cette méthode Dans les exemples ci-dessous, on suppose que le bit de poids fort sert de bit de parité paire.

- Si exactement un bit a été corrompu, on est certains de détecter qu'il y a eu une erreur. *Exemple :* si l'on reçoit l'octet 0110 0100, on détecte que le bit de parité n'est pas valide donc il y a eu (au moins) une erreur de transmission.
- Si plusieurs bits ont été corrompus, nous n'avons pas de garantie sur la détection ou non de l'erreur. *Exemple :* si l'expéditeur envoie l'octet 1001 1100 et que le destinataire reçoit l'octet 1000 0100, ce dernier ne peut pas détecter qu'il y a eu des erreurs dans la transmission.

II.2 Code de Hamming

Proposé par R. Hamming (1952), cette méthode est plus élaborée que l'ajout d'un simple bit de parité. On va rajouter un certain nombre de bits de parité de telle sorte que, si un unique bit arrive corrompu parmi le mot binaire transféré, on puisse détecter lequel. On peut ainsi corriger l'erreur et retrouver le message d'origine envoyé par l'expéditeur.

Plus précisément : l'encodage d'un mot binaire de n bits en code de Hamming consiste à rajouter k bits de parité (k est choisi de telle sorte que $n+k < 2^k$). Le nouveau mot est ainsi constitué de $(n+k)$ bits, numérotés de b_1 à b_{n+k}, et contient donc n bits d'informations et k bits de contrôle positionnés sur les indices correspondant à des puissances de 2 (b_1, b_2, b_4, b_8, etc... sont des bits de contrôle). Chaque bit de parité b_i contrôle ainsi les bits d'informations dont les indices contiennent i en RBNS.

Encodage Pour coder avec Hamming, on liste les positions des bits à 1 dans le message d'origine (hors bits de contrôle), on écrit ces positions en RBNS sur k bits (où k est le nombre de bits de contrôle) les unes en-dessous des autres, puis on décide pour chaque colonne quel est le bit de parité à ajouter. Chacune de ces colonnes correspond à un bit de contrôle, donc on reporte ces bits de parité aux emplacements prévus pour les bits de contrôle.

Dans ce cours, nous prendrons la convention de numéroter les bits de droite à gauche pour les codes de Hamming, donc b_1 est toujours le plus à droite.

Exemple 19 *L'expéditeur veut envoyer le message* $010\ 0011_2$ *(qui correspond d'ailleurs au caractère # en ASCII) avec un code de Hamming en parité paire. Il faut donc choisir le nombre de bits de contrôle k de telle sorte que* $k+7 < 2^k$*, et s'il n'y a pas d'autres contraintes sur la taille du message, on choisit le plus petit k qui vérifie cette inégalité. Il nous faut donc 4 bits de contrôle, qui seront placés aux positions qui sont des puissances de 2, soit* b_1*,* b_2*,* b_4 *et* b_8*. Le message aura donc la forme :*

b_{11}	b_{10}	b_9	$\mathbf{b_8}$	b_7	b_6	b_5	$\mathbf{b_4}$	b_3	$\mathbf{b_2}$	$\mathbf{b_1}$
0	*1*	*0*	*?*	*0*	*0*	*1*	*?*	*1*	*?*	*?*

Les bits à 1 sont b_3*,* b_5 *et* b_{10}*. On va donc écrire les nombres 3, 5 et 10 en RBNS, les uns en-dessous des autres, puis on regarde colonne par colonne quel est le bit de parité à ajouter :*

Position des bits à 1	*Ecriture en RBNS de la position*
3	*0 0 1 1*
5	*0 1 0 1*
10	*1 0 1 0*
Bits de contrôle (parité paire)	***1 1 0 0***
	b_8 b_4 b_2 b_1

Le message encodé à envoyer est donc :

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
0	*1*	*0*	**1**	*0*	*0*	*1*	**1**	*1*	**0**	**0**

Décodage : détection et correction d'une erreur si elle existe L'avantage du code de Hamming provient du fait que chaque bit d'information est contrôlé par au

moins deux bits de parité : par exemple, b_5 est contrôlé par b_4 et b_1. Ainsi, lorsqu'un bit d'information est erroné, plusieurs bits de parité deviennent incohérents alors que, lorsqu'il s'agit d'un bit de contrôle erroné, lui seul est incohérent.

