Introduction à l'informatique
Partie 1 – Fonctionnement des ordinateurs

Yann Thorimbert

Préface

Ce document a été rédigé à partir de 2024 pour servir de support de cours aux étudiants du cours d'introduction à l'informatique de l'Université de Genève. Il est fait pour aller de pair des exercices en séances de travaux pratiques, bien que des exercices soient également donnés ici.

Cette première partie du cours est consacrée au fonctionnement des ordinateurs. Les aspects de programmation et d'algorithmique sont abordés dans une seconde partie, spécifique aux différents cursus suivis par les étudiants.

Le contenu général du cours ainsi que sa structure sont largement basés sur celui de J. Lätt [2]. Les ressources de référence utilisées sont *The Elements of Computing Systems : Building a Modern Computer from First Principles* [3] en général et, dans une moindre mesure, *Computer Organization and Design MIPS Edition* [4], en particulier pour le chapitre dévolu à l'architecture des ordinateurs. Les sources plus spécifiques sont indiquées au fil du texte.

Le lecteur est invité à signaler par e-mail toute erreur ou question afin de participer à l'amélioration du polycopié : `yann.thorimbert@unige.ch`.

Enfin, signalons que pour des raisons de clarté, certains exemples de code sont donnés au fil du texte. Ces exemples remplissent le rôle habituellement dévolu au pseudocode ; dans ce document, le Python endosse ce rôle, tandis que des codes en C et en Python sont parfois donnés en annexe. Il est néanmoins important de noter que l'essentiel de ce que le lecteur doit retenir, au travers de ces exemples, ne dépend pas du langage utilisé. Par ailleurs, les lecteurs trop peu familiers avec la programmation peuvent simplement ignorer ces exemples de code, tout en gardant en tête que ce cours est surtout destiné à des étudiants exercant le programmation par ailleurs, dans des séances d'exercices à part.

Table des matières

Chapitre 1

Les origines de l'informatique

Dans ce chapitre, nous effectuons un tour d'horizon de l'histoire de l'informatique. C'est une bonne occasion d'aborder des notions de base qui seront utiles pour la suite du cours, où nous les traiterons plus en profondeur. En effet, pour comprendre les chapitres plus techniques qui suivent, il est capital d'avoir une idée générale du paysage au sein duquel s'inscrivent les concepts abordés tout au long du cours. En effet, ces sujets étant interdépendants, il n'est pas aisé de les aborder les uns après les autres ; nous commençons donc par en donner ici une vue d'ensemble.

1.1 Amplifier et relayer un signal

Dans la perspective historique que nous adoptons dans ce chapitre, choisissons pour point de départ l'un des plus gros problèmes auxquels sont confrontés les réseaux de télécommunications du dix-neuvième siècle et, en particulier, le télégraphe électrique (souvent associé au code Morse), qui est le principal moyen de télécommunication sur de longues distances dans la seconde moitié du dix-neuvième siècle. Ce problème est celui de l'affaiblissement du signal.

En effet, un signal électrique transitant au sein d'un câble [1] perd en intensité à mesure qu'il parcourt de la distance. Pour résoudre ce problème, une solution naturelle est de récupérer à intervalles réguliers le signal avant qu'il ne soit trop faible et, de là, le retransmettre avec une intensité renouvellée. C'est là l'origine des **relais électromécaniques**, qui sont des dispositifs qui répètent et amplifient les signaux électriques de façon automatique grâce à l'exploitation du phénomène d'induction électromagnétique, comme suggéré sur les figures 1.1 et 1.2.

Ainsi, on peut voir un relais électromécanique comme un interrupteur (ou commutateur) automatique : quand un signal arrive, même faible, il déclenche un certain mé-

1. En fait, le même problème se pose pour tout signal qui doit être transmis sur une certaine distance, que ce soit un signal électrique, optique, acoustique, etc ; de la même façon, ce phénomène intervient quel que soit le milieu au sein duquel le signal est transmis, bien que ce dernier ait évidemment un fort impact sur le taux d'affaiblissement.

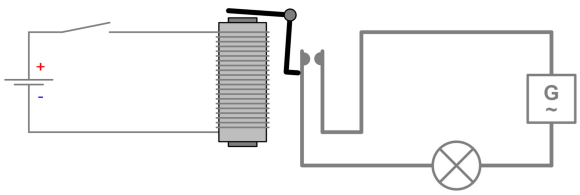


FIGURE 1.1 – Schéma d'un relais électromécanique lorsqu'aucun courant ne parvient à la bobine. Crédit : Cloullin

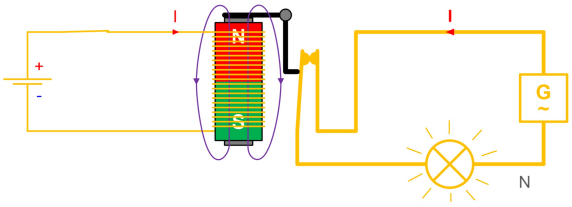


FIGURE 1.2 – Schéma d'un relais électromécanique lorsqu'un courant parvient à la bobine. Crédit : Cloullin

canisme. En l'occurrence, ledit mécanisme répète le signal d'origine avec une nouvelle intensité, au sein d'un circuit électrique voisin. Il faut bien comprendre que l'automatisation du processus est fondamentale : en effet, non seulement les signaux dont il est question sont très nombreux et impliqueraient donc une grande quantité d'être humains pour être traités manuellement, mais surtout leur fréquence (taux de variation) est telle que l'humain n'est, dans bien des cas, pas capable de les relayer de façon viable.

Après leur utilisation au sein des réseaux télégraphiques, les relais électromécaniques ont également été utilisés dans les réseaux téléphoniques. Cependant, dans ce dernier cas, leur efficacité laissait à désirer ; en effet, la voix humaine, au travers du large specte de fréquence qu'elle peut engendrer, est un signal plus délicat à transmettre que les signaux souvent binaires du télégraphe. C'est ainsi que, quelques décennies plus tard, les tubes à vide ont pris le relais.

1.2 Les tubes à vide

Un tube à vide permet de faire passer un flux d'électrons d'un point à un autre de façon contrôlée. C'est en quelque sorte un robinet à électrons[2]. La source (cathode) émet des électrons ; la cible (anode) récolte les électrons ; enfin une grille, située entre les deux, permet de contrôler le flux via la charge électrique qu'elle contient. Plus la grille est négative relativement à la cathode, plus il est difficile pour les électrons qui veulent traverser le tube de terminer leur parcours.

Initialement, les tubes à vide sont apparus grâce à une observation accidentelle concernant les ampoules à incandescence. Au sein de ces dernières, le filament était chauffé à 2000 °C environ afin de produire de la lumière ; or, à cette température, si de l'oxygène

2. D'ailleurs, en anglais, on peut également les nommer « valve ».

est présent, une réaction de combustion peut avoir lieu. Par conséquent, les concepteurs des ampoules retiraient l'air au sein des bulbes (et donc l'oxygène avec lui). Dès lors, les ampoules remplissaient bien leur rôle premier, qui était d'émettre de la lumière, mais un effet secondaire accompagnait la mise sous vide du bulbe. On remarqua en effet qu'avec le temps, le verre des ampoules avait tendance à subir une décoloration asymétrique ; il semblerait que ce soit entre autres ainsi que l'on comprit qu'un flux d'électrons était émis par le filament, parcouru d'un courant électrique continu, et que ce flux était dirigé dans un sens bien précis en raison de la différence de potentiel entre les deux extrémités du filament [3]. En raison du vide ambiant au sein du bulbe, rien ne stoppait le « vol » des électrons accidentellement émis par le filament jusqu'à ce qu'ils frappent le verre du bulbe, finissant par le décolorer.

À partir de cette observation, on construisit des sortes d'« ampoules à électrons » composées de la cathode chaude et de l'anode froide, ainsi que d'un simple vide les séparant. Avec un tel dispositif, si flux d'électron il y a, celui-ci va nécessairement de la cathode à l'anode, et jamais en sens inverse, puisque cette dernière n'éjecte pas d'électrons. Ainsi, ce dispositif peut convertir du courant alternatif (qui change périodiquement de sens) en un courant qui possède toujours le même sens. Ces « routes à sens unique pour électrons », comme elles ont été baptisées par leur inventeur John Fleming en 1904, portent le nom de **diodes** [4].

Comme on va le voir, les diodes ont joué un rôle fondamental dans l'histoire de l'informatique et, par conséquent, dans l'histoire récente de l'humanité. La figure 1.3 montre le symbole de la diode dans les circuits électriques et électroniques [5].

FIGURE 1.3 – Symbole de la diode, composant fondamental des circuits électroniques.

1.2.1 Première implication : amplification du signal

Dans un premier temps, les diodes ont joué un rôle important dans les communications téléphoniques. En effet, jusqu'ici, les relais éléctromécaniques étaient utilisés pour les télégrammes, comme discuté plus haut. Si cette méthode fonctionnait bien pour la nature binaire du morse (par exemple), elle était mal adaptée à la propagation des signaux vocaux, beaucoup plus nuancés et nécessitant une fréquence de commutation plus rapide, ainsi que des commutateurs plus solides. En 1906, on eut l'idée d'ajouter la grille chargée,

3. Bien que la décoloration elle-même soit imputables à différentes phénomènes physiques.

4. Le nom de « diode » (« deux chemins » en grec ancien) se réfère surtout au fait que c'est un dispositif composé de deux électrodes (la cathode et l'anode). Les diodes de tubes à vide ne sont pas les premières diodes électriques, mais leur importance est capitale car elle constituent la base qui servira à des améliorations significatives discutées plus bas.

5. Notons au passage qu'il est d'usage de qualifier d'« électrique » les dispositifs ou circuits dont la fonction de base est de transmettre de l'électricité ou de l'énergie, tandis qu'il est commun de qualifier d'« électronique » les dispositifs ou circuits dont la fonction de base est de manipuler un signal, une information. On peut trouver les diodes au sein des deux types de circuits, en l'occurrence.

dont nous avons parlé plus haut, entre la cathode et l'anode des diodes, le tout étant baptisé **triode**. Avec ce nouveau dispositif, un petit changement de charge sur la grille pouvait faire « levier » et provoquer un grand changement de courant entre la cathode et l'anode[6]. C'est ainsi que l'on a pu amplifier et reproduire les signaux électriques, et donc les signaux vocaux au travers des lignes téléphoniques. Le symbole de la triode est illustré sur la figure 1.4.

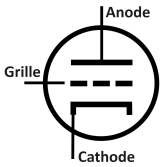


FIGURE 1.4 – Symbole de la triode au sein des circuits électroniques.

1.2.2 Seconde implication : implémentation de la logique binaire

Si la conséquence immédiate de l'invention des diodes fut de nature technologique avec l'amélioration des communications téléphoniques et télégraphiques, une autre implication fut de nature plus conceptuelle. Elle a pour origine l'algèbre de Boole (ou logique binaire), un domaine des mathématiques qui permet d'exprimer des propositions de logique, et donc une forme de raisonnement, sous forme algébrique. Pour ce faire, on utilise fréquemment des **tables de vérité**, qui servent à résumer un raisonnement de façon exhaustive, et dont nous reparlerons dans le chapitre 4. En algèbre de Boole, on choisit de travailler avec deux valeurs, arbitrairement dénotées 0 et 1, ou *faux* et *vrai*. Une variable qui ne peut prendre que deux valeurs possibles est qualifée de « binaire ». La table 1.1 ci-dessous illustre un exemple de table de vérité sous-jacente à un type d'excès de vitesse en Suisse.

TABLE 1.1 – Exemple de table de vérité pour deux inputs binaires, l'un à propos de la vitesse d'un véhicule, l'autre à propos du lieu où il roule. La valeur de l'output (colonne *Est en infraction*) est 1 si le véhicule est en infraction, 0 sinon. La situation est simplifiée par rapport au véritable code de la route.

Roule à plus de 50 km/h	Est hors localité	Est en infraction
0	0	0
0	1	0
1	0	1
1	1	0

Claude Shannon[7], qui s'intéressait à l'algèbre de Boole vers 1938, eut l'idée d'une **équivalence entre table de vérité et circuits utilisant des commutateurs**. Cette

6. Un peu comme un petit changement sur un barrage hydraulique (ouverture d'une vanne coûtant peu d'efforts) peut provoquer un grand changement (fleuve entier déversé). Il est commun de trouver des analogies entre circuits électriques et circuits hydrauliques dans la littérature.

7. Aussi connu pour de nombreux autres travaux, notamment en théorie de l'information, et à l'origine (avec Boltzmann) de la formalisation du concept d'entropie, utilisé dans de nombreux domaines scientifiques.

idée est fondamentale pour la suite de l'histoire de l'informatique, car elle permet de passer du monde des concepts mathématiques à celui, concret, du matériel technologique. Autrement dit : des commutateurs (tels que les triodes par exemple) permettent d'implémenter dans le monde physique n'importe quel raisonnement qui peut être exprimé dans une table de vérité. Dès lors, on peut déléguer à des machines des tâches mathématiques complexes. C'est le point de départ de l'informatique électronique. Nous discuterons plus loin de la façon dont les tubes à vides peuvent être utilisés pour réaliser une table de vérité en pratique.

Par ailleurs, les tables de vérité peuvent être combinées avec des algorithmes, c'est-à-dire des séquences d'instructions qui permettent de résoudre un problème. Il est important de comprendre que les ordinateurs sont utiles uniquement pour résoudre des problèmes qui peuvent être décrits de façon algorithmique. Autrement dit, on peut programmer un ordinateur de façon à ce qu'il résolve un type de problème sans même posséder à l'avance les données exactes sur lesquelles porte le problème, mais uniquement la façon de traiter ces données. Pour reprendre l'exemple de l'excès de vitesse, on peut imaginer un radar routier relié à un ordinateur qui aurait pour instruction d'appliquer la table de vérité 1.1 à chaque véhicule qui passe devant lui. L'algorithme ressemblerait alors à la recette suivante :

1. Mesurer la vitesse du véhicule.
2. Appliquer la table de vérité pour déterminer si le véhicule est en infraction.
3. Si le véhicule est en infraction, enregistrer la plaque dans la base de données.
4. Revenir au point 1 pour le véhicule suivant.

Pour pouvoir implémenter un algorithme, cependant, un ordinateur a besoin d'autres composants que de commutateurs simples. Notamment, il a besoin de pouvoir effectuer des calculs, ainsi que d'une mémoire indépendante des données concernées par la table de vérité, afin de se souvenir des instructions à exécuter. Cela dit, chacun de ces composants peut être réalisé grâce à une combinaison de commutateurs électroniques. Nous traiterons plus loin de ce sujet, dans le chapitre 5.

1.3 Le terme « ordinateur »

Comment définir ce qu'est un ordinateur ? Pourquoi une calculatrice de poche n'est-elle pas nécessairement un ordinateur, pas plus qu'un boulier ? Le terme anglais pour « ordinateur » est « *computer* », qui signifie à la fois « calculateur » et « ordinateur ». Jusqu'en 1955, le terme français était également calculateur, ce qui était ambigu et restrictif, au vu des capacités de ces nouvelles machines à aller bien au-delà du simple calcul : en effet, elles pouvaient être programmées pour appliquer des algorithmes sur un ensemble de données, ce qui est différent d'un simple traitement arithmétique de nombres. Pour palier à ce problème de vocabulaire, la filiale française d'IBM a proposé le néologisme « ordinateur » afin de refléter le fait que ces machines ordonnent l'information. Par ailleurs, le terme « informatik », raccourci pour « traitement automatique de l'information », fait son apparition en 1957 en Allemagne, puis en 1962 en France.

Nous dirons d'une machine que c'est un ordinateur si elle peut être programmée pour réaliser tout algorithme, à supposer qu'elle ait assez de temps et de mémoire pour traiter

les données sur lesquelles porte l'algorithme.

Nous dirons d'une machine que c'est un ordinateur électronique si elle utilise des composants électroniques pour appliquer l'algorithme en question. Par habitude, nous omettrons le qualificatif « électronique » dans la suite de ce document lorsqu'il s'applique à « ordinateur », sauf mention contraire.

Ainsi, de nombreuses machines qui ont vu le jour vers la moitié du vingtième siècle sont proches d'être des ordinateurs au sens où nous venons de l'entendre. Nous allons ici nous intéresser à quelques unes d'entre elles.

1.4 Le premier ordinateur complet

ENIAC[8], construite en 1946. Cette machine, qui pesait 30 tonnes et occupait 140 mètres carrés, a initialement été construite pour des applications militaires. Elle était composée de dizaines de milliers de tubes à vide et permettait en pratique d'effectuer quelques milliers d'opérations décimales par seconde[9]. Cette capacité de calcul a contribué à l'élaboration de la bombe H par les USA. C'était une machine extrêmement coûteuse, qui nécessitait une équipe de plusieurs personnes pour fonctionner et ne restait en état de marche que quelques jours consécutifs avant de subir une ou de nécessiter des modifications. La figure 1.5 ci-dessous illustre une partie de l'ordinateur.

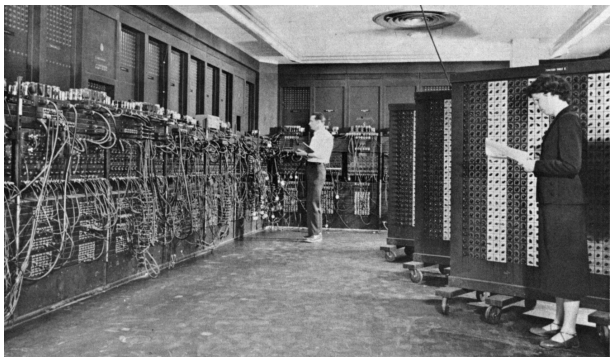


FIGURE 1.5 – Une programmeuse et un programmeur au travail sur ENIAC.

L'EDVAC (1950), sera le successeur de l'ENIAC. Tout comme ce dernier, l'EDVAC est programmable, mais il est également doté d'une mémoire indépendante des données, ce qui permet de stocker les instructions à exécuter, plutôt que de programmer l'ordinateur en recâblant les circuits électroniques, comme c'était le cas sur ENIAC. C'est le début de l'architecture de von Neumann, qui est encore utilisée aujourd'hui, et dont nous

8. Acronyme de Electronic Numerical Integrator And Computer est généralement la première machine électronique à être considérée comme un ordinateur programmable capable en théorie d'appliquer n'importe quel algorithme, Il s'agissait initialement d'une machine aidant à intégrer les équations du mouvement de problèmes de balistique.

9. Cela dépend de bien des facteurs, à commencer par le type précis d'opération. Par ailleurs, en théorie, ENIAC pouvait effectuer de l'ordre de 100'000 additions par seconde, grâce à la parallélisation des calculs permise par son architecture. Cependant, pour des raisons de manque de mémoire principalement, cette limite théorique était quasiment impossible à atteindre.

reparlerons dans le chapitre 5. Par ailleurs, contrairement à son prédecesseur, l'EDVAC travaille avec une représentation binaire des données (cf. chapitre 2).

Comme dit plus haut, de nombreuses autres machines dignes d'intérêt ont été conçues durant la même période, signe que le concept technologique de l'ordinateur était dans l'air du temps à cette époque-là. On peut notamment mentionner, parmi les machines les plus célèbres et les plus proches de remplir nos critères de définition de l'ordinateur :

- Z3 (1941) - Conçu par Konrad Zuse à des fins commerciales [10], le Z3 était une machine programmable représentant les nombres en binaire, et presque capable d'exécuter tout algorithme. Il utilisait des relais électromécaniques (et en ce sens, il ne peut constituer un ordinateur électronique).
- ABC (1942) - L'Atanasoff–Berry Computer utilise des tubes à vide et représentait les nombres en binaire. Cependant, il ne permettait pas d'implémenter n'importe quel algorithme et n'était donc pas un ordinateur au sens où nous l'entendons, mais plutôt un calculateur au sens classique du terme, bien qu'extrêment novateur pour l'époque.
- Mark I (1943) - Le Mark I de l'université de Harvard était une machine programmable électromécanique, tout comme le Z3. Elle était réputée fiable et opérait de façon autonome, nécessitant très peu d'interventions humaines. Le programme et les données étaient séparés sur des bandes perforées distinctes, où tout algorithme pouvait être codé (bien que les possibilité de branchement conditionnel fussent limitées dans sa forme initiale), ce qui constitue également un choix important dans l'histoire des ordinateurs (cf. chapitre 5).
- Colossus (1943) - Célèbre pour son rôle – secret à l'époque – consistant à déchiffrer les messages codés de l'armée allemande (*via* le chiffrement de Lorenz notamment, utilisé par les dirigeants allemands et souvent amalgamé avec la machine Enigma, plus connue), Colossus désigne une série de machines programmables qui utilisaient des tubes à vide et une représentation binaire des nombres. La machine pouvait atteindre une fréquence de calcul de quelques milliers d'opérations par seconde, ce qui est comparable à l'ENIAC, mais ne permettait pas vraiment d'implémenter n'importe quel algorithme.

Finalement, remarquons que le concept de machine programmable n'est pas né au cours du vingtième siècle. Plus d'une centaine d'années auparavant, autour de 1820, Charles Babbage avait pour ambition de construire une machine destinée à calculer des tables astronomiques pour navigateurs qui ne soient pas entachées d'erreurs humaines, comme c'était souvent le cas à l'époque. Cependant, au cours de la conception de cette machine, Babbage eut l'idée de la rendre programmable, c'est-à-dire de lui permettre d'effectuer n'importe quel calcul générique qui puisse être appliqué aux nombres qu'on lui donnera en entrée via des cartes perforées (voir figure 1.6). Bien que le principe de la machine soit abouti vers 1834, les technologies de l'époque ne permettent pas sa construction. Il n'en reste pas moins que de nombreuses réflexions furent déjà menées au sujet de l'algorithmique et de la programmation, notamment avec la collaboration d'Ada Lovelace, qui pour

10. Contrairement à nombre de ses contemporains, Konrad Zuse ne publiait pas le fruit de ses recherches au sein de journaux scientifiques, d'où sa renommée peut-être moins prononcée que celle de ses contemporains.

cette raison est souvent considérée comme la première programmeuse de l'histoire [11].

Encore plus tôt, on peut mentionner le métier Jacquard, un métier à tisser automatique et programmable via des cartes perforées qui guident les aiguilles au bon endroit, datant de 1801. C'est en fait ce dernier qui a inspiré Babbage pour l'analytical engine. Un ancien métier Jacquard est montré sur la figure 1.7. Ce type de technologie est encore utilisé aujourd'hui dans l'industrie textile.

FIGURE 1.6 – Cartes d'instructions et de données qui auraient servi à coder l'information à donner en entrée à la machine analytique de Babbage, si celle-ci avait été terminée. Auteur : Karoly Lorentey, licence CC BY 2.0 DEED

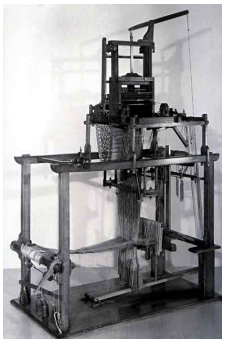


FIGURE 1.7 – Un métier Jacquard ancien, dispositif ayant utilisé des cartes perforées à un niveau industriel dès le début du dix-neuvième siècle.

Enfin, mentionnons que les premiers calculateurs mécaniques, qui ne répondent pas aux critères que nous avons établis pour qualifier une machine d'« ordinateur », remontent à bien plus tôt encore. Sans parler des ordinateurs analogiques tels que la machine d'Anticythère, dont nous parlerons plus loin, on peut remarquer que dès 1642 Blaise Pascal a conçu une machine à calculer mécanique, la Pascaline, qui permettait d'effectuer des additions et des soustractions. Quelques décennies plus tard, Leibniz espère que les scientifiques puissent prochainement être délestés de tâches liées au simple calcul arithmétique, qui devrait être automatisé puisqu'étant rébarbatif et ne nécessitant pas de réflexion profonde [12] :

> « Also the astronomers surely will not have to continue to exercise the patience which is required for computation. It is this that deters them from computing or correcting tables, from the construction of Ephemerldes, from working on hypotheses, and from discussions of observations with each other. For it is unworthy of excellent men to lose hours like slaves in the labor of calculation, which could be safely relegated to anyone else if the machine were used. What I have said about the construction and future use [of the machine], should be sufficient, and I believe will become absolutely clear to the observers [when completed]. »

11. D'ailleurs, un célèbre langage de programmation nommé Ada s'appelle ainsi en son hommage.
12. Leibniz, "Machina Arithmetica in qua non Aditio tantum Subtractio." Traduit par Mark Kormes, *A Source Book in Mathematics*, édité par David Eugene Smith, 1929, p. 181.

1.5 Les transistors

Au sein des ordinateurs, les tubes à vide ont petit à petit été remplacés par des **transistors**, qui sont plus petits, plus fiables et qui consomment moins d'énergie [13]. La figure 1.8 ci-dessous illustre la différence de taille entre ces deux types de commutateurs au moment de la popularisation des transistors.

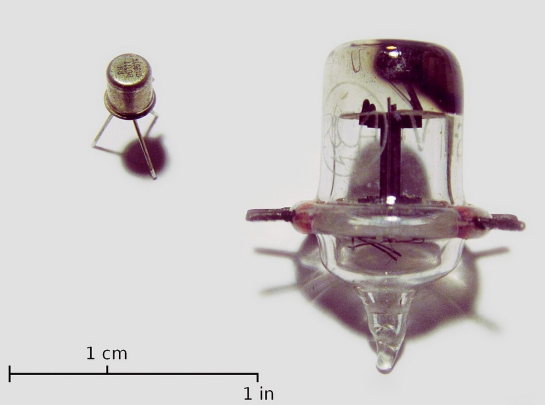


FIGURE 1.8 – Un transistor (à gauche) comparé à un tube à vide (à droite) dans les années 1950-1960.

Les transistors sont des composants électroniques qui, tout comme les triodes, permettent de contrôler le courant entre deux électrodes en fonction d'un signal de commande appliqué à une troisième borne. Cependant, au sein d'un transistor, le principe physique sous-jacent à la méthode de contrôle du flux d'électrons repose sur les propriétés des semi-conducteurs. Un semi-conducteur est un matériau qui, contrairement aux métaux, ne conduit pas le courant dans n'importe quelles conditions. Il devient conducteur en fonction du potentiel électrique qu'on lui applique. C'est le phénomène exploité au sein d'un transistor. Ainsi, d'un point de vue conceptuel, transistors et triodes s'utilisent tous deux de façon tout à fait similaire, c'est-à-dire comme des robinets à électrons ; en pratique, ils diffèrent beaucoup pour des raisons physiques et techniques. La figure 1.9 illustre le symbole d'un transistor dans les circuits électroniques.

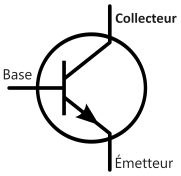


FIGURE 1.9 – Symbole du transistor au sein des circuits électroniques.

Il faut remarquer que, contrairement à l'idée répandue, ce ne sont pas les transistors qui ont conditionné la création des ordinateurs. Comme on l'a vu, la première machine a être considérée comme un ordinateur n'en utilisait pas ! Cependant, il est vrai que le degré de miniaturisation des transistors ainsi que leurs performances ont permis l'essor de l'informatique telle que nous la connaissons aujourd'hui. De nos jours, la taille typique d'un transistor pour processeurs est de l'ordre du nanomètre. Un phénomène similaire

13. Ici, pas besoin de chauffer un filament à 2000°C.

a eu lieu concernant les récepteurs radios. La radio existait bien avant l'apparition des transistors sur le marché ; mais quand ces derniers ont pu être utilisés pour remplacer les tubes à vide, la qualité des appareils a été drastiquement améliorée puisqu'ils sont devenus tout à la fois plus petits, plus solides, plus économes et, même, plus immédiats à utiliser, les filaments des tubes à vide ayant besoin d'un certain temps avant d'atteindre leur température de fonctionnement. L'impact des transistors sur la radio a été tel qu'il est même devenu courant, à une certaine époque, de désigner les récepteurs radios par le terme « transistor ».

La figure 1.10 illustre la famille des commutateurs. Ce qu'il est important de comprendre ici, c'est que la fonction remplie par ces dispositifs peut être réalisée *via* toutes sortes de principes physiques ; cependant, au moment de la rédaction de ce document, les dispositifs électroniques basés sur les semi-conducteurs sont les plus efficaces pour cette tâche en terme de rapidité, fiabilité et miniaturisation. C'est la raison pour laquelle on les retrouve dans les ordinateurs modernes. Mais si, demain, une nouvelle technologie permettait de réaliser la même fonction de commutation de façon plus efficace, alors les transistors pourraient être remplacés par ces nouveaux dispositifs, et les machines qui en résulteraient seraient tout de même des ordinateurs au sens où nous l'entendons, mais pas nécessairement des ordinateurs électroniques.

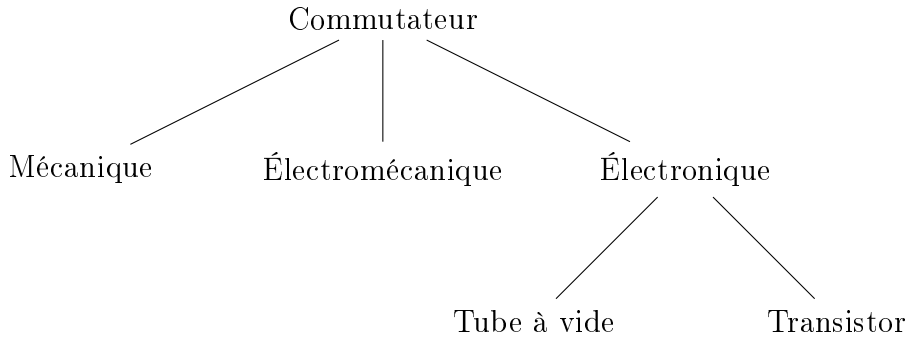


FIGURE 1.10 – Une visualisation possible de la famille des commutateurs. Seuls les types de composants discutés dans ce document sont représentés.

1.6 Évolution des ordinateurs

1.6.1 Classification des générations d'ordinateurs

Dans les années qui suivent l'apparition des transistors, la façon dont ces derniers sont combinés entre eux devient une nouvelle préoccupation d'autant plus importante que les transistors, hautement miniaturisables comparés aux tubes à vide, autorisent davantage de liberté. D'une part, les câblages utilisés dans les décennies précédentes vont vite s'avérer fastidieux en comparaison de « pistes » conductrices gravées sur une plaque isolante (concept du **circuit imprimé**, existant déjà avant les transistors) ; d'autre part, il est possible de fabriquer tous les transistors et leurs connexions « d'un seul tenant » sur un même matériau semi-conducteur, suivant une idée proposée à la fin des années 1950 par les ingénieurs et physiciens Jack Kilby et Robert Noyce. Cette technique dite des **cir-**

cuits intégrés permet de produire des combinaisons de composants à la fois plus fiables et davantage miniaturisables. Il est commun de désigner ces circuits intégrés par leur nom anglais de **chips**, ou encore **puces électroniques**.

Ces circuits vont se populariser de la fin des années 50 jusqu'au début des années 70, où une nouvelle étape majeure sera franchie. En effet, plutôt que de continuer à fabriquer des circuits intégrés spécifiquement dédiés à des tâches précises, un ingénieur d'Intel proposera d'utiliser un circuit à usage général, capable d'assumer toutes les tâches habituellement rencontrées dans l'utilisation d'un ordinateur, sur une seule puce. On nommera ce type de circuit intégré un **microprocesseur** et, par conséquent, les ordinateurs qui en font usage seront désignés apr le terme de **micro-ordinateurs**. C'était la dernière étape manquante avant que l'ordinateur ne devienne un objet de consommation courante, et non plus seulement un outil pointu dévolu à la recherche académique, industrielle ou militaire. La figure 1.11 illustre l'Altair 8800, l'un des premiers ordinateurs personnels à utiliser un microprocesseur.

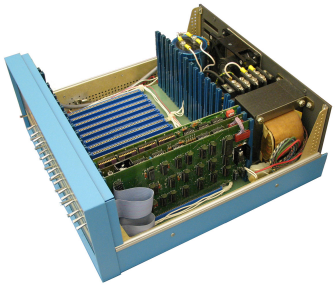


FIGURE 1.11 – Intérieur de l'Altair 8800, souvent considéré comme le premier micro-ordinateur grand public à rencontrer du succès.

Pour terminer ce chapitre, nous reprenons ici une classification arbitraire (mais populaire au sein de la littérature) des types historiques d'ordinateurs. Au sein de cette classification, chaque génération d'ordinateur représente une étape technologique considérée comme majeure.

1. Première génération (1940 - 1956) – ordinateurs à tubes à vide.
2. Deuxième génération (1956 - 1963) – ordinateurs à transistors.
3. Troisième génération (1964 - 1971) – ordinateurs à circuits intégrés.
4. Quatrième génération (1971 - aujourd'hui) – ordinateurs à microprocesseurs.

Il est important de comprendre que les deux premières générations sont caractérisées par la technologie utilisée pour les composants de base (tubes à vide ou transistors), tandis que les deux dernières générations sont relatives à la façon dont ces composants sont combinés.

Les développements ultérieurs à la quatrième génération concernent essentiellement des avancées dans les domaines suivants, qui ne sont pas considérés comme des « sauts » technologiques majeurs relatifs aux ordinateurs eux-mêmes :

- Les dispositifs périphériques (technologies des écrans, des réseaux, des mémoires persistantes, cartes graphiques...)

- Les technologies liées à l'utilisation des ordinateurs (batteries pour ordinateurs portables, pavés tactiles, ...)
- La performance des processeurs et des mémoires (mais pas leur nature fondamentale)

Parmi les exemples cités ci-dessus, la question du statut de l'internet est digne d'intérêt : elle ne concerne absolument pas le principe de fonctionnement des ordinateurs, mais uniquement une façon de faire communiquer ces derniers. Néanmoins, son impact sur les dernières décennies est conséquent. Pour clore cette section, nous traitons donc ici brièvement de l'histoire du développement des réseaux informatiques et de l'internet en particulier.

1.6.2 Réseaux informatiques

Comme nous l'avons vu en section 1.2.1, la résolution de problèmes techniques liés aux télécommunications (essentiellement télégramme et téléphone) a enfanté les technologies (tubes à vide et transistors) permettant l'avènement de l'informatique. Il est intéressant de voir que l'informatique, en retour, a permis d'engendrer un système de télécommunication particulièrement performant : l'internet.

Réseaux locaux

Dès les années 1950, le besoin s'est fait sentir de pouvoir connecter des machines proches physiquement, comme celles d'un même centre de recherche scientifique. Cela permettait par exemple d'éliminer le besoin de transporter et d'insérer manuellement des cartes perforées d'un appareil à un autre, une tâche fastidieuse et chronophage. Ces réseaux de faible étendue géographique (Local Area Networks, **LAN**) ont été les premiers à voir le jour, puis les premiers à être standardisés dans les années 1970. En plus de connecter des machines entre elles, ils permettent de partager des ressources coûteuses (imprimantes, mémoire persistante, etc.) entre les ordinateurs d'un même centre de recherche. La technologie de communication Ethernet, qui désigne à la fois des règles de communication et des technologies matérielles de transmission, est née dans les années 1970 et est encore la norme aujourd'hui pour les réseaux locaux. Avec ce système, chaque machine possède une adresse unique au sein du réseau local, nommée adresse MAC (Media Access Control).

Les premiers réseaux étendus et la commutation de paquets

Dans les années 1960, il semblait évident que la réutilisation des lignes de télécommunication du type de celles du téléphone, pour lesquelles des technologies étaient déjà existantes, s'imposait pour la communication entre machines distantes. Cela devait permettre la réalisation de réseaux plus étendus (Wide Area Networks, **WAN**). Cependant, la façon de communiquer via les lignes téléphoniques, où la voix des interlocuteurs est transmise en continu le long d'un chemin physique constant (technique nommée « commutation de

circuits »[14]), ne se prêtait pas bien à la transmission de données informatiques. En effet, là où deux interlocuteurs humains peuvent spontanément se mettre d'accord sur une façon de répéter les informations perdues lors de la transmission (par exemple en raison de perturbations externes), les ordinateurs manquaient d'un moyen de se « mettre d'accord » en cas d'imprévu. Il a donc fallu concevoir des réseaux informatiques où des données peuvent être transmises de façon discontinue. La solution qui s'est imposée consiste à découper les données en **paquets** envoyés séparément et reconstitués à l'arrivée, peu importe le chemin suivi et l'ordre d'arrivée. C'est le principe de la commutation de paquets, qui est à la base des communications informatiques les plus répandues aujourd'hui. L'ARPANET (vers 1970), un réseau destiné à l'armée américaine et reliant entre eux des ordinateurs au sein du pays, met en oeuvre la commutation de paquets et est souvent considéré comme le premier réseau de grande envergure. Selon certains auteurs[5], la menace d'un conflit nucléaire a également été un facteur déterminant dans la mise en place de ce réseau, qui devait permettre de maintenir les communications en cas de destruction d'une partie de l'infrastructure, là où la commutation de circuits (à chemin « fixe ») aurait été plus vulnérable.