Exemple 20 *On a reçu le message suivant, codé suivant Hamming en parité paire :*

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
0	*1*	*0*	*1*	*1*	*0*	*1*	*1*	*1*	*0*	*0*

Dans un tableau similaire à celui utilisé pour l'encodage, on écrit en RBNS les positions de tous les bits à 1 (y compris les bits de contrôle, cette fois-ci), puis on contrôle la cohérence de la parité colonne par colonne :

Position des bits à 1	*Ecriture en RBNS de la position*			
3	*0*	*0*	*1*	*1*
4	*0*	*1*	*0*	*0*
5	*0*	*1*	*0*	*1*
7	*0*	*1*	*1*	*1*
8	*1*	*0*	*0*	*0*
10	*1*	*0*	*1*	*0*
Cohérence de la parité paire	*Oui*	***Non***	***Non***	***Non***
	b_8	b_4	b_2	b_1

Il y a incohérence des bits b_1, b_2 et b_4. Pour détecter l'erreur, il suffit de calculer la somme des numéros de positions incohérentes, c'est-à-dire 1+2+4 =7. On peut donc en déduire que le message envoyé était, avec correction de b_7 :

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
0	*1*	*0*	*1*	$\underline{0}$	*0*	*1*	*1*	*1*	*0*	*0*

Le message d'origine, avant encodage, s'obtient en supprimant les bits de contrôle b_1, b_2, b_4 et b_8 : il s'agit de 0100011_2.

Ce code fonctionne également lorsque c'est un bit de contrôle qui a été inversé.

Exemple 21 *On a reçu le message suivant, codé suivant Hamming en parité paire :*

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
0	*1*	*1*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*1*

Dans un tableau similaire à celui utilisé pour l'encodage, on écrit en RBNS les positions de tous les bits à 1 (y compris les bits de contrôle), puis on contrôle la cohérence de la parité colonne par colonne :

Position des bits à 1	*Ecriture en RBNS de la position*			
1	*0*	*0*	*0*	*1*
6	*0*	*1*	*1*	*0*
9	*1*	*0*	*0*	*1*
10	*1*	*0*	*1*	*0*
Cohérence de la parité paire	*Oui*	***Non***	*Oui*	*Oui*
	b_8	b_4	b_2	b_1

Il y a incohérence du bit b_4. Il y a donc eu un bit corrompu, c'est b_4. On peut donc en déduire que le message envoyé était :

b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
0	*1*	*1*	*0*	*0*	*1*	*0*	*1*	*0*	*0*	*1*

Le message d'origine, avant encodage, s'obtient en supprimant les bits de contrôle b_1, b_2, b_4 et b_8 : il s'agit de 0110100_2.

Remarque 12 *Lorsque le bit corrompu est un bit d'information, on aura au moins 2 bits de contrôle incohérents. Par contre, lorsque le bit corrompu est un bit de contrôle, on n'aura qu'un seul bit de contrôle incohérent (celui-ci).*

Propriétés de cette méthode

- Si exactement un bit du message arrive corrompu, on est certain de pouvoir détecter qu'il y a eu une erreur et identifier la position du bit corrompu. On peut ainsi reconstituer le message d'origine.
- Si au moins deux bits du message arrivent corrompus, on n'a pas de garantie sur la détection d'erreur (et encore moins sur l'identification des positions corrompues).

Il existe beaucoup d'autres codes correcteurs, dont certains sont plus puissants (par exemple : permettent de détecter voire corriger plusieurs erreurs), mais nous n'en verrons pas d'autres dans ce cours.

Attention ! Dans les exercices de TD et d'examens, prenez bien soin de lire la consigne correctement pour savoir si l'on vous demande **d'encoder** un message grâce à un code de Hamming **ou bien de le décoder** (et corrigeant l'éventuel bit corrompu). *Il s'agit malheureusement d'une erreur fréquente à l'examen.*

Remarque 13 *L'ajout d'information redondante pour détecter une erreur est en fait une pratique très courante, et pas spécifiquement réservée au binaire. Cette information redondante est parfois appelée* clé de contrôle*. Dans les applications de la vie courante, son but est généralement de détecter une faute de frappe. Par exemple, on retrouve une clé de contrôle :*

- *Dans les numéros de Sécurité Sociale français : les deux derniers chiffres forment une clé de contrôle à deux chiffres décimaux, obtenue grâce à une méthode appelée complément à 97 modulo 97*[1].
- *Dans les numéros de carte bancaire : le dernier chiffre est une clé de contrôle à un chiffre décimal, obtenue à partir d'une méthode appelée algorithme de Luhn*[2].

1. Pour en savoir plus : `https://fr.wikipedia.org/wiki/Numéro_de_sécurité_sociale_en_France`
2. Pour en savoir plus : `https://fr.wikipedia.org/wiki/Formule_de_Luhn`

Chapitre 4

Logique booléenne

Sommaire

Dans ce chapitre, nous allons nous concentrer sur les cas où la valeur 1 ou 0 d'un bit est utilisée pour encoder l'information *Vrai* ou *Faux*. On dit alors que la valeur du bit est un *booléen*. Ce cas-là est extrêmement important dans toute l'informatique, tant d'un point de vue de l'architecture que de la programmation, de l'algorithmique, etc...

I Algèbre de Boole

Boole a défini une algèbre qui s'applique à des fonctions logiques de variables logiques (appelées variables booléennes). Une variable booléenne ne peut prendre que deux valeurs : **Vrai** ou **Faux**, représentées respectivement par 1 et 0. Il a montré que toute fonction logique peut être réalisée à partir d'un petit nombre de fonctions logiques de base. Il a également montré que les opérations arithmétiques de base pouvaient être réalisées à l'aide d'opérations logiques de base.