Protocoles de communication et début de l'internet

Au sein d'un réseau à commutation de paquets, chaque paquet est divisé en deux parties : la première, l'en-tête, contient l'adresse du destinataire ainsi que des informations de contrôle, tandis que la seconde contient les données à transmettre. Les paquets sont envoyés de proche en proche, de routeur en routeur, jusqu'à atteindre leur destination. Les routeurs sont des machines spécialisées dans le transfert de paquets, qui se chargent de les acheminer vers leur destination.

Pour que ce système de paquets fonctionne, il faut nécessairement qu'un ensemble de règles régisse la façon dont ils sont acheminés d'un point à un autre ainsi que la façon d'interpréter les en-têtes. Cet ensemble de règles porte le nom de **protocole**. Le protocole **IP** (Internet Protocol) est à la base de l'internet et spécifie essentiellement la façon dont les paquets portent l'adresse de leur destinataire. Il est souvent associé à un autre protocole, le protocole **TCP** (Transmission Control Protocol), qui met en place des mécanismes permettant de réordonner les données reçues et de retransmettre celles qui auraient été perdues ou altérées. Les deux pris ensemble forment le protocole TCP/IP, mis en place dès les années 1980 au sein d'ARPANET[15]. Grâce à cela, on peut abstraire la technologie de transmission sous-jacente, qu'il s'agisse de lignes téléphoniques, de fibres optiques, de satellites, ou autres ; ce qui compte, c'est que les destinataires et les expéditeurs respectent

14. Puisque la communication, dans ce cas, repose sur le fait que le chemin physique (fil électrique) soit bien établi entre les interlocuteurs, alors il était nécessaire de « brancher » les interlocuteurs entre eux avant que ces derniers ne puissent se parler. Ce travail a, dans un premier temps, été délégué à des humains travaillant dans centrales d'appels. Entre les années 30 et 60, cependant, des commutateurs automatiques les ont progressivement remplacés.

15. Notons également qu'un protocole plus « léger », nommé UDP, permet de réaliser le transfert de données sans gestion des paquets perdus : cela évite d'avoir à numéroter les paquets, ainsi que d'avoir à établir des sortes d'accusés de réception (*acknowledgement*) entre les interlocuteurs : ce protocole est souvent utilisé dans les télécommunications pour lesquelles la vitesse de transmission est prioritaire sur l'intégrité des données, comme les visioconférences par exemple, où la fluidité de l'échange prime sur l'exactitude de la couleur de chaque pixel.

des règles communes.

Notons que, afin d'acheminer un message d'un point à un autre du réseau, il est nécessaire de connaître l'adresse du destinataire. Cette adresse est un nombre unique attribué à chaque machine connectée au réseau, et qui permet de l'identifier de façon univoque. Une même machine peut posséder une adresse IP privée, utilisée pour être discriminée au sein du réseau local [16] ou encore une adresse IP publique, qui permet de la distinguer au sein de l'internet. La façon dont la route d'un message est déterminée en fonction de l'IP publique se nomme le **routage** de l'information, et nous n'en traitons pas dans ce document.

La commutation de paquets et les protocoles mis en place pour garantir son bon fonctionnement ouvrent la porte à la communication entre machines issues de réseaux quelconques, et donc à la réalisation de communications dites « inter-réseaux », ou encore *inter-network* en anglais. C'est la naissance d'un internet mondial, au sein duquel peuvent alors venir se greffer tous les ordinateurs capables d'implémenter les protocoles concernés. Cet internet, cependant, est uniquement à la portée des spécialistes, et est typiquement utilisé pour la communication entre universités ou laboratoires de recherche. Par exemple, dès les années 1980, des messages électroniques (*e-mails*) sont standardisés par un protocole nommé SMTP. Cependant, jusqu'aux années 1990, l'internet reste un outil dévolu aux experts, quand bien même les ordinaturs personnels sont déjà populaires.

World Wide Web et popularisation de l'internet

La dernière décennie du siècle, cependant, voit la mise en place d'un protocole central dans le développement du World Wide Web, qui désigne un ensemble de règles qui permettent de grandement faciliter un type d'utilisation de l'internet. Il s'agit du protocole HTTP (HyperText Transfer Protocol), qui régit l'envoi de données relatives à un type de document spécifique nommé « page web », regroupé en ensembles appelés « site web » ; ces sites web peuvent alors être interprétés au sein de logiciels nommés « navigateurs web ». Bien qu'historiquement lié au Web, le protocole HTTP est aujourd'hui générique et non pas spécifique aux ressources de type page Web.

C'est l'arrivée de ce World Wide Web (communément appelé « web ») qui va populariser l'internet et en faire un outil de communication de masse. Le web est un ensemble de pages reliées entre elles. Ces pages web sont le plus souvent écrites en HTML (HyperText Markup Language), un langage de balisage qui permet de décrire la structure d'un document susceptible de contenir des hyperliens, c'est-à-dire des éléments menant vers d'autres documents hypertextes. Les navigateurs web, tels que Chrome, Firefox ou Safari, sont des programmes qui permettent d'afficher des pages web et de naviguer entre elles. Enfin, à mesure que les capacités de transmission des réseaux ainsi que les capacités de calcul des ordinateurs grandissent, des fonctionnalités annexes apparaissent sur les navigateurs, qui deviennent capables d'interpréter du code allant de pair avec les pages web,

16. Nous avons dit plus haut que chaque machine possède une adresse MAC unique au sein d'un réseau local. Cette adresse identifie la machine physique et n'est pas censée être modifiée (comme une empreinte digitale pour un être humain), tandis que l'IP privée identifie la machine d'un point de vue logique (comme le numéro de la chambre). De façon générale, les adresses IP et MAC n'opèrent pas au même niveau d'abstraction (voir le paragraphe « couches du modèle OSI » plus bas).

comme le JavaScript notamment (cf. chapitre 6.1), qui permet de rendre les pages web plus interactives. Ces codes sont faits pour être exécutés sur la machine du destinataire et sont donc envoyés à travers des paquets comme n'importe quelle autre donnée. Ce n'est qu'après leur réception que le navigateur les exécute.

Le web est l'une des utilisations possibles de l'internet, mais il en existe bien d'autres, tels que le courrier électronique, le transfert de fichiers, la visioconférence, les jeux en ligne, chacune étant associée à un ensemble de protocoles spécifiques – l'« internet » désigne l'infrastructure sous-jacente connectant les machines entre elles, quel que soit le service et les protocoles utilisés.

Le modèle client-serveur

Un **serveur** est une machine qui stocke des données et qui les envoie à des **clients** qui les demandent. Par exemple, un serveur web stocke des pages web et les envoie à des clients. Les clients formulent ces demandes sous la forme de **requêtes** respectant un protocole donné ; à la réception d'une requête, le serveur interprète celle-ci puis génère la réponse appropriée, qui est ensuite envoyée au client. Comme dit précédemment, le protocole HTTP est un exemple de protocole couramment utilisé dans le cadre du web, mais le modèle client-serveur est utilisé dans bien d'autres services que le web. Une « app » de réseau social sur un téléphone mobile, par exemple, est un client qui envoie des requêtes à un serveur distant (hors du téléphone, potentiellement à l'autre bout de la planète) pour obtenir des données (images, vidéos ou tout autre contenu d'un « feed »).

Le modèle client-serveur est le plus souvent utilisé de manière asynchrone, c'est-à-dire que les requêtes peuvent êtres envoyées à tout moment ; le client peut alors continuer à travailler sur d'autres données en attendant la réponse. En général, le client n'est pas « bloqué » pendant le traitement de la requête par le serveur. Cet aspect est l'une des raisons pour lesquelles la programmation web, qui doit souvent traiter de problématiques relatives à l'asynchronicité, est particulière en regard de la programmation de logiciels.

Concernant le web spécifiquement, un ensemble de serveurs nommés « DNS » (Domain Name System) est utilisé pour traduire les noms de domaines (comme `www.google.com`) en adresses IP. En effet, les ordinateurs ne peuvent communiquer qu'avec des adresses IP (donc des nombres), et non pas avec des noms de domaines, tandis qu'il est plus commode pour l'être humain de se souvenir et de travailler avec du texte qu'avec des nombres. Le serveur DNS est donc un serveur qui permet de faire la correspondance entre les deux. La chaîne de caractères identifiant une ressource (telle qu'une page web) et un protocole donnés se nomme l'**URL** (Uniform Resource Locator), et elle comprend typiquement le nom du protocole utilisé (par exemple `https`), le nom de domaine (par exemple `www.google.com`) et le chemin de la ressource demandée au sein du domaine (par exemple `/search`).

Les couches du modèle OSI

Pour terminer, mentionnons le modèle OSI (Open Systems Interconnection), qui définit une catégorisation des différents aspects relatifs à la transmission des données au sein d'un réseau. Ce modèle est composé de sept couches, chacune correspondant à un aspect particulier de la communication entre machines. L'étude plus détaillée des réseaux sortant du cadre de ce cours, nous nous bornons ici à mentionner à titre d'exemple :

- La couche physique, qui concerne les aspects matériels du réseau, comme la façon physique de coder le signal électrique ou optique.
- La couche réseau, responsable du chemin suivi par les données entre les points du réseau. Un exemple célèbre est le protocole IP discuté plus haut.
- La couche applicative, qui définit le format des messages envoyés entre applications. Un exemple célèbre est le protocole DNS discuté plus haut.

Les technologies de transmission de l'information

En 2024, les réseaux informatiques utilisant les lignes téléphoniques comme support de transmission tendent à disparaître, remplacés par des réseaux de fibres optiques. Ces derniers sont constitués de fils utilisant le principe physique de la réflexion totale pour transmettre la lumière d'un point à un autre en minimisant les pertes. Les fibres optiques sont capables de transmettre des informations à des fréquences bien plus élevées que les câbles électriques, et sont donc utilisées pour les transmissions aujourd'hui qualifiées de « haut débit ». Ces lignes, souvent sous-marines, relient les différents continents de la planète entre eux. Les données sont transmises sous forme de lumière avant d'être converties en signaux électriques à l'arrivée. Il est également possible d'utiliser une transmission par ondes électromagnétiques transitant par des satellites en orbite autour de la Terre, mais cette technique impose un chemin plus long aux données, qui arrivent avec un retard systématique (latence) au destinataire. Ce retard est bien plus faible dans le cas des câbles sur Terre. Bien que les satellites soient utilisés pour des communications où les connexions terrestres ne sont pas possibles (comme pour les communications maritimes, aériennes ou dans des régions très isolées), pour la plupart des applications de télécommunication terrestre, les câbles sont privilégiés. Ainsi, la vaste majorité des données de l'internet transitent aujourd'hui via des câbles, y compris celles reçues par des téléphones mobiles ; seuls les derniers hectomètres du chemin parcouru par les données (par exemple d'une antenne 4G au téléphone, ou d'une borne WiFi au téléphone) transitent éventuellement par les airs. Cette dernière partie du trajet est en général parcourue au sein d'un réseau local ou d'un réseau mobile, où des équipements spécifiques se chargent de la conversion du signal en un format interprétable par les machines destinataires.

1.7 Technologie digitale

1.7.1 Signaux analogiques et digitaux

Les types de commutateurs que nous avons considérés dans les dernières sections permettent tous de véhiculer des quantités discrètes d'information. Que l'on parle des ordinateurs travaillant en base 10, en base 3 ou en base 2, l'information est représentée grâce à un nombre fini de valeurs réparties sur un nombre fini de « cases » de la mémoire de l'ordinateur. Ces quantités sont dites **digitales** [17], ce qui désigne le fait que l'information soit découpée en paliers discrets (à l'instar des doigts de la main), appartenant à un ensemble fini d'états, et s'oppose à une information **analogique**, qui désigne une quantité dont les variations sont continues et possèdent donc une infinité d'états possibles. Tandis que la mesure d'une quantité analogique doit être approximée (il faudrait une précision infinie pour écrire le résultat complet), la lecture d'une quantité digitale livre un résultat exact (à l'intérieur du système de codage discret qui a été choisi).

Dans les deux prochains chapitres, nous traitons de la façon de faire **correspondre** des quantités analogiques (issues du « monde extérieur », continu) à des quantités digitales (représentables par un ordinateur, discrètes). Cette correspondance, la plupart du temps nécessairement imparfaite, est cruciale pour comprendre la façon dont les ordinateurs traitent l'information, et porte le nom de **codage** de l'information.

Notons enfin que les signaux digitaux ont l'avantage, en plus de bien se prêter à la conception d'ordinateurs, d'être plus robustes aux perturbations extérieures que les signaux analogiques. Etant donné qu'un signal digital est constitué d'un nombre fini (et souvent petit) d'états distincts, il est plus facile de déterminer si un signal est « correct » ou « incorrect » que dans le cas d'un signal analogique, qui peut être plus difficilement reconstructible suite à une perturbation par des bruits de fond, des interférences électromagnétiques, etc. Cela a pour conséquence que des calculs sur ordinateur peuvent être **reproductibles**, tandis que des calculs analogique ne peuvent totalement l'être.

1.7.2 Le cas des calculateurs analogiques

La résolution de problèmes géométriques grâce au compas est un exemple de méthode analogique de résolution de problèmes. La célèbre machine d'Anticythère est un autre exemple antique : une manivelle permettait de faire tourner une série d'engrenages dimensionnés pour prédire les positions des astres conformément à un modèle astronomique de l'époque. Ainsi, une équation était résolue en déléguant les calculs à un mécanisme physique continu (en bonne approximation). La figure 1.12 ci-dessous illustre une reconstitution de cette machine. Plus récemment, des calculateurs analogiques ont été utilisés pour résoudre des équations différentielles et, donc, simuler des systèmes physiques. Ces calculateurs peuvent dans certains cas se montrer plus rapides que leurs homologues numériques, mais sont également plus coûteux et moins flexibles. Ils sont donc susceptibles

17. Il est également courant d'employer dans le langage courant le terme « numérique » pour se référer à une information exprimable ou exprimée par des ordinateurs, bien qu'il semble que ce terme puisse également être utilisé pour désigner des ensembles de nombres continus, contrairement à « digital ».

FIGURE 1.12 – Une proposition de reconstitution du mécanisme d'Anticythère en 2007. Auteur : I. Mogi, licence CC BY 2.5

d'être utilisés dans des cas spécifiques où la rapidité est un facteur critique. Néanmoins, ils ne sauraient se rapprocher de la définition de l'ordinateur que nous avons donnée précédemment, et constituent plutôt une classe à part de machines à calculer. Notons finalement que, tout comme des équations différentielles peuvent être émulées par des circuits électriques non digitaux au sein de calculateurs analogiques, des équations booléennes peuvent être émulées par des circuits électroniques digitaux – ceux que l'on nomme aujourd'hui « ordinateurs ».

Chapitre 2

Codage de l'information – Les nombres

Comme nous l'avons vu au sein du chapitre précédent, si l'ordinateur est un dispositif électronique, c'est avant tout pour des raisons pratiques ; si, par exemple, les relais hydrauliques étaient les plus performants, alors l'ordinateur serait un dispositif hydraulique. Or, il se trouve que le composant fondamental ayant permis la miniaturisation et la fiabilisation des ordinateurs est le transistor, un composant électronique qui se prête bien à la manipulation de quantités **binaires**, c'est-à-dire possédant deux états possibles : *on* et *off*, *vrai* et *faux* ou encore *1* et *0*, qui est la notation que nous suivrons désormais. Par ailleurs, comme nous l'avons mentionné au moment de discuter du calcul analogique, les signaux sont d'autant plus robustes aux perturbations environnantes qu'ils portent un nombre restreint de valeurs possibles. Ainsi, le choix de la base 2 pour représenter les informations au sein d'un ordinateur est un choix naturel.

Nous avons discuté de la différence entre des signaux analogiques et des signaux digitaux au sein de la section 1.7 — la question se pose maintenant de savoir comment représenter des informations de types variés telles que des nombres, des lettres, des images, des vidéos ou des sons, à l'aide d'un signal binaire uniquement. En réalité, cette question cache deux problèmes de natures différentes, que nous allons aborder dans ce chapitre :

1. Comment exprimer un nombre quelconque à l'aide de symboles binaires ?
2. Comment coder une information qui n'est pas un nombre (par exemple une lettre, une couleur ou un son) à l'aide de nombres ?

2.1 Systèmes de numération

Plusieurs façons d'écrire les nombres ont été développées selon les époques et les régions. Ces « façons d'écrire les nombres » sont appelées des **systèmes de numération**. Chacune comporte ses avantages et ses inconvénients ; nous verrons plus bas, en section 2.1.2, que les systèmes de numération positionnels sont particulièrement adaptés à la manipulation de quantités potentiellement réparties sur plusieurs ordres de grandeur et sont donc, à ce titre, d'une grande utilité dans tous les domaines techniques.

2.1.1 Systèmes de numération non positionnels additifs

Certains systèmes simples consistent à associer à chaque symbole un nombre et à additionner ces nombres pour obtenir le total. Pour cette raison, on dit que ce sont des systèmes additifs ; toujours pour cette raison, la position de chaque symbole au sein du nombre n'a pas d'importance [1]. L'exemple le plus simple est le système n'utilisant qu'un symbole, celui de l'unité. Si le symbole choisi est « I », alors la quantité représentant trois unités se note : III.

Les symboles qui servent à exprimer un **nombre** sont appelés des **chiffres**. Cette terminologie a une importance pour désigner sans ambiguité ce dont nous parlons. Par exemple, nous allons souvent devoir dire des choses comme « le nombre de chiffres de ce nombre est plus petit que celui de tel autre nombre ». Pour que ce type de phrase ait un sens, il faut être rigoureux dans l'emploi du vocabulaire.

Il est courant d'envisager l'analogie suivante : les chiffres sont en quelque sorte aux nombres ce que les lettres sont aux mots. Une lettre ne porte pas de signification en soi ; ce sont les combinaisons de lettres qui portent une signification. Par ailleurs, tout comme certains mots s'écrivent avec une seule lettre (« a » par exemple), certains nombres peuvent être exprimés à l'aide d'un seul chiffre.

Considérons un autre exemple de système non positionnel. Si l'on fixe que $A = 1$, $B = 2$ et $C = 3$, alors le nombre $ABC = ACB = BAC = BCA = CAB = CBA = 1+2+3 = 6$. Cette quantité représente 6 unités, et elle est exprimée à l'aide des trois chiffres A, B et C, disposés dans n'importe quel ordre. Il est important de voir qu'en plus de cet ordre quelconque, les chiffres que l'on vient d'utiliser pour exprimer 6 ne forment eux-mêmes pas un ensemble unique. En effet, $BBB = 6$ également, tout comme $AAAAAA$.

Remarquons pour terminer qu'au sein des systèmes non positionnels, le concept de « quantité nulle » n'est d'aucune utilité apparente. En effet, la quantité nulle correspond ici à l'absence de symbole. Cette apparente simplicité cache en réalité le plus grand défaut de ce système : nous allons maintenant voir pourquoi la capacité à pouvoir désigner une quantité nulle est fondamentale au sein des systèmes positionnels, et les avantages que cela confère sur les systèmes non positionnels.

2.1.2 Systèmes de numération positionnels

Les systèmes non positionnels offrent l'avantage d'être faciles à appréhender. Néanmoins, ils comportent des défauts importants tels qu'un manque de compacité ou de généralité dans l'écriture de nombres peu communs, pour lesquels aucun symbole commode n'est déjà prévu au sein du système. Par ailleurs, il est peu aisé d'effectuer des calculs avancés en utilisant ces systèmes. Pour ces raisons, les systèmes positionnels, qui offrent un niveau de généralité et de compacité supérieurs, sont devenus les systèmes de numération les plus utilisés dans le monde moderne.

1. Dans des systèmes tels que le système de numération romain, d'autres règles viennent s'ajouter à celle qui vient d'être décrite, mais cette dernière constitue néanmoins le cœur du système.

L'étude des systèmes de numération dans le cas général sort du cadre de ce cours[2]. Nous nous limiterons ici aux systèmes de numération positionnels à base entière, qui sont les plus courants et les plus utiles en informatique.

Un exemple bien connu : le système décimal

Le système décimal, qui nous est enseigné à l'école primaire, est un système de numération positionnel à base entière dans lequel chaque chiffre a une valeur qui dépend de sa position dans le nombre. Par « **base 10** » ou, ce qui est équivalent, « **système décimal** », nous entendons que le chiffre dans la colonne numéro i est **implicitement** à multiplier par 10^i avant d'être ajouté au total du nombre que l'on est en train d'exprimer, lui-même obtenu par addition de la contribution de chacun des termes ainsi composés. Quand on exprime un entier, la colonne la plus à droite porte le numéro $i = 0$. Chaque chiffre du nombre correspond à un entier compris entre 0 et 9, c'est-à-dire que l'ensemble des symboles possibles contient 10 symboles. Ainsi, un nombre entier n en base 10 s'exprime de la façon suivante :

$$n = \sum_{i=0}^{k-1} a_i \cdot 10^i, \tag{2.1}$$

où a_i est le chiffre en position i du nombre, k est le nombre de chiffres du nombre et $a_{k-1} \neq 0$, c'est-à-dire que le nombre ne commence pas par des zéros inutiles.

Par exemple, le nombre 42 s'exprime en base 10 comme $42 = 4 \cdot 10^1 + 2 \cdot 10^0$. On retrouve là les « colonnes » des dizaines et des unités, qui sont des concepts familiers à tout élève de primaire.

De façon analogue, on peut exprimer des valeurs non entières. En effet, il suffit d'étendre la combinaison des termes $a_i \cdot 10^i$ à des valeurs de i négatives. Par exemple, le nombre 42.03 s'exprime en base 10 comme $42.03 = 4 \cdot 10^1 + 2 \cdot 10^0 + 0 \cdot 10^{-1} + 3 \cdot 10^{-2}$. Tout comme nous n'indiquons pas de zéros inutiles « avant » le nombre, nous n'indiquons pas non plus de zéros inutiles « après » lui. Par convention, on utilisera au sein de ce cours le point pour désigner la transition de $i = 0$ à $i = -1$ au sein de la succession des chiffres ; c'est ce que nous appelons la « virgule ».

Le système binaire

Le système binaire désigne la base 2. Par analogie avec tout ce qui vient d'être dit pour le système décimal, un nombre entier n en base 2 s'exprime de la façon suivante :

$$n = \sum_{i=0}^{k-1} a_i \cdot 2^i, \tag{2.2}$$

où a_i est le chiffre en position i du nombre, k est le nombre de chiffres du nombre et $a_{k-1} \neq 0$.

2. En effet, le champ des systèmes envisageables est très large. Par exemple, il est possible de s'intéresser aux systèmes de numération à bases non entières.

Ainsi, par exemple, la valeur décimale 42 s'exprime en binaire comme 101010. En effet, $42 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. On s'aperçoit ici que, pour éviter toute confusion, il est bon de signifier clairement lorsque des nombres sont exprimés dans des bases différentes. Ainsi, plutôt que d'écrire $42 = 101010$, nous utiliserons dorénavant la convention d'indiquer en indice des nombres la base dans laquelle ils sont exprimés :

$$42_{10} = 101010_2, \tag{2.3}$$

tout en gardant la possibilité d'omettre l'indice de la base uniquement lorsque celle-ci est décimale. Ainsi, $42 = 101010_2$ est également non ambigu dans ce cours.

Pour clore cette sous-section, notons qu'il est commun de qualifier de « bit de **poids fort** » le bit le plus à gauche d'une série de bits représentant un nombre. Inversement, on parlera de « bit de **poids faible** » pour désigner le bit le plus à droite.

Base entière positive quelconque

Les bases binaire (2), octale (8), décimale (10) et hexadécimale (16) sont couramment utilisées en informatique. Néanmoins, comme le concept de base entière se généralise aisément, nous en donnons ici une brève description.

Un système de numération positionnel à base entière b est un système de numération dans lequel chaque chiffre possède une valeur qui dépend de sa position i dans le nombre. La position i a pour poids la valeur b^i au sein du nombre. Le nombre exprimé vaut la somme des contributions de chacun des chiffres qui le composent. La base b est un entier strictement supérieur à 1. Tout chiffre du nombre correspond à un entier compris entre la valeur nulle et $b - 1$. Un nombre n en base b, composé de k chiffres, s'exprime alors de la façon suivante :

$$n = (a_{k-1} \dots a_1 a_0)_b = \sum_{i=0}^{k-1} a_i \cdot b^i, \tag{2.4}$$

où a_i est le chiffre en position i du nombre, k est le nombre de chiffres du nombre n exprimé en base b, et $a_{k-1} \neq 0$.

Par exemple, le nombre 61_7 désigne le nombre décimal 43 écrit en base 7, car $43 = 6 \cdot 7^1 + 1 \cdot 7^0$.

Notons que, par convention, lorsque la base utilisée est supérieure à 10, nous utilisons lorsque c'est possible les lettres majuscules de l'alphabet latin pour désigner les chiffres supplémentaires. Ainsi, en base 16, les chiffres A, B, C, D, E et F correspondent aux quantités décimales 10, 11, 12, 13, 14 et 15, respectivement.

Expression d'un nombre à virgule

Enfin, les nombres non entiers s'expriment de façon analogue à ce qui a été vu plus haut pour le système décimal ou binaire, c'est-à-dire en étendant la combinaison des termes

$a_i \cdot b^i$ à des valeurs de i négatives :

$$n_b = (a_{k-1} \dots a_1 a_0 a_{-1} \dots a_{-v})_b = \sum_{i=-v}^{k-1} a_i \cdot b^i, \tag{2.5}$$

où v est le nombre de chiffres après la virgule et $a_{-v} \neq 0$.

Par exemple, le nombre 61.4_7 désigne un nombre décimal égal à $43 + \frac{4}{7} \approx 43.57$, car $6 \cdot 7^1 + 1 \cdot 7^0 + 4 \cdot 7^{-1} = 43.57142\dots$

2.1.3 Conversions entre systèmes de numération positionnels

Conversion des nombres entiers

La conversion d'un entier exprimé en base b vers la base 10 est donnée par la définition en équation 2.4. Cependant, pour convertir un nombre entier n exprimé en base 10 vers une base quelconque b, nous pouvons utiliser l'algorithme suivant :

1. Effectuer une division entière du nombre n par b (notée $\lfloor n \div b \rfloor$) et noter le reste.
2. Répéter l'opération depuis la première étape jusqu'à ce que le quotient soit nul. Le nombre final est constitué des restes des divisions successives, lus de gauche à droite de la dernière division à la première.

Par exemple, pour convertir le nombre 547 en base 7, nous obtenons la suite d'étapes :

- $\lfloor 547 \div 7 \rfloor = 78$ avec un reste de 1.
- $\lfloor 78 \div 7 \rfloor = 11$ avec un reste de 1.
- $\lfloor 11 \div 7 \rfloor = 1$ avec un reste de 4.
- $\lfloor 1 \div 7 \rfloor = 0$ avec un reste de 1.

Ainsi, le nombre 547_{10} s'exprime comme 1411_7. Notons que, dans tout ce qui suit, les quantités encadrées par « $\lfloor$ » et « $\rfloor$ » sont arrondies à l'entier inférieur. Par exemple, $\lfloor 3.8 \rfloor = 3$.

Bien que l'algorithme qui vient d'être mentionné soit général, il est bon de savoir que, de la base 10 à la base 2, l'algorithme suivant permet d'accélérer quelque peu le procédé en trouvant directement les chiffres non nuls qui servent à écrire le nombre :

1. Trouver la puissance i à laquelle il faut élever le nombre 2 pour obtenir un nombre inférieur ou égal au nombre n à exprimer. Ce nombre vaut $i = \lfloor \log_2 n \rfloor$. Retenir la valeur de i.
2. La nouvelle valeur n à exprimer est alors $n - 2^i$.
3. Répéter les étapes depuis le point 1 jusqu'à ce que $n = 0$.
4. Le nombre de départ s'exprime en base b comme $\sum b^i$, où l'on utilise les valeurs de i retenues à chaque étape.

Voici un exemple de conversion du nombre 547_{10} vers la base 2 utilisant la méthode qui vient d'être décrite :

- On note la position du premier chiffre non nul : $\lfloor \log_2 547 \rfloor = 9$.
- On calcule ce qu'il reste à exprimer : $547 - 2^9 = 35$.
- On note la position du second chiffre non nul : $\lfloor \log_2 35 \rfloor = 5$.
- On calcule ce qu'il reste à exprimer : $35 - 2^5 = 3$.
- On note la position du troisième chiffre non nul : $\lfloor \log_2 3 \rfloor = 1$.
- On calcule ce qu'il reste à exprimer : $3 - 2^1 = 1$.
- On note la position du quatrième chiffre non nul : $\lfloor \log_2 1 \rfloor = 0$.
- On calcule ce qu'il reste à exprimer : $1 - 2^0 = 0$. On a donc terminé.
- Le nombre 547_{10} s'exprime en binaire comme 1000100011_2.

Conversion de nombres non entiers

Lorsque nous convertissons un entier, la nature du système de numération positionnel implique que l'on doit effectuer des divisions pour trouver les chiffres qui constituent le nombre. Or, après un nombre suffisant de divisions, la partie entière de ce nombre finit toujours par être nulle, puisque les diviseurs sont tous positifs. À ce moment, nous avons terminé la conversion.

En revanche, pour la partie fractionnaire d'un nombre (exprimée par les chiffres situés après la virgule), l'argument précédent ne tient plus. En effet, les divisions effectuées impliquent alors un diviseur toujours plus petit, si bien qu'il n'y a aucune garantie de ne devoir en effectuer une infinité. Nous discutons en section 2.5.2 des implications qui en découlent.

Ainsi, la conversion de la base décimale vers une base quelconque dépend le plus souvent d'un niveau de précision arbitraire que nous devons décider. Voici un exemple pour le nombre 42.301 converti en base 7 :

- $\lfloor 42 \div 7 \rfloor = 6$ avec un reste de 0.
- $\lfloor 6 \div 7 \rfloor = 0$ avec un reste de 6.
- La partie entière vaut donc $42 = 60_7$.
- Si l'on veut un seul chiffre après la virgule, on cherche le plus grand a tel que $a \cdot 7^{-1} \leq 0.301$. On trouve $a = 2$ et donc $42.301 \approx 60.2_7$.
- Si l'on veut un second chiffre après la virgule, on cherche le plus grand a tel que $2 \cdot 7^{-1} + a \cdot 7^{-2} \leq 0.301$. On trouve $a = 0$ et donc $42.301 \approx 60.20_7$.
- Si l'on veut un troisième chiffre après la virgule, on cherche le plus grand a tel que $2 \cdot 7^{-1} + 0 \cdot 7^{-2} + a \cdot 7^{-3} \leq 0.301$. On trouve $a = 5$ et donc $42.301 \approx 60.205_7$.

Remarquons que, selon l'utilisation que l'on souhaite faire de notre expression, on pourrait choisir le dernier chiffre après la virgule de façon à minimiser l'écart avec le nombre à exprimer n, plutôt que de façon à ce que le nombre exprimé soit strictement inférieur à n. Ainsi, en appliquant notre méthode pour 4 chiffres après la virgule, on trouverait $42.301 \approx 60.2051_7$, ce qui est environ égal à 42.3007 ; une meilleure valeur avec le même nombre de chiffres serait $42.301 \approx 60.2052_7$, qui vaut environ 42.3011. Avec un nombre fini de chiffres après la virgule, on doit en général choisir entre une troncature et un arrondi.

2.1.4 Additions en binaire

La méthode manuscrite classique (« en colonnes ») pour l'addition de nombres décimaux s'applique de la même façon pour les nombres binaires –ou pour toute autre base entière. Il s'agit d'aligner les nombres à additionner en colonnes et de les additionner de droite à gauche, en reportant les retenues d'une colonne à la suivante. Il faut donc garder en tête qu'une colonne dont le résultat vaut 10_2 possède donc un résultat de zéro, mais reporte un 1 en retenue à la colonne suivante. De même, une colonne dont le résultat vaut 11_2 (ce qui n'arrive que lorsque chacun des nombres à additionner vaut 1 et qu'en plus une retenue était également présente à l'origine) possède un résultat de 1 et reporte un 1 en retenue à la colonne suivante. Dans l'exemple ci-dessous, on additionne les nombres $81 = 1010001_2$ et $71 = 1000111_2$. Les retenues sont écrites en exposant afin d'imiter la notation écrite pour les additions décimales.

	0^1	1	0	1	0^1	0^1	0^1	1
+	0	1	0	0	0	1	1	1
=	1	0	0	1	1	0	0	0

Le résultat se lit dans la ligne du bas. En l'occurrence, il vaut $10011000_2 = 152$. Ici, une colonne a dû être ajoutée à gauche pour accueillir la retenue engendrée par la somme des deux chiffres les plus à gauche de chaque nombre. Tout comme en décimal, si deux nombres ne s'écrivent pas avec le même nombre de chiffres, il faut garder en tête qu'il est possible de préfixer un nombre avec des zéros autant de fois que possible, étant donné que cela n'affecte pas sa valeur.

2.1.5 Soustractions en binaire

La méthode manuscrite classique pour la soustraction de nombres décimaux implique un nombre d'opérations et une complexité supérieures à la méthode de l'addition. D'un point de vue informatique, il semble plus avantageux de trouver un moyen de codage des nombres tels que toute soustraction puisse être vue comme une addition de nombres qui, eux-mêmes, peuvent être négatifs. Par ailleurs, ainsi que discuté dans le chapitre 4, il existe des avantages technologiques à procéder de la sorte. Ainsi, au lieu d'avoir à définir deux opérations, nous n'avons qu'à en définir une seule. C'est ce que nous ferons en section 2.4.

2.1.6 Multiplications en binaire

Multiplication par une puissance de 2

Avant de voir comment multiplier deux nombres binaires sans passer par une conversion en décimal, il est utile de se rappeler la définition même d'un nombre dans un système positionnel, qui est constitué d'additions de puissances de la base. En base 10, cela signifie que multiplier un nombre par 1000 revient à décaler tous les chiffres de trois colonnes vers

la gauche, en laissant des zéros dans les colonnes vides sur la droite. En effet, et pour prendre un exemple, il se trouve que :

$$\begin{aligned} 23 \cdot 1000 &= \left(2 \cdot 10^1 + 3 \cdot 10^0\right) \cdot 10^3 && (2.6)\\ &= 2 \cdot 10^1 \cdot 10^3 + 3 \cdot 10^0 \cdot 10^3 && (2.7)\\ &= 2 \cdot 10^4 + 3 \cdot 10^3 && (2.8)\\ &= 23000. && (2.9) \end{aligned}$$

De façon tout à fait analogue, on comprend la multiplication d'un nombre binaire par une puissance de 2. En effet, en utilisant la définition de la notation positionnelle (équation 2.4) et en multipliant un nombre n par la 2^m, on obtient :

$$n \cdot 2^m = 2^m \sum_{i=0}^{k-1} a_i \cdot 2^i = \sum_{i=0}^{k-1} a_i \cdot 2^{i+m}, \tag{2.10}$$

ce qui signifie que le chiffre de poids le plus faible, ici noté a_0, participe à la somme *via* le facteur 2^{0+m} et, donc, que la séquence de chiffres de a_0 à a_{k-1} est décalée de m colonnes vers la gauche.

Multiplier un nombre binaire par 2^m revient donc à décaler tous ses chiffres de m colonnes vers la gauche, en laissant des zéros dans les colonnes vides sur la droite.