Une fonction logique a un nombre fini de valeurs d'entrées (ou états d'entrées dans le cas de circuits) et donc un nombre fini de valeurs de sortie. On peut donc entièrement décrire une fonction par la liste exhaustive des valeurs de sortie associées aux valeurs d'entrée. On range ces valeurs dans un tableau que l'on appelle *table de vérité* de la fonction logique.

Exemple 22 *Soit f la fonction logique sur 2 variables définie par la table de vérité suivante :*

entrées		$f(a,b)$
a	b	
0	*0*	*0*
0	*1*	*1*
1	*0*	*0*
1	*1*	*1*

On a ainsi :
$f(Vrai, Faux) = f(1,0) = 0 = Faux$ *ou encore* $f(Vrai, Vrai) = f(1,1) = 1 = Vrai$.

La table de vérité d'une fonction logique à n variables d'entrées aura 2^n états d'entrées. Comme pour chacun de ces états d'entrées, on peut avoir deux valeurs de sorties, cela nous donne 2^{2^n} fonctions possibles à n variables.

Exemple 23
- *pour* 1 *variable,* 4 *fonctions ;*
- *pour* 2 *variables,* 16 *fonctions ;*
- *pour* 3 *variables,* 256 *fonctions ;*
- *pour* 4 *variables,* 65 536 *fonctions ;*
- *pour* 5 *variables, ...*

I.1 Fonctions de base

D'après ce qui a été énoncé précédemment, on peut donc définir $2^{2^1} = 4$ fonctions logiques à une variable. Chacune d'elles porte un nom :

Constante Faux

Entrée	Sortie
0	0
1	0

Constante Vrai

Entrée	Sortie
0	1
1	1

Identité

Entrée	Sortie
0	0
1	1

Négation, aussi appelée **NON**

Entrée	Sortie
0	1
1	0

TABLE 4.1 – Les 4 fonctions logiques à une seule variable

La seule fonction logique à une variable non triviale (ni constante, ni l'identité) est la fonction de complémentation correspondante à l'opérateur logique *NON*, opérateur désigné par une barre horizontale au-dessus : $\overline{a}$ se lit NON a ou "a barre".

Toujours d'après la même formule, on voit qu'il est possible de définir 16 fonctions logiques à 2 variables, les deux non triviales les plus importantes correspondent aux opérateurs *ET* et *OU* notés respectivement comme la multiplication arithmétique ($a.b$ ou même ab pour exprimer a ET b) et comme l'addition arithmétique ($a + b$ pour exprimer a OU b).

Leurs tables de vérité sont données ci-dessous :

entrées		ET	OU
a	b	$a.b$	$a+b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Ces trois opérateurs de base (NON, ET, OU) permettent de définir la structure de l'algèbre de Boole.

On peut alors construire des formules logiques en utilisant ces opérateurs, de la même façon que l'on construit des formules arithmétiques en mathématiques grâce l'addition, la soustraction, la multiplication et la division. Voici un exemple de formule logique à 3 variables : $f(a,b,c) = a.(b+\overline{c}) + \overline{a}.c$. Si l'on nous donne les valeurs des variables, alors on peut évaluer la valeur de sortie de la formule de manière similaire à l'arithmétique.

Exemple 24 *Si* $f(a,b,c) = a.(b+\overline{c}) + \overline{a}.c$ *et que* $a = Vrai = 1$, $b = Vrai = 1$ *et* $c = Faux = 0$, *alors*

$$\begin{array}{lll} f(1,1,0) & = 1.(1+\overline{0}) + \overline{1}.0 & \textit{en remplaçant les variables par leur valeur} \\ & = 1.(1+1) + 0.0 & \textit{car } \overline{0} = 1 \textit{ et } \overline{1} = 0 \textit{ (opérateur NON)} \\ & = 1.1 + 0 & \textit{car } 1+1 = 1 \textit{ (opérateur OU) et } 0.1 = 0 \textit{ (opérateur ET)} \\ & = 1 + 0 & \textit{car } 1.1 = 1 \textit{ (opérateur ET)} \\ & = 1 & \textit{car } 1+0 = 1 \textit{ (opérateur OU)} \end{array}$$

On a donc $f(1,1,0) = 1 = Vrai$.

Deux formules logiques différentes peuvent exprimer la même fonction logique, c'est-à-dire que les tables de vérités sont identiques.

Exemple 25 *On définit les deux formules logiques suivantes :* $F_1(a,b) = \overline{a}.b + a.\overline{b}$ *et* $F_2(a,b) = (a+b).(\overline{a}+\overline{b})$. *On souhaite savoir si elles représentent la même fonction, donc on va écrire la table de vérité de chacune et comparer si elles sont identiques. Cela signifie que, pour toutes les combinaisons d'entrée possibles (ici, 4 combinaisons possibles car il y a deux variables ayant chacune deux choix possibles), on doit trouver la valeur booléenne que nous donne la formule. Par exemple, si* $a = 0$ *et* $b = 0$, *on a*

$$\begin{array}{lll} F_1(a,b) = F_1(0,0) & = \overline{0}.0 + 0.\overline{0} & \\ & = 1.0 + 0.1 & \textit{car } \overline{0} = 1 \\ & = 0 + 0 & \textit{car } 1.0 = 0 \textit{ et } 0.1 = 0 \\ & = 0 & \end{array}$$

Dressons alors les tables de vérité :

Entrées		*Sortie*
a	*b*	$F_1(a,b)$
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Entrées		*Sortie*
a	*b*	$F_2(a,b)$
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Les deux tables de vérité sont identiques, autrement dit pour tous booléens a et b, $F_1(a,b) = F_2(a,b)$, donc les deux formules expriment la même fonction logique.

I.2 Théorèmes fondamentaux de l'algèbre de Boole

Plutôt que de devoir dresser les tables de vérité à chaque fois, les Théorèmes Fondamentaux de l'Algèbre de Boole regroupent des relations souvent utiles pour simplifier des formules ou pour vérifier que deux formules sont équivalentes (c'est-à-dire, les deux formules expriment la même fonction).

élément neutre :	$a+0=a$	$a.1=a$	
élément absorbant :	$a+1=1$	$a.0=0$	
idempotence :	$a+a=a$	$a.a=a$	
complémentation :	$a+\overline{a}=1$	$a\overline{a}=0$	
commutativité :	$a+b=b+a$	$ab=ba$	
théorème de De Morgan :	$\overline{ab}=\overline{a}+\overline{b}$	$\overline{a+b}=\overline{a}\overline{b}$	
involutivité du complément :	$\overline{\overline{a}}=a$		
distributivité :	$a+(bc)=(a+b)(a+c)$		$a(b+c)=(ab)+(ac)$
associativité :	$a+(b+c)=(a+b)+c=a+b+c$		$a(bc)=(ab)c=abc$
autres relations :	$a+(ab)=a$	$a+(\overline{a}b)=a+b$	
	$a(a+b)=a$	$(a+b)(a+\overline{b})=a$	

TABLE 4.2 – Théorèmes fondamentaux de l'algèbre de Boole

Exemple 26 *Prouvons que les formules F_1 et F_2 de l'exemple 25 sont équivalentes en utilisant les théorèmes fondamentaux de l'algèbre de Boole :*

$$
\begin{array}{lll}
F_2(a,b) & = (a+b).(\overline{a}+\overline{b}) & \\
& = (a+b).\overline{a}+(a+b).\overline{b} & \textit{par distributivité} \\
& = a.\overline{a}+b.\overline{a}+a.\overline{b}+b.\overline{b} & \textit{par distributivité} \\
& = 0+\overline{a}.b+a.\overline{b}+0 & \textit{car } a.\overline{a}=0 \textit{ par complémentation (idem } b.\overline{b}=0\textit{)} \\
& & \textit{et } b.\overline{a}=\overline{a}.b \textit{ par commutativité} \\
& = \overline{a}.b+a.\overline{b} & \textit{car 0 est l'élément neutre pour } + \\
& = F_1(a,b) &
\end{array}
$$

De plus, l'algèbre de Boole nous assure qu'il est possible d'exprimer toute fonction logique à l'aide des opérateurs NON, ET et OU, comme on le verra lorsque nous construirons des formules logiques à partir de la table de vérité d'une fonction.

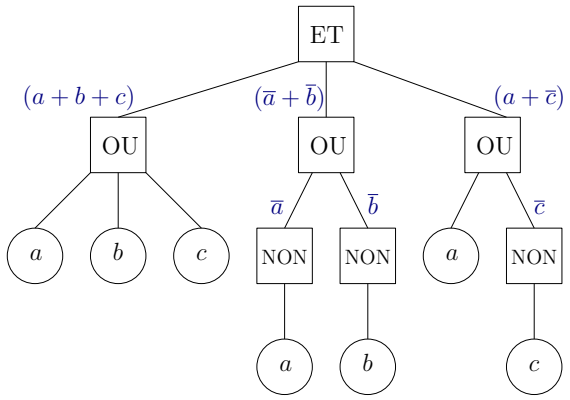
Certains formats de formules sont considérés comme plus simples ou plus importants dans certaines applications, il s'agit des deux Formes Canoniques (conjonctive ou disjonctive) et des deux Formes Normales (là encore, conjonctive ou disjonctive). Dans ces 4 formes-là, la même règle s'applique : on autorise l'opérateur NON à agir sur les variables directement (exemple : $\overline{a}, \overline{b}$), mais pas sur des formules plus complexes (exemple : $\overline{(a+b)}$ n'est pas autorisé). Le détail des autres règles n'est pas au programme de ce cours. On illustrera ici

seulement l'une de ces 4 formes, la Forme Normale Conjonctive (particulièrement utile en Théorie de la Complexité, sous-domaine de l'informatique théorique qui permet de classifier quels problèmes sont *algorithmiquement durs* ou *algorithmiquement faciles*) sur un exemple :