Multiplication entre deux entiers quelconques

Le principe que l'on vient de voir peut être exploité pour écrire une addition binaire à partir d'une multiplication binaire. Il suffit en effet de voir tout nombre binaire de k chiffres comme une somme d'au plus k nombres binaires de 1 chiffre, chacun multiplié par une puissance de 2 qui lui est propre. Par exemple, le nombre 1101_2 peut être vu comme le résultat de l'addition suivante :

	1	0	0	0
+	0	1	0	0
+	0	0	0	0
+	0	0	0	1
=	1	1	0	1

À présent que nous sommes à l'aise avec la décomposition d'un nombre, nous pouvons le multiplier avec un autre. Prenoms l'exemple de 1101_2 multiplié par 101_2. Multiplier par cinq, c'est additionner le résultat de la multiplication par 100_2 avec celui de la multiplication par 1_2. Ainsi, $1101_2 \cdot 101_2 = 1101_2 \cdot (100_2 + 1_2) = 110100_2 + 1101_2$, en vertu de la règle du décalage vue ci-dessus. Le résultat final est ensuite calculé grâce à l'opération d'addition déjà définie plus haut. Ainsi, à aucun moment nous n'avons dû définir de « méthode en colonne » pour la multiplication. Pour résumer l'exemple précédent, nous avons :

				1	1	0	1
×					1	0	1
=	0^1	1^1	1^1	0^1	1	0	0
+	0		0	1	1	0	1
=	1	0	0	0	0	0	1

2.1.7 Divisions en binaire

Les méthodes manuscrites classiques pour la division de nombres binaires quelconques sont, à l'image de ce qui se fait en décimal, légèrement plus fastidieuses que les méthodes d'addition et de multiplication. Bien qu'elles ne soient pas abordées ici, il est bon de savoir que des algorithmes existent pour effectuer ces opérations de façon efficace. Le lecteur intéressé est invité à trouver dans la littérature et sur Internet les ressources idoines.

En revanche, nous discutons ici du cas des divisions par une puissance de 2. Celles-ci sont particulièrement simples à effectuer, car elles reviennent à décaler tous les chiffres du dividende vers la droite, à l'inverse des multiplications par une puissance de 2. En effet, de façon analogue à l'équation 2.10, on peut écrire :

$$\frac{n}{2^m} = 2^{-m} \sum_{i=0}^{k-1} a_i \cdot 2^i = \sum_{i=0}^{k-1} a_i \cdot 2^{i-m}. \tag{2.11}$$

Encore une fois, l'analogie avec toute autre base entière est possible. Par exemple $\lfloor 4781 \div 10^3 \rfloor = 4$. Diviser un nombre binaire par 2^m revient donc à décaler tous ses chiffres de m colonnes vers la droite. Il y a ici une nuance à apporter malgré tout : dans le cas de la division entière, certains chiffres du nombre divisé sont perdus lors du décalage vers la droite, peu importe leur valeur. Ainsi, la division par une puissance de 2 s'entend ici comme une division entière avec arrondi vers le bas.

2.2 Stockage des quantités binaires

En préambule à cette section, remarquons que le concept de **convention** est fondamental dans tout le reste du chapitre. Par « convention », nous entendons un choix **arbitraire** grâce auquel nous pouvons représenter une information, et qui contient des règles implicites. Par exemple, en Suisse, les codes postaux codent les communes de façon unique mais arbitraire (1217 pour Meyrin, 1227 pour Carouge, etc.) ; sans connaître la convention utilisée, il est impossible de déchiffrer un code postal. De même, en informatique, il est nécessaire de connaître les conventions de codage pour comprendre comment une information est représentée. Sans convention, les nombres qui représentent l'information n'endossent aucune signification particulière[3].

3. Notons que le poids de chaque chiffre au sein d'un système de numération positionnel contient lui-même un implicite très fort et pourtant indispensable (en plus de la signification intrinsèque des chiffres) : par exemple, 42 signifie implicitement $4 \cdot 10^1 + 2 \cdot 10^0$; deux additions, deux multiplications et deux puissances de la base sont en quelque sortes cachées au sein de l'écriture « 42 ».

2.2.1 Bits, octets, mots et préfixes

Dans un système de numération binaire, les chiffres sont souvent appelés des **bits**[4]. Plus généralement, on désigne par là un **état binaire**, c'est-à-dire une quantité qui ne peut prendre que deux valeurs différentes. Comme nous le verrons, il est possible de faire correspondre un dispositif électronique à un état binaire au sein d'un ordinateur. Pour des raisons pratiques, au sein des ordinateurs modernes, les bits sont communément regroupés par « paquets » de huit nommés **octets** en français ou **bytes**[5] en anglais. Enfin, un **mot** est un groupe de bits d'une taille donnée qui est spécifique à un type de processeur ainsi qu'à certains éléments qui lui sont liés. Le tableau 2.1 ci-dessous résume les termes que nous venons de voir. Comme nous le verrons plus loin, les ordinateurs travaillent avec

TABLE 2.1 – Terminologie utilisée pour désigner différentes quantités d'information binaire. Dans ce cours, pour éviter toute confusion entre l'abbréviation « O » et le chiffre 0, l'abbréviation anglaise B sera toujours utilisée pour désigner l'octet.

Nom	Abbréviation	Longueur
bit	b	1 bit
octet (byte)	B	8 bits
mot de longueur k		k bits

des mots entiers plutôt qu'avec des bits isolés. Ces mots sont en général des multiples d'un octet, bien qu'on les exprime communément en bits. Comme cette quantité est une constante pour un type d'ordinateur donné, on parle communément d'**architecture** utilisant des mots de n bits. La taille des mots conditionne celle des données sur lesquelles le processeur effectue des opérations ou encore celle des données lues dans la mémoire centrale. Pour cette raison, la taille des mots a un fort impact sur les types numériques en programmation (cf. section 2.3.1).

Il est à noter que ces différentes quantités d'informations sont souvent précédées d'un préfixe indiquant une puissance de 1000 pour désigner des quantités plus grandes[6]. La table 2.2 présente les préfixes les plus communément utilisés, en informatique aussi bien que dans d'autres domaines :

2.2.2 Nombre de configurations représentables avec k bits

Une séquence de k bits peut posséder plusieurs valeurs différentes. Chaque bit étant susceptible de prendre deux valeurs, une séquence de k bits peut prendre 2^k valeurs

4. Contraction de **bi**nary digi**t** et jeu de mot avec la signification de « petit morceau » en anglais. En effet, un bit est la plus petite quantité d'information imaginable.

5. Il est à noter que dans certains systèmes, les bytes ne sont pas nécessairement de 8 bits ; en ce sens, le terme français d'octet est moins ambigu. Dans tous les cas, « byte » sera entendu comme traduction exacte d'« octet » (8 bits) dans ce cours.

6. Il n'est pas rare de trouver des documents où le choix a été fait de travailler avec des puissances de 1024 plutôt que de 1000, ce que nous ne faisons pas dans ce cours, où nous utilisons uniquement les préfixes du système international. Dans le cas où des puissances de 1024 sont utilisées, les préfixes sont alors de la forme Ki, Mi, Gi, ... au lieu de k, M, G, ...

TABLE 2.2 – Quelques préfixes communément utilisés.

Préfixe	Symbole	Valeur
kilo	k	10^3
méga	M	10^6
giga	G	10^9
téra	T	10^{12}
péta	P	10^{15}
exa	E	10^{18}

différentes. Par exemple, un octet peut prendre $2^8 = 256$ valeurs différentes. Nous ne parlons pas ici de la valeur de la séquence de bits si on l'interprète via un système de numération positionnel, mais bien du nombre de **configurations différentes** que la séquence peut prendre. Ces valeurs peuvent par exemple être « numérotées » de 0 à $2^k - 1$ (ou tout autre choix), mais cela est arbitraire ; ce qui ne l'est pas, en revanche, est le nombre de configurations possibles.

Comme nous venons de le signaler, si l'on **interprète** une séquence de k bits comme un nombre naturel exprimé en base 2, alors ce nombre peut prendre des valeurs allant de 0 à $2^k - 1$. Cette façon d'interpréter la séquence de bits, en correspondance directe avec la lecture d'un nombre en base 2, est utilisée dans plusieurs contextes dans la suite du cours. Quoi qu'il en soit, dès lors que l'on interprète une séquence de bits (peu importe la méthode), on dit que la séquence de bits **code** une valeur. Par exemple, un octet interprété comme un entier naturel écrit en base 2 peut coder des valeurs allant de 0 à 255.

Il est tentant de se demander si le prix à payer pour travailler en binaire n'est pas trop élevé ; en travaillant dans une base plus élevée [7], ne pourrait-on pas compactifier l'écriture des nombres à un point tel que, même si la technologie pour ce faire est plus lente, le gain en compacité de l'information serait suffisant pour compenser cette lenteur de calcul ?

La réponse touche bien entendu à la technologie et à l'ingénierie de la couche matérielle des ordinateurs. Cependant, il faut ici se méfier de l'intuition humaine, qui a peu de prises sur les quantités qui évoluent de façon exponentielle, telles que l'équation 2.4. Posons-nous donc la question : combien de chiffres faut-il pour représenter un nombre n dans une base quelconque b ? Pour simplifier, supposons que n est une puissance de b. Un tel nombre s'exprimerait sous la forme $n = b^k$ et le nombre de chiffres nécessaires à son écriture vaut :

$$k = 1 + \lfloor \log_b n \rfloor. \tag{2.12}$$

En effet, en base b, le nombre b^k s'écrit avec le chiffre 1 suivi de k chiffres 0. En utilisant les propriétés des logarithmes, on peut exprimer le ratio entre le nombre de chiffres nécessaires au sein de la base b_1 comparé à la base b_2, lorsque b est grand :

$$\frac{k_1}{k_2} = \frac{1 + \log_{b_1} n}{1 + \log_{b_2} n} \approx \frac{\log_{b_1} n}{\log_{b_2} n} = \log_{b_1} b_2, \tag{2.13}$$

7. Rappelons-nous qu'il est tout à fait possible, comme nous l'avons vu dans le premier chapitre de ce cours, de construire des ordinateurs représentant les nombres dans d'autres base. Des machines fonctionnant en base 3 et en base 10, notamment, ont été utilisées (mais de façon marginale) jusque dans la seconde moitié du vingtième siècle.

ce qui illustre le fait que l'avantage procuré par la compacité de l'écriture croît peu avec la base b_2 choisie, et donc que les désavantages qui découlent du surplus de complexité technologique pèsent très lourd dans la balance.

2.2.3 Utilité de la base hexadécimale

Le nombre de valeurs représentables avec un octet vaut $2^8 = 256$. Ainsi, si l'on pouvait travailler en base 256, il serait possible de représenter un octet par un seul chiffre. Malheureusement, la mémoire humaine rend l'utilisation d'une telle base peu commode, puisqu'il faudrait se souvenir de 256 symboles différents pour noter les chiffres. En revanche, $2^4 = 16$ correspond à un nombre de chiffres beaucoup plus abordable. Bien entendu, d'un point de vue technologique il est également plus commode de travailler dans des bases faibles (comme 2), comme nous l'avons vu dans le chapitre 1. Cependant, comme $2^8 = (2^4)^2 = 16^2$, il est possible d'exprimer n'importe quelle valeur stockable dans un octet à l'aide de seulement deux chiffres en base 16. C'est pourquoi la base 16, appelée système **hexadécimal**, est couramment utilisée en informatique pour afficher des nombres binaires codés sur un octet. Nous avons mentionné précédemment (cf. section 2.1.2) la convention d'utiliser les lettres latines pour désigner les chiffres supplémentaires en base 16. Il faut néanmoins garder à l'esprit que la machine, elle, travaille bien en base 2 ; c'est l'affichage des nombres destinés à leur lecture par l'humain qui est souvent effectué en base 16, en particulier lorsque ceux-ci sont stockés dans un multiple d'un octet. Nous examinons ci-dessous quelques cas où l'hexadécimal est particulièrement utile ou, du moins, utilisé.

Etats binaires et nombres

Une suite de bits ne représente pas nécessairement un entier naturel. Comme nous le verrons dans ce chapitre, on peut décider d'interpréter ces bits comme codant une lettre, une couleur, etc. Au final, bien évidemment, rien ne nous interdit de faire correspondre cette signification à l'entier naturel que ces bits représenteraient s'ils étaient interprétés comme tel.

Toujours est-il qu'une séquence de k bits correspond, en toute généralité, à un « état binaire », et pas nécessairement à son interprétation en tant que nombre. Au sein de la communauté informatique, il est courant de se référer à des états binaires en utilisant le système hexadécimal. On préfixe alors par `0x` l'entier naturel qui correspond à l'état binaire pour indiquer que le nombre est exprimé en base 16. Cette façon de noter est empruntée au langage C. Par exemple, si le contenu de deux octets est `1011011111100100`, alors on se réfère à cet état binaire comme valant `0xB7E4`, car $B7_{16} = 10110111_2$ et $E4_{16} = 11100100_2$.

Codage des couleurs

Dans le domaine du graphisme, par exemple, la représentation hexadécimale des couleurs primaires au format Rouge-Vert-Bleu (RGB, cf. chapitre 3.3) est très commune, chaque couleur primaire étant représentée comme un couple de chiffres hexadécimaux ; ainsi une couleur donnée, qui est un triplet de couleurs primaires, est donc codée sous la forme de 6 chiffres en base 16. Par exemple, la couleur bleu ciel, correspondant au triplet RGB $(135, 206, 235)$, est représentée par le code hexadécimal `#87CEEB`, le dièse étant un préfixe courant pour désigner les couleurs en hexadécimal. En effet, $135 = 87_{16}$, $206 = CE_{16}$ et $235 = EB_{16}$.

Concaténation des chiffres

Nous allons ici prouver que tout nombre n en base 16 peut être converti en base 2 en concaténant simplement l'expression binaire de chaque chiffre au sein de n_{16}.

$$\begin{aligned} n_{16} &= \text{« XY »} && (2.14)\\ &= X \cdot 16^1 + Y \cdot 16^0 && (2.15)\\ &= X \cdot 2^4 + Y \cdot 2^0, && (2.16) \end{aligned}$$

où, comme X et Y sont des nombres entre 0 et 15, ils s'expriment en binaire sur 4 bits, et donc :

$$X = X_3 \cdot 2^3 + X_2 \cdot 2^2 + X_1 \cdot 2^1 + X_0 \cdot 2^0 \tag{2.17}$$

et

$$Y = Y_3 \cdot 2^3 + Y_2 \cdot 2^2 + Y_1 \cdot 2^1 + Y_0 \cdot 2^0. \tag{2.18}$$

Par conséquent,

$$X \cdot 2^4 = X_3 \cdot 2^7 + X_2 \cdot 2^6 + X_1 \cdot 2^5 + X_0 \cdot 2^4, \tag{2.19}$$

et l'équation 2.16 devient

$$n = X_3 \cdot 2^7 + X_2 \cdot 2^6 + X_1 \cdot 2^5 + X_0 \cdot 2^4 + Y_3 \cdot 2^3 + Y_2 \cdot 2^2 + Y_1 \cdot 2^1 + Y_0 \cdot 2^0. \tag{2.20}$$

Autrement dit, n_{16} s'exprime en base 2 comme :

$$n_2 = \text{«} \underbrace{X_3X_2X_1X_0}_{=X} \underbrace{Y_3Y_2Y_1Y_0}_{=Y} \text{»}, \tag{2.21}$$

où l'on voit bien que l'expression binaire de n_{16} est simplement la concaténation des expressions binaires des chiffres X et Y.

Par exemple, la valeur $A7_{16}$ s'exprime en binaire comme 10100111_2 car $A_{16} = 1010_2$ et $7_{16} = 0111_2$.

La démonstration qui vient d'être donnée fonctionnerait pour toute base b qui est une puissance de 2. La méthode d'écriture des nombres par concaténation fonctionne donc également en octal. Par exemple, la valeur 57_8 s'exprime en binaire comme 101111_2 car $5_8 = 101_2$ et $7_8 = 111_2$.

2.3 Codage des entiers naturels

Les entiers naturels (nombres nuls ou positifs et sans chiffres après la virgule) peuvent être représentés de façon directe en binaire, en établissant une correspondance entre les chiffres binaires utilisés pour leur écriture et la séquence de bits stockés en mémoire.

Comme discuté au sein de la section 2.2.2, un point important à prendre en compte est le nombre de bits utilisés pour représenter l'entier. Par exemple, une séquence de 8 bits peut représenter $2^8 = 256$ valeurs différentes (de 0 à 255 si l'on choisit arbitrairement de commencer à zéro). Si l'on souhaite représenter davantage de valeurs différentes, il est nécessaire d'utiliser davantage de bits. Par exemple, une séquence de 16 bits peut représenter $2^{16} = 65536$ valeurs différentes, de 0 à 65535.

Cette distinction entre la séquence de bits codant le nombre, d'un côté, et le nombre représenté, de l'autre, peut paraître superflue tant que nous nous intéressons uniquement au codage des entiers naturels. Cependant, dès que nous aborderons les entiers relatifs, elle sera cruciale. En effet, la séquence de bits stockée en mémoire peut toujours s'interpréter comme un entier naturel, quelle que soit la convention utilisée pour en tirer une autre interprétation.

Par la suite, nous désignerons $cod_{\mathbb{N}_{(k)}}(n)$ la séquence de bits qui code le nombre n sur k bits. Inversement, nous désignerons $dec_{\mathbb{N}_{(k)}}(c)$ l'entier naturel codé par la suite de bits c de longueur k. Par exemple, $cod_{\mathbb{N}_{(8)}}(42) = 00101010$. Inversement, $dec_{\mathbb{N}_{(8)}}(00101010) = 42$. Ici, nous n'indiquons pas que la suite de bits est en base 2, précisément parce qu'il s'agit d'un état binaire et non pas d'un nombre exprimé dans une base donnée !

2.3.1 Taille de la représentation

En programmation, il est donc capital de préciser le nombre de bits dévolus à la représentation d'un nombre. Pour cela, on utilise des types de variables qui définissent entre autres la taille de la mémoire allouée pour stocker un nombre. Dans un langage comme le C, on rencontre des types numériques d'entiers tels que `char`, `short`, `int`, `long` ou `long long`. Étant donnée l'importance du langage C au sein de la programmation scientifique moderne, le tableau 2.3 résume les tailles minimales dans ce langage. De nombreux langages populaires dérivant du C, tels que C++ ou Java, ont également hérité de la nomenclature de certains de ces types de variables. D'autres langages, tels que Python, masquent volontairement cette distinction au programmeur.

TABLE 2.3 – Résumé des différentes tailles minimales des variables en C. En pratique, ces tailles peuvent être supérieures et dépendent de l'architecture utilisée et du compilateur. Pour cette raison, une taille habituelle est également donnée à titre indicatif. Certains types, tels que `long`, sont très dépendant de l'architecture et du langage. Enfin, il faut noter que sur les systèmes 32 bits, le type `long long` ne peut être représenté sur un seul mot mémoire.

Nom de type	`char`	`short`	`int`	`long`	`long long`
Taille minimale (bits)	8	16	16	32	64
Taille courante (bits)	8	16	32	32	64

Ces noms de types sont souvent couplé avec d'autres spécificateurs définissant la façon d'interpréter la variable, tels que `signed` et `unsigned`, dont nous parlerons plus loin.

Il est important de comprendre que le compilateur (cf. 6.1.2) détermine le nombre de bits sur lesquels les variables sont réellement codées (en adéquation avec l'architecture de la machine) ; les noms des types sont des conventions qui fixent un nombre minimum de bits mais ne garantissent pas la taille exacte. Par exemple, un `int` peut être codé sur 2 octets ; or, dans de très nombreux cas, il est codé sur 4 octets. L'annexe A propose un code d'exemple dans le langage C pour connaître la taille de certains types de données sur une architecture donnée ainsi que pour illustrer d'autres concepts développés dans les sections suivantes.

2.3.2 Gestion des débordements

Supposons que l'on effectue l'addition suivante des entiers naturels 129 et 7, chacun étant exprimé sur un octet. Cela s'écrit :

	1	0	0	0	0^1	0^1	0^1	1
+	0	0	0	0	0	1	1	1
=	1	0	0	0	1	0	0	0

Ici, le résultat $10001000_2 = 136$ ne déborde pas de l'octet car la colonne de poids fort (tout à gauche) ne contient pas de retenue. En revanche, que se passe-t-il si l'on additionne par exemple 150 et 150, toujours sur un octet ? En adaptant l'exemple précédent, on voit que le résultat, $100101100_2 = 300$, ne peut être représenté sur un octet, car il nécessite 9 bits pour être codé. On dit qu'il « **déborde** » ou qu'il « dépasse ». Il faut donc gérer les débordements (*overflow* en anglais) d'une façon ou d'une autre.

Dans le langage C, par exemple, un débordement d'entier relatif mène à un comportement dit « indéfini » car la façon dont le débordement est géré peut varier selon le processeur sur lequel il a lieu, et aucune règle générale n'est spécifiée quant à la manière de

traiter ce débordement dans le standard du langage [8]. Cela dit, nous discutons ci-dessous une méthode courante pour gérer les débordements ; il faut néanmoins garder à l'esprit qu'elle n'est pas universelle.

Règle de l'ignorance du bit de poids fort

Une manière de réguler ces débordements de façon arbitraire mais systématique est d'ignorer le bit qui déborde. Dans notre cas, cela signifie que la valeur obtenue après l'addition de 150 et 150 est $\not{1}00101100_2$, ce qui nous laisse avec $00101100_2 = 44$. Comme on le verra dans le chapitre 4.4.2 sur les circuits logiques, il est simple au niveau des circuits électroniques de détecter les débordements ; cependant, tous les langages de programmation ne permettent pas de le faire aisément.

La gestion des débordements par l'ignorance du bit de poids fort est motivée à la fois par le fait qu'elle soit simple et efficace à implémenter de façon électronique, et par le fait qu'elle soit reproductible, c'est-à-dire que deux résultats débordant de la même façon donneront le même résultat après gestion du débordement. Par ailleurs, il faut noter la nature cyclique ou **périodique** de l'arithmétique avec cette gestion du débordement. Par exemple, le bit qui déborde d'un octet étant toujours le neuvième, l'ignorer revient toujours à soustraire $2^{9-1} = 256$ du résultat. Ainsi, si l'on additionne 150 et 150 sur un octet, on obtient $150 + 150 - 256 = 44$. Or, dans le pire des cas où chaque octet à additionner est rempli de 1, on obtient $255 + 255 = 510$, ce qui tient également sur 9 bits. Ainsi, en additionnant deux nombres de n bits, le résultat est toujours un nombre occupant au maximum $n + 1$ bits. La méthode que l'on vient de décrire pour prévoir la valeur d'un entier naturel suite à un débordement fonctionne dans tous les cas où le débordement est géré par la méthode de l'ignorance du bit de poids fort. Comme nous le verrons plus loin, cela s'applique également aux soustractions.

Notons que, dans certains cas, un comportement cyclique peut être souhaitable pour des raisons de logique. Par exemple, si l'on code une donnée temporelle où l'entier 255 représente la dernière unité de temps avant le tour d'horloge complet, il est souhaitable que l'addition de 1 à 255 donne 0, car dans une horloge l'aiguille « revient » à zéro après avoir franchi sa « dernière » valeur.

De nombreux langages modernes permettent de travailler avec des nombres de taille « arbitraire » (dans les limites de la mémoire disponible). Cela signifie qu'un algorithme est implémenté au sein même du langage pour allouer l'espace nécessaire dans la mémoire de l'ordinateur afin de pouvoir manipuler les nombres demandés [9]. Mais, les ordinateurs étant optimisés pour travailler avec des mots de taille donnée, cette méthode n'exploite pas forcément au mieux les capacités matérielles de l'ordinateur, en plus d'introduire un traitement supplémentaire des nombres impliqués. Les opérations sur des nombres de taille

8. En C, il faut distinguer plusieurs situations. Lors d'une opération arithmétique sur des entiers signés, un résultat hors de l'intervalle représentable conduit à un comportement indéfini. En revanche, pour les entiers non signés, l'arithmétique est définie modulo 2^n avec n le nombre de bits du type, ainsi que discuté dans cette section. Voir `https://en.cppreference.com/w/c/language/conversion.html`, à la mention du terme « overflow ».

9. En Python par exemple, un entier est représenté par une liste de chiffres en base 2^15 ou 2^30 (selon l'architecture), chaque chiffre étant lui-même représenté par un entier « classique ».

fixe sont donc plus rapides que celles sur des nombres de taille variable (mais cet aspect de la performance n'est de loin pas toujours le plus important au sein d'un programme).

Sous-ensemble des entiers naturels

Il est courant de désigner par $\mathbb{N}_{(k)}$ l'ensemble des entiers naturels qui peuvent être représentés sur k bits en utilisant la convention discutée plus haut. Par exemple, $\mathbb{N}_{(8)}$ est l'ensemble des entiers naturels qui peuvent être représentés sur un octet, c'est-à-dire les entiers de 0 à 255. En général, $\mathbb{N}_{(k)}$ est l'ensemble des entiers naturels qui peuvent être représentés sur k bits, c'est-à-dire les entiers de 0 à $2^k - 1$:

$$\mathbb{N}_{(k)} = \{x \in \mathbb{N} | 0 \leq x < 2^k\}. \tag{2.22}$$

Comme discuté plus haut, deux nombres pris au sein de cet ensemble peuvent livrer un résultat qui n'en fait pas partie, lorsqu'on les additionne ou les soustrait.

2.4 Codage des entiers relatifs

Les nombres sans chiffres après la virgule mais possédant un signe (+ ou −) appartiennent à l'ensemble des entiers relatifs. Pour cette raison, on parle également d'« entier **signé** ». Pour ce qui est de leur codage, la différence avec les entiers naturels réside dans la façon de tenir compte du signe. Lorsque l'on considère non pas uniquement le stockage, mais également les possibilités d'appliquer un algorithme simple d'addition sur ces nombres, l'ajout du signe implique des modifications non triviales au codage des nombres. Nous allons considérer ici trois méthodes de codage des entiers relatifs en binaire. Afin de rendre la lecture plus aisée, nous annoncons que la troisième méthode, celle du complément à deux, est celle qui est la plus couramment utilisée en informatique pour représenter les entiers relatifs ; néanmoins, les deux autres, qui sont plus simples, nous aideront par la suite à comprendre la représentation des nombres à virgule, raison pour laquelle nous les étudions dès à présent.

En C et dans d'autres langages, les tailles des types discutés dans le tableau 2.3 peuvent être couplées avec le spécificateur `unsigned` afin de les interpréter comme des entiers naturels et non pas relatifs (ce dernier étant le cas par défaut). Ainsi, un `short` est un entier signé codé sur au moins deux octets, tandis qu'un `unsigned short` est un entier naturel codé sur au moins deux octets. Chacune de ces variables devrait donc en théorie pouvoir prendre 2^{16} valeurs différentes, mais tandis que le premier peut représenter des valeurs de -32768 à 32767, le second peut représenter des valeurs de 0 à 65535.

2.4.1 Codage signe-norme

Le signe étant lui-même une information binaire, il est tentant de représenter un entier signé comme un entier naturel auquel un bit de signe est ajouté (disons arbitrairement au début du mot, afin de ressembler à l'écriture mathématique manuscrite). Pour un mot d'un octet, un tel codage serait de la forme :

b_s	b_6	b_5	b_4	b_3	b_2	b_1	b_0

où b_6 à b_0 sont les bits de la **norme** du nombre, et b_s est le bit de **signe**. Fixons $b_s = 0$ pour le signe positif et $b_s = 1$ pour le signe négatif. En effet, cela nous permet d'obtenir le signe du nombre n avec la définition $sgn(n) = (-1)^{b_s}$. Ainsi, le nombre -42 s'exprime en binaire signe-norme sur un octet comme 10101010, tandis que le nombre 42 s'exprime comme 00101010. On voit ici l'importance de la convention pour interpréter correctement les nombres stockés, car 10101010 peut également être interprété comme un entier naturel valant 170_{10}, ou encore toute autre valeur suivant la convention utilisée (cf. sections suivantes). Le codage signe-norme d'un entier dont la valeur absolue peut se coder sur $k-1$ bits possède une longueur de k bits et s'exprime donc

$$cod_{\mathbb{Z}SN_{(k)}}(n) = \begin{cases} cod_{\mathbb{N}_{(k)}}(|n| + 2^{k-1}) & \text{si } n < 0, \\ cod_{\mathbb{N}_{(k)}}(n) & \text{sinon.} \end{cases} \tag{2.23}$$

En effet, le bit de poids fort étant utilisé comme bit de signe, il compte pour 2^{k-1} dans l'interprétation de la séquence de bits en tant qu'entier naturel. Le codage de $|n|$, quant à lui, est réalisé sur les $k-1$ bits de droite.

Pour décoder[10] une séquence binaire, on utilise alors la fonction inverse, en prenant garde à la valeur du bit de signe, en position $k-1$:

$$dec_{\mathbb{Z}SN_{(k)}}(c) = \begin{cases} 2^{k-1} - dec_{\mathbb{N}_{(k)}}(c) & \text{si } c_{k-1} = 1, \\ dec_{\mathbb{N}_{(k)}}(c) & \text{sinon.} \end{cases} \tag{2.24}$$

Notons que le nombre d'états binaires codables ne diffère pas de celui des entiers naturels. Dans les deux cas, il s'agit de 2^k états différents pour une séquence de longueur k. Cependant, la **plage de valeurs** exprimables est différente. Par exemple, une séquence de 8 bits peut exprimer des entiers naturels de 0 à 255, mais des entiers relatifs de -127 à 127 avec le codage dont on vient de parler. On remarquera que le premier cas correspond à un total de 256 valeurs différentes, tandis que le second correspond seulement à 255 valeurs différentes. En effet, une conséquence de la méthode de codage signe-norme est que le zéro n'est pas codé de façon unique, car il peut se coder $+0$ aussi bien que -0. En soi, ce « gaspillage » n'est pas préoccupant, mais il implique que l'opération qui teste si un nombre est nul est plus complexe, ce qui est davantage problématique.

Un autre inconvénient de la méthode est la gestion des opérations arithmétiques. On voit que, de façon générale, le codage du résultat de $(-n) + n$ n'est pas égal au codage de 0. Par exemple, pour effectuer l'addition $(-1) + (1)$ sur 3 bits, on ne peut se contenter de décoder le codage de leur addition : en effet, l'un étant codé 101 et l'autre étant codé 001, l'addition en colonne livre le résultat 110, ce qui devrait être décodé comme valant -2. On voit que l'on ne peut pas réutiliser le circuit qui permet d'additionner des entiers naturels afin d'additionner des entiers relatifs sans s'inquiéter de la nature des nombres impliqués. La méthode utilisée jusqu'ici (addition en colonnes avec retenue) n'est plus valable telle quelle avec le codage signe-norme.

10. Nous détaillons en annexe B la façon d'obtenir la fonction de décodage à partir de la fonction de codage

Une manière intuitive de comprendre l'origine de ce problème est de voir que ce codage implique un « changement de sens de l'addition », selon que le nombre à coder soit négatif ou positif. En effet, si l'on ajoute une unité au codage d'un nombre positif, celui-ci croît ; si l'on ajoute une unité au codage d'un nombre négatif, celui-ci décroît. On peut visualiser ceci en disposant les nombres représentés sur un cercle, afin de tenir compte de la règle de débordement cyclique discutée précédemment. La figure 2.1 illustre cela pour $k = 3$ bits. On voit que, outre la frontière de débordement entre -3 et 0, l'addition d'une unité à un nombre représente un déplacement dans le sens horaire du cercle pour les nombres positifs, et dans le sens anti-horaire pour les nombres négatifs. Or, on cherche un codage pour lequel il n'y a pas besoin de se soucier de la nature du nombre représenté (représentation dite « externe »), puisque la machine opère directement sur les états binaires (représentation dite « interne »).

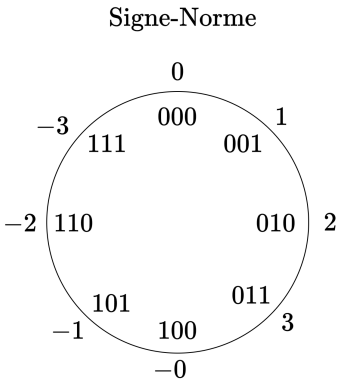


FIGURE 2.1 – Représentations externes et internes des nombres codés en signe-norme sur 3 bits, disposés sur un cercle afin de retranscrire la règle de débordement. Les circuits électroniques (par exemple pour additionner deux nombres) travaillent sur la représentation interne.

2.4.2 Codage avec biais

Une autre façon d'envisager les choses serait de voir les nombres signés comme des nombres naturels décalés (*offset* en anglais) d'une certaine quantité fixe. Par exemple, si l'on décale toute valeur de $+128$, alors -128 est représenté par 0, -127 par 1, -126 par 2, etc. Puisque, avec une séquence de k bits on peut représenter 2^k valeurs différentes, alors on doit décaler les nombres de $\frac{2^k}{2} = 2^{k-1}$ si l'on veut une répartition aussi équilibrée que possible entre les nombres positifs codables et les nombres négatifs codables.

Ainsi, un entier relatif n codé sur k bits est représenté par une séquence de bits équivalente au codage d'un entier naturel valant $n + 2^{k-1}$. Nous noterons ici $cod_{\mathbb{Z}B^-_{(k)}}(n))$ ce codage :

$$cod_{\mathbb{Z}B^-_{(k)}}(n) = cod_{\mathbb{N}_{(k)}}(n + 2^{k-1}), \tag{2.25}$$

Autrement dit, il faut retrancher 2^{k-1} de l'entier naturel codé en mémoire pour obtenir l'entier relatif à interpréter. Cette dernière relation permet de représenter davantage de valeurs négatives que positives. C'est pour cela que nous avons noté B^- ce codage. Il

existe une autre transormation qui permet également une répartition presque équilibrée entre les négatifs et les positifs, favorisant quant à elle les nombres positifs :

$$cod_{\mathbb{Z}B^{+}_{(k)}}(n) = cod_{\mathbb{N}_{(k)}}(n + 2^{k-1} - 1). \tag{2.26}$$

Par la suite, la variante $cod_{\mathbb{Z}B^{+}_{(k)}}$ sera impliquée dans le codage des nombre réels.

Il est possible d'exprimer la relation inverse de chaque variante, qui permet d'obtenir l'entier relatif à interpréter à partir de la représentation interne c :

$$dec_{\mathbb{Z}B^{-}_{(k)}}(c) = dec_{\mathbb{N}_{(k)}}(c) - 2^{k-1}, \tag{2.27}$$

et

$$dec_{\mathbb{Z}B^{+}_{(k)}}(c) = dec_{\mathbb{N}_{(k)}}(c) - 2^{k-1} + 1. \tag{2.28}$$

Par exemple, si l'on utilise le codage 2.26 pour coder sur 3 bits la valeur $n = 2$ en tant qu'entier relatif, il faut écrire en mémoire une séquence de bits qui correspond à $cod_{\mathbb{Z}B^{+}_{(3)}}(2) = cod_{\mathbb{N}_{(3)}}(2 + 2^{3-1} - 1) = cod_{\mathbb{N}_{(3)}}(5)$. La séquence de bits codée est donc 101. Inversement, pour décoder la séquence de bits 101 en tant qu'entier relatif, il faut effectuer le calcul $dec_{\mathbb{Z}B^{+}_{(3)}}(101) = dec_{\mathbb{N}_{(3)}}(101) - 2^{3-1} + 1 = 5 - 2^2 + 1 = 2$. L'annexe B illustre la façon d'obtenir une fonction de décodage à partir de son inverse.

Le nombre total de valeurs représentables étant nécessairement une puissance de 2, c'est-à-dire un nombre pair, la présence du zéro introduit forcément un déséquilibre entre le nombre de valeurs négatives et le nombre de valeurs positives représentables avec la méthode du biais. L'un ou l'autre des codages précédents doit être choisi de façon arbitraire. Le tableau 2.4 ci-dessous permet de comparer les deux codages possible sur un exemple où $k = 3$ bits.

TABLE 2.4 – Comparaison du décodage de séquences de 3 bits interprétés comme des entiers naturels (seconde ligne) ou comme des entiers relatifs utilisant les biais des équations 2.25 et 2.26 (troisième et quatrième lignes).