Exemple 27 *La formule suivante est en Forme Normale Conjonctive :*

$$f(a,b,c) = (a+b+c).(\overline{a}+\overline{b}).(a+\overline{c})$$

On peut la représenter schématiquement ainsi :



La formule est en Forme Normale Conjonctive car elle se compose d'un opérateur ET liant plusieurs opérandes (plusieurs parties, ici 3 parties : $(a+b+c)$ puis $(\overline{a}+\overline{b})$ puis $(a+\overline{c})$), et chacune de ces opérandes se compose elle-même d'un opérateur OU entre plusieurs "sous-opérandes" ; ces "sous-opérandes" étant soit une variable, soit la négation d'une variable. Cela se traduit dans la représentation schématique par le fait que l'étage du haut est un seul opérateur ET, l'étage d'en-dessous ne contient que des opérateurs OU, et l'étage d'en-dessous ne contient que des variables ou des opérateurs NON appliqués sur des variables.

La Forme Normale Disjonctive se définit similairement à la Forme Normale Conjonctive, en échangeant le rôle de l'opérateur ET et de l'opérateur OU : l'étage du haut doit être un unique opérateur OU, et l'étage d'en-dessous doit contenir uniquement des opérateurs ET.

I.3 Autres fonctions : XOR, NAND, NOR

Un autre opérateur logique à deux variables très utilisé est l'opérateur *XOR*, appelé aussi *ou exclusif*, de symbole $\oplus$. Il permet d'exprimer qu'exactement une des deux variables est à Vrai, et l'autre est à Faux. Sa table de vérité est la suivante :

entrées		*XOR*
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

On remarque que la table de vérité est la même que celles obtenues dans l'exemple 25, on peut donc en déduire que l'opérateur XOR s'exprime à l'aide des formules F_1 et F_2 étudiées dans l'exemple :

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

Il y a deux autres opérateurs importants du point de vue théorique dans l'algèbre de Boole : les opérateurs *NAND* (non et) et *NOR* (non ou). En effet, ces fonctions forment chacune un ensemble complet minimal, c'est à dire qu'elles peuvent exprimer tous les autres opérateurs. L'ensemble utilisé en pratique, {ET,OU,NON} permet bien d'exprimer tous les opérateurs mais il n'est pas minimal. En effet, on peut réaliser la fonction ET avec des OU et des NON et la fonction OU avec des ET et des NON. Le tableau ci-dessous donne la table de vérité des opérateurs NAND et NOR.

Entrées		a NAND b	a NOR b
a	b	$\overline{a.b}$	$\overline{a+b}$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

I.4 Formule à partir de la table de vérité

On considère la fonction logique décrite par la table de vérité suivante :

a	b	c	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Pour obtenir une formule décrivant la fonction logique S, on repère chaque ligne où la sortie vaut 1, et on lui associe la formule correspondant à la combinaison des variables d'entrée sur cette ligne. Par exemple, la dernière ligne du tableau correspond à la formule abc car les trois variables sont à 1. L'avant-dernière ligne correspond à la formule $ab\overline{c}$ (car $a = b = 1$ mais $c = 0$), etc... On obtient ainsi : $S = \overline{a}\overline{b}c + \overline{a}bc + ab\overline{c} + abc$. On essaye ensuite de simplifier la formule en utilisant les théorèmes fondamentaux de l'algèbre de Boole. Cependant, cette tâche n'est pas toujours facile, c'est pourquoi nous allons voir une méthode plus efficace pour obtenir une formule simplifiée.

I.5 Tableaux de Karnaugh

Pour obtenir une expression simplifiée d'une fonction logique, on utilise un *tableau de Karnaugh*. Pour simplifier, on ne traitera que les cas où le nombre de variables est inférieur ou égal à 4.

Principe Au lieu d'écrire la table de vérité avec une ligne par combinaison d'entrées, on va écrire la table de vérité dans un tableau à 2 dimensions, où chaque case correspond à une combinaison d'entrée. Par exemple, dans le tableau ci-dessous, on lit les valeurs de a et b sur la ligne, et les valeurs de c et d sur la colonne. La case verte correspond à la

combinaison $a = 0, b = 1, c = 0, d = 0$ et la case bleue correspond à la combinaison $a = 0, b = 0, c = 1, d = 1$.

Table de vérité

a	b	c	d	S
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Tableau de Karnaugh

a, b \ c, d	00	01	11	10
00	0	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

Attention : il faut que le passage d'une ligne à une autre (ou d'une colonne à une autre) n'entraîne le changement de valeur que **d'une seule** variable. Par conséquent, on ne pourra jamais mettre les valeurs 10 et 01 consécutivement (ni 00 suivi de 11).