Séquence de bits c	000	001	010	011	100	101	110	111
$dec_{\mathbb{N}_{(3)}}(c)$	0	1	2	3	4	5	6	7
$dec_{\mathbb{Z}B^{-}_{(3)}}(c)$	-4	-3	-2	-1	0	1	2	3
$dec_{\mathbb{Z}B^{+}_{(3)}}(c)$	-3	-2	-1	0	1	2	3	4

Avec cette méthode, le signe est codé implicitement au sein de la séquence de bits et le zéro est bien représenté de façon unique. Par ailleurs, on voit que cette fois-ci l'addition d'une unité sur la représentation interne des nombres implique toujours un passage à la valeur suivante dans l'ordre cyclique de la représentation externe, ce qui règle également le second problème du codage signe-norme. La figure 2.2 illustre la représentation externe et interne des nombres codés en utilisant le biais $B^{-}_{(3)}$, disposés sur un cercle pour retranscrire la règle de débordement.

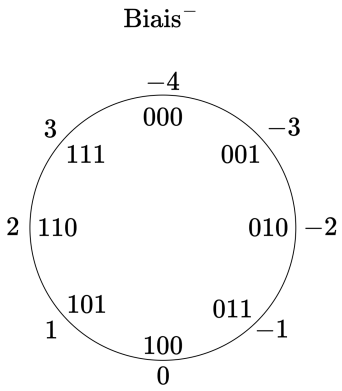


FIGURE 2.2 – Représentations externes et internes des nombres codés avec la méthode $cod_{\mathbb{Z}B^-_{(k)}}$ sur 3 bits, disposés sur un cercle afin de retranscrire la règle de débordement.

Malheureusement, cette méthode n'est toujours pas idéale pour l'unification des opérations d'addition sur les nombres positifs et négatifs. En effet, $cod(-n)+cod(n) \neq cod(0)$. Pour s'en convaincre, considérons l'addition de 3 et de -3 sur 3 bits avec le codage $\mathbb{Z}B^+_{(k)}$. Le résultat de cette addition est l'entier relatif 0, qui devrait être codé comme l'entier naturel $0+2^{3-1}-1=3=011_2$. Le codage de 3 est celui de l'entier naturel $3+2^{3-1}-1=6=110_2$. Le codage de (-3) est celui de l'entier naturel $(-3)+2^{3-1}-1=0=000_2$. Or, l'addition de leurs représentations internes donne donc $110_2 \neq cod(0)$. Encore une fois, il faudrait donc appliquer des modifications supplémentaires pour décoder correctement le résultat ; le circuit électronique d'addition ne peut être utilisé indifféremment pour opérer sur des entiers naturels ou relatifs. Il est envisageable de construire un algorithme pour travailler avec des nombres codés de cette façon, mais il serait préférable d'avoir une façon unifiée de traiter l'addition des nombres quel que soit leur signe. Remarquons que $cod(n)+cod(0) \neq cod(n)$ avec la méthode du biais. Cela nous donne un indice sur la prochaine modification à apporter, que nous utilisons dans la section à venir. Nous ne disons pas que c'est là une condition suffisante pour l'unicité de l'algorithme d'addition-soustraction, mais c'est assurément une condition nécessaire.

2.4.3 Complément à 2

La méthode du complément à 2, dont le nom sera expliqué plus bas, est une méthode de codage des entiers relatifs en binaire qui résout les problèmes du codage signe-norme ainsi que ceux du codage avec biais.

En résumé, nous sommes à la recherche d'une méthode pour laquelle :

1. Chaque nombre est codé de façon unique (pas respecté par le codage signe-norme).
2. La relation $cod(-n)+cod(n)=cod(0)$ est vérifiée (respecté ni par le codage signe-norme, ni par celui du biais).

Principe

Comme discuté plus haut, le fait que, dans la méthode du biais, $cod_{\mathbb{Z}B_{(k)}}(0) \neq cod_{\mathbb{N}_{(k)}}(0)$ est problématique. Nous essayons donc de faire « tourner » la représentation externe des nombres en figure 2.2, de sorte à ce que les représentations du zéro coïcindent. Après $2^k/2$ rotations d'une unité, on obtient la répartition présentée en figure 2.3 ci-dessous.

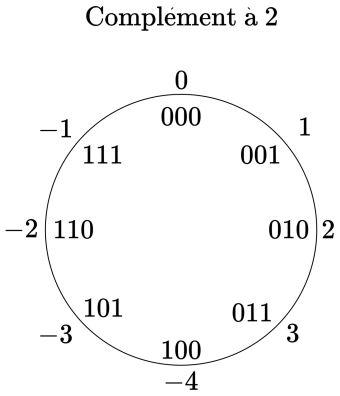


FIGURE 2.3 – Représentations externes et internes des nombres codés avec la méthode du complément à 2 sur 3 bits, disposés sur un cercle afin de retranscrire la règle de débordement.

Le nouveau codage obtenu arbore un caractère hybride relativement aux méthodes vues précédemment, car il utilise un « bit de signe », mais le cycle des nombres correspond à celui de la méthode du biais. Ici, le bit de poids fort joue le rôle d'indicateur de signe tout comme dans le codage signe-norme, tout en restant intégré à la valeur du mot, contrairement au codage signe-norme.

On se convainc également qu'à présent, il semble que $cod(n) + cod(-n) = cod(0)$, ce que nous démontrerons plus loin. Ainsi, pour additionner deux nombres a et b, il semble que l'on puisse utiliser le même circuit électronique, que les nombres soient naturels ou relatifs. En effet, le circuit n'a pas besoin de traiter d'information relative à la nature des nombres impliqués, il se contente de traiter des nombres dans leur représentation interne ; c'est l'interprétation (décodage) qu'on en fait (naturel ou relatif) qui détermine leur représentation externe.

Pour effectuer la soustraction $a - b = a + (-b)$, il faut néanmoins d'abord pouvoir trouver l'opposé de b, c'est-à-dire calculer le codage de $-b$ comme un nombre relatif en utilisant notre nouvelle méthode. Puisque la plage de valeurs représentables est la même que dans le codage avec biais [11], c'est-à-dire $\mathbb{Z}_{(k)} = \{x \in \mathbb{Z} | -2^{k-1} \leq x < 2^{k-1}\}$, le codage de l'opposé d'un nombre n vaut $cod_{\mathbb{Z}_{(k)}}(-n) = cod_{\mathbb{N}_{(k)}}(2^k - n)$. Malheureusement, on ne peut utiliser directement cette dernière expression, puisqu'elle contient une soustraction et que nous cherchons justement à caractériser la méthode de soustraction de deux nombres.

11. En revanche, cette fois on observe (cf. figure 2.3) que les nombres et leurs opposés sont distribués symétriquement par rapport au zéro, hormis pour le nombre le plus négatif, qui n'a pas d'opposé.

Néanmoins, on peut observer que $2^k - n = 2^k - 1 + 1 - n$, ce qui peut se voir comme :

$$\underbrace{2^k - 1}_{111\ldots1_2 \text{ avec } k \text{ bits}} \quad -n + 1 = \overline{n} + 1, \tag{2.29}$$

où l'on comprend que $\overline{n} = 2^k - 1 - n$ est une inversion des bits de n. En effet, $2^k - 1$ étant une suite de k bits sont à 1, lui soustraire une séquence de bits quelconque n donnera les valeurs 0 là où les bits de n sont à 1, et laissera 1 là où les bits de n sont à 0. Ainsi, pour obtenir l'opposé d'un nombre n dans ce contexte, il suffit d'inverser tous ses bits, puis d'ajouter 1. C'est une chose à garder en tête lorsqu'on exprime le codage en complément à 2 sous la forme qui contient une soustraction, puisque cela nous fournit une procédure pour obtenir l'opposé d'un nombre sans explicitement effectuer de soustraction.

Pour résumer formellement, la règle du complément à 2 sur k bits permet de coder les nombres appartenant à l'ensemble $\mathbb{Z}_{(k)} = \{x \in \mathbb{Z} | -2^{k-1} \leq x < 2^{k-1}\}$ et s'écrit :

$$cod_{\mathbb{Z}_{(k)}}(n) = \begin{cases} cod_{\mathbb{N}_{(k)}}(2^k - |n|) & \text{si } n < 0, \\ cod_{\mathbb{N}_{(k)}}(n) & \text{sinon.} \end{cases} \tag{2.30}$$

Inversement, on a :

$$dec_{\mathbb{Z}_{(k)}}(c) = \begin{cases} dec_{\mathbb{N}_{(k)}}(c) - 2^k & \text{si } c_{k-1} = 1, \\ dec_{\mathbb{N}_{(k)}}(c) & \text{sinon.} \end{cases} \tag{2.31}$$

Prenons un exemple. Pour obtenir le codage du nombre -1 sur 3 bits, on inverse les bits du codage de l'entier naturel correspondant à sa valeur absolue ($|-1| = 1 = 001_2$) pour obtenir 110, puis on ajoute 1 pour obtenir 111. On peut également adopter l'approche calculatoire grâce à la relation 2.30 ci-dessus : $cod_{\mathbb{Z}_{(3)}}(-1) = cod_{\mathbb{N}_{(3)}}(2^3 - |-1|) = cod_{\mathbb{N}_{(3)}}(7)$.

Inversement, pour décoder la séquence 111, on commence dans tous les cas par remarquer qu'il est négatif en raison de son bit de poids fort. Ensuite, on peut soustraire 1 puis inverser les bits (en n'oubliant pas que le nombre est négatif à la fin), ou bien utiliser la relation 2.31 ci-dessus : $dec_{\mathbb{Z}_{(3)}}(111) = dec_{\mathbb{N}_{(3)}}(111) - 2^3 = 7 - 8 = -1$. Notons que, puisque appliquer le complément à 2 à la quantité n revient à calculer $2^k - n$, appliquer le complément à 2 deux fois de suite revient à calculer $2^k - (2^k - n) = n$. Autrement dit, si le bit de poids fort d'une séquence est à 1, la machine peut simplement calculer son complément à 2 pour obtenir la valeur absolue de l'entier relatif qu'elle code (il n'y a toujours pas de soustraction à effectuer), à une exception près. En effet, tout comme avec la méthode du biais, il y a toujours avec le complément à 2 un nombre qui n'a pas d'opposé dans $\mathbb{Z}_{(k)}$: il s'agit du nombre -2^{k-1}, qui est codé par une séquence de k bits égale à $100\ldots0$.

Munis des relations précédentes, on peut prouver que $cod(n) + cod(-n) = cod(0)$ avec ce codage. En effet, si n est positif, alors $cod(n) = cod_{\mathbb{N}_{(k)}}(n)$ et $cod(-n) = cod_{\mathbb{N}_{(k)}}(2^k - n)$, et donc $cod(n) + cod(-n) = cod_{\mathbb{N}_{(k)}}(n) + cod_{\mathbb{N}_{(k)}}(2^k - n)$. Or, comme nous l'avons vu ci-dessus, la valeur de $2^k - n = \bar{n} + 1$ s'écrit en binaire comme une inversion des bits de n à laquelle on doit encore ajouter une unité. Dès lors, $cod_{\mathbb{N}_{(k)}}(n) + cod_{\mathbb{N}_{(k)}}(2^k - n)$ s'écrit $n + \bar{n} + 1$. Avant d'ajouter l'unité, cela nous donne donc une séquence de k bits valant tous

1. Après avoir ajouté l'unité, on obtient alors une séquence de $k+1$ bits égale à $100\ldots0$; quand la règle de débordement est appliquée, cela revient à la façon de coder le zéro.

Le terme de « complément à deux » résulte d'un raccourci de langage. Dans une base b donnée, le complément d'un nombre n écrit avec k chiffres est défini comme la valeur m qu'il faut ajouter à ce nombre pour atteindre b^k. Autrement dit, $n+m=b^k \iff m=b^k-n$. La reformulation $m=(b^k-1)-n+1$ fait apparaître un terme (entre parenthèses) qui est pratique, parce qu'il est constitué de k fois le plus grand chiffre de la base. Par abus de langage, on appelle ce terme le « complément à $b-1$ » du nombre n, qui en base 2 correspond à l'inversion des bits, ou conjugué. Par ailleurs, on peut imaginer qu'un nom tel que « codage signe-norme réarrangé » ou encore « codage par biais réarrangé », suivant le raisonnement que nous avons vu, serait plus parlant.

Exemple

Afin de nous familiariser avec la méthode, prenons un exemple quelconque d'addition sur un octet et effectuons l'addition $-42+7$ en complément à 2. Le codage de -42 en complément à 2 est 11010110 car $2^8-42=214=11010110_2$ et celui de 7 est 00000111. L'addition donne alors :

	1	1	0	1	0^1	1^1	1	0
+	0	0	0	0	0	1	1	1
=	1	1	0	1	1	1	0	1

Le résultat se lit donc : 11011101. Interprété avec la règle du complément à deux, on obtient $-(00100010+1)_2=-00100011_2=-35$.

Afin de considérer un cas avec débordement issu d'une soustraction, prenons également pour exemple l'addition $-42-126$ sur un octet. Le résultat, -168, excède la valeur minimale possible avec le complément à 2 sur un octet, à savoir -128. En effet :

	1	1	0	1	0^1	1^1	1	0
+	1	0	0	0	0	0	1	0
=	0	1	0	1	1	0	0	0

Le bit de poids fort aurait dû être écrit sur un neuvième bit, mais ce bit n'existe pas. Le résultat tel qu'on peut le lire ici vaut 88, ce qui vaut $2^8-168=88$. Ainsi, le caractère cyclique discuté en section 2.3.2 se retrouve pour les nombres négatifs codés en complément à 2. On peut aisément vérifier que si l'on additione la valeur 1 à la valeur maximale codable en complément à 2 sur un octet, c'est-à-dire 01111111, on obtient 10000000, ce qui s'interprète comme -128.

2.5 Codage des nombres réels

Si, comme nous l'avons vu dans la section précédente, les nombres entiers peuvent être représentés à l'aide de deux informations (un signe et une norme), les nombres réels, eux, nécessitent une troisième information. Ils sont en effet constitués d'un signe, d'une partie entière et d'une partie fractionnaire.

Il est légitime de se demander pourquoi la norme d'un réel ne peut pas être considérée comme incluant sa partie fractionnaire (son développement décimal). Une différence fondamentale, cependant, les sépare : comme discuté en section 2.1.3, la partie fractionnaire d'un nombre exprimé dans une base donnée peut parfois être finie, mais elle ne l'est pas nécessairement dans une autre base, dans le cas général. Par exemple, pour exprimer le nombre exact 42.301 en base 7 comme nous l'avons fait en section 2.1.3, il a fallu se fixer un degré de précision arbitraire, sans quoi le développement des chiffres après la virgule peut se poursuivre indéfiniment. Il en va de même pour la valeur décimale 0.3 quand elle est exprimée en base 2.

Une autre façon de se convaincre du plus grand nombre d'informations intrinsèques contenues au sein d'un réel est de considérer le problème des nombres irrationnels. Pour ces nombres, il n'existe pas de base au sein de laquelle le nombre de chiffres après la virgule est fini. Il faut donc arbitrairement décider d'une précision pour les représenter au sein de la mémoire d'un ordinateur, quelle que soit la place dont on dispose et quelle que soit la base entière choisie. Dès lors, l'expression d'un nombre réel peut se faire à l'aide d'un signe, d'une norme entière et d'une précision (ou nombre de chiffres après la virgule), qui comme nous le verrons peut être vue comme un facteur de mise à l'échelle à appliquer à la norme.

2.5.1 Virgule fixe

La représentation en virgule fixe est sans doute la plus intuitive vis-à-vis de la définition suivante de l'approximation d'un nombre réel en binaire :

$$n = s\left(\sum_{i=0}^{k-1} a_i 2^i + \sum_{i=1}^{\ell} f_i 2^{-i}\right), \tag{2.32}$$

où s est le signe et où la somme impliquant les quantités a_i représente la partie entière du nombre, tandis que celle impliquant les quantités f_i représente sa partie fractionnaire. Ainsi, un nombre donné est représenté par $k+\ell+1$ valeurs. Notons qu'ici, pour des raisons de commodité, nous avons choisi de commencer l'indexation de la partie fractionnaire à 1 afin que, à l'instar de ce qui se fait pour la partie entière, le chiffre numéro i corresponde à la puissance i de la base (au signe près). Remarquons enfin que l'équation qui précède découle directement de la définition de l'expression d'un nombre réel dans une base entière, que nous avons vue en section 2.1.2.

Une façon intuitive de représenter l'approximation d'un réel est donc de fixer à l'avance k et ℓ. Par exemple, on pourrait décider qu'un nombre réel approximé sur un octet suit la

structure suivante, où l'on a $k = 3$ bits pour la partie entière et $\ell = 4$ bits pour la partie fractionnaire :

s	a_2	a_1	a_0	f_1	f_2	f_3	f_4

Cette méthode est dite à « virgule fixe » parce qu'une fois que l'on a fixé les valeurs de ℓ et de k, tous les nombres que l'on code possèdent le même nombre de chiffres après la virgule. L'avantage de cette méthode est sa simplicité et le fait qu'elle permette une représentation « sur mesure » de quantités spécifiques. Par exemple, si l'on est certain que toutes les quantités que notre machine doit traiter possèdent une partie entière à trois chiffres maximum, alors on peut fixer $k = 3$ et adapter les autres paramètres de la représentation en conséquence, en exploitant au mieux l'espace mémoire disponible et en garantissant que tous les nombres sont exprimés avec la même précision.

En raison de sa simplicité et pour des raisons techniques, la représentation en virgule fixe a été utilisée sur de nombreuses calculatrices par le passé. Cependant, le défaut de la méthode est évident : en pratique, il est très utile de pouvoir coder des valeurs dont le nombre de chiffres après la virgule sont différents les uns des autres. Autrement dit, le nombre de chiffres significatifs des nombres exprimés est variable si la virgule est fixe. Pour cette raison, la représentation en virgule flottante, que nous allons discuter ci-dessous, est devenue la norme dans le monde de l'informatique pour la représentation des nombres réels.

2.5.2 Virgule flottante

Principe

Une autre façon de représenter une quantité telle que définie dans l'équation 2.32 serait de coder l'emplacement de la virgule au sein même du nombre. Par exemple, en base 10, l'approximation $\pi \approx 3.141$ pourrait être représentée à l'aide de l'entier 3141 et de la « mise à l'échelle » 10^{-3}, ce qui revient à indiquer que la virgule se trouve après le troisième chiffre en partant de la droite. On parle alors de virgule **flottante** parce que l'emplacement du séparateur entre la partie entière et la partie fractionnaire n'est pas fixé une fois pour toutes et pour tous les nombres, mais codé au sein même du nombre représenté.

Ainsi, en représentation en virgule flottante, on ne choisit plus un nombre ℓ de bits après la virgule, mais plutôt un nombre ℓ de bits servant à **coder la position de la virgule**. Par exemple, sur un octet avec $\ell = 3$, on pourrait avoir la structure suivante, où l'on nomme v_i les bits servant à coder la position de la virgule :

s	a_3	a_2	a_1	a_0	v_3	v_2	v_1

Dans cet exemple, on voit que la position de la virgule peut posséder 2^3 configurations différentes, tandis que les chiffres du nombre que l'on exprime peuvent posséder 2^4 configurations différentes. Evidemment, le nombre total de valeurs représentables est exactement

le même qu'en virgule fixe sur un mot de même longueur, c'est-à-dire 2^8 dans cet exemple, mais la plage des valeurs représentables est différente et dépend de certains paramètres de la représentation que nous discutons plus bas.

Mantisse et exposant

Comme nous l'avons illustré précédemment (ainsi qu'au sein des sections 2.1.6 et 2.1.7), indiquer la position de la virgule au sein d'un nombre revient à multiplier un entier par un facteur de mise à l'échelle. En décimal, nous avons pris pour exemple $3.141 = 3141 \cdot 10^{-3}$. Bien entendu, cette façon d'exprimer les nombres est valable en base 2 également [12]. Par exemple : $1.1011_2 = 11011_2 \cdot 2^{-4}$. Ainsi, de façon générale, un nombre réel n peut être exprimé comme suit :

$$n = s \cdot m \cdot b^E, \tag{2.33}$$

où s est le signe, m est la **mantisse** et E est l'**exposant**. La mantisse est la partie du nombre qui spécifie les chiffres composant ce dernier, tandis que l'exposant est la partie du nombre qui spécifie la position de la virgule au sein de la mantisse. Par exemple, pour le nombre -312.7 en base 10, si l'on décide que $1 \leq m < 10$, alors le signe est -1, la mantisse est 3.127 et l'exposant est 2.

On reconnaît au sein de l'équation 2.33 la structure d'un nombre exprimé en notation scientifique ; en l'occurrence, il s'agirait de notation scientifique décimale, si $1 \leq m < 10$ et $b = 10$. La notation scientifique est extrêmement utilisée (en particulier dans les sciences appliquées) pour exprimer des nombres très grands ou très petits, car elle permet une séparation entre l'ordre de grandeur d'un nombre et les chiffres significatifs de ce dernier. Autrement dit : elle permet d'exprimer uniquement les informations importantes à propos d'une quantité donnée, en faisant fi de la redondance inhérente à la notation décimale classique. C'est précisément ce que nous allons exploiter ici pour représenter des nombres réels en binaire : la mantisse contient uniquement les chiffres du nombre, tandis que l'exposant indique l'ordre de grandeur du nombre. Ainsi, la précision est liée au contenu de la mantisse, tandis que la taille, ou ordre de grandeur, est liée à l'exposant.

Notation scientifique en base 2 et choix de codage

Le fait de contraindre $1 \leq m < b$ signifie, pour le binaire, que le premier chiffre de la mantisse est forcément 1. Ainsi, une mantisse codée par la séquence de bits c_m de longueur k_m correspond à la valeur :

$$m = 1 + \sum_{i=1}^{k_m} c_{m,i} 2^{-i}, \tag{2.34}$$

Par exemple, si la mantisse vaut 1.1011, on la codera 1011, puisqu'il est inutile de préciser que le premier chiffre est 1.

12. Notons au passage que, la base b étant toujours codée 10_b, le facteur de mise à l'échelle est toujours une puissance de 10_b si le nombre est intégralement exprimé en base b. Ainsi, par exemple, $11.011_2 = 11011_2 \cdot 10_2^{-11_2}$.

Finalement, il nous reste à contraindre la valeur de E, qui est un entier relatif. En effet, pour pouvoir représenter des nombres plus petits que la base b, E doit pouvoir être négatif. Pour pouvoir représenter nombres plus grands que la base b, E doit pouvoir être positif. Il est alors tentant de coder la valeur de E en utilisant le complément à deux que nous avons précédemment défini pour les entiers relatifs. Cependant, pour des raisons techniques, la norme IEEE 754, qui est la norme la plus couramment utilisée pour la représentation des nombres réels en informatique, utilise la méthode du biais dont nous avons parlé précédemment. Il est important de garder en tête que les raisons qui ont mené à ce choix ne sont pas fondamentales mais pratiques ; l'important est bien de pouvoir coder E comme un entier relatif.

La méthode de codage par biais correspondant à la fonction 2.26 est choisie pour représenter l'exposant. On parle alors d'**exposant biaisé**. Ainsi, si la valeur de l'exposant est E, son codage sur k' bits est donné par :

$$cod(E) = cod_{\mathbb{Z}B^+_{(k')}}(E) = cod_{\mathbb{N}_{(k')}}(E + 2^{k'-1} - 1), \tag{2.35}$$

tandis que le décodage de la séquence binaire b_E associée à l'exposant est donné par :

$$dec(b_E) = dec_{\mathbb{Z}B^+_{(k')}}(b_E) = dec_{\mathbb{N}_{(k')}}(b_E) - 2^{k'-1} + 1. \tag{2.36}$$

Par exemple, si l'exposant vaut $E = 12$, alors son codage sur 8 bits est $cod(E) = cod_{\mathbb{N}_{(8)}}(2^{8-1} + 12 - 1) = cod_{\mathbb{N}_{(8)}}(139) = 10001011$. Lorsqu'on fixe le nombre de bits sur lesquels l'exposant est codé, on fixe également le biais. Sur un octet, le décalage au moment du décodage vaut toujours $-2^{8-1} + 1 = -127$.

L'ensemble des choix de codage qui sont présentés ci-dessus correspond à la norme IEEE 754, qui est la norme la plus couramment utilisée pour la représentation des nombres réels en informatique.

Avant de donner un exemple concret de la norme IEEE 754, il nous reste à discuter du nombre de bits dévolus au codage de chaque composante du nombre à approximer.

Formats de réels

Comme on vient de le voir, le standard doit fixer le nombre de bits consacrés au codage des différentes parties du nombre. La norme IEEE 754 définit plusieurs formats de nombres réels, en fonction du nombre de bits totaux utilisés. Les formats les plus courants sont les suivants :

- **Simple précision (32 bits)** : 1 bit de signe, 8 bits pour l'exposant et 23 bits pour la mantisse. Ce format est souvent appelé ***float*** en informatique. Dans ce cas, le décalage appliqué au biais est de 127. Cela permet d'exprimer des nombres décimaux comportant au plus 7 chiffres significatifs.
- **Double précision (64 bits)** : 1 bit de signe, 11 bits pour l'exposant et 52 bits pour la mantisse. Ce format est souvent appelé ***double*** en informatique. Dans ce cas, le décalage appliqué au biais est de 1023. Cela permet d'exprimer des nombres décimaux comportant au plus 16 chiffres significatifs.

Notons que si l'on décodait en tant qu'entier naturel la valeur d'une mantisse de k_m bits, alors celle-ci prendrait au maximum la valeur $n_m = 2^{k_m} - 1$, à laquelle on doit ajouter 1 pour le bit implicite. Comme on l'a vu plus tôt dans ce document, le nombre de chiffres nécessaires pour écrire cette valeur en base 10 est alors donné par $\lfloor \log_{10}(2^{k_m}) \rfloor + 1$. Cela explique les valeurs données ci-dessus quant au nombre de chiffres décimaux significatifs permis par les deux standards présentés.

Afin de donner un exemple concret de codage en virgule flottante, la table 2.5 ci-dessous illustre la séquence de bits correspondant au codage du nombre 9.125 en simple précision.

TABLE 2.5 – Exemple de nombre réel codé avec la méthode de la virgule flottante sur 32 bits. Le bit de signe est affiché en gris, les 8 bits de l'exposant biaisé en bleu et les 23 bits de la mantisse en noir. Ici, le signe vaut $(-1)^0 = +1$, l'exposant $cod(E) = 10000010$ correspond à $E = 130 - 2^7 + 1 = 3$ et la mantisse à $1 + 2^{-3} + 2^{-6} = 1.140625$. Le nombre exprimé vaut donc $+1.140625 \cdot 2^3 = 9.125$.

0	1	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Erreurs d'arrondi

Comme discuté au sein de la section 2.1.3, le développement de la partie fractionnaire d'un nombre n'est pas nécessairement fini dans une base donnée, quand bien même il l'est dans une autre. Par ailleurs, si le développement est fini, il peut être plus long que le nombre de bits dévolus à la mantisse. Pour ces raisons, des erreurs d'arrondi peuvent apparaître lors du codage d'un nombre réel en virgule flottante ; si les nombres qui contiennent ces erreurs sont impliqués dans des calculs, ces erreurs peuvent alors se propager et affecter le résultat final de façon très significative. L'annexe C donne des exemples variés de ce qui est discuté dans cette section du document.

Prenons l'exemple du nombre $0.3 = 3/10$. En base 10, son développement fractionnaire est fini[13] car $0.3 = 3 \cdot 10^{-1}$. En base 2, cependant, il n'est pas possible de l'exprimer exactement à l'aide d'une mantisse m finie. Examinons l'approximation obtenue avec une mantisse de 23 bits. Le lecteur peut se convaincre que le codage correspondant au nombre

13. Comme on l'a vu, une multiplication par 2^k revient à décaler la virgule de k chiffres. Les nombres dont le développement fractionnaire en base 2 est fini sont ceux qui s'expriment sous la forme d'une fraction simplifiée du type $n = \frac{x}{y}$, où $x \in \mathbb{N}$ et y est une puissance de 2. De façon générale, si le nombre fractionnaire $n = \frac{x}{y}$ sous forme irréductible possède un développement décimal fini de longueur k en base b, cela signifie que $n \cdot b^k = (xb^k)/y$ n'est pas fractionnaire. Par conséquent, tous les facteurs premiers de y doivent se trouver dans xb^k. Mais comme par définition x et y n'ont pas de facteurs communs, cela signifie que tous les facteurs premiers de y font également partie de ceux de b. Par exemple, 3/4 a un développement fini en base 10 car c'est une fraction simplifiée dont le dénominateur ($4 = 2^2$) ne possède que des facteurs premiers qui sont également des facteurs de la base $b = 10 = 2 \cdot 5$. Ainsi, il est certain qu'il existe une puissance k à laquelle élever la base de telle sorte que 2^2 compte parmi les facteurs de 10^k ; autrement dit, il est certain que l'on puisse décaler la virgule un nombre fini de fois pour obtenir un nombre entier à partir de n fractionnaire.

décimal 0.3 est

$$\underbrace{0}_{cod(s)}\ \underbrace{01111101}_{cod(E)}\ \underbrace{00110011001100110011010}_{cod(m)}, \tag{2.37}$$

et le nombre approximé qui en découle vaut donc $n = 0.30000001192092895507812 5$, soit une erreur relative de l'ordre de 10^{-8}.

Pour comprendre l'impact de ce type d'erreurs, supposons que n soit amené à être additionné de multiples fois à d'autres valeurs, elles-mêmes réinjectées dans les calculs. L'erreur d'arrondi initiale peut alors se propager, s'amplifier, et affecter significativement le résultat final. Nous donnons en annexe C un exemple de code illustrant ce phénomène. En particulier, on montre que si l'on additionne 0.2 (codé en simple précision) à lui-même dix million de fois, le résultat vaudra 2'175'874 au lieu des 2 millions attendus. On peut créer une erreur relative quasiment arbitrairement grande en choisissant le nombre d'itérations (ici, « seulement » un million) ainsi que la nature des opérations appliquées aux nombres. Nous encourageons le lecteur à tester et modifier le code donné en annexe C pour s'en convaincre.

Enfin, remarquons que lorsque deux nombres proches l'un de l'autre sont soutraits, le résultat est un nombre très proche de zéro. Ainsi, les éventuels bits erronés issus d'arrondis précédents endossent une importance relative qui peut être grande, quand bien même la soutraction elle-même est exacte. Prenons un exemple où $A = 25.4$ et $B = 25.6$. Ainsi, $\Delta = B - A = 0.2$. Si le codage de A et B impose une approximation (disons à l'unité pour simplifier), alors $A' = 25$ et $B' = 26$. Le résultat de la soustraction est alors $\Delta' = 1$. Ainsi, alors même que l'approximation de A et B est plutôt bonne ($A'/A \approx 98\%$ et $B'/B \approx 102\%$) l'approximation de leur différence est mauvaise : $\Delta'/\Delta = 500\%$. Ce phénomène est connu sous le nom d'« annulation catastrophique » et doit parfois être pris en compte en programmation scientifiques. De nombreux problèmes de physique, par exemple, peuvent être simulés en faisant évoler des valeurs *via* une relaxation vers un équilibre : $x' = f(\lambda \cdot (x - x_0))$, où x et x_0 peuvent être extrêmement proches. L'annexe C.4 donne un exemple de code illustrant une annulation catastrophique.

Nombres dénormalisés et nombres spéciaux

Il est important de noter qu'au sein de la norme IEEE 754, tous les nombres pour lesquels le codage de l'exposant est composé uniquement de zéros mais pas la mantisse sont des nombres dits **dénormalisés**. Il a été décidé que cette plage de nombres devait rompre avec la convention décrite plus haut. Ces nombres dénormalisés (on devrait plutôt dire « codage dénormalisé ») sont à la place utilisés pour représenter des nombres très petits. Dans un tel cas, on utilise une technique similaire à celle du codage en virgule fixe, c'est-à-dire que l'exposant est fixé à une valeur spécifique [14] et la mantisse est codée telle qu'elle est, sans le bit implicite de poids fort valant 1. Ainsi, le codage dénormalisé autorise une moins grande souplesse de précision ainsi qu'un moins grand nombre de chiffres significatifs que le codage normal du standard, mais permet en échange de représenter des quantités très petites. Cependant, en raison de l'augmentation de complexité engendrée par la cohabitation entre nombres normalisés et dénormalisés, leur manipulation explicite

14. Par exemple, $E = -126$ en simple précision et $E = -1022$ en double précision, car il s'agit là, comme on le voit en détail plus loin, des valeurs minimales de l'exposant pour les nombres normaux.

en programmation scientifique est peu commune et nous n'en discuterons pas plus en détail dans ce cours. Gardons cependant en tête que la façon de coder les nombres que nous avons vue dans les sous-sections précédentes concerne les nombres à virgule flottante **normalisés**.

Par ailleurs, la norme IEEE 754 permet de représenter des nombres spéciaux tels que le zéro exact, $+\infty$, $-\infty$ et **N**ot-**a**-**N**umber, abbrégé `NaN`. Pour ce faire, des valeurs spécifiques de l'exposant et de la mantisse ont été réservées. Ces valeurs sont résumées au sein du tableau 2.6 ci-dessous.

TABLE 2.6 – Valeurs de l'exposant et de la mantisse pour les nombres spéciaux en virgule flottante. Ici, les valeurs sont données pour le format simple précision.

Codage de l'exposant	Valeur de la mantisse	Signification
00000000	0	± 0 (zéro exact)
00000000	$\neq 0$	Nombre dénormalisé
11111111	0	$\pm\infty$ (en fonction du signe)
11111111	$\neq 0$	`NaN` (Not a Number)

Les valeurs spéciales $+\infty$ et $-\infty$ se comprennent aisément de par leur similarité avec les cas de débordement d'un entier. Ici, lorsque la valeur du nombre à exprimer dépasse de la plage des valeurs normales (on parle d'**overflow**), elle est remplacée par $+\infty$ ou $-\infty$ en fonction du signe du nombre.

De même, lorsque la valeur est trop faible pour la plage de valeurs, on parle d'**underflow**. Dans ce cas, la valeur du nombre est remplacée par un nombre dénormalisé, voire par le zéro exact. Enfin, la valeur spéciale `NaN` est utilisée pour représenter des cas où le résultat d'une opération mathématique n'est pas un nombre défini, tel que 0/0 par exemple.

Valeurs maximales et minimales

Connaissant les valeurs spéciales qui viennent d'être définies, nous pouvons maintenant calculer les valeurs maximales et minimales représentables pour un format de réel normalisé donné, tout en gardant en tête que ces valeurs seraient plus extrêmes si les nombres spéciaux n'avaient pas été définis, car les valeurs maximales et minimales seraient alors celles pour lesquelles l'exposant est maximal ou minimal.

Le réel maximal est celui pour lequel le codage de E ne contient que des 1 sauf pour le bit de poids le plus faible (faute de quoi, le nombre représenté serait l'infini, en vertu des règles de nombres spéciaux). En simple précision, cela signifie que $cod(E) = 11111110$, ce qui correspond à $E = 254 - 127 = 127$. La mantisse, quant à elle, est alors composée de 23 bits à 1, ce qui donne $1 + \sum_{i=1}^{23} 2^{-i}$ (autrement dit aussi proche de 2 que possible). Le nombre maximal est donc $\left(1 + \sum_{i=1}^{23} 2^{-i}\right) \cdot 2^{127} \approx 3.4 \cdot 10^{38}$, et le nombre minimal vaut donc environ $-3.4 \cdot 10^{38}$.

En utilisant la même approche avec le plus petit exposant non-nul possible, c'est-à-dire

$cod(E) = 00000001$, ce qui correspond à $E = 1 - 127 = -126$, ainsi que la plus petite mantisse possible, c'est-à-dire $1 + 0 = 1$, on trouve que le nombre le plus proche de zéro que l'on puisse représenter de façon normalisée est $\pm 2^{-126} \approx \pm 10^{-38}$. C'est ce « trou » entre le véritable zéro et ces valeurs de faible exposant qui est comblé par les nombres dénormalisés ainsi que le nombre spécial zéro exact.

Pour terminer, la table 2.7 ci-dessous résume les différentes plages de valeurs représentables pour un nombre à virgule flottante en simple précision selon la norme IEEE 754. On y indique également le type de débordement qui se produit lorsque l'on sort de ces plages, ainsi que le type de nombre (spécial, normalisé ou dénormalisé) qui est représenté dans ces plages.

TABLE 2.7 – Résumé des différentes plages de valeurs représentables pour un nombre à virgule flottante en simple précision selon la norme IEEE 754. La seconde ligne représente le type de débordement, tandis que la troisième ligne indique le type de nombre (spécial, normalisé ou dénormalisé).

$-\infty$ à $-3.40 \cdot 10^{38}$		$-1.18 \cdot 10^{-38}$ à $1.18 \cdot 10^{-38}$		$3.40 \cdot 10^{38}$ à ∞
overflow		underflow		overflow
$-\infty$	normalisés	0 et dénormalisés	normalisés	$+\infty$

Écart entre nombres représentables

À première vue, il peut paraître étonnant que la valeur maximale représentable sur 32 bits avec un nombre à virgule flottante soit de l'ordre de 10^{38}, alors que la valeur maximale représentable avec un codage d'entier non signé, qui est incapable de représenter des quantités fractionnaires, n'est que de l'ordre de $2^{32} \approx 10^9$.

Comme nous l'avons déjà noté, la spécificité de la virgule flottante est de travailler avec un nombre de chiffres significatifs donné ; une conséquence directe à cela est qu'entre deux flottants dont l'exposant possède une valeur différente, alors le poids d'un bit donné de la mantisse est différente. Par exemple, et pour prendre un exemple en base décimale, dans $3.57 \cdot 10^3$, le premier chiffre après la virgule « pèse » 10^2, tandis que dans $3.57 \cdot 10^8$ il pèse 10^7. Ainsi, il est évident que le prix à payer pour travailler avec un nombre constant de chiffres significatifs tout en représentant des nombres sur une large plage d'ordre de grandeurs, est que l'écart entre deux nombres représentables n'est pas constant. C'est la raison pour laquelle un entier non signé sur 32 bits ne peut représenter une valeur maximale qui ne vaut que 10^9 de plus que la valeur minimale ; en échange, l'écart entre deux entiers non signés est toujours d'une unité.

Caractérisons cet écart. Prenons un nombre exprimé en virgule flottante, et soustrayons-lui un nombre flottant voisin, c'est-à-dire dont la mantisse ne diffère de celui-ci que d'un seul bit ; dans ce cas, la mantisse du résultat ne possède que des bits nuls, sauf celui de poids le plus faible. Dans ce cas, en appliquant la définition du décodage d'un flottant, l'écart entre les deux flottants doit valoir $2^{-k_m} \cdot 2^E = 2^{E-k_m}$, où k_m est le nombre de bits

de la mantisse et E est l'exposant. Il est évident que l'écart entre deux nombres voisins est une fonction exponentielle de l'exposant du nombre représenté, c'est-à-dire proportionnel au nombre représenté lui-même.

Cette variabilité de l'écart entre deux flottants a des conséquences importantes en programmation scientifique. Par exemple, supposons que l'on veuille simuler l'évolution de la masse du soleil (disons $M = 2 \cdot 10^{30}$ kg) en codant cette dernière comme un flottant en précision simple. Dans ce cas, la valeur stockée en mémoire serait :

$$cod(2 \cdot 10^{30} \text{ kg}) = 01110001110010011111001011001010 \tag{2.38}$$

À présent, supposons que notre modèle de l'évolution massique du soleil implique une opération du type $M(t+1) = M(t) + \Delta m$. On voudrait que les petits écarts de masse Δm soient, par exemple, de l'ordre de grandeur de la masse d'un grand astéroïde comme Cérès, à savoir 10^{21} kg. Une telle opération ne serait alors pas possible en précision simple. En effet, le nombre voisin de la masse du soleil telle que représentée en mémoire est :

$$dec(01110001110010011111001011001010 + 1) = 2.0000002 \cdot 10^{30} \text{ kg}, \tag{2.39}$$

Ici, l'écart à la prochaine valeur représentable est de l'ordre de $(2.0000002 - 2) \cdot 10^{30} \approx 10^{23}$ kg, soit un dixième de masse terrestre. En l'occurrence un passage à une représentation en double précision serait pertinent.

Ce problème aurait pu être anticipé en réutilisant l'argument discuté plus haut, à savoir que l'écart au flottant le plus proche est d'environ 2^{E-k_m}. En précision simple, $k_m = 23$, tandis que 10^{30} s'exprime en base 2 comme $2^{\log_2(10^{30})} \approx 2^{100}$. Ainsi, l'écart entre deux flottants voisins est d'environ $2^{100-23} \approx 2 \cdot 10^{23}$. En précision double, où la mantisse possède 52 bits, on aurait $2^{100-52} \approx 2 \cdot 10^{14}$.

Nous donnons en annexe D un algorithme pour approximer facilement la valeur de l'écart entre deux flottants en détectant à partir de quelle incrément la valeur d'un nombre donné ne varie plus. Une implémentation en Python est également proposée. À noter que le rapport entre cet écart et le nombre représenté est constant, puisque $2^{E-k_m}/2^E = 2^{-k_m}$. On nomme communément « epsilon machine » l'écart entre l'unité et le plus petit flottant strictement supérieur à l'unité.

Chapitre 3

Codage de l'information – Textes, images et sons

À présent que nous avons vu comment représenter les nombres, nous pouvons représenter toute information pour laquelle il est possible d'établir une **convention de correspondance**, car cette correspondance peut toujours prendre la forme d'un nombre ou d'une séquence de nombres du côté de la machine.

3.1 Codage des caractères

La représentation des caractères au sein d'un texte est un exemple à la fois important et simple de correspondance entre nombres et information non numérique, c'est pourquoi nous l'abordons en premier.

3.1.1 Convention ASCII

L'American Standard Code for Information Interchange (ASCII) est une convention de codage de caractères qui a été très largement utilisée dans les premiers temps de l'informatique.

Principe de base

Comme son nom peut le laisser supposer, cette convention (ou « norme ») est basée sur l'alphabet latin. En effet, pour les programmeurs occidentaux des années 1960, il était essentiel de pouvoir *a minima* coder des caractères de la langue anglaise (ainsi que les symboles mathématiques les plus courants). L'héritage de ce choix est encore très important aujourd'hui et se fait ressentir, par exemple, dans le jeu limité des caractères disponibles pour renseigner l'URL d'un site internet, les adresses e-mail, les noms de fichiers, etc.

La norme ASCII utilise 7 bits coder chaque caractère. Ainsi, $2^7 = 128$ caractères différents peuvent être représentés en ASCII. Cependant, 1 bit étant souvent utilisé pour contrôler les erreurs (ou alors le bit est inutilisé), les caractères ASCII sont typiquement stockés sur un octet. La table 3.1 ci-dessous présente les caractères ASCII les plus courants.

b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

FIGURE 3.1 – Table ASCII des caractères datant de 1972. Par exemple, le code ASCII de la lettre « A » est `1000001`, ou $0x41$ en hexadécimal, tandis que celui de la lettre « N » est `1001110`, ou $0x4E$ en hexadécimal.

L'annexe E montre un exemple de code C dans lequel un message est écrit dans un fichier en indiquant explicitement le code ASCII de chaque caractère.

Bit de parité

Le bit qui n'est pas utilisé pour coder les caractères peut être utilisé pour ce que l'on nomme un contrôle de parité. Cela sert à diminuer les chances que le message ait été corrompu lors de sa transmission sans qu'on le sache. Supposons que l'on veuille transmettre un message et que nous ayons fixé arbitrairement que tout octet « sain » doit comporter un nombre pair de bits possédant la valeur 1. Dès lors, si l'on veut par exemple transmettre le message composé des 7 bits `1001011`, alors le bit de parité doit être égal à 0. En revanche, si l'on veut transmettre le message composé des 7 bits `1001111`, alors le bit de parité doit être égal à 1. Si un nombre impair de bits sont modifiés par erreur lors de la transmission du message, alors le nombre de bits à 1 que reçoit le destinataire est impair et il est certain que l'octet reçu est corrompu (ou mal codé).

Limitations

Dès lors qu'il faut coder d'autres caractères que les minuscules et majuscules latines de base, la norme ASCII est limitante. En effet, elle n'établit aucune convention pour représenter des caractères accentués, des caractères non latins, des symboles mathématiques élaborés, des émoticônes, etc. La norme ISO/CEI 8859-1, plus connue sous le nom de **Latin-1**, peut être utilisée pour pallier ces limitations en incorporant un bit supplémentaire pour coder des caractères, c'est-à-dire un huitième bit. Cependant, le nombre de caractères codables demeurant relativement faible (256) en comparaison de l'étendue des symboles de toutes sortes utilisés dans le monde, il est clair qu'une norme plus étendue est nécessaire.

3.1.2 Standard Unicode

La convention Unicode, née dans les années 1990, établit un standard de correspondance entre caractères et symboles pour environ 160'000 caractères différents [1]. Il est important de comprendre ici une nuance : si « Unicode » désigne le standard, la façon précise de coder les caractères en utilisant ce standard dépend du type exact de codage utilisé, qui spécifie par exemple le nombre de bits sur lesquels les caractères sont codés, ou encore le fait que ce nombre soit variable ou constant, ce que le simple terme d'« Unicode » n'indique pas.

Points de code

En Unicode, le terme « point de code » est utilisé pour désigner les numéros spécifiques attribués aux caractères. Cette terminologie permet d'illustrer le fait que la façon dont les caractères sont représentés dépend du type de codage utilisé et n'est pas une constante, comme dans un codage fixe tel que l'ASCII. Les implémentations les plus populaires – comme UTF-8, dont nous discutons ci-dessous – utilisent un codage de taille variable.

Les points de code sont communément écrits en hexadécimal et préfixés par « U+ », par exemple `U+0041` pour la lettre « A », ou encore `U+2207` pour « ∇ ». Les points de code sont divisés en « plans » de 65'536 points de code chacun. Le plan 0, appelé **plan multilingue de base** (BMP), contient les caractères les plus courants, tels que les caractères latins, les chiffres arabes, les symboles mathématiques, etc. Au total, les points de code s'étalent de `U+000000` à `U+10FFFF`.

Taille fixe et taille variable

L'avantage évident des codages en taille fixe est la simplicité et l'efficacité de l'implémentation qu'ils offrent. Par exemple, le codage UCS-2, utilisé au sein de plusieurs

1. En réalité, le standard Unicode va même plus loin, en établissant d'autres règles que nous ne traitons pas ici (par exemple, le sens d'écriture).

systèmes d'exploitation par le passé, utilise deux octets par caractère (et peut donc encoder de U+0000 à U+FFFF), ce qui permet en théorie d'accéder à chaque caractère en mémoire en temps constant, c'est-à-dire que le nombre d'étapes requises par l'algorithme pour associer le codage au caractère est toujours le même. En effet, on sait que le n-ième caractère d'un texte se situe en position kn de la mémoire par rapport au début du texte, si chaque caractère est codé sur k octets. En l'occurrence, en UCS-2, le point de code du caractère interprété comme entier coïncide avec le codage $cod_{\mathbb{N}_{(16)}}$ de ce caractère. Le codage UCS-32 est un autre exemple (plus actuel) de codage à taille fixe, qui utilise quatre octets par caractère et permet donc d'accéder à tous les points de code Unicode.

Cependant, cette simplicité a un coût : le gaspillage de mémoire. En effet, un texte en anglais, par exemple, pourrait être codé avec un seul octet par caractère, ce qui signifie que la moitié de la mémoire utilisée pour coder un texte anglais en UCS-2 est inutile. À l'inverse, un codage à taille variable permet donc de coder sur moins d'octets certains caractères, mais au prix d'une complexité accrue de l'algorithme d'interprétation du codage. En effet, dans ce cas, le code d'un caractère doit permettre de déduire où celui-ci commence ou s'arrête, et on ne peut pas directement accéder à un caractère en mémoire en temps constant : il faut parcourir individuellement les caractères du texte pour savoir où ces derniers commencent et se terminent, car le nombre de bits qui les compose dépend justement de leur point de code et figure au sein même de leur représentation en mémoire. Nous voyons plus loin, dans la partie algorithmique du cours, comment un codage à taille variable peut être mis en oeuvre, en partie pour compresser des données.

Le codage UTF-8 constitue un exemple commun de codage à taille variable, et compose par exemple l'immense majorité des pages web aujourd'hui. En effet, de nos jours, le débit de la connexion au réseau a un impact bien plus grand sur la vitesse d'affichage d'une page Web que la vitesse de calcul d'un ordinateur, qui est de toute manière largement suffisante pour décoder un contenu textuel typique ; dès lors, on peut se permettre de complexifier le traitement des données d'un media si cela autorise à raccourcir la taille de ce dernier. En UTF-8 les caractères sont codés sur 1, 2, 3 ou 4 octets, dépendemment de leur point de code. Il faut noter que le codage UTF a été choisi spécialement pour assurer la compatibilité avec les standards ASCII, de sorte que les points de code des caractères ASCII (et partiellement de Latin-1) sont les mêmes en Unicode. « UTF » signifie « Unicode Transformation Format », tandis que le nombre qui suit indique la taille des unités de code utilisées.

Il est fréquent de rencontrer des erreurs de codage en naviguant sur le web ou, encore davantage, lorsqu'on programme en lisant ou écrivant dans des fichiers texte. Ces erreurs sont souvent dues à des incompatibilités entre les codages par défaut utilisés par les différents logiciels ou systèmes d'exploitation. En particulier, elles surviennent dans les cas où un texte a été codé avec un certain codage et décodé avec un autre. Le script Python ci-dessous permet de générer volontairement une telle incompatibilité :

```
1 original_string = "Ce décodage n'est pas réussi !"
2
3 encoded_string = original_string.encode('utf-8') # Codage en utf-8
4
5 decoded_string = encoded_string.decode('latin-1') # Décodage en latin-1
6
```

```
7 print(decoded_string) #affiche "Ce dÃ©codage n'est pas rÃ©ussi !"
```

Sachant qu'en latin-1 tous les caractères sont codés sur un octet, on peut par exemple deviner qu'en utf-8, le caractère « é » est codé sur deux octets, puisque il devient un « Ã » suivi d'un « © », lorsqu'il est interprété en latin-1.

3.2 Codage des sons

D'un point de vue physique, le son est une onde qui se propage au sein d'un milieu donné, comme l'air de notre quotidien par exemple. Cette vibration (changement de pression) du milieu peut être perçue par des appareils de mesure, dont l'oreille humaine est un exemple. Le microphone, qui est un autre appareil captant le son grâce à une membrane sensible aux vibrations de l'air, est un transducteur qui convertit les variations de pression acoustique (les vibrations de l'air) en variations de tension électrique (« voltage »). Le haut-parleur, quant à lui, agit comme un microphone inversé : il convertit les variations de tension électrique en vibrations d'une membrane, et donc du milieu ambiant. Plus la fréquence de vibration est haute, plus le son est perçu comme aigu, et plus la fréquence est basse, plus le son est perçu comme grave.

À l'échelle humaine, on peut faire l'excellente approximation que le nombre de niveaux de pression acoustique de l'air est infini. Dès lors, si l'on disposait d'un appareil capable de mesurer cette pression acoustique de façon continue dans le temps et avec une précision infinie, nous obtiendrions un signal tel que celui montré sur le haut de la figure 3.2, qui correspond en quelque sorte au « véritable » signal physique du phénomène. Si, à l'inverse, nous utilisons un appareil possédant une résolution finie, nous serions contraints d'« échantillonner » ce signal en mesurant son amplitude uniquement en des instants donnés, ce qui résulterait en une séquence de mesures telle qu'illustrée au centre de la figure 3.2. Cette séquence de nombres serait celle que nous pourrions stocker de façon digitale au sein de la mémoire d'un ordinateur. Lorsque la séquence de nombres est transmise à un haut-parleur, les vibrations que ce dernier transmet à l'air sont alors une version continue du signal digital, comme montré en bas de la figure 3.2. Si l'échantillonnage désigne l'action de discrétiser un signal dans le temps, le terme de quantification désigne l'action de discrétiser un signal dans l'amplitude, c'est-à-dire de le représenter par un nombre fini de niveaux de pression acoustique.

L'oreille humaine, quant à elle, est capable de distinguer un nombre fini de niveaux de pression acoustique, et c'est pourquoi un son peut être enregistré de façon digitale tout en donnant l'illusion que sa reconstruction est un signal continu. Ce principe est également utilisé pour les dégradés des couleurs au sein d'une image, ou encore dans l'illusion de continuité donnée par les pixels d'une image sur un écran, malgré leur nombre fini. Dans le cas de l'illusion de « dégradé sonore » pour l'oreille humaine, une fréquence d'échantillonnage courante est d'un peu plus de 40'000 Hz, ce qui peut constituer une

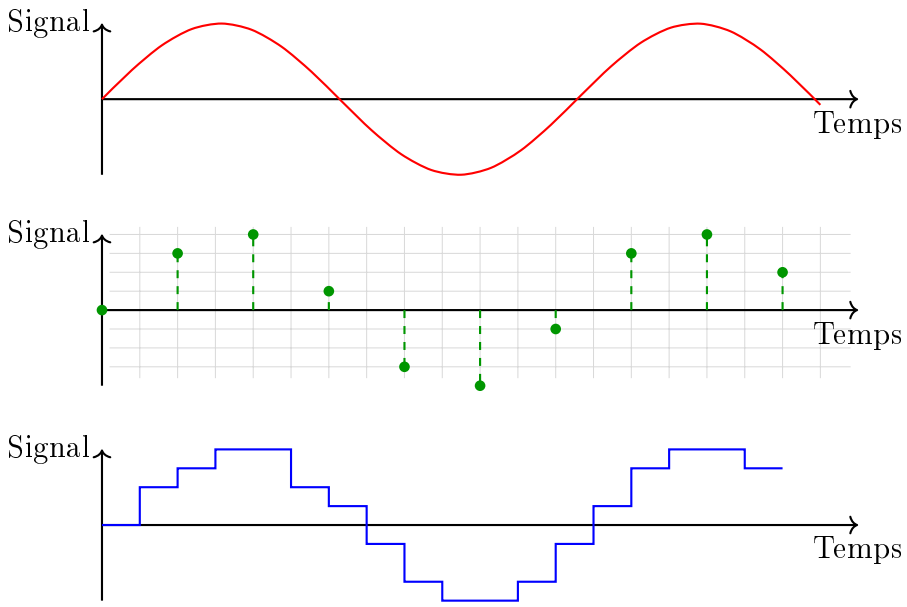


FIGURE 3.2 – Signal sonore analogique (en haut), discrétisé (au milieu) et analogique reconstruit sans lissage (en bas). C'est la version du milieu, discrétisée en une séquence finie valeurs quantifiées (représentée par les points de la grille), qui est stockée au sein de la mémoire d'un ordinateur.

grande quantité de données à stocker en mémoire suivant la longueur de l'enregistrement [2].

Notons pour terminer que, dans bien des cas, la qualité d'un son digitalisé et stocké au sein de la mémoire est souvent dégradée suite à son traitement par des algorithmes de compression, qui réduisent la taille du fichier audio en éliminant des informations jugées superflues. Les fichiers audio de type MP3, par exemple, sont des fichiers audio compressés, tandis que les fichiers WAV sont des fichiers audio non-compressés (nous verrons que le même principe de compression peut être utilisé pour tout autre type d'information, comme les images ou le texte). Par ailleurs, il existe également des algorithmes de compression de l'information qui n'infligent aucune perte d'information aux données. D'une manière générale, les algorithmes de compression sont un sujet vaste qui sort du cadre de ce cours, bien que nous en donnions un exemple dans la partie du cours dévolue à l'algorithmique.

Dans certains cas, comme celui de la composition musicale, on peut exploiter la nature de l'information pour la coder sous une forme synthétique. Par exemple, le format de fichier MIDI permet en quelque sorte de coder une partition : il ne stocke pas directement un signal sonore échantillonné, mais une suite d'événements musicaux tels que le début et fin de note, hauteur, vélocité, changements de programme et divers contrôles. Par exemple, au format MIDI les notes sont codées sur 7 bits. Les données audio relatives à un morceau de musique issues d'un codage synthétique sont bien plus concises que celles provenant de l'échantillonnage d'un signal physique, car elles ne stockent pas une série temporelle

2. L'humain est capable de percevoir des sons dont la fréquence se situe approximativement entre 20 Hz et 20 kHz. Or, pour que l'échantillonnage permette de reconstruire une oscillation d'une fréquence donnée, la fréquence d'échantillonnage doit valoir plus du double de la fréquence mesurée. Ceci est formalisé par le théorème de Nyquist-Shannon. Par exemple, si la fréquence d'échantillonnage est égale à celle d'une sinusoïde, les prélèvements ont lieu chaque fois au même endroit de la période et donnent donc toujours la même valeur !)

de fréquences sonores, mais uniquement les instructions pour les générer. C'est un peu l'équivalent des images vectorielles, dont nous parlons brièvement ci-dessous. L'annexe F donne un exemple de code Python qui écrit les premières notes d'*Au clair de la lune* au format MIDI.

3.3 Représentation des couleurs au sein des images

3.3.1 Types d'images

On peut distinguer deux grandes familles de codage des images : les formats de type **bitmaps** (aussi nommés images matricielles ou raster) et les formats de type **vectoriels**. Les formats matriciels, tels que le JPEG, le PNG, le GIF ou le BMP, sont les plus courants et sont basés sur une grille de pixels, chaque pixel se voyant associé à une couleur. Les formats vectoriels, tels que le SVG ou l'EPS sont basés sur des formules mathématiques qui décrivent les formes géométriques des objets représentés sur l'image, et sont en quelque sorte le pendant graphique des formats audio synthétiques tels que MIDI. Les formats raster sont plus adaptés pour les photographies, tandis que les formats vectoriels sont plus adaptés pour les dessins et les illustrations amenées à être redimensionnées et tournées.

3.3.2 Aparté sur les formats de fichier

Par « format de fichier », nous entendons un ensemble de conventions de codage qui spécifient non seulement comment encoder et décoder l'information, mais aussi comment compresser, organiser, et stocker les données dans le fichier. En plus du codage lui-même, le format spécifie comment les informations sont organisées et stockées dans le fichier. Enfin, concernant les données audiovisuelles, le codec (pour « encodeur-décodeur »), est la couche logicielle ou le composant qui s'occupe de manipuler les données selon les spécifications du format de fichier, en appliquant les méthodes de compression et de décompression nécessaires. Les formats de fichiers peuvent concerner tout type d'information stockée sur un ordinateur, et pas uniquement les images.

Il faut remarquer que le nom attribué aux fichiers, au sein d'un système d'exploitation, est souvent lié à leur format, le but étant de permettre facilement d'identifier le logiciel adapté à l'ouverture du fichier. Cependant, ce n'est pas une règle stricte. Ainsi, un fichier dont le nom se termine par `.jpeg` est probablement un fichier image au format JPEG, mais rien n'empêche de renommer ce fichier tout autrement.

3.3.3 Principe du codage RGB

Dans cette section, nous nous attachons à la description du codage des couleurs au sein des images. Pour simplifier la discussion, nous considérerons des images au format matriciel, mais le codage des couleurs elles-mêmes est indépendant de la façon de décrire leur répartition au sein de l'image.

Supposons une image au sein de laquelle chaque pixel correspond à un octet au sein de la mémoire de l'ordinateur. Dans ce cas, il peut exister $2^8 = 256$ valeurs (couleurs) différentes pour chaque pixel de l'image. On peut tout à fait décider d'une correspondance arbitraire entre des entiers et des couleurs. Par exemple, on peut décider que le nombre 0 correspond au noir et le nombre 255 au blanc, et que la couleur associée à chaque valeur entre ces deux bornes est une nuance de gris. De la même façon, on aurait pu dire que le 0 correspond au noir et le 255 au bleu « pur ». Comme nous allons le voir maintenant, il existe des façons intéressantes de décomposer les couleurs d'une façon utiles pour l'oeil humain, tout en s'inspirant de l'exemple qui vient d'être donné pour les 256 nuances de gris ou de bleu.

Commençons par noter que, d'un point de vue biologique, les couleurs peuvent en bonne approximation être décomposées en trois composantes de base : le rouge, le vert et le bleu. Ces trois couleurs sont appelées **couleurs primaires** et sont suffisantes pour coder une grande partie des couleurs visibles par l'oeil humain. En effet, celui-ci possède trois types de cônes, chacun sensible à une plage de longueurs d'onde du spectre lumineux. En mélangeant trois couleurs primaires bien choisies, on peut obtenir d'autres couleurs, puisque les sensations de couleur sont issues de la combinaison de l'influx nerveux lié aux différents cônes. C'est le principe de la synthèse additive des couleurs[3]. Ainsi, n'importe quel triplet de couleurs dont aucune ne peut être obtenue à partir des deux autres[4] peut également faire office de couleurs primaires en synthèse additive, bien qu'en pratique l'ensemble des couleurs que l'on peut reproduire sur un écran ne soit pas le même dans tous les cas. Puisque les cônes au sein de l'oeil sont sensibles à des *plages* de longueurs d'onde se chevauchant et non pas à des longueur d'onde précises, il est en réalité impossible de n'exciter qu'un seul type de cône à la fois ; c'est l'une des raisons pour lesquelles nous disons plus haut que ces couleurs primaires doivent être « bien choisies ». Une lumière monochromatique stimule généralement plusieurs types de cônes à des degrés différents, puisque leurs plages de sensibilité se recouvrent. Ainsi, une sensation de couleur n'est pas équivalente à une longueur d'onde du spectre électromagnétique (comme le brun, par exemple). Une lumière monochromatique possède une longueur d'onde donnée mais, par ailleurs, beaucoup de sensations de couleur ne correspondent à aucune lumière monochromatique.

Il se trouve que l'oeil humain sain est capable de distinguer entre 1 et 10 millions de couleurs différentes, ce qui dans tous les cas est bien plus que les 256 couleurs que l'on peut coder avec un octet. En supposant que ces 10 millions de couleurs distinguables par l'humain soient réparties de façon uniforme dans l'espace des couleurs (ce qui n'est pas le cas), on peut estimer le nombre de bits nécessaires pour coder les couleurs de telle sorte qu'une séquence de couleurs différentes soit perçue comme continue par l'oeil humain :

$$2^n = 10^7 \iff n = \log_2(10^7) \approx 23.25. \tag{3.1}$$

Ainsi, il faudrait environ 24 bits pour coder une couleur de façon à ce que l'oeil humain ne puisse pas distinguer les différentes nuances entre les pixels (ce que l'on nomme souvent un « dégradé de couleurs »). C'est un heureux hasard, car 24 bits est justement divisible

3. Il faut insister sur le fait que cette décomposition a pour origine la biologie humaine et ne découle pas d'une propriété fondamentale de la lumière.

4. Un peu comme trois vecteurs linéairement indépendants peuvent servir à former une base de l'espace tridimensionnel.

en 3 – qui est le nombre de couleurs primaires pour l'humain – octets. C'est pourquoi les images non-compressées sont souvent codées sur 3 octets par pixel, soit 1 octet pour le rouge, 1 octet pour le vert et 1 octet pour le bleu. Cela permet de coder environ dix-sept millions de couleurs, à raison de 256 nuances par couleur primaire. On parle alors de codage RGB (pour *Red, Green, Blue*) ; chaque composante de couleur est donc une valeur entre 0 et 255. Ce codage est également souvent désigné par les termes « *True Color* »On peut bien entendu envisager des codages différents en consacrant plus ou moins de mémoire à chaque couleur primaire, ce qui a pour effet d'augmenter ou de réduire la gamme de couleurs possibles autour de certaines nuances.

Notons qu'il est également courant de coder l'opacité du pixel, en plus des couleurs primaires qui décrivent sa couleur. On parle alors de codage RGBA (pour les 4 canaux *Red, Green, Blue, Alpha*). Par exemple, si l'opacité est codée sur 8 bits, une valeur alpha de 255 correspond à un pixel complètement opaque, tandis qu'une valeur de 0 correspond à un pixel complètement transparent.

3.3.4 Écriture des triplets RGB

Il est courant, dans les applications de traitement d'image, en design Web ou encore en programmation d'application visuelles, de désigner une couleur par un triplet (R, G, B). Une représentation alternative utilise le système hexadécimal, où chaque composante, codée sur un octet, correspond à deux chiffres. Par exemple, la couleur rouge pure est codée en RGB-24bits par `(255,0,0)` et hexadécimal par `#FF0000`, où l'utilisation du dièse est une convention pour indiquer que l'on parle de couleurs au format hexadécimal pour le codage RGB-24bits. La table 3.1 ci-dessous donne quelques exemples de couleurs importantes et leur codage en RGB et en hexadécimal.

TABLE 3.1 – Nom et codage de quelques couleurs en RGB et en hexadécimal.

Nom	RGB	Hexadécimal	Échantillon
Noir	(0,0,0)	#000000	
Rouge	(255,0,0)	#FF0000	
Vert	(0,255,0)	#00FF00	
Bleu	(0,0,255)	#0000FF	
Blanc	(255,255,255)	#FFFFFF	
Gris	(127,127,127)	#7F7F7F	
Jaune	(255,255,0)	#FFFF00	
Magenta	(255,0,255)	#FF00FF	
Cyan	(0,255,255)	#00FFFF	

L'annexe G propose un code python permettant d'écrire un fichier image contenant un dégradé de bleu, ainsi que d'autres couleurs. Ce code est une bonne illustration de la façon dont les couleurs sont codées dans les images matricielles, bien que l'étude d'un format d'image donné sorte du cadre de ce cours.

3.3.5 Autres codages de couleurs

Pour terminer, remarquons que bien d'autres méthodes de codage des couleurs existent, bien qu'elles utilisent également 3 composantes. HSV (Teinte, Saturation, Valeur) représente les couleurs en termes de leur teinte (la couleur proprement dite), saturation (l'intensité de la couleur) et valeur (la luminosité de la couleur). Teinte est représentée comme un angle sur un cercle de couleur (0 à 360 degrés), saturation est représentée comme un pourcentage, et la valeur est également un pourcentage, décrivant la luminosité. HSL (Teinte, Saturation, Luminosité) est une variante de HSV où la luminosité est utilisée à la place de la valeur.

Enfin, signalons le codage CMJN (pour Cyan, Magenta, Jaune, Noir), utilisé en imprimerie pour des raisons techniques et basé sur la synthèse soustractive des couleurs. Contrairement à RGB, où les couleurs sont créées en ajoutant de la lumière, CMYK fonctionne en soustrayant la lumière à partir d'une feuille blanche. Les encres cyan, magenta et jaune absorbent chacune des parties spécifiques du spectre lumineux, et la superposition de ces encres crée une gamme de couleurs. Le noir est ajouté (représenté par 'K' pour 'Key') parce que les trois autres encres mélangées ne produisent pas un noir pur et économique.

3.3.6 Compression de l'information

Le cas de la compression des images est un sujet très vaste qui sort du cadre de ce cours. Tout comme le format MP3 est un format impliquant une compression des fichiers audio, le format JPEG est un format populaire impliquant une compression des images, afin de limiter l'espace en mémoire qu'elles occupent. À l'instar du format MP3, le format JPEG inflige une **perte d'information** au contenu compressé ; le format PNG, à l'inverse, est un exemple de format de compression sans pertes pour les images. Enfin, un format tel que PPM ou BMP énumère sans compression les couleurs associées aux pixels d'une image, comme le format WAV le fait pour les sons.

Le tableau 3.2 ci-dessous donne un aperçu de quelques formats d'images courants et de leurs caractéristiques principales. Il permet également de dresser un parallèle avec les formats de fichiers audio.

TABLE 3.2 – Quelques formats de fichier ou familles de formats de ficher pour les images et les sons, avec leurs caractéristiques de compression principales couramment utilisées. L'indication « mixte » signifie que le format existe en plusieurs versions et peut être utilisé avec ou sans pertes. Les exemples audio ci-dessous correspondent à des usages courants et constituent une simplification : certains formats peuvent exister sous plusieurs variantes ou encapsuler différents codages.

Format image	Format audio	Compression	Perte d'information
PPM, BMP	WAV	Non	Non
PNG, GIF	FLAC	Oui	Non
JPEG	MP3, AAC, Ogg Vorbis	Oui	Oui
WEBP	WMA	Oui	Mixte

Chapitre 4

Circuits logiques

Dans le chapitre sur les origines historiques de l'informatique, nous avons mentionné (section 1.2.2) l'importance capitale de la réalisation, par Claude Shannon, du fait que des machines pouvaient effectuer des calculs logiques, par opposition à de simples calculs arithmétiques. Dans le chapitre 2, par ailleurs, nous avons vu comment différents types d'information pouvaient être représentés par des états binaires.

Dans ce chapitre, nous allons nous intéresser à la façon dont ces calculs logiques peuvent être effectués au travers de circuits logiques binaires. Il est important de comprendre qu'ici, bien que nous discutions brièvement le cas des transistors, nous nous situerons à un niveau d'abstraction où la façon physique dont les composants traitent l'information nous importe peu : nous partirons du principe qu'il existe des relais quels qu'ils soient (électromécaniques, tubes à vide, transistors) qui nous permettent de réaliser certaines fonctions fondamentales discutées plus bas.

4.1 Un exemple de circuit logique

Afin de donner au lecteur une vision globale des sections qui suivent, nous proposons ici un exemple de circuit logique à interpréter de façon intuitive. Par la suite, nous reviendrons de façon plus formelle sur les concepts utilisés.

Reprenons la table de vérité introduite dans la section 1.2.2. Cette fois-ci, nous allons également renommer les différentes conditions binaires afin de les désigner de façon plus commodes. Le fait de rouler à plus de 50 km/h sera dénoté A, tandis que le fait de se situer hors localité sera dénoté B. Enfin, le résultat d'être en infraction sera dénoté R. Quelle que soit la proposition A, B ou R, nous noterons 1 si elle est vraie et 0 sinon ; il s'agit de propositions binaires. La table de vérité correspondante est donnée dans le tableau 4.1 ci-dessous.

TABLE 4.1 – Table de vérité pour deux inputs binaires, l'un (A) à propos de la vitesse d'un véhicule, l'autre (B) à propos du lieu où il roule. Le résultat (R) est binaire également et indique si le véhicule est en infraction.

Signification	Roule à plus de 50 km/h	Est hors localité	Est en infraction
Nom raccourci	A	B	R
	0	0	0
	0	1	0
	1	0	1
	1	1	0

En examinant la table, on peut parvenir à la résumer par une phrase du langage naturel (français, dans notre cas) qui résume le raisonnement sous-jacent. Ici, une façon de résumer la table de vérité pourrait être : « Le véhicule est en infraction si et seulement si la condition suivante est réalisée : il roule à plus de 50 km/h et n'est pas hors localité. » [1]

Ainsi, pour qu'une machine puisse calculer le résultat de ce raisonnement logique, il lui faut une façon de combiner les inputs A et B de façon à obtenir le résultat R. Cela pourrait ressembler au schéma du circuit ci-dessous en figure 4.1, où nous décidons que la **porte logique** représentée par un triangle suivi d'un petit cercle a pour rôle d'inverser la valeur de l'input qu'elle reçoit (0 devient 1 et *vice-versa*), tandis que la porte logique représentée par une demie-ellipse retourne la valeur de 1 uniquement si chacun de ses inputs vaut 1 également.

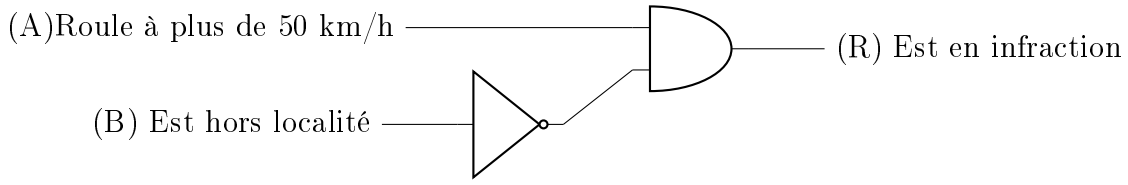


FIGURE 4.1 – Circuit logique pour le problème de l'excès de vitesse.

On dit que le circuit de la figure 4.1, composé de deux portes logiques, **implémente** la table de vérité du problème de l'excès de vitesse. Quelle que soit la valeur binaire transitant par les inputs A et B, l'output R correspondra toujours au résultat indiqué dans la table. Pour construire une machine électronique effectuant un raisonnement logique quant aux excès de vitesses, il ne nous reste plus qu'à trouver les éléments électroniques concrets qui réalisent les portes logiques du schéma. C'est ce que nous discuterons brièvement en section 4.2.2.