Ensuite, on recouvre les 1 avec des rectangles les plus grands possibles respectant les conditions suivantes :

- Chaque 1 doit être dans au moins un rectangle (il peut être dans plusieurs).
- Aucun zéro ne doit être dans un rectangle.
- Le nombre de cases dans un rectangle doit être une puissance de 2.

En reprenant le tableau ci-dessus, on recouvre les 1 avec deux rectangles :

a, b \ c, d	00	01	11	10
00	0	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

Pour chaque rectangle, on identifie les valeurs de variables qui restent fixes en toutes cases du rectangle et on les combine avec un ET. Pour le rectangle orange, on a toujours $b = 1$ et $c = 0$, mais les valeurs de a et d varient. La formule correspondant au rectangle orange est donc $b\bar{c}$. Pour le rectangle violet, on a toujours $d = 1$, mais les valeurs des trois autres variables varient. La formule correspond au rectangle violet est donc d. On remarque que, plus le rectangle est grand, plus la formule associée au rectangle est petite. Pour finir, on combine les formules des rectangles en les séparant par un OU : on obtient donc la formule $b\bar{c} + d$.

Rectangles étendus Les rectangles choisis pour couvrir les 1 peuvent être des rectangles *étendus* : la dernière ligne est considérée comme adjacente à la première ligne, et la

première colonne est adjacente à la dernière colonne. En particulier, les rectangles de taille 4 dessinés ci-dessous sont tous valides :

a,b \ c,d	00	01	11	10
00	0	1	1	0
01	1	0	0	0
11	1	0	0	0
10	0	1	1	0

$\overline{b}d$

a,b \ c,d	00	01	11	10
00	0	0	1	0
01	1	0	1	1
11	1	0	0	1
10	0	1	0	0

$b\overline{d}$

a,b \ c,d	00	01	11	10
00	1	0	1	1
01	0	0	1	0
11	1	0	0	0
10	1	1	0	1

$\overline{b}.\overline{d}$

Méthode de la somme, méthode du produit La méthode expliquée ci-dessus, consistant à recouvrir les 1, est la plus courante. Elle est appelée *méthode de la somme* car elle donne toujours la formule finale sous la forme d'une somme de produits. Il existe une deuxième méthode, appelée *méthode du produit*, qui donne la formule finale sous la forme d'un produit de sommes. Elle consiste à calculer une formule simplifiée pour $\overline{S}$ avec la méthode de la somme en couvrant les 0 au lieu des 1, puis à appliquer les règles de De Morgan. Cette méthode est donc appliquée lorsque les 0 se couvrent avec moins de rectangles que les 1.

Exemple 28 *On considère le tableau de Karnaugh suivant, dans lequel on recouvre les 0 au lieu des 1 :*

a,b \ c,d	00	01	11	10
00	1	1	1	1
01	0	1	1	0
11	0	1	0	0
10	1	1	0	0

On obtient donc : $\overline{S} = b\overline{d} + ac$. *Or* $S = \overline{\overline{S}}$ *donc en appliquant les règles de De Morgan :* $S = \overline{b\overline{d} + ac} = (\overline{b\overline{d}}).(\overline{ac}) = (\overline{b} + \overline{\overline{d}}).(\overline{a} + \overline{c}) = (\overline{b} + d).(\overline{a} + \overline{c})$. *Pour comparaison, on aurait obtenu avec la méthode de la somme la formule suivante :* $S = \overline{a}\overline{b} + \overline{a}d + \overline{b}\overline{c} + \overline{c}d$.

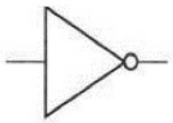
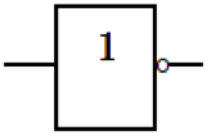
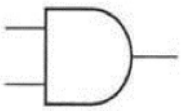
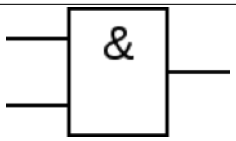
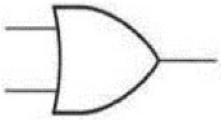
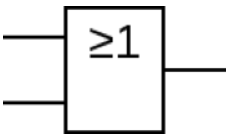
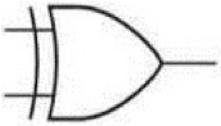
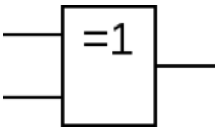
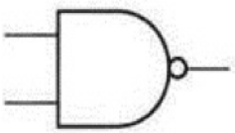
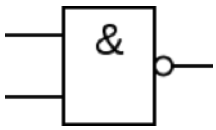
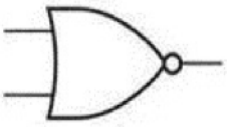
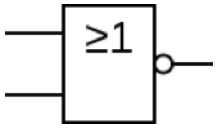
II Circuits logiques

Le comportement des opérateurs booléens vus précédemment est réalisé en pratique par des circuits électroniques. En combinant plusieurs circuits, on obtient des composants électroniques de plus en plus complexes, jusqu'à l'obtention d'un processeur. Les circuits logiques constituent la couche la plus basse de l'architecture d'un ordinateur. Nous ne traiterons dans le cadre de ce cours que des circuits logiques simples.