1. Cependant, il faut faire ici très attention : nous avons perdu en précision en utilisant le langage naturel. En particulier, l'utilisation du *et* et du *ou* est quelque peu ambiguë en français, où la délimitation du bloc auquel il s'applique n'est pas claire.

4.2 Portes logiques

4.2.1 Les différents types de portes logiques

Comme on l'a vu dans l'exemple d'introduction, il est important qu'un circuit logique puisse contenir des éléments qui modifient la valeur de l'information qu'ils reçoivent. Ils remplissent des fonctions logiques mathématiques. De tels éléments sont nommés les portes logiques et on peut en imaginer plusieurs types. Les plus courantes sont les portes *NOT*, *AND*, *OR*, *NAND*, *NOR* et *XOR*. Chacune de ces portes logiques possède un symbole (voir figure 4.2) ainsi qu'une table de vérité associée. Cette dernière indique comment les valeurs d'entrée (inputs) sont combinées pour donner le résultat en sortie (output). Ci-dessous, les tables 4.2 à 4.7 présentent les inputs et l'output (colonne de droite) pour chacune des portes logiques précitées.

TABLE 4.2 – Table de vérité de la fonction logique *NOT*.

A	$NOT(A)$
0	1
1	0

TABLE 4.3 – Table de vérité de la fonction logique *AND*.

A	B	$AND(A,B)$
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 4.4 – Table de vérité de la fonction logique *OR*.

A	B	$OR(A,B)$
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 4.5 – Table de vérité de la fonction logique *NAND*.

A	B	$NAND(A,B)$
0	0	1
0	1	1
1	0	1
1	1	0

TABLE 4.6 – Table de vérité de la fonction logique *NOR*.

A	B	$NOR(A,B)$
0	0	1
0	1	0
1	0	0
1	1	0

TABLE 4.7 – Table de vérité de la fonction logique XOR.

A	B	$XOR(A, B)$
0	0	0
0	1	1
1	0	1
1	1	0

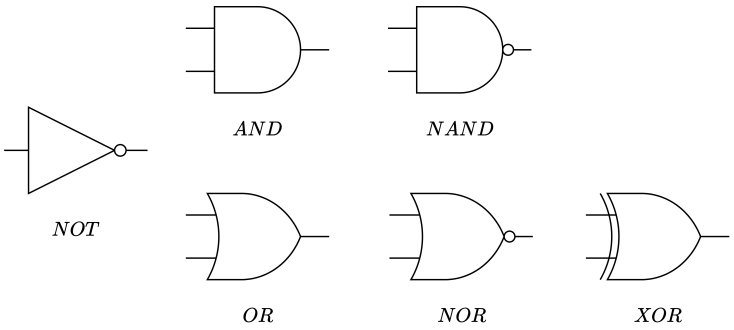
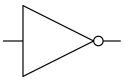
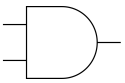
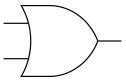


FIGURE 4.2 – Symboles pour les six types de portes logiques utilisées dans ce cours. Chaque porte est située au-dessus du texte indiquant son nom.

Il est à noter qu'en combinant des portes entre elles, on obtient des équivalences. À titre d'exemple, $NOR(A, B) = NOT(OR(A, B))$.

Pour terminer, chacune des fonctions logiques que nous venons de décrire peut être désignée par une notation plus compacte afin de faciliter l'écriture d'expressions logiques. La table 4.8 ci-dessous résume les notations utilisées dans différents contextes.

TABLE 4.8 – Notations d'usage pour les trois fonctions logiques de base en algèbre, en logique, en informatique et au sein des diagrammes logiques. Ces notations ne sont pas rigides, mais il est commun de les trouver sous cette forme dans de nombreux ouvrages. Attention, pour des raisons de facilité d'écriture sur ordinateur, nous adoptons dans ce cours une notation mixte.

Fonction	Ce cours	Algèbre	Logique	Programmation	Diagrammes
NOT(X)	$!X$	$\bar{X}$	$\neg X$ ou $\sim X$	$!X$	
AND(X,Y)	$X \cdot Y$ ou $X * Y$	$X \cdot Y$	$X \wedge Y$	$X\&\&Y$ ou X *and* Y	
OR(X,Y)	$X + Y$	$X + Y$	$X \vee Y$	$X\|\|Y$ ou X *or* Y	

4.2.2 De transistors à portes logiques

Bien que cela ne soit pas le sujet principal de ce chapitre, nous donnons ici un bref aperçu de la façon dont des transistors peuvent être utilisés pour réaliser les portes logiques

décrites plus haut.

Nous avons vu en section 1.5 que les transistors agissent un peu comme des robinets (ou barrages hydrauliques) à électrons. En effet, en fonction d'un petit signal de commande (la vanne), un transistor peut laisser passer ou non un courant électrique.

Pour des raisons de cohérence avec les technologies utilisées dans la réalité, nous allons ici considérer des tensions électriques plutôt que des courants [2]. Afin de traiter un signal binaire, nous avons besoin de deux niveaux de tension : un niveau « bas », qui convoiera la valeur binaire 0, et un niveau « haut », qui convoiera la valeur binaire 1. Il y a donc, pour un circuit donné, un seuil de tension à choisir pour distinguer les deux niveaux. Dans la littérature, il est commun de trouver des circuits où le niveau bas est de 0V et le niveau haut de 5V ; c'est donc la convention que nous suivrons dans cette sous-section également.

Considérons la situation où un transistor est représenté au sein d'un circuit allant d'une source d'énergie à la terre, comme dans la figure 4.3 ci-dessous. Entre la source et la terre, on représente des résistances [3] ainsi que le transistor qui nous intéresse. Dans un circuit réel, il y a évidemment d'autres dispositifs en amont de la résistance, et d'autres encore avant la terre, qui représente simplement une référence de tension (0V par convention) pour l'ensemble du circuit. En l'occurrence, le transistor de la figure 4.3 permet de réaliser une porte logique *NOT*, où l'on peut lire l'output de la fonction logique au niveau du collecteur du transistor, et où l'input correspond à la tension appliquée à la base.

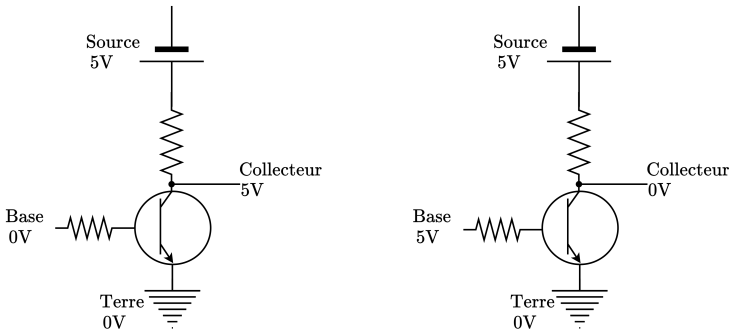


FIGURE 4.3 – Transistor réalisant la fonction logique *NOT* au sein d'un circuit électronique. À gauche, la base n'activant pas le transistor, aucun courant ne s'établit dans le circuit et une tension de 5V est mesurée en output au niveau du collecteur. À droite, la base activant le transistor, un courant s'établit dans le circuit et la résistance fait chuter la tension, qui est mesurée à 0V en output.

En combinant deux transistors soit en série, soit en parallèle, on peut réaliser les fonctions logiques *NAND* (nom donné à *NOT(AND)*) ainsi que *NOR* (nom donné à *NOT(OR)*, mais qui signifie également « ni » en anglais). Lorsque le courant doit franchir

2. Notons cependant que de nombreux auteurs et vulgarisateurs donnent des versions alternatives de la réalisation des portes logiques par des circuits électroniques. Par exemple, il serait en théorie envisageable de traiter le courant (et non pas la tension) comme signal d'importance pour déterminer l'information binaire, ce qui permettrait de simplifier quelque peu les diagrammes des portes de base que nous venons de voir, mais complexifierait en pratique leur réalisation.

3. Ces résistances ne sont pas nécessaires pour comprendre la réalisation de la fonction logique, et elle ne constituent pas ici l'objet étudié ; nous les représentons car, en pratique, elles sont nécessaires pour contrôler le courant subi par les différents éléments du transistor.

successivement l'un, puis l'autre des transistors pour pouvoir s'installer, on dit bien qu'il doit franchir l'un *et* l'autre ; inversement, lorsque les transistors sont branchés en parallèle le courant doit franchir l'un *ou* l'autre des transistors afin de s'installer. La figure 4.4 résume les trois fonctions logiques discutées jusqu'ici et construites à base de transistors.

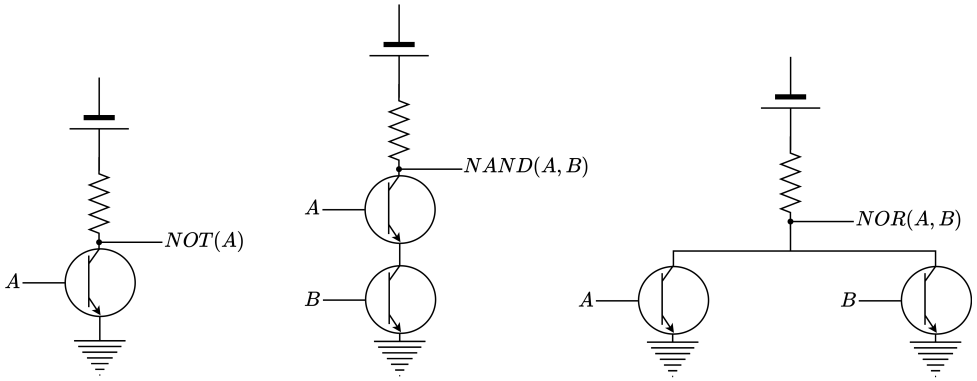


FIGURE 4.4 – Fonctions logiques *NOT*, *NAND* et *NOR* réalisées grâce à des transistors au sein d'un circuit électronique. Par souci de lisibilité, les légendes pour la source d'énergie et la terre sont omises ainsi que les résistances sur les lignes d'input.

Sur la base de ces seules portes, il est possible de construire les autres portes mentionnées précédemment. Par exemple, la porte *AND* peut être réalisée en combinant une porte *NAND* et une porte *NOT* comme sur la figure 4.5 ci-dessous.

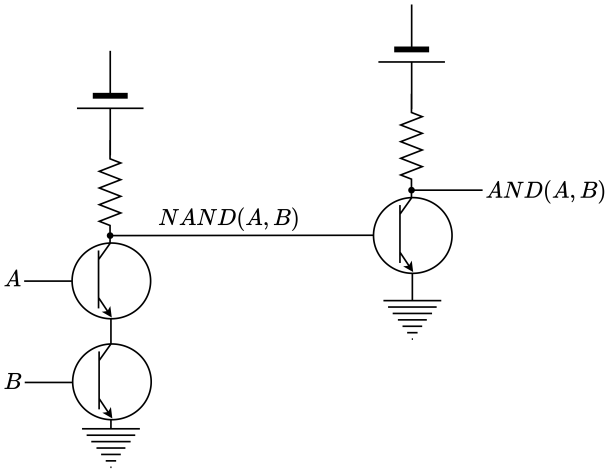


FIGURE 4.5 – Fonction logique *AND* réalisée grâce à des transistors au sein d'un circuit électronique. Ici, la combinaison d'une porte *NAND* et d'une porte *NOT* a servi à réaliser le circuit. Par souci de lisibilité, les légendes pour la source d'énergie et la terre sont omises ainsi que les résistances sur les lignes d'input.

De tels circuits électroniques peuvent maintenant être abstraits comme les portes logiques discutées dans la sous-section précédente.

Avec ces combinaisons de transistors, nous sommes maintenant parés pour passer à l'étape suivante : considérer la combinaison de plusieurs portes logiques afin de construire un circuit dit « **combinatoire** ». Ce sujet sera traité en section 4.4.

4.3 Algèbre de Boole

Nous sommes maintenant proches de combiner plusieurs portes logiques entre elles afin de constituer des circuits capables de réaliser n'importe quelle table de vérité. Cependant, comme on a pu le voir plus haut dans l'exemple où nous avons combiné plusieurs transistors pour obtenir un circuit réalisant la fonction *AND*, il est important de comprendre les mathématiques liées au traitement binaire de l'information afin de pouvoir énoncer des propositions utiles telles que *NOT(NAND(A,B))=AND(A,B)*.

Pour des raisonnements particulièrement simples, l'intuition humaine est suffisante, mais dès lors que nous abordons des combinaisons plus complexes ou que nous voulons formellement prouver des propositions, un cadre mathématique est désirable. En effet, toutes les tables de vérité ne sont pas aussi intuitives à retranscrire sous forme de circuit que la table 4.1. Nous nous intéressons donc ici à l'algèbre de Boole, développée autour de 1850 par George Boole, qui avait dans l'idée de formaliser une algèbre du raisonnement, une algèbre de la logique. Elle constitue une formalisation mathématique abstraite qui capture les règles de logique classique (un point de vocabulaire est donné en annexe H concernant le lien avec les structures mathématiques que l'on peut prêter à l'algèbre de Boole ; mais dans le reste de ce document, nous adoptons une approche pragmatique et non formelle).

4.3.1 Notation et priorité des opérations

Précédemment, nous avons décidé de noter 0 et 1 les deux valeurs possibles pour convoyer l'information binaire (on parle de « valeurs Booléennes [4] ») ; nous garderons ici cette notation. Cependant, afin de simplifier l'écriture des propositions logiques que nous formulerons, nous allons maintenant utiliser les notations décrites dans la table 4.9 pour les fonctions logiques NOT, OR et AND. L'ordre de priorité des opérations (comme pour les opérations arithmétiques traditionnelles) est également indiqué. Ainsi, par exemple :

$$\underbrace{0 \cdot 1}_{=0} + \underbrace{!0}_{=1} = 0 + 1 = 1, \tag{4.1}$$

tandis que :

$$0 \cdot \underbrace{(1+!0)}_{=1} = 0 \cdot 1 = 0. \tag{4.2}$$

4.3.2 Règles de manipulation

La fonction $f(A,B) = NAND(A,B) = NOT((AND(A,B)))$, que nous avons déjà manipulée dans la section précédente, s'écrit dans notre nouvelle notation, $f(A,B) = !(A \cdot B)$. Si l'on reprend l'exemple donné plus haut pour réaliser la porte logique AND, à savoir

4. Par la suite, en programmation, on parlera également de « variables Booléennes », qui seront des variables ne pouvant prendre que deux valeurs, usuellement 0 ou 1, ou encore True et False.

TABLE 4.9 – Notations utilisées pour les fonctions logiques NOT, OR et AND en logique Booléenne ainsi que leur ordre de priorité.

Priorité	Fonction logique	Notation	Exemple
Haute	*NOT(x)*	$!x$	$!0 = 1$
Moyenne	*AND(x,y)*	$x \cdot y$	$1 \cdot 0 = 0$
Faible	*OR(x,y)*	$x + y$	$1 + 1 = 1$

$f(A, B) = NOT(NAND(A, B))$, cela s'écrit maintenant :

$$f(A, B) = !(!(A \cdot B)). \tag{4.3}$$

On voit bien qu'il doit exister un ensemble de règles permettant de simplifier grandement des propositions Booléennes. Ici, à l'évidence, on voit que $!(!x) = x$. Cette double négation est souvent utilisée par inadvertance dans le langage naturel. Par exemple, « ce n'est pas faux » signifie « c'est vrai », et « ce n'est pas vrai » signifie « c'est faux ».

Il paraît maintenant clair qu'une fonction logique donnée peut être exprimée (et donc réalisée) de plusieurs façons différentes, dont certaines sont plus longues que d'autres. C'est là une troisième raison d'étudier l'algèbre de Boole : il est courant, en manipulant une expression Booléenne qui semble complexe, de se rendre compte qu'elle est en réalité remarquablement simple après quelques reformulations. Or, lorsqu'on applique cela à des circuits électroniques ou à des algorithmes, de vraies différences de performance, de coût ou de robustesse peuvent en découler.

On peut dégager de l'algèbre de Boole un ensemble de règles très fréquemment utiles et résumées au sein de la table 4.11. Le but de ce cours n'est pas de toutes les démontrer. Cependant, comme elles portent toutes sur des propositions impliquant très peu de variables discrètes, il est aisé de les démontrer en établissant une liste exhaustive des possibilités ; c'est ce que nous faisons, pour l'exemple, concernant la preuve de $!(x + y) = !x!y$ dans la table 4.10 ci-dessous.

TABLE 4.10 – Table de vérité pour $!(x + y)$ ainsi que pour $!x!y$. On s'aperçoit par simple épuisement des possibilités que les deux formulations sont équivalentes.

x	y	$!(x + y)$	$!x!y$
0	0	$!(0 + 0) = !0 = 1$	$!0\ !0 = 1 \cdot 1 = 1$
0	1	$!(0 + 1) = !1 = 0$	$!0\ !1 = 1 \cdot 0 = 0$
1	0	$!(1 + 0) = !1 = 0$	$!1\ !0 = 0 \cdot 1 = 0$
1	1	$!(1 + 1) = !1 = 0$	$!1\ !1 = 0 \cdot 0 = 0$

4.3.3 Portes logiques universelles

Comme nous l'avons vu en section 4.2.2, un seul type de relais permet de créer toutes les portes, en exploitant le fait que deux dispositifs, selon qu'ils soient branchés en série ou

TABLE 4.11 – Règles de l'algèbre de Boole utiles dans ce cours déclinées dans leurs version à propos de la fonction AND et de la fonction OR.

Nom	Version AND	Version OR
Valeur nulle	$0 \cdot x = 0$	$1 + x = 1$
Valeur neutre	$1 \cdot x = x$	$0 + x = x$
Complément	$x!x = 0$	$x+!x = 1$
Commutativité	$xy = yx$	$x + y = y + x$
Associativité	$x(yz) = (xy)z$	$x + (y + z) = (x + y) + z$
Distributivité	$x(y + z) = xy + xz$	$x + yz = (x + y) \cdot (x + z)$
Idempotence	$xx = x$	$x + x = x$
Absorption	$x(x + y) = x$	$x + (xy) = x$
Théorème de De Morgan	$!(xy) = !x+!y$	$!(x + y) = !x!y$

en parallèle, permettent de réaliser physiquement la logique du OR et du AND. Mais qu'en est-il des fonctions logiques ? Existe-t-il une fonction à deux arguments qui permettrait à elle seule d'exprimer toutes les autres ?

La réponse est oui. Pour le montrer, commençons par montrer que l'une des trois fonctions NOT, AND et OR est réalisables grâce aux deux autres [5]. Par exemple, à l'aide de la version OR du théorème de De Morgan, il est aisé d'exprimer AND ainsi :

$$x \cdot y = !!x \cdot !!y = !(!x+!y), \tag{4.4}$$

tout comme on peut exprimer OR de la façon suivante en utilisant la version AND du théorème de De Morgan :

$$x + y = !!x+!!y = !(!x \cdot !y). \tag{4.5}$$

Il est maintenant clair que deux portes suffisent : soit NOT et AND, soit NOT et OR.

Cependant, la porte $NAND$ est précisément la combinaison de NOT et de AND. On peut donc tenter de formuler NOT et AND comme des fonctions de $NAND$ uniquement. Pour NOT, on a :

$$!x = !(x \cdot x) = NAND(x, x), \tag{4.6}$$

où l'idempotence a été utilisée. Pour AND, on a :

$$x \cdot y = ! \underbrace{!(xy)}_{\equiv NAND(x,y)} = !NAND(x, y), \tag{4.7}$$

ce qui, en vertu de 4.6, donne

$$x \cdot y = NAND\left(NAND(x, y),\ NAND(x, y)\right). \tag{4.8}$$

Nous avons donc montré que la porte $NAND$ est universelle.

5. Nous avons déjà vu que les autres portes, telles que $NAND$, NOR et XOR, sont des combinaisons de NOT, AND et OR. Ainsi, il nous suffit de montrer que ces trois dernières peuvent être réduites à une seule des autres pour montrer l'universalité de celle-ci.

On peut effectuer le même raisonnement avec OR :

$$!x = !(x + x) = NOR(x, x), \tag{4.9}$$

et

$$x + y = !\underbrace{!(x + y)}_{\equiv NOR(x,y)} = !NOR(x, y), \tag{4.10}$$

ce qui en vertu de 4.9 donne :

$$x + y = NOR\left(NOR(x, y),\ NOR(x, y)\right). \tag{4.11}$$

Nous avons donc montré que la porte NOR est universelle au même titre que $NAND$. Ainsi, les circuits formant un ordinateur électronique peuvent être réalisés à l'aide d'un unique type de porte logique, elle-même composée de deux transistors !

4.3.4 Exemple de simplification – Nombres premiers à 3 bits

Afin d'exemplifier l'utilisation des règles du calcul booléen, nous allons à présent étudier un problème non trivial. Examinons la table de vérité 4.12, qui associe à chaque entier de 0 à 7 une valeur de vérité quant au fait que cet entier est un nombre premier. On considère que 1 n'est pas premier. Comme il est ici plus difficile de formaliser une expression ou un circuit logique de façon intuitive (comme cela a été fait précédemment pour le problème de l'excès de vitesse, par exemple), nous allons appliquer les règles du calcul booléen aux lignes de la table de vérité qui nous intéressent.

TABLE 4.12 – Table de vérité pour $P(x, y, z)$, où $P = 1$ si et seulement le nombre décimal $n = z \cdot 2^2 + y \cdot 2^1 + x \cdot 2^0$ est premier. Les entrées de cette dernière sont x, y, z et son unique sortie est P.

z	y	x	P
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

On peut lister toutes les combinaisons des inputs qui donnent un output p égal à 1. Une telle expression, qui prend donc la forme d'une somme de produits, se nomme la **forme canonique** de l'expression. Ici, les combinaisons d'inputs qui donnent un output égal à 1 sont 010, 011, 101 et 111, et la forme canonique est donc $P = !zy!x + !zyx + z!yx + zyx$.

Simplifions maintenant la forme canonique de P :

$$P = !zy!x + !zyx + z!yx + zyx \tag{4.12}$$
$$= !zy(!x + x) + zx(!y + y) \tag{4.13}$$
$$= !zy + zx, \tag{4.14}$$

où la règle de distributivité pour AND a été utilisée pour passer de l'équation 4.12 à l'équation 4.13. Enfin, la règle du complément pour OR a été utilisée pour passer de l'équation 4.13 à l'équation 4.14.

L'expression simplifiée de P contient une fois l'opération logique OR, deux fois l'opération logique AND et une fois l'opération logique NOT. On pourrait donc réaliser un circuit logique qui détermine si notre nombre est premier de la façon indiquée sur la figure 4.6 ci-dessous. Il est important de noter que, dans ce cas relativement simple, on peut toujours lier l'expression logique finale à notre intuition. Ici, notre résultat montre que le nombre est premier à chaque fois que les bits x et z sont à 1 (premier terme de l'expression finale). Par ailleurs, le nombre est également premier dans un autre cas : celui où z est à 0 et y est à 1 (second terme de l'expression finale). Grâce à l'application de règles du calcul booléen, nous avons donc réussi à condenser en deux « conditions » ce que la table de vérité semblait dire en huit.

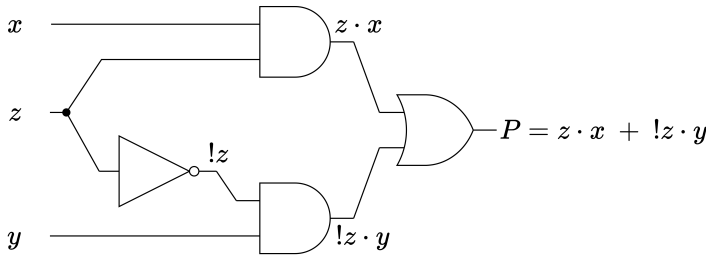


FIGURE 4.6 – Fonction P relative à la table 4.12 réalisée grâce à des portes logiques au sein d'un circuit combinatoire. L'expression d'output des portes est indiqué à proximité de ces dernières.

4.3.5 Méthodes des mintermes et des maxtermes

Dans l'exemple exploré en section 4.3.4, nous avons appliqué la méthode suivante :

1. Nous avons écrit la table de vérité de la fonction logique à réaliser.
2. Nous avons listé les combinaisons des inputs qui donnent un output égal à 1 sous la forme d'une somme de produits (forme canonique).
3. Nous avons simplifié cette fonction logique en utilisant les règles de l'algèbre de Boole.

Cette méthode se nomme la méthode des **mintermes** car l'expression de départ, une somme de produits, regroupe les termes tels qu'au minimum l'un d'entre eux doit être vrai pour que le tout soit vrai.

La méthode des **maxtermes**, à l'inverse, consiste à effectuer la négation de combinaisons d'inputs livrant un output égal à 0, et de les lier entre elles par des ET logiques. Si on l'applique au problème du nombre premier à trois bits (fonction P en table 4.12), on obtient :

$$P = (z + y + x)(z + y + !x)(!z + y + x)(!z + !y + x), \tag{4.15}$$

dont l'expression peut éventuellement être simplifiée.

On peut voir de la manière suivante la raison pour laquelle un minterme (chaque terme de la somme 4.12 est un « minterme ») s'appelle ainsi : il est minimum dans le sens où une seule ligne de la table de vérité peut le vérifier. À l'inverse, un maxterme (chaque terme du produit 4.15 se nomme un « maxterme ») est maximum dans le sens où il est vrai partout sauf sur une ligne de la table de vérité.

4.3.6 Méthode du complément des lignes fausses

Enfin, notons que l'on peut lister les combinaisons des inputs qui donnent un output égal à 0. On obtient alors une expression résumant les conditions pour que l'output ne soit *pas* égal à 1. Appliquons cette dernière méthode au problème du nombre premier à trois bits. On peut lister toutes les combinaisons des inputs x, y et z qui donnent un output égal à 0. Ces inputs sont 000, 001, 100 et 110. Autrement dit :

$$\begin{aligned} !P &= !z!y!x + !z!yx + z!y!x + zy!x && (4.16)\\ &= !z!y(!x + x) + z!x(!y + y) && (4.17)\\ &= !z!y + z!x, && (4.18) \end{aligned}$$

où, comme pour les mintermes, la règle de distributivité pour AND ainsi que la règle du complément pour OR ont été utilisées. Enfin, comme nous voulons connaître l'expression pour P, nous utilisons le fait que $P = !(!P)$:

$$\begin{aligned} P &= !(!z!y + z!x) && (4.19)\\ &= !(!z!y) \cdot !(z!x) && (4.20)\\ &= (!!z + !!y) \cdot (!z + !!x) && (4.21)\\ &= (z + y) \cdot (!z + x), && (4.22) \end{aligned}$$

où le théorème de De Morgan dans ses versions AND et OR a été utilisé plusieurs fois. Le circuit correspondant est montré sur la figure 4.7 ci-dessous. Ce circuit réalise une fonction tout à fait équivalente à celui de la figure 4.6 obtenu avec les mintermes. En l'occurrence, on a besoin du même nombre de portes logiques dans chacun des deux circuits obtenus[6].

En fin de compte, on choisit en général la méthode la plus facile à utiliser en fonction de la table de vérité. Si cette dernière est dominée par des valeurs d'output de 1, on préférera les maxtermes ou encore la méthode qui vient d'être vue, tandis que l'on préférera les mintermes dans le cas contraire.

6. Mais ce n'est pas vrai dans le cas général. Dans la réalité, il peut être important de choisir le circuit le plus simple ou le plus performant à réaliser. Par ailleurs, il faut noter qu'une infinité de circuits équivalents peuvent toujours être produits (par exemple en utilisant les doubles négations et les compléments).

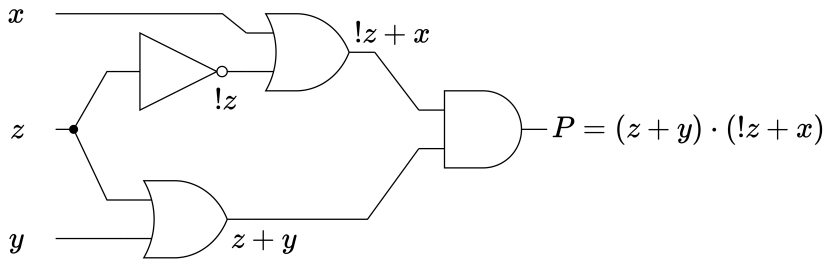


FIGURE 4.7 – Fonction P relative à la table 4.12 réalisée grâce à des portes logiques au sein d'un circuit combinatoire. L'expression pour P a été trouvée en prenant le complément des lignes fausses de la table de vérité. L'output des portes est indiqué à proximité de ces dernières.

Pour conclure cette comparaison, assurons-nous que les deux expressions obtenues, à savoir $P_m = zx+!zy$ pour les mintermes et $P_c = (z+y)(!z+x)$ pour la méthode du complément, sont bien équivalentes. On peut développer puis simplifier P_c :

$$P_c = (z+y)(!z+x) \tag{4.23}$$
$$= \underbrace{z!z}_{=0} + zx + !zy + xy \tag{4.24}$$
$$= zx+!zy + xy, \tag{4.25}$$

où l'on a utilisé la règle de distributivité et la règle du complément pour AND. À première vue, il semble qu'un terme xy en trop soit apparu. Considérons néanmoins les deux cas possibles :

- Soit $xy = 0$, et dans ce cas $P_c = zx+!zy + 0 = P_m$.
- Soit $xy = 1$, et dans ce cas $x = 1$ et $y = 1$, et donc $P_c = z \cdot 1 + !z \cdot 1 + 1$, et le terme xy ne change aucunement le résultat.

Autrement dit, les deux expressions sont bien équivalentes. Il faut noter que le dernier cas mentionné, qui montre que $zx+!zy + xy = zx+!zy$, est tout à fait général et souvent désigné par le nom de « théorème du consensus », ou encore « théorème de l'élément redondant ». Il peut également être démontré en utilisant des règles déjà vues :

$$zx+!zy + xy = zx+!zy + xy(!z+z) \tag{4.26}$$
$$= zx+!zy + xy!z + xyz \tag{4.27}$$
$$= zx(1+y)+!zy(1+x) \tag{4.28}$$
$$= zx+!zy. \tag{4.29}$$

4.3.7 Méthode de Karnaugh

Bien qu'il n'existe pas de méthode à la fosi générale et simple qui garantisse une simplification optimale d'expressions booléennes correspondant à une table de vérité, il existe néanmoins des méthodes qui permettent de simplifier des expressions de manière relativement efficace. La méthode de Karnaugh, développée au milieu des années 50 par un ingénieur des laboratoires Bell, est l'une d'elles. Elle consiste à regrouper les mintermes ou

les maxtermes de la table de vérité en blocs dont le nombre d'éléments est une puissance de 2, et à les simplifier en utilisant les règles de l'algèbre de Boole. Cette méthode est particulièrement efficace pour les fonctions logiques à 3 ou 4 variables.

Dans une table de vérité standard, tous les inputs possibles, ainsi que l'output, sont disposés en colonnes, c'est-à-dire au sein d'un tableau dont toutes les entrées sont côte-à-côte. Pour appliquer la méthode de Karnaugh, en revanche, nous allons disposer les inputs dans un tableau à double entrée, de telle sorte que :

1. Seule la valeur d'output est indiquée dans les cases.
2. Lorsqu'on passe d'une ligne d'input (ou colonne d'input) à une autre qui lui est adjacente, un seul bit change.

Suite à cela, nous pouvons soit appliquer la méthode de Karnaugh pour les mintermes, soit l'appliquer pour les maxtermes. Nous donnons ici uniquement la version pour les mintermes. Les étapes à suivre sont alors les suivantes :

1. Créer des groupes rectangulaires contenant uniquement des cellules valant 1.
2. Un groupe doit contenir un nombre de cellules qui est une puissance de 2.
3. Un groupe peut être périodique (« sortir » par un côté de la table et « revenir » par l'autre.)
4. Afin de simplifier le processus, créer aussi peu de groupes que possibles, et des groupes aussi grands que possibles.
5. Tous les 1 doivent appartenir à au moins un groupe (mais il est possible qu'une valeur appartienne à plusieurs groupes). Aucun 0 ne doit être dans un groupe.

Finalement, on peut écrire l'expression booléenne correspondant aux groupes sélectionnés. Pour profiter du fait que nous créons des groupes aussi grands que possible, il est important de repérer des variables **redondantes** en leur sein[7]. Une variable est redondante au sein d'un groupe si le bit qui lui est associé prend toutes les valeurs possibles au sein des cellules du groupe. L'expression booléenne associée au groupe peut alors ignorer la variable en question.

La table 4.13 ci-dessous montre une façon (mais ce n'est pas la seule) de choisir les groupes pour la fonction logique P prise en exemple jusqu'ici. Le groupe en gris possède x comme variable redondante. L'expression qui en découle est $P_{gris} = !zy$. De façon similaire, on remarque que le groupe en vert possède y comme variable redondante, et que l'expression correspondante est donc $P_{vert} = zx$. Ainsi, l'expression finale associée à la table de vérité pour P est bien $P = !zy + zx$, comme trouvé en équation 4.14

Pour les tables de vérité simples, où l'intuition humaine est forte, ou bien où la méthode des mintermes et maxtermes fonctionne bien, il peut paraître fastidieux d'appliquer la méthode de Karnaugh. Cependant, elle est d'un intérêt particulier pour toutes les autres situations, étant donné la nature algorithmique de son fonctionnement ; elle permet de systématiser le développement d'expressions booléennes, ce qui peut être très utile dans bien des cas.

7. Dans le cas extrême où nous ne créons que des groupes d'une seule cellule, alors la méthode revient à celle des mintermes.

TABLE 4.13 – Table de Karnaugh pour la fonction logique P relative à la table 4.12. Pour des raisons pédagogiques, l'expression littérale est donnée en gras en première ligne et colonne, suivie de l'expression sous forme binaire, en gras également. Les groupes choisis sont représentés en couleur. Ici, il y a deux groupes de 1 ligne par 2 colonnes.

		!y!x	**!yx**	**yx**	**y!x**
		00	**01**	**11**	**10**
!z	**0**	0	0	1	1
z	**1**	0	1	1	0

4.4 Exemples de circuits logiques combinatoires

Comme nous l'avons dit plus haut, un circuit combinatoire est un circuit logique dont la sortie dépend d'une combinaison de portes logiques. Nous étudions dans cette section deux circuits combinatoires d'une grande importance en informatique : le multiplexeur et l'additionneur.

4.4.1 Multiplexeur

Multiplexeur à 2 voies

Il est courant de décrire un algorithme comme une certaine condition de sélection, que nous nommons ici S. Par exemple : « Si S est faux, alors la valeur de sortie est A, sinon la valeur de sortie est B. » Un circuit logique permettant de réaliser cette fonction est appelé un multiplexeur. Il est représenté par un symbole comme celui de la figure 4.8 ci-dessous.

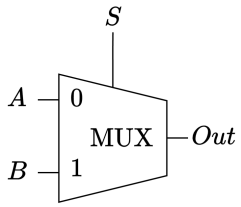


FIGURE 4.8 – Symbole d'un multiplexeur. Si S est nul, la sortie vaut A, sinon elle vaut B – cela ne dit rien de la valeur de A ou de B.

Il est important de comprendre que le multiplexeur renvoie une **ligne de donnée**, quelle que soit la valeur de cette dernière, en fonction de la valeur de la **ligne de contrôle**. En effet, dans la phrase d'exemple donnée plus haut, la valeur de sortie est soit A soit B, peu importe la valeur de A et de B. En particulier, la notation sur le symbole du multiplexeur peut paraître trompeuse : le zéro à côté de la première entrée, ou le 1 à côté de la seconde, ne nous dit rien de la valeur de l'entrée en question (qui comme toute entrée peut être soit 0, soit 1), mais nous informe plutôt quant au numéro de sélection qui est

attribué à chaque ligne de donnée. Le multiplexeur, s'il était décrit par un code, serait de la forme :

```
if S == 0:
    out = A
else:
    out = B
```

Expression logique d'un multiplexeur à 2 voies

La table 4.14 ci-dessous représente la table de vérité d'un multiplexeur à 2 voies.