II.1 Portes logiques

Une porte logique ou *gate* est un circuit électronique qui combine des transistors pour réaliser une des fonctions booléennes de base, à une ou deux variables booléennes (NON, ET,

OU, ...). Elle a un comportement binaire : soit elle laisse passer le courant (état ouvert, sortie valant le booléen 1), soit elle le bloque (état fermé, sortie valant 0). Il s'agit du composant élémentaire des circuits logiques, que l'on verra comme une "boîte noire" qui possède (une ou) deux entrée(s) et une sortie, et dont on connaît le comportement extérieur sans connaître le fonctionnement intérieur. Voici le schéma utilisé pour chaque type de porte :

Type	Nom anglais	Symbole américain	Symbole français	Table de vérité
NON	**NOT**			A || Sortie 0 || 1 1 || 0
ET	**AND**			A B || Sortie 0 0 || 0 0 1 || 0 1 0 || 0 1 1 || 1
OU	**OR**			A B || Sortie 0 0 || 0 0 1 || 1 1 0 || 1 1 1 || 1
OU-exclusif	**XOR**			A B || Sortie 0 0 || 0 0 1 || 1 1 0 || 1 1 1 || 0
NON-ET	**NAND**			A B || Sortie 0 0 || 1 0 1 || 1 1 0 || 1 1 1 || 0
NON-OU	**NOR**			A B || Sortie 0 0 || 1 0 1 || 0 1 0 || 0 1 1 || 0

II.2 Circuits logiques combinatoires

On crée un circuit logique combinatoire en combinant des portes logiques et des fils, comme sur l'exemple de la Figure 4.1.

Les fils transportent l'information 0 ou 1 (1 correspondant à VRAI, le courant passe ; et 0 correspondant à FAUX, le courant ne passe pas). Sur le dessin, comme souvent par convention, les entrées sont les fils les plus à gauche, et la ou les sorties sont les fils les plus à droite. Un fil peut relier la sortie d'une porte logique à l'entrée d'une autre porte logique.

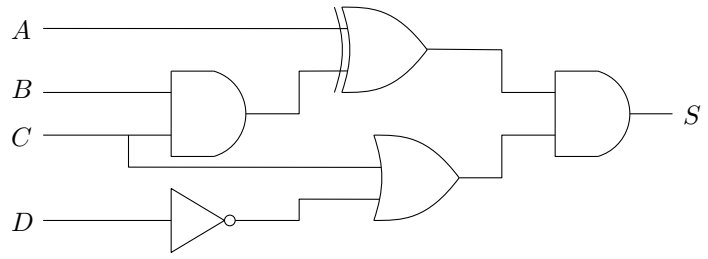


FIGURE 4.1 – Un circuit combinatoire ayant 4 entrées A, B, C, D et une sortie S. La formule correspondante est : $S = (A \oplus BC).(C + \overline{D})$.

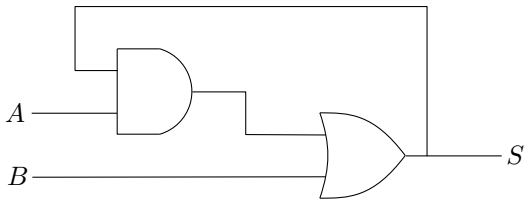
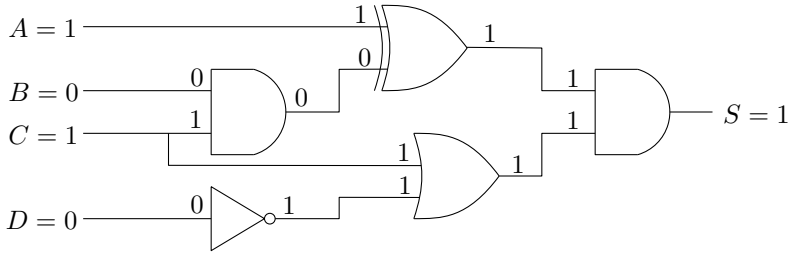


FIGURE 4.2 – Un circuit non-combinatoire car il contient une boucle.

La valeur de l'entrée de la seconde porte dépend donc de la valeur de sortie de la première porte. Un fil peut également se dédoubler pour qu'une même valeur soit utilisée plusieurs fois comme entrée sur différentes portes (ou sur la même porte, même si ce n'est généralement pas très utile), comme le fil de l'entrée C sur la Figure 4.1. Pour qu'un circuit soit appelé *combinatoire*, il ne faut pas qu'il comporte de boucle. Voir Figure 4.2 pour un exemple de circuit logique non-combinatoire.

Grâce à cette condition, on est sûrs de pouvoir calculer la valeur de la (ou des) sortie(s) si l'on nous donne la valeur de toutes les entrées.