TABLE 4.14 – Table de vérité pour d'un multiplexeur à 2 voies

S	In_1	In_2	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

On constate que la table de vérité d'un multiplexeur à 2 voies est exactement la même que celle qui décrit le problème des nombres premiers à 3 bits précédemment discutés en table 4.12. Cela signifie que le circuit logique qui détermine si un nombre à 3 bits est premier ou non est en fait équivalent à un multiplexeur à 2 voies. Le lecteur trouvera donc de façon indirecte une discussion de la réalisation électronique d'un multiplexeur à 2 voies au sein des sections 4.3.4 et 4.3.6, en gardant en tête que le bit z est ici la ligne de contrôle S, que le bit y est l'entrée In_1 et que le bit x est l'entrée In_2. Enfin, la sortie P est l'output *out* du multiplexeur. Afin de nous exercer à la méthode des mintermes, nous en redonnons ici une démonstration rapide. On a donc, d'après la table de vérité 4.14 :

$$\begin{aligned} out = \; & In_1!In_2!S + In_1In_2!S + !In_1In_2S + In_1In_2S && (4.30)\\ = \; & In_1!S(!In_2 + In_2) + In_2S(!In_1 + In_1) && (4.31)\\ = \; & In_1!S + In_2S && (4.32) \end{aligned}$$

Multiplexeur à k voies

Dans le cas où plus de deux choix sont possibles, la ligne de contrôle S est étendue à plusieurs bits. Par exemple, un multiplexeur à 4 entrées (nommé « multiplexeur à 4 voies ») nécessitera 2 bits de contrôle, tandis qu'un multiplexeur de 3 bits de contrôle

autorise 8 choix. De façon générale, un multiplexeur à n lignes de contrôle est un multiplexeur à 2^n voies. Autrement dit :

$$k = 2^n \iff n = \log_2(k), \tag{4.33}$$

où n est le nombre de bits de contrôle et k le nombre de voies du multiplexeur. Un arrondi vers l'entier supérieur est à effectuer puisque le nombre de bits doit être entier ; on gardera donc $n = \lceil \log_2(k) \rceil$

À titre d'exemple, la figure 4.9 ci-dessous montre un multiplexeur à $k = 4$ voies. Comme il est peu commode d'indiquer toutes les voies sur un multiplexeur à plus que

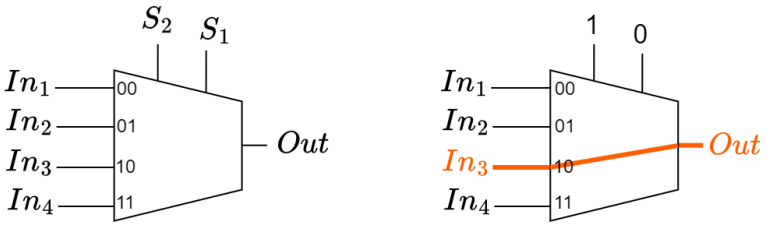


FIGURE 4.9 – (gauche) Multiplexeur à 4 voies. (droite) Représentation informelle de la sélection d'une ligne de données parmi les 4 possibles.

quelques entrées, on abstrait en général ces dernières, comme indiqué sur la figure 4.10, qui illustre le symbole générique d'un multiplexeur à voies multiples au sein d'un circuit électronique. Dans un tel cas, une barre oblique indique au lecteur du diagramme qu'une ligne du dessin correspond à de multiples lignes dans le circuit. Le nombre de bits impliqués est également indiqué au-dessus de la barre oblique.

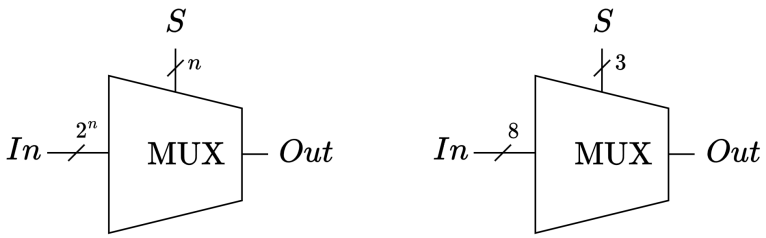


FIGURE 4.10 – (gauche) Symbole d'un multiplexeur à voies multiples. (droite) Exemple pour un multiplexeur à 8 voies.

Ci-dessous, nous illustrons une façon de réaliser un multiplexeur à k voies *via* un arbre de multiplexeurs à 2 voies. Dans cette approche, le nombre de portes que doit traverser le signal d'un multiplexeur à k voies est proportionnel à $\log(k)$. On peut intuitivement s'en convaincre si l'on essaie de réaliser un multiplexeur à 4 ou 8 voies en combinant des multiplexeurs à 2 voies. On s'aperçoit qu'un arbre binaire de multiplexeurs à 2 voies est nécessaire, où la valeur de chaque bit de S est distribuée entre les différents étages de l'arbre, en commençant par S_1 du côté des « feuilles ». Si le nombre de feuilles de l'arbre est égal à k, alors le nombre d'étages vaut $\log_2(k)$. Ceci est illustré pour un multiplexeur à 8 voies sur la figure 4.11 ci-dessous. Comme les multiplexeurs (ou leurs inverse, discuté plus bas) sont impliqués dans la sélection des données dans la mémoire de l'ordinateur,

la longueur du chemin que doit parcourir le signal en leur sein est un facteur important pour la vitesse de l'ordinateur. Nous réutiliserons donc ce résultat dans le chapitre dévolu à l'architecture des ordinateurs.

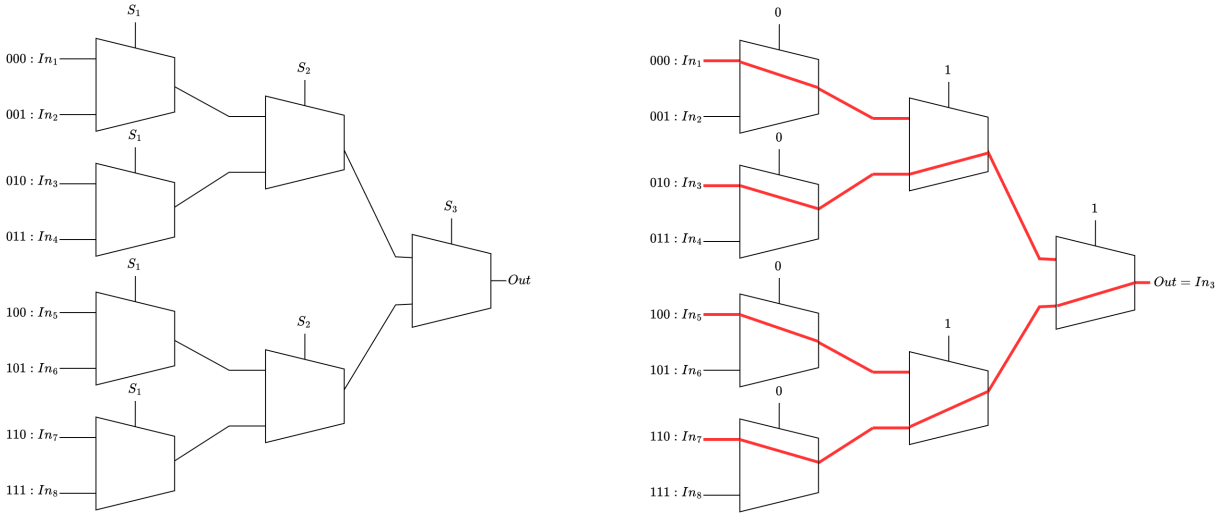


FIGURE 4.11 – Combinaisons de multiplexeurs à 2 voies pour réaliser un multiplexeur à 8 voies. La figure de droite montre l'exemple de la sélection de l'entrée In_7, correspondant à $S = 110$. Notons que l'ordre des lignes de contrôle va du bit de poids faible de S (côté feuilles) au bit de poids fort (côté racine).

Pour terminer, remarquons qu'un dispositif effectuant le travail inverse de celui d'un multiplexeur se nomme un **démultiplexeur**. Un tel dispositif prend en entrée un bit de donnée et un ensemble de bits de contrôle désignant le numéro d'une ligne de sortie. Le bit de donnée est alors transmis à l'une des 2^n sorties possibles désignées par les n bits de la ligne de contrôle. Les démultiplexeurs sont des circuits logiques importants dans les mécanismes permettant de lire et d'écrire dans la mémoire d'un ordinateur (cf. chapitre 5).

4.4.2 Additionneur

Au sein d'un processeur, l'additionneur est un circuit logique faisant partie de l'unité arithmétique et logique (cf. section 5.1.2) et permettant d'additionner deux nombres. Ici, nous nous intéressons à deux nombres binaires exprimés sous la forme de mots de même longueur, à l'image de ce qui a été vu précédemment en section 2.1.4. Dans ce qui suit, nous allons utiliser les variables d'input nommées a_i et b_i, qui représentent les bits de poids i des deux nombres a et b reçus en entrée par l'additionneur. Les outputs de l'additionneur sont les bits s_i de la somme s ainsi que le bit R de la retenue finale.

Version naïve

Lorsqu'on additionne deux nombres binaires et que l'on a la garantie que leur résultat n'est pas plus grand que 1, alors ce dernier est simplement le résultat de l'opération XOR

sur les deux bits d'input. En effet, les seuls cas possibles sont $0+0=0$, $0+1=1$ et $1+0=1$.

Voici donc un exemple qui fonctionnerait pour des mots de 5 bits, si l'on avait la garantie qu'aucune des colonnes ne génère de retenue :

	a_4	a_3	a_2	a_1	a_0
$+$	b_4	b_3	b_2	b_1	b_0
$=$	$a_4 \oplus b_4$	$a_3 \oplus b_3$	$a_2 \oplus b_2$	$a_1 \oplus b_1$	$a_0 \oplus b_0$

Version générale

Malheureusement, comme nous l'avons vu en section 2.1.4, les retenues sont omniprésentes au sein des calculs, et la simple utilisation de XOR sur chaque paire de bits d'inputs ne peut mener à des résultats corrects en général. En particulier, la difficulté vient ici du fait que chaque colonne peut recevoir une retenue provenant de sa voisine de poids faible, tout comme elle peut donner lieu à une retenue à reporter sur la colonne voisine de poids fort. Par conséquent, dans le cas général, chaque colonne i de l'addition implique :

- Deux bits a_i et b_i (input) ;
- Un bit r_i de retenue provenant de la colonne à sa droite (input) ;
- Un bit s_i de somme (output) ;
- Un bit r_{i+1} de retenue à reporter sur la colonne à sa gauche (output).

Ainsi, on peut modéliser le comportement d'une seule colonne par la table de vérité 4.15 ci-dessous. Comme nous nous intéressons à une seule colonne, l'indice i est omis et r_{i+1} est noté R.

TABLE 4.15 – Table de vérité pour un additionneur. Les inputs sont les bits a et b des deux nombres à additionner, ainsi que la retenue r provenant de la colonne à droite. Les outputs sont le bit de somme s et le bit de retenue à reporter R.

r	a	b	s	R
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

En examinant la table de vérité, on peut appliquer la méthode des mintermes pour s :

$$\begin{aligned} s &= !r!ab + !ra!b + r!a!b + rab && (4.34)\\ &= !r(!ab + a!b) + r(!a!b + ab) && (4.35)\\ &= !r(a \oplus b) + r!(a \oplus b) && (4.36)\\ &= r \oplus (a \oplus b), && (4.37) \end{aligned}$$

où l'on a utilisé la définition de XOR : $x \oplus y \equiv x!y + !xy$. Il est bien entendu possible d'utiliser une version sans XOR, mais cette dernière variante est choisie par commodité, puisqu'elle fait intervenir une seule fonction logique.

De même, cherchons une expression pour R qui privilégie l'utilisation du XOR. On pourrait une fois de plus appliquer la méthode des mintermes, mais on voit aisément que la condition pour que R vale 1 est :

- Soit a et b valent 1 tous les deux (peu importe la valeur de r) ;
- Soit r vaut 1 et uniquement l'un des deux bits a et b vaut 1.

Cela s'exprime ainsi :

$$R = ab + r(a \oplus b). \tag{4.38}$$

Par conséquent, si l'on veut réaliser un circuit logique correspondant à la table de vérité de l'additionneur, il faut que ses deux outputs réalisent les fonctions données par les équations 4.38 et 4.37. Le circuit logique correspondant est montré sur la figure 4.12 ci-dessous.

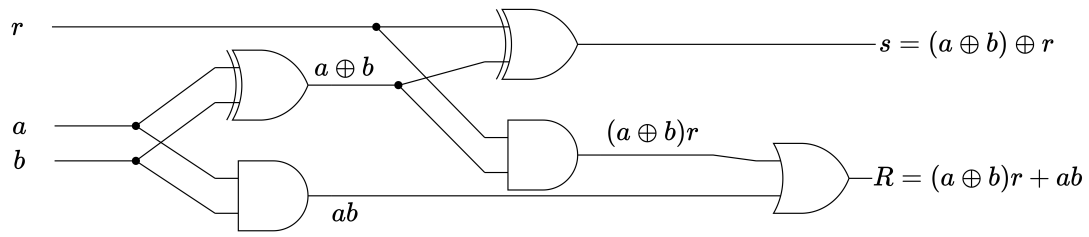


FIGURE 4.12 – Additionneur (cf. table 4.15) réalisé grâce à des portes logiques au sein d'un circuit combinatoire.

Finalement, pour parvenir à notre objectif d'additioneur deux mots a et b et non pas deux bits a et b, il nous reste encore à combiner plusieurs additionneurs entre eux. Afin de simplifier les diagrammes à dessiner, nous allons dorénavant abstraire le circuit donné en figure 4.12 par le symbole donné dans la figure 4.13 ci-dessous.

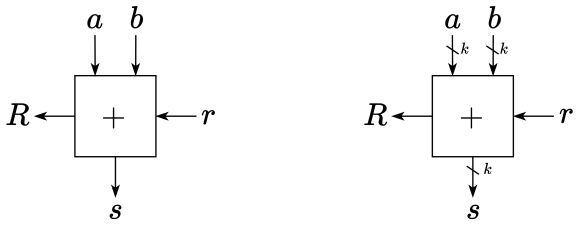


FIGURE 4.13 – Symboles d'un additionneur complet pour des mots de 1 bit (à gauche) et de k bits (à droite). Ici, le sens de l'information est donné par les flèches de façon informelle (les flèches ne sont pas à indiquer au sein d'un véritable diagramme).

Afin d'implémenter notre algorithme d'addition de deux mots binaires avec un nombre arbitraire de bits, il suffit alors de relier les additionneurs de façon à ce que la retenue d'output de chaque additionneur soit reliée à la retenue d'input de l'additionneur à sa gauche. Par exemple, le circuit logique correspondant à l'addition de deux mots binaires de 4 bits est montré sur la figure 4.14 ci-dessous. Il faut noter que cette façon de procéder

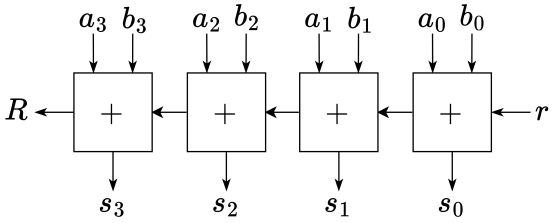


FIGURE 4.14 – Combinaisons de plusieurs additionneurs afin de réaliser l'addition de deux mots binaires de 4 bits. Pour cela, il faut faire en sorte que $r_i = R_{i-1}$ pour toutes les colonnes qui ont deux voisins.

montre uniquement l'essence d'un circuit logique permettant d'additionner deux nombres ; en réalité, de nombreuses optimisations qui sortent du cadre de ce cours sont effectuées afin d'améliorer les performances des processeurs. Nombre de ces optimisations portent notamment sur l'anticipation des valeurs de retenue.

Finalement, notons que la retenue initiale r vaut 0 dans le cas où l'on cherche simplement à additionner a et b ; comme on va le voir, r a cependant une utilité dans le cas de la soustraction. Dans le cas d'un entier non signé, la retenue finale R, quant à elle, permet de signaler quand le résultat final de l'addition ne peut être exprimé avec les k bits [8]. Dans le cas où ce débordement est ignoré, le comportement de l'additionneur correspond à la règle de débordement cyclique discutée dans le chapitre 2.3.2.

Soustraction de nombres

Plutôt que de procéder à partir de la table de vérité comme nous l'avons fait dans le cas de l'additionneur, on peut envisager la soustraction de deux nombres comme une addition impliquant un nombre négatif. Cela faisait partie des critères que nous avons fixés dans le chapitre 2.4 pour décider d'un bon codage des entiers relatifs. En effet, cela permet de réutiliser le circuit de l'additionneur.

Comme nous l'avons vu dans le chapitre 2.4.3, le codage d'un nombre négatif est obtenu en inversant tous les bits de sa valeur absolue, puis en y ajoutant une unité. Ainsi, la soustraction $a - b$ correspond à l'addition $a + (-b)$, où $-b$ est obtenu comme $not(b) + 1$. Afin de tirer profit de la retenue d'entrée de l'additionneur, jusqu'ici non utilisée, nous pouvons donc envisager $a - b$ comme étant égal à $a + not(b) + 1$. Le circuit permettant de réaliser une soustraction et réutilisant l'additionneur est présenté dans la figure 4.15 ci-dessous.

8. Dans le cas des entiers signés, cette détection est plus compliquée et repose sur la vérification de la cohérence du changement de signe du nombre

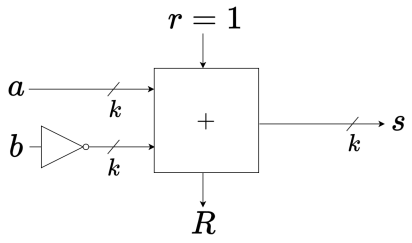


FIGURE 4.15 – Circuit permettant de réaliser une soustraction entre deux entiers a et b.

4.5 Circuits logiques séquentiels

Nous avons jusqu'ici considéré des circuits au sein desquels la notion de temps n'entrait pas en compte. Nous avons procédé comme si toutes les quantités physiques représentant les données s'établissaient instantanément. Cependant, dans la réalité, une certaine durée s'écoule entre l'application d'une tension électrique en entrée et la stabilisation d'une tension électrique en sortie du circuit. Ce **délai de propagation** est d'autant plus important que le nombre de portes logiques que doit traverser le signal est grand.

Le circuit de l'additionneur vu plus haut est un bon exemple du problème que le délai de propagation engendre. En effet, chaque additionneur complet du bit i doit attendre que la tension soit stabilisée en sortie de l'additionneur $i-1$ qui le précède afin de pouvoir reprendre la retenue de ce dernier pour le calcul.

Cela constitue un problème important car, dès lors, il devient difficile de déterminer à un instant donné si la sortie d'un circuit est valide en l'état ou bien s'il faut attendre un peu plus longtemps. Ce problème, évidemment, se répercute en cascade sur les circuits qui utilisent l'output en question comme valeur d'entrée. Ainsi, il est particulièrement intéressant de pouvoir incorporer une notion temporelle à nos circuits logiques afin de maîtriser ces délais de propagation et de fixer des instants où les sorties sont valides.

4.5.1 Circuits synchrones

On peut imaginer différentes façons de prendre en compte la notion de temps ; nous traitons ici celle qui est utilisée au sein des circuits séquentiels modernes les plus communs. Il s'agit de circuits **synchrones**.

Au sein d'un circuit synchrone, une horloge [9] « bat la mesure ». Les éléments du circuit qui sont synchronisés sur cette horloge, dont les pulsations leur parviennent à intervalle régulier, se nomment les **éléments d'état**. L'état du système ne peut changer qu'au signal de l'horloge ; il oscille entre un signal « bas », ici noté 0, et un signal « haut », ici noté 1. Evidemment, plus les pulsations sont rapprochées dans le temps, plus le circuit peut effectuer une tâche rapidement ; mais il faut entre autres s'assurer que le laps de

9. Dans les machines modernes, le signal d'horloge est produit grâce à un matériau piézoélectrique, dont la déformation et la polarisation électrique sont interdépendants ; couplé à une source d'énergie, cela permet de générer un signal électrique périodique.

temps écoulé entre deux pulsations soit supérieur à la durée de stabilisation de la tension électrique en sortie des portes logiques. La figure 4.16 ci-dessous montre un exemple de signal possible. Chaque période complète du signal est appelée un **cycle**.

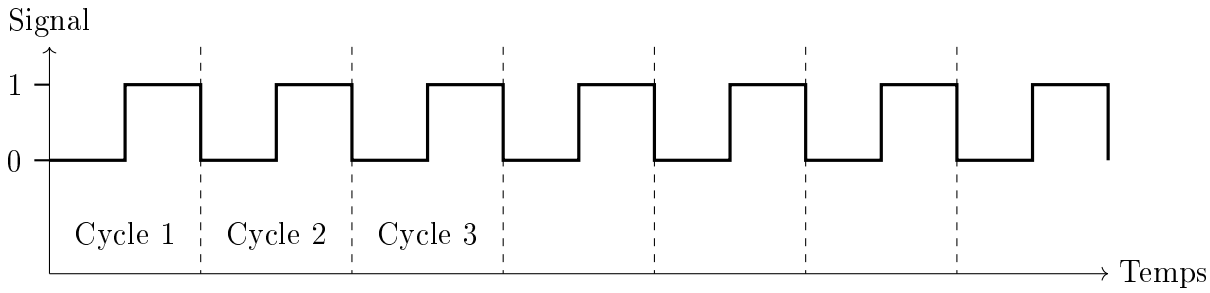


FIGURE 4.16 – Signal d'horloge idéalisé, ici sous forme d'onde carrée.

Au sein de la figure 4.17 ci-dessous est illustré un exemple de signal d'horloge, ainsi que les signaux aux entrées et à la sortie d'un circuit logique représentant la porte logique *AND*. On peut voir que la tension électrique en sortie du circuit prend plus de temps à s'adapter que l'entrée, en raison du délai de propagation du signal au sein du circuit électronique implémentant la fonction logique *AND*. Si l'on s'intéresse à la valeur précise de la tension électrique à un instant arbitraire, on peut tout à fait mesurer que les deux entrées sont à 1 mais la sortie à 0, ou bien que l'une des entrées est à 0 mais pas la sortie, ce qui serait incohérent avec la fonction assignée à la porte logique *AND*.

Ainsi, il est important de ne pas considérer les valeurs des signaux à un instant quelconque, mais uniquement à la fin d'un cycle d'horloge, afin de pouvoir faire abstraction de ces processus d'ordre physique. Autrement dit, l'introduction d'une horloge permet de se débarrasser de considérations d'ordre électronique ou physique, pour autant que la période d'oscillation du signal soit adaptée aux délais de propagation des circuits utilisés.

4.5.2 Circuits séquentiels

En plus du problème des délais de propagation, un autre aspect reste à régler afin de pouvoir implémenter aisément des algorithmes au sein d'une machine. En effet, nous ne disposons pour l'instant d'aucun moyen de faire des calculs qui prennent en compte l'historique des résultats. Cela serait utile, par exemple pour calculer la somme d'une liste de nombres.

Considérons par exemple le code Python ci-dessous, qui affiche les entiers de 1 à 7 :

```
1 i = 0
2 while i < 7:
3     i = i + 1
4     print(i)
```

Pour que de telles instructions puisse être réalisées, il faudrait que l'ordinateur puisse stocker la valeur de `i` en mémoire, afin que l'instruction `i = i + 1`, qui signifie « la

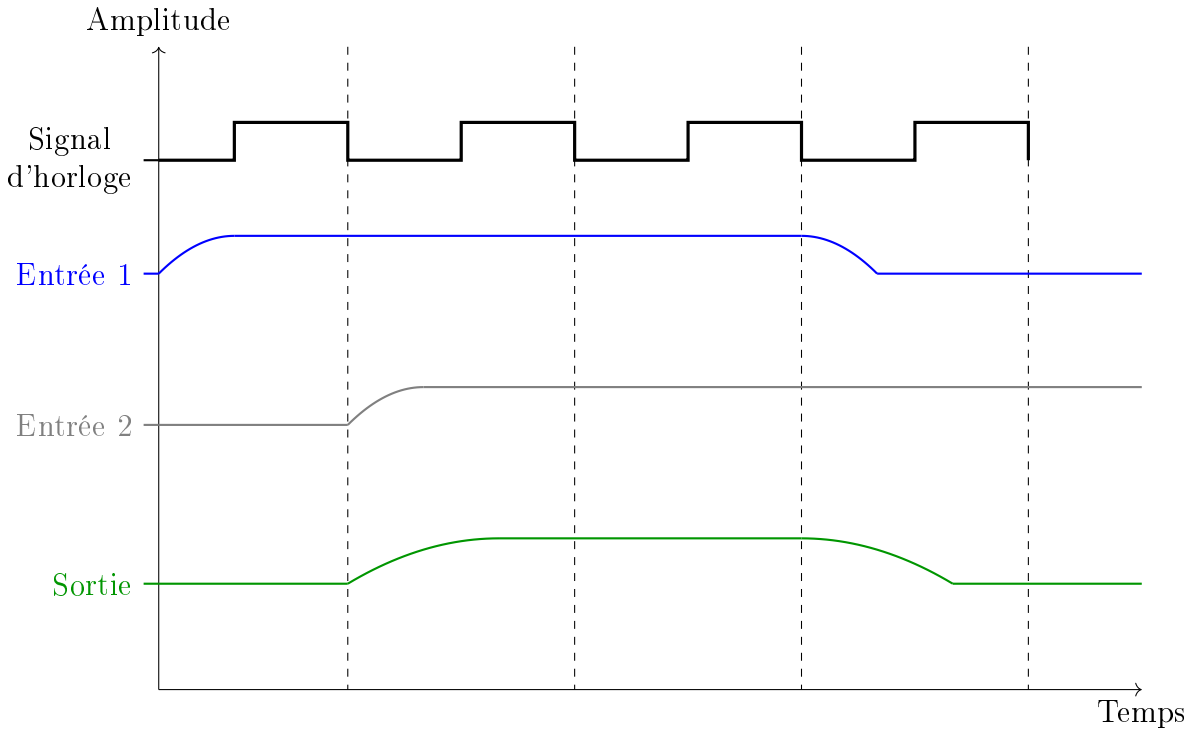


FIGURE 4.17 – Évolution schématique de la tension aux entrées et à la sortie d'un circuit pour la porte logique *AND* au sein d'un circuit séquentiel. La première entrée prend la valeur 1 durant le premier cycle, tandis que la seconde entrée prend la valeur 1 durant le second cycle, ainsi que la sortie, mais après un laps de temps supérieur. Durant le quatrième cycle, la première entrée retrouve une valeur de zéro, ainsi que la sortie, à nouveau sur une durée supérieure. Si l'on considère la valeur du signal hors des instants de début/fin de cycle, on peut mesurer une sortie semblant incohérente avec la logique du *AND* ; c'est en fait qu'elle ne correspond pas encore à la valeur stable attendue après propagation du signal.

nouvelle valeur de `i` est égale à l'ancienne valeur de `i` à laquelle on additionne 1 », ait un sens. Par ailleurs, il faudrait un moyen de garantir que l'ordre des instructions est respecté, c'est-à-dire que l'ordinateur commence par lire la valeur de `i`, puis l'incrémente, et non pas l'inverse – et ceci de façon répétée.

Pour ce faire, nous allons introduire le concept de circuit séquentiel, qui est un circuit logique dans lequel la notion de temps est prise en compte. Nous allons voir comment stocker des informations en mémoire, et comment réaliser des calculs qui prennent en compte l'historique des résultats.

4.5.3 Bascules Set-Reset

Les bascules Set-Reset sont les éléments fondamentaux permettant de mémoriser des valeurs. Elles sont à la base des registres, dont nous discutons dans le chapitre 5, mais également des circuits permettant de réaliser des compteurs, par exemple.

Le rôle d'une bascule est de maintenir un état[10] donné, à moins qu'elle ne recoive l'ordre de basculer dans un autre état. Elles sont donc plus que de simples stockeurs de bits, mais des éléments dynamiques dans la logique des circuits. On les nomme également « verrous » (*latch* en anglais) en raison de leur capacité à capturer un état donné et à le conserver malgré certains changement des entrées.

Une bascule peut donc soit conserver son bit, soit le mettre à 0, soit le mettre à 1. Si l'on résume par un code Python informel le comportement d'une bascule, on obtient le code suivant, où `que_faire` est l'ordre donné à la bascule, `Q_in` est la valeur en mémoire au sein de la bascule, et `Q_out` est la nouvelle valeur en mémoire :

```
if que_faire == "Set":
    Q_out = 1
elif que_faire == "Reset":
    Q_out = 0
elif que_faire == "Keep":
    Q_out = Q_in #la valeur en mémoire est conservée
```

Autrement dit, on veut pouvoir piloter une bascule *via* trois ordres différents en entrée ; cela signifie que la table de vérité d'une bascule prend nécessairement deux bits en entrée (et non pas un seul). Nous nommerons ces deux bits S et R, pour *Set* et *Reset* respectivement, comme cela sera justifié plus bas.

La table de vérité 4.16 ci-dessous résume le comportement attendu d'une bascule, à savoir qu'un certain couple de valeur d'entrée permet de garder en mémoire la valeur précédente, tandis que deux autre couples de valeurs d'entrée permettent de changer le bit en mémoire. Ici, un choix de codage des trois cas de figures a été fait, où $S = 0$ et $R = 0$ est le codage pour garder une valeur en mémoire, tandis que $S = 1$ et $R = 0$ sert à mettre la valeur à 1, et $S = 0$ et $R = 1$ sert à mettre la valeur à 0.

10. On entend par « état » l'ensemble des valeurs des signaux du circuit à la fin du cycle.

TABLE 4.16 – Table de vérité d'une bascule, où l'on décidé arbitrairement du codage des entrées. Les lignes horizontales délimitent les principaux cas de figures, correspondent aux trois ordres discutés plus haut. Les valeurs d'entrée sont S, R et Q_t (valeur issue du cycle précédent), tandis que la valeur de sortie est Q_{t+1}.

S	R	Q_t	Q_{t+1}
0	0	1	1
0	0	0	0
1	0	0	1
1	0	1	1
0	1	0	0
0	1	1	0
1	1	0	Non défini
1	1	1	Non défini

L'expression associée à cette table, si l'on ignore les deux cas non définis, est :

$$Q_{t+1} = !S!RQ_t + S!R = !R(!SQ_t + S) = !R(S + Q_t) \tag{4.39}$$

On peut manipuler cette dernière expression de manière à la reformuler en termes de la fonction NOR uniquement [11] :

$$Q_{t+1} = !R(S + Q_t) = !R!!(S + Q_t) = !(R + !(S + Q_t)) = NOR(R, NOR(S, Q_t)), \tag{4.40}$$

où le théorème de De Morgan a été utilisée dans l'avant-dernière égalité. Une telle expression logique correspond au circuit suivant, en figure 4.18.

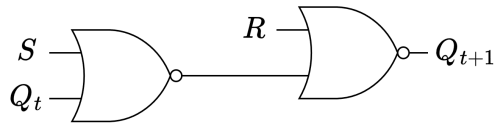


FIGURE 4.18 – Circuit logique correspondant à l'expression logique $NOR(R, NOR(S, Q_t))$ obtenue pour caractériser une bascule NOR réalisant la table de vérité 4.16.

Le fait que Q_t serve de valeur d'entrée au prochain cycle $t+1$ peut être illustré en reliant la sortie du circuit à sa propre entrée pour Q_t, comme montré sur la figure 4.19 ci-dessous. Pour le reste de ce document, nous visualiserons le circuit en utilisant la représentaiton de droite sur la figure 4.19, mais la plus courante au sein de la littérature est celle de gauche. Par ailleurs, nous omettons l'indice temporel pour Q, car il est implicite que la valeur en mémoire dépend du cycle considéré. Afin de se convaincre que le circuit que l'on a obtenu correspond bien à la table de vérité, il est utile de vérifier que les différents cas de figures possibles sont bien respectés. La figure 4.20 ci-dessous illustre donc les quatre cas de figures possibles pour une bascule NOR.

11. En faisant des autres choix de codage des entrées que ceux indiqués dans la table 4.16, on peut également aboutir à une expression à base de $NAND$ uniquement. Cela donne lieu à la bascule $NAND$, que nous ne traitons pas dans ce document.

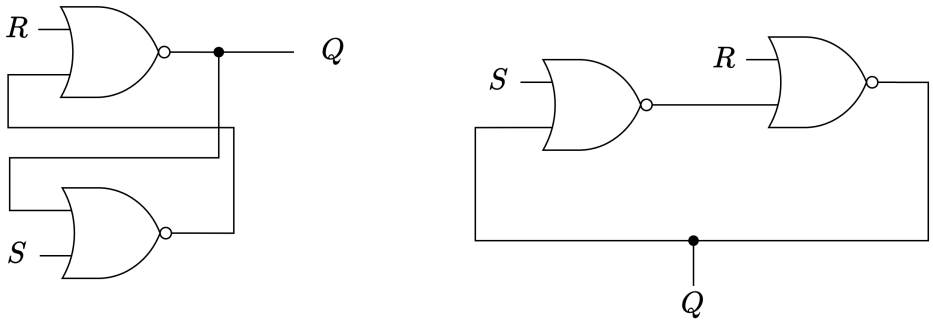


FIGURE 4.19 – Circuit logique correspondant à la bascule *NOR* réalisant la table de vérité 4.16 ; (gauche) visualisation courante au sein de la littérature, (droite) visualisation utilisée dans ce document.

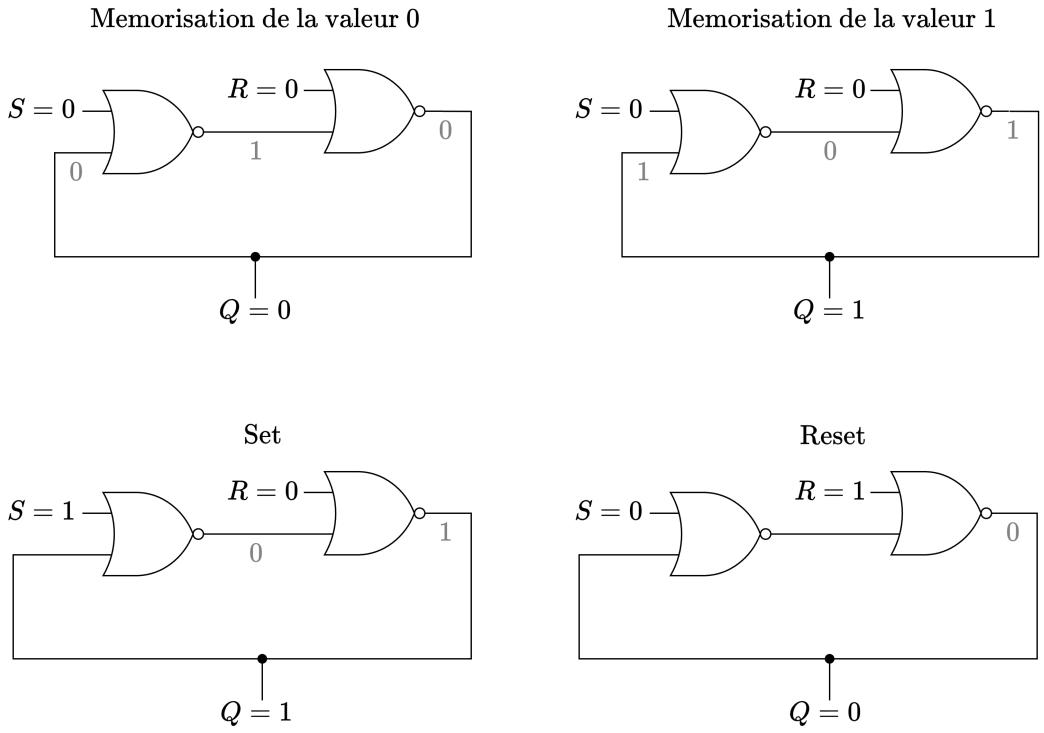


FIGURE 4.20 – Quatre cas de figures valides pour une bascule *NOR*. La ligne du haut représente les cas où la valeur Q est préservée d'un cycle à l'autre. Sur la ligne du bas à gauche, $S = 1$ et $R = 0$ (« Set ») et la nouvelle valeur en mémoire est nécessairement 1, quelle que soit la valeur précédente ; à l'inverse, sur la ligne du bas à droite, $S = 0$ et $R = 1$ (« Reset ») et la nouvelle valeur en mémoire est nécessairement 0. Les valeurs grisées correspondent aux entrées ou sorties des portes.