Exemple 29 *En reprenant le circuit de la Figure 4.1 et en choisissant la valeur des entrées comme indiqué à l'extrémité gauche de la figure ci-dessous, on propage les valeurs de gauche à droite jusqu'à la sortie S. On obtient $S = 1$:*



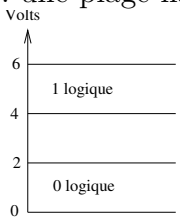
Remarque 14 • *Les portes NAND et NOR sont particulièrement importantes. En effet, leur implémentation électronique est en fait plus simple que celle des portes ET et OU, bien qu'elles paraissent conceptuellement plus compliquées. De plus, comme elles forment chacune un ensemble complet minimal, n'importe quel circuit logique combinatoire peut être réalisé en n'utilisant que des portes NAND ou que des portes NOR.*

- *Les simplifications d'écriture de fonctions booléennes vues en section précédente prennent maintenant tout leur sens : écrire la fonction sous forme plus simple permet d'avoir un circuit moins complexe à implémenter, utilisant généralement moins de composants, donc diminuant le coût du matériel.*

III Informations supplémentaires pour votre culture

Le contenu de cette section n'est pas exigible dans le cadre de ce cours.

Pour représenter le passage de "0" ou de "1" sur les fils électriques, on découpe en fait la tension en deux plages : une plage haute pour le 1, et une plage basse pour le 0.



Remarque 15 *Ce découpage des tensions de sortie des circuits en deux plages représentant le 0 et 1 est tout à fait arbitraire et on a déjà envisagé d'autres découpages en 3 , 4 et même 10 plages de potentiels. Avec 10 plages représentant les chiffres de 0 à 9, on obtient une machine travaillant directement en décimal ! Cependant, ces architectures restent du domaine de la recherche, en raison :*

1. *de leur coût : il faut développer tous les circuits composant la machine ...*
2. *de leur fiabilité plus faible : les 10 plages étant plus "minces " et plus proches les unes des autres, des parasites peuvent plus facilement provoquer des erreurs.*

III.1 Implémentation électronique d'une porte NON

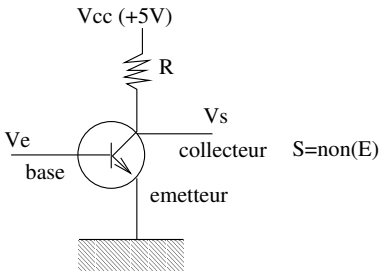


FIGURE 4.3 – Transistor

Lorsque V_e est inférieur à la valeur critique du transistor, celui-ci est bloqué et est équivalent à un interrupteur ouvert : V_s est donc proche de V_{cc}. Lorsque V_e est supérieur à la

tension critique, le transistor bascule (la porte se ferme) et équivaut à une résistance quasi-nulle : V_s est donc proche de 0 .

Remarquons que l'inversion de l'entrée sur la sortie n'est pas instantanée : selon les technologies d'intégration, elle tourne autour des quelques nano-secondes ($10^{-9}s$).

III.2 Implémentation électronique des portes NAND et NOR

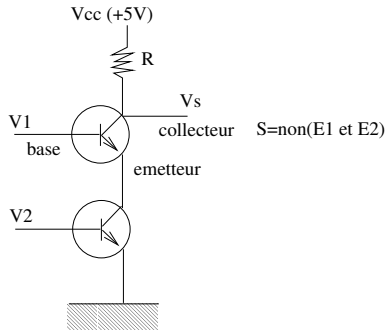


FIGURE 4.4 – La porte logique **NON ET** (NAND)

Si $V_1 < V_{critique}$ ou $V_2 < V_{critique}$ alors $V_s = V_{cc}$ sinon $V_s = 0$.

En plaçant deux transistors en série, on obtient une porte NON-OU :

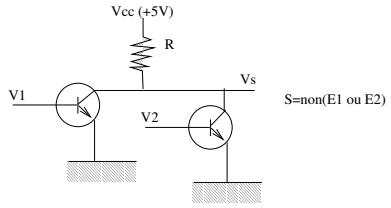


FIGURE 4.5 – La porte logique **NON OU** (NOR)

III.3 Composants de base

Les circuits intégrés (puce, chip) permettent d'utiliser plusieurs fonctions logiques sur une plaquette de silicium (5mm*5mm). Différents procédés technologiques permettent cette intégration à plus ou moins forte densité. La densité est calculée en nombre de portes ou de transistors par mm^2 ou par circuit. Ensuite, chaque plaquette est encapsulée dans un boîtier (noir) rectangulaire en plastique ou en céramique, d'où sortent des broches (pattes) de connexion.

Il existe 4 classes de produits classées selon leur densité d'intégration :

- **SSI** (Small Scale Integration) 1 à 10 portes.
- **MSI** (Medium Scale Integration) 10 à 100 portes.
- **LSI** (Large Scale Integration) 100 à 100000 portes.
- **VLSI** (Very Large Scale Integration) plus de 100000 portes.