Pour terminer, signalons que les deux états non définis de la table de vérité 4.16 constituent des cas de figures qui ne sont pas souhaitables, car ils correspondent à des situations où les deux ordres « Set » et « Reset » sont donnés en même temps. Dans ce cas, la valeur en mémoire n'est pas définie, et le circuit peut se retrouver dans un état instable.

4.5.4 Delay Flip-Flop

La bascule discutée ci-dessus permet de mémoriser des valeurs, mais il lui manque toujours une façon de se synchroniser à une horloge. Pour ce faire, il faudrait y incorporer un élément qui bloque l'état de la bascule tant que le prochain signal d'horloge n'est pas reçu. En effet, le délai de propagation de la bascule (seulement constituée de deux *NOR* ou deux *NAND*) est en général bien plus faible que la période d'oscillation du signal d'horloge ; du point de vue de l'horloge, la bascule agit quasi-instantanément sur les données. Or, comme nous l'avons dit, il est important de ne traiter le signal provenant d'autres portes logiques qu'à un moment donné du cycle. En particulier, si les entrées souffrent d'une certaine instabilité (même sur une échelle de temps bien inférieure au cycle d'horloge), ces valeurs erronnées vont se transmettre à la sortie de la bascule, et donc au reste du circuit. Par ailleurs, dès lors que les sorties dépendent de l'ordre dans lequel les valeurs d'entrée arrivent, la capacité à reproduire une situation donnée est perdue et le circuit est en quelque sorte « indéterministe », ce qui est hautement indésirable. D'une façon plus générale : puisque, pour des raisons pratiques, l'ordinateur est conçu autour des pulsations globales de l'horloge, alors le fonctionnement de l'ordinateur est mis en danger par des éléments qui ne respecteraient pas ces pulsations.

Les bascules set-reset vues précédemment sont des bascules dites « asynchrones », tandis que nous allons maintenant nous intéresser à des bascules « synchrones » afin de garantir une cohérence entre la valeur issue de la bascule et celle issue d'autres circuits logiques du circuit. Ce que l'on cherche à faire est en quelque sorte un « synchronisateur » à ajouter aux circuits combinatoires. Cette mémoire à bascule spéciale, que l'on détaille plus bas, est nommé **Delay Flip-Flop** (DFF) et peut être vue comme une bascule qui ne prend qu'un input In en rapport avec la donnée à mémoriser (contre deux pour les bascules set-reset) ; en revanche, un second input est lié au signal d'horloge Clk et a pour effet d'activer la mémorisation de In.

Le symbole du DFF au sein des circuits est représenté sur la figure 4.21 ci-dessous. La figure 4.22, quant à elle, illustre comment un DFF s'intègre au sein d'un circuit combinatoire pour en faire un circuit séquentiel.

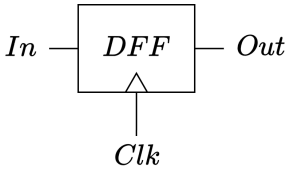


FIGURE 4.21 – Représentation d'un DFF au sein d'un circuit électronique. En général, l'entrée Clk est implicite, mais ce n'est pas le cas ici.

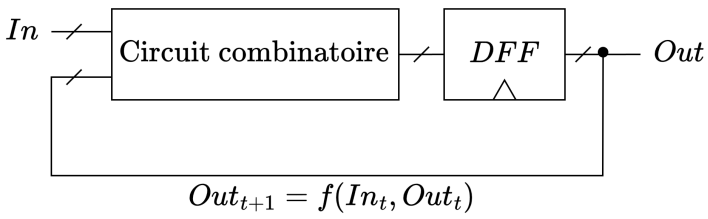


FIGURE 4.22 – Ajout d'un DFF à un circuit combinatoire quelconque de telle sorte que le circuit devienne un circuit séquentiel, c'est-à-dire pour lequel un état au cycle t dépend de l'état au cycle $t-1$.

Puisqu'il permet de synchroniser d'autres circuits avec l'horloge, le DFF est l'**élément d'état** fondamental dans les circuits séquentiels modernes. C'est en quelque sorte le pendant séquentiel de la porte identité [12]. Il est constitué de deux bascules : la première mémorise la prochaine valeur d'entrée Q_{t+1} du DFF, tandis que la seconde produit la valeur de sortie Q_t. Toutes deux sont précédées de circuits logiques qui servent à tenir compte du fait que l'horloge ait envoyé un signal (ou non) afin de conditionner les valeurs de S et de R de chaque bascule. En particulier, l'une des bascules est en mode mémorisation durant la phase basse du signal d'horloge, tandis que l'autre charge la nouvelle valeur ; les rôles sont inversés durant la phase haute du signal d'horloge. Ceci peut être résumé comme suit :

```
if signal bas:
    bascule 1 accepte valeur IN et produit valeur Q1
    bascule 2 conserve valeur Q
elif signal haut:
    bascule 1 conserve valeur Q1
    bascule 2 accepte valeur Q1 et produit valeur Q
```

Cette idée est également résumée dans la figure 4.23 ci-dessous, où l'on utilise les notations Clk pour le signal d'horloge (« Clock ») et In pour la valeur d'entrée.

Pour résumer, le circuit que l'on est en train d'imaginer reçoit deux entrées : le signal d'horloge Clk et une valeur quelconque In. Il produit une sortie Q, qui ne peut changer qu'à un instant prédéfini du cycle. Pour simplifier, nous supposerons ici que cet instant est le front d'horloge marquant la fin du cycle. Hors contexte, ce circuit ressemble à un circuit combinatoire comme un autre ; mais puisqu'il est conditionné à l'entrée spéciale que constitue le signal d'horloge, c'est un élément d'état soumis aux pulsations de l'horloge, et donc la brique de base d'un circuit séquentiel.

Il nous reste à présent à spécifier le contenu des circuits 1 et 2, qui sont responsables de piloter les valeurs S et R de chaque bascule. Reformulons le dernier code python de manière plus précise. Voici donc la description sous forme de code du circuit 1 :

12. La porte identité est la porte logique qui ne fait rien, c'est-à-dire implémentée comme un simple fil ; son pendant temporel, un élément d'état « identité » comme le DFF, transporte un état du cycle t au cycle $t+1$ en le laissant inchangé.

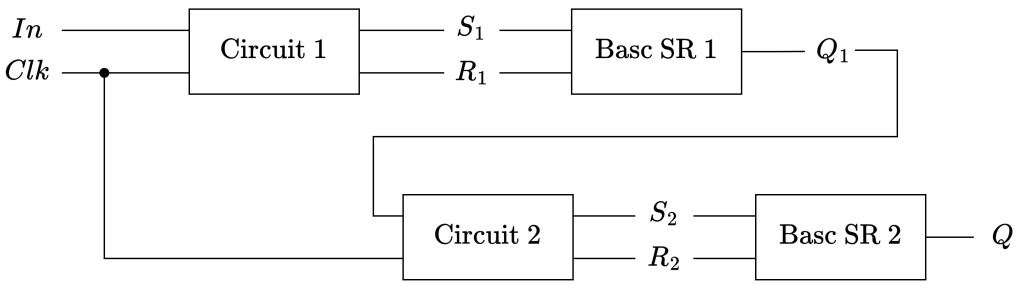


FIGURE 4.23 – Principe de fonctionnement d'un DFF composé de deux bascules, où Circuit 1 et Circuit 2, responsables du pilotage des valeurs S et R de chaque bascule, ne sont pas détaillés. Basc SR correspond à une bascule Set-Reset.

```
#Circuit 1
if Clk == 0: #Signal bas : accepte valeur In
    if In:
        S1 = 1
        R1 = 0
    else:
        S1 = 0
        R1 = 1
else: #Signal haut : ne fait rien d'autre que mémoriser
    S1 = 0
    R1 = 0
```

Voici la description sous forme de code du circuit 2 :

```
#Circuit 2
if Clk == 1:
    if Q1:
        S2 = 1
        R2 = 0
    else:
        S2 = 0
        R2 = 1
else:
    S2 = 0
    R2 = 0
```

On voit que le rôle des deux circuits est essentiellement le même. Nous pouvons écrire leur table de vérité, qui est donnée dans la table 4.17 ci-dessous. On obtient alors les expressions logiques suivantes pour le circuit 1 :

$$\begin{aligned} S_1 &= !Clk \ \cdot In \\ R_1 &= !Clk \cdot !In \end{aligned} \tag{4.41}$$

et pour le circuit 2 :

$$\begin{aligned} S_2 &= Clk \cdot Q_1 \\ R_2 &= Clk \cdot !Q_1 \end{aligned} \tag{4.42}$$

TABLE 4.17 – Table de vérité des circuits 1 et 2 au sein du Delay Flip-Flop. Ces circuits pilotent les lignes set et reset de leurs bascules respectives. Attention, ici Q_1 est à voir comme une entrée pour le calcul de S_2 et R_2.

Clk	In	S_1	R_1	Q_1	S_2	R_2
0	0	0	1	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	1
1	1	0	0	1	1	0

Comme discuté en section 4.3.3, on peut toujours reformuler ces expressions uniquement en termes de portes *NOR* ou *NAND*, afin de préserver l'unicité des portes utilisées.

Avec ces dernières expressions, le DFF est entièrement caractérisé. Nous pouvons donc maintenant, en un cycle, charger une donnée relative au cycle t, et transporter cet état dans le temps jusqu'au cycle $t + 1$.

4.5.5 Exemples de circuits logiques séquentiels

Compteur périodique

Un exemple classique de circuit séquentiel est le compteur périodique. Il s'agit d'un circuit qui, à chaque cycle, incrémente un compteur de 1, et revient à 0 une fois que la valeur maximale est atteinte. Dans cet exemple, nous considérons un compteur périodique à 3 bits ; un tel compteur s'incrémente de 0 à 7, puis revient à 0, et ceci indéfiniment.

Ce type de compteur permet par exemple de tenir le compte du nombre de cycles d'horloge écoulés (et donc du temps), ou de générer des signaux périodiques de fréquence donnée. La table de vérité de ce compteur est donnée dans la table 4.18 ci-dessous.

Dans cet exemple, par souci de concision, nous noterons b_i le bit numéro i au temps t et b'_i le bit numéro i au temps $t + 1$. En observant le tableau, on voit que :

$$b'_0 = NOT(b_0) \tag{4.43}$$

$$b'_1 = XOR(b_1, b_0) \tag{4.44}$$

$$b'_2 = XOR(b_2, AND(b_1, b_0)) \tag{4.45}$$

Dans le cas général du compteur périodique à n bits, on aurait :

$$b'_n = XOR(b_n, AND(b_{n-1}, ..., b_0)) \tag{4.46}$$

La figure 4.24 ci-dessous représente le circuit correspondant aux expressions séquentielles obtenues.

TABLE 4.18 – Table de vérité du compteur périodique à 3 bits.

t	b_2	b_1	b_0	b'_2	b'_1	b'_0
0	0	0	0	0	0	1
1	0	0	1	0	1	0
2	0	1	0	0	1	1
3	0	1	1	1	0	0
4	1	0	0	1	0	1
5	1	0	1	1	1	0
6	1	1	0	1	1	1
7	1	1	1	0	0	0

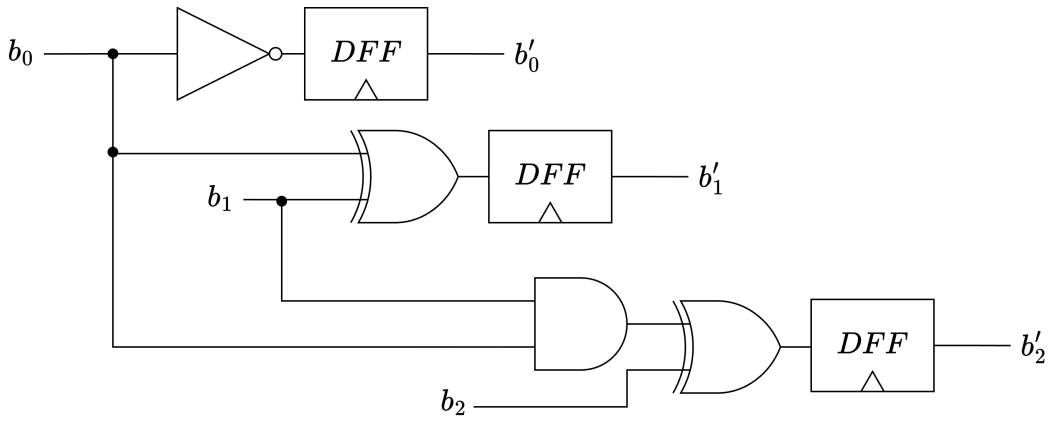


FIGURE 4.24 – Compteur périodique à 3 bits réalisé à l'aide de DFF et de portes logiques. Par souci de lisibilité, les b'_i sont représentés en sortie et les b_i en entrée, mais ils sont en réalités reliés entre eux (b' étant l'état qui suit b).

Registre

Comme nous le verrons dans le chapitre 5, les registres sont de petits éléments de mémoire au sein même du processeur de l'ordinateur, et revêtent à ce titre une importance capitale au sein des machines modernes. Un registre est un circuit séquentiel qui permet de stocker plusieurs bits. Nous étudions ici le registre à 1 bit, élément de base des registres servant à stocker des mots entiers.

Un registre doit pouvoir être lu (consulté) ou écrit (modifié), et doit mémoriser une valeur pour un nombre de cycles arbitraire. Il reçoit donc deux commandes en entrée : l'une, nommée `Load`, qui spécifie ce qu'il faut faire (lire ou écrire), et l'autre, nommée `In`, qui spécifie la valeur à écrire. Il reçoit également un signal d'horloge Clk, afin de pouvoir ordonner les états, comme discuté dans les sections précédentes. Enfin, il retourne une valeur nommé `Out`.

Voici un code décrivant le comportement d'un registre à 1 bit :

```
if Load(t):
    Out(t+1) = In(t) #écriture
else:
    Out(t+1) = Out(t) #lecture
```

Pour déterminer la valeur de sortie $Out(t+1)$ d'un tel registre à 1 bit, il faut donc spécifier la valeur de $Out(t)$, $In(t)$, Load(t) et Clk. Il serait possible d'écrire les 16 lignes de la table de vérité, mais cela serait fastidieux. À la place, nous pouvons utiliser le multiplexeur, déjà vu en section 4.4.1, afin de retranscrire sous forme de circuit logique la condition de branchement du code ci-dessus. La figure 4.25 ci-dessous illustre le circuit séquentiel correspondant à un registre à 1 bit.

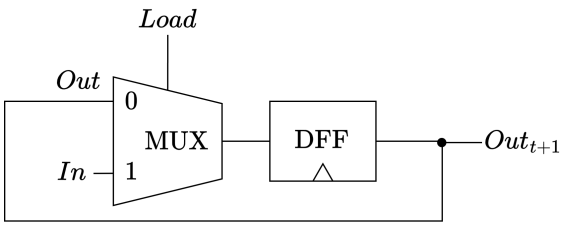


FIGURE 4.25 – Registre à 1 bit réalisé à l'aide de DFF et de portes logiques.

Pour réaliser des registres à plusieurs bits, on peut regrouper plusieurs registres 1 bit et envoyer la même valeur de `Load` à chacun, comme suggéré dans la figure 4.26 ci-dessous, ou encore dans la figure 4.27 en utilisant la notation vue plus haut pour les lignes à plusieurs bits.

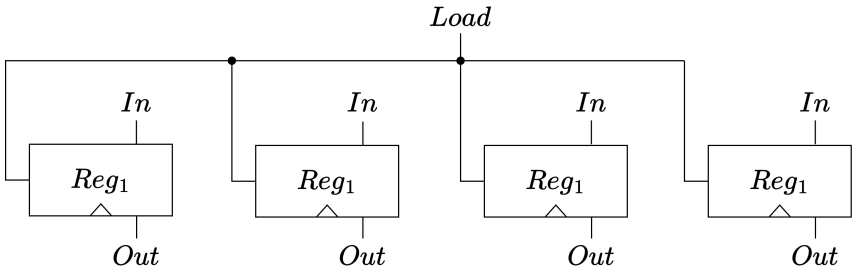


FIGURE 4.26 – Branchements de plusieurs registres 1 bit pour réaliser un registre à 4 bits.

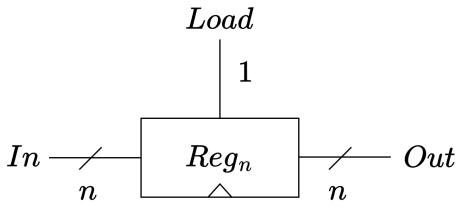


FIGURE 4.27 – Branchements de plusieurs registres 1 bit pour réaliser un registre à n bits.

Chapitre 5

Architecture des ordinateurs

Dès le milieu des années quarante, les informaticiens ont réalisé qu'il était commode de séparer la machine en tant que matériel (hardware) des instructions qu'elles doit exécuter (software). En effet, la partie matérielle est pour ainsi dire « immuable » tandis que la partie logicielle, le **programme** à exécuter, varie fortement en fonction du problème abordé. En séparant le programme de la machine elle-même, cette dernière peut être utilisée pour résoudre une grande variété de problèmes simplement en changeant le programme qu'elle exécute, sans toucher à ses autres composants.

Il faut se rappeler que la partie matérielle de certaines machines peut être conçue pour épouser au mieux les spécificités d'un problème donné ; il nous semble naturel, aujourd'hui, qu'un ordinateur doive être « universel », mais cela n'a pas toujours été le cas, en particulier lorsque le gain de performance offert par une architecture dite « dédiée », c'est-à-dire épousant la tâche logicielle a exécuter, était substantiel. Comme on le verra plus loin, les machines modernes contiennent d'ailleurs des composants spécialements conçus pour exécuter certaines tâches précises, ce qui en fait des sortes d'architectures dédiées à l'intérieur même d'une machine plus universelle.

Cette séparation des aspects matériels et logiciels de l'ordinateur est à la base de l'architecture dite de « von Neumann » [1], dont les ordinateurs modernes sont des descendants directs.

5.1 L'architecture de von Neumann

Une façon de réaliser le concept de programme stocké est schématisée sur la figure 5.1 ci-dessous, où les périphériques d'entrée (par exemple clavier, souris, carte réseau, ...) et de sortie (par exemple écran, haut-parleur, imprimante, carte réseau, ...) sont représentés, ainsi que les périphériques de stockage auxiliaire (par exemple disque dur, clé USB, carte

1. Nommée ainsi en référence au mathématicien prolifique John von Neumann, qui a participé au projet de l'EDVAC (ordinateur précurseur en matière d'architecture à programme séparé). Il semble néanmoins que de nombreuses autres personnes aient contribué de façon indépendante à l'idée de l'architecture nommée ainsi aujourd'hui.

SSD, ...). Cependant, comme leur nom l'indique, ces périphériques – ou dispositifs, pour mieux coller au terme anglais de « device » – ne constituent pas l'essence de l'architecture de von Neumann, bien qu'un être humain ait en fin de compte besoin d'un minimum de ces éléments pour interagir avec l'ordinateur d'une façon ou d'une autre.

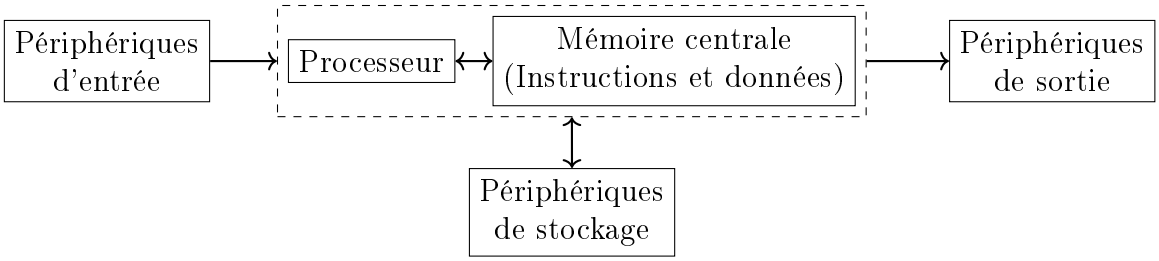


FIGURE 5.1 – Base de l'architecture de von Neumann.

Le cœur de l'ordinateur est donc composé de deux éléments : le **processeur** central d'un côté (souvent abrégé processeur ou CPU pour Central Processing Unit), et la **mémoire centrale** de l'autre (souvent nommée mémoire vive ou RAM, pour des raisons discutées plus bas).

5.1.1 La mémoire centrale

Terminologie et technologies

Comme nous venons de le voir, le terme de « mémoire centrale » est souvent interverti au quotidien avec d'autres termes, que nous examinons ci-dessous, mais au sujet desquels on peut trouver des défauts. Dans ce cours, on préférera donc utiliser le terme « mémoire centrale ».

Le terme de **mémoire vive**, fréquent en français, sert à rappeler qu'il s'agit d'un type de mémoire qui peut être lu et écrit, et non pas seulement accédé en lecture comme une mémoire dite « morte » (Read Only Memory en anglais, donnant l'abbréviation **ROM**). Par ailleurs, la mémoire vive des ordinateurs modernes est **volatile**, c'est-à-dire non persistante ; on associe donc volontiers (et à tort, si l'on est rigoureux) mémoire morte avec mémoire persistante, et à l'inverse mémoire vive avec mémoire volatile. À proprement parler, puisque la mémoire centrale n'est pas le seul dispositif de mémoire qui puisse également être accédé en écriture, le terme « mémoire vive » peut paraître mal choisi dans un contexte moderne.

Le terme de **RAM**, quant à lui, signifie Random Access Memory, et rappelle qu'à la différence des mémoires dites « séquentielles », la mémoire centrale possède des adresses pouvant être lues ou écrites dans un ordre arbitraire, c'est-à-dire sans avoir à parcourir les données qui précèdent celles qui nous intéresse. Cette faculté de pouvoir accéder aux données dans un ordre arbitraire a été décrite comme un accès « aléatoire » par le passé, mais « mémoire à accès direct » serait un terme moins ambigu, du moins en français.

Rôle général

La mémoire centrale stocke toutes les donnée nécessaires au fonctionnement du programme exécuté par l'ordinateur, c'est-à-dire les instructions du progamme lui-même ainsi que les données sur lesquelles il agit (une image, un texte, ...). La mémoire centrale est donc en quelque sorte une mémoire de travail pour le processeur. Les mémoires centrales modernes des ordinateurs personnels (portables ou non) possèdent, depuis les années 2010, une capacité typique de plusieurs gigaoctets.

Le nombre représentant chaque octet en mémoire suit celui de l'octet précédente : on parle d'« adressage continu ». Nous verrons plus loin dans ce cours que cette propriété peut être exploitée lorsqu'on programme afin de créer des structures de données efficaces nommées « tableaux de valeurs ».

La mémoire centrale ne doit pas être confondue avec les registres ou la mémoire cache, dont nous discutons plus loin, ni avec les mémoires auxiliaires qui sont utilisées pour stocker des données de façon persistante ; il s'agit de nos jours des disques-durs magnétiques ou des mémoires de type flash (cartes SSD, clés USB). Les mémoires auxiliaires sont en général beaucoup plus lentes d'accès que la mémoire centrale, tout en permettant en revanche de stocker une plus grande quantité de données. Les mémoires auxiliaires modernes du quotidien sont, tout comme la mémoire centrale, accessibles en écriture aussi bien qu'en lecture. Par ailleurs et surtout, ces mémoires auxiliaires sont persistantes : contrairement à la mémoire centrale, elles conservent leurs données même hors tension et sur une échelle de temps significative.

Les types de mémoires persistantes

L'un des rôles – évident au quotidien – d'une mémoire auxiliaire est de stocker les données des utilisateurs sur le long terme, lorsque la machine est éteinte. Afin de simplifier la discussion, nous supposons ici une mémoire auxiliaire persitante [2]. La mémoire auxiliaire de la machine peut également avoir un rôle durant l'exécution d'instructions sur un ordinateur à programme stocké, en plus de stocker ce dernier lorsque la machine est hors tension. C'est pourquoi nous en traitons dans ce document.

Il existe plusieurs familles de technologies permettant de stocker une information binaire de façon persistante. Voici une proposition de classification :

- Les supports de type **mécanique**, où l'information est codée grâce à une modification physique de la forme du support lui-même. Exemples : carte perforée (binaire), disque vinyle (analogique). À noter que, dans le cas des cartes perforées, pour détecter la présence ou l'absence d'un trou, on peut employer soit des moyens électriques (le courant passe ou ne passe pas d'un côté à l'autre de la carte), soit des moyens optiques (auquel cas, on pourrait arguer que ce type de carte perforée appartient à la famille des supports optiques, discutés plus bas).

2. Rappelons que le fait qu'une mémoire est auxiliaire signifie qu'elle ne constitue pas la mémoire centrale de la machine ; le fait qu'elle soit persistante signifie qu'elle ne perd pas son information une fois privée d'énergie. Les deux propriétés ne vont pas nécessairement de pair, mais c'est la plupart du temps le cas dans notre quotidien.

- Les supports de type **magnétique**, où l'on exploite l'aimantation locale d'un matériau pour stocker l'information. Exemples : disquette (binaire), cassette audio et vidéo (analogique), disque-dur (binaire), bande magnétique (binaire et analogique).
- Les supports de type **optique**, où l'on exploite les propriétés réflectives locales d'un matériau pour stocker l'information. Un faisceau de lumière LASER, s'il est dirigé vers un endroit du support qui donne lieu à une réflection, peut ainsi être interprété comme la valeur *1*, tandis que l'absence de réflection peut être interprétée comme la valeur *0*. Exemples : CD, DVD, Blu-Ray.
- Les supports de type **flash**, où des transistors spéciaux permettent de piéger (ou non) des électrons au sein d'une petite aire isolante et ainsi de stocker l'information même lorsqu'ils sont coupés d'une source d'énergie.

De nombreux dispositifs venant d'être cités ne sont plus utilisés – hors marchés de niche – de nos jours mais revêtent une importance historique, à l'instar de la disquette (voir image 5.2), qui a été un support de stockage très populaire dans les années 80 et 90, et qui a donné son image à l'icône de sauvegarde dans de nombreux logiciels. Enfin, la table 5.1 donne un aperçu des capacités typiques de certains supports de stockage.

FIGURE 5.2 – (gauche) Dessin d'une disquette 3.5 pouces. (droite) Représentation schématique d'une disquette utilisée dans le logiciel Micrososft Word comme icône de sauvegarde.

TABLE 5.1 – Capacités typiques des supports de stockage en 2024.

Type de support	Exemple	Capacité typique approximative
Mécanique	Carte perforée	Plusieurs dizaines d'octets
Magnétique	Disquette	1.44 MB (3.5 pouces)
Magnétique	Disque dur	Quelques TB
Magnétique	Bande magnétique	Plusieurs TB
Optique	CD	700 MB
Optique	DVD	4.7 GB (simple couche)
Optique	Blu-Ray	Jusqu'à 100 GB
Flash	SSD	Dizaines de GB à quelques TB

Pour terminer, il faut noter que puisque le contenu de la mémoire centrale est perdu après une mise hors tension, il est important qu'un ordinateur possède une petite quantité de mémoire persistante (une mémoire morte, ou du moins non volatile, incorporée dans la carte mère de la machine) à partir de laquelle charger un certain minimum vital nommé le BIOS, afin de pouvoir démarrer l'ordinateur. Ce démarrage consiste à charger le noyau du

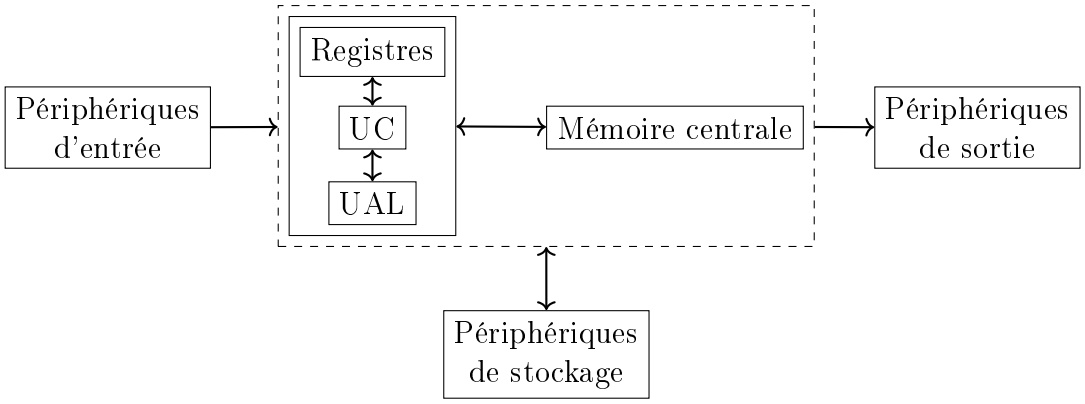


FIGURE 5.3 – Base de l'architecture de von Neumann, où l'on a détaillé les composants du processeur. UC signifie unité de contrôle, UAL signifie unité arithmétique et logique.

système d'exploitation (cf. section 6.2) durant la phase dite d'« amorçage », ou « boot » en anglais. Le BIOS (Basic Input/Output System) est un programme « fixe » intégré au sein du matériel [3], qui permet d'interagir de façon minimale avec la machine juste après le démarrage, ainsi que de s'assurer du bon fonctionnement du matériel. Sur beaucoup de machines modernes, le BIOS est étendu par une interface nommée UEFI (Unified Extensible Firmware Interface).

5.1.2 Le processeur

Le processeur peut être divisé en composants, illustrés sur la figure 5.3. On y compte l'**unité de contrôle**, qui comme son nom l'indique commande les autres composants du processeur, l'**unité arithmétique et logique** (UAL) qui effectue les opérations arithmétiques et logiques, et les **registres**, qui sont des emplacements de mémoire très rapides et de petite taille, utilisés pour stocker des données temporaires.

Par souci de clarté et de concision, nous ne traitons que des machines à un seul processeur dans ce cours. Dans le cas où l'ordinateur possède plusieurs processeurs, ceux-ci communiquent en général tous avec la mémoire centrale [4]. Un processeur donné peut également posséder plusieurs unités de calcul distinctes (ou **cœurs**), c'est-à-dire plusieurs ensembles UAL-registres-unité de contrôle au sein de la même puce et se partageant davantage de ressources (alimentation, certains éléments de mémoire, ...) qu'un ordinateur multiprocesseur. La plupart des ordinateurs domestiques modernes incorporent plusieurs cœurs mais pas nécessairement plusieurs processeurs, ce dernier cas étant en revanche commun pour les superordinateurs et les serveurs.

L'étude des programmes exploitant plusieurs unités de calcul en parallèle sur un même problème constitue un vaste domaine de l'informatique nommé **parallélisme**. Nous n'aborderons pas ce sujet dans ce cours, mais il faut garder en tête que des types d'architectures spéciaux pour le calcul parallèle, dérivant de l'architecture de base décrite dans

3. Ce type de programme bas-niveau intégré au matériel est nommé « firmware » en anglais. Néanmoins, et de façon informelle, il est parfois employé pour désigner un logiciel très proche du hardware.

4. Dans le cas où il s'agirait de couples processeur-mémoire centrale distincts, il s'agirait alors d'ordinateurs distincts selon notre vision de l'ordinateur.

ce cours, sont utilisés notamment pour le calcul scientifique, qui peut en faire un usage intensif. Par ailleurs, des dispositifs tels que les cartes graphiques (Graphics Processing Unit, abrégé GPU) sont massivement parallèles et possèdent des points communs avec les processeurs, mais nous n'en traitons également pas dans ce cours.

Les registres

Les registres contiennent les valeurs intermédiaires des calculs effectués par le processeur, des adresses d'instructions à lire, ainsi que des adresses de données à lire ou écrire en mémoire centrale. Deux registres spéciaux, sur lesquels nous reviendrons plus loin, sont pour nous d'une importance particulière : le **registre d'instruction** et le **compteur ordinal** (« program counter » en anglais, souvent abrégé PC).

Comme nous l'avons dit précédemment, les registres sont des unités de mémoire très rapides et se trouvant au sein même du processeur. Ils sont réalisés grâce aux Delay Flip-Flop présentés dans le chapitre 4.5.5. On peut alors se demander pourquoi la mémoire centrale toute entière n'est pas réalisée sous forme de registres, puisqu'ils sont si efficaces. Il faut comprendre que le fait que les registres soient peu nombreux est précisément l'une des raisons qui les rend si rapides, car cela permet d'une part de simplifier la façon dont ils sont adressés, et d'autre part de les placer très près du processeur, ce qui réduit le temps de propagation des signaux électriques.

Pour comprendre la première des raisons précitées, à savoir le problème de l'adressage des registres, rappelons-nous que le circuit électronique qui réalise un registre (cf. figure 4.25) contient un multiplexeur. Par ailleurs, pour sélectionner l'adresse du registre à lire ou à écrire parmi ceux du **banc** de registres, on utilise un décodeur, qui est un cas particulier des démultiplexeurs discutés en section 4.4.1. Or, comme nous l'avons vu en section 4.4.1, les multiplexeurs et les décodeurs sont des circuits à $\log(k)$ étages, où k est le nombre d'entrées du multiplexeur ou du décodeur. En augmentant le nombre de registres, on augmente donc nécessairement le temps d'accès à celui qui nous intéresse, puisqu'il faut le chercher parmi ceux qui ne nous intéressent pas. Autrement dit, il existe un compromis entre le nombre de registres et la rapidité d'accès à chacun d'eux.

L'unité de contrôle

Pour piloter le CPU et son interaction avec les autres composants, l'unité de contrôle envoie des signaux. Cela est nécessaire par exemple pour spécifier certains paramètres, tel que le type d'opération que l'UAL doit effectuer, ou encore la nature (lecture ou écriture) de l'accès à la mémoire, etc. L'unité de contrôle s'occupe également de synchroniser les transferts de données, gérer les interruptions (par exemple parce qu'un périphérique le demande). Ces signaux font partie d'une stratégie de communication plus générale qui consiste en un cycle de trois étapes nommé **Fetch-Decode-Execute**. Nous détaillerons ce cycle plus loin dans le chapitre.

L'unité arithmétique et logique

L'UAL est le composant du processeur qui effectue les opérations arithmétiques et logiques. Il est réalisée grâce à un circuit logique (cf. chapitre 4). Les opérations arithmétiques sont les opérations de base de l'arithmétique telles que l'addition, la soustraction, la multiplication et la division (sur les machines modernes, certains composants spécialisés peuvent endosser ce rôle, mais nous n'en discutons pas ici). Les opérations logiques correspondent aux opérations vues dans le chapitre 4 telles que *AND*, *OR*, *NOR*, etc. L'UAL prend toujours (au plus) deux mots en entrée pour en produire un seul en sortie [5] ; c'est pourquoi elle est communément représentée dans les schémas par le symbole indiqué en figure 5.4, qui reflète cette fusion de l'information.

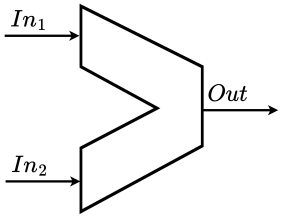


FIGURE 5.4 – Symbole de l'UAL dans les schémas de circuits, symbolisant le fait qu'elle prend deux mots en entrée (gauche) pour en produire un en sortie (droite). Des lignes de contrôle, non représentées ici, sont également nécessaires pour spécifier à l'UAL le type d'opération à effectuer.

5.2 Flux de l'information

Les trois composants fondamentaux du CPU (UAL, UC, registres) communiquent entre eux et avec la mémoire centrale par l'intermédiaire d'un **bus système**. Le bus système est un ensemble de connexions qui transportent les signaux entre les composants du CPU et la mémoire centrale. La figure 5.5 ci-dessous illustre le flux de l'information dans l'architecture de von Neumann de façon abstraite, en symbolisant le bus par un symbole. Il faut cependant garder en tête que ce bus est en réalité un ensemble de fils de connexion, et non un élément isolé.

Les signaux transportés par le bus système sont en réalité de plusieurs types : signaux de contrôle, d'adresse et de données. Ainsi, le bus est en réalité divisé en ces sous-catégories, car le flux d'information n'est pas le même dans chacune.

5.2.1 Flux de données

Les signaux de données sont essentiellement bidirectionnels : les données sont accédées en mémoire, des calculs sont effectués, et de nouvelles valeurs sont écrites en mémoire.

5. Ainsi qu'éventuellement des *flags* ou indicateurs d'état, qui apportent une information supplémentaire sur l'opération qui vient d'être effectuée (par exemple, overflow d'un type numérique).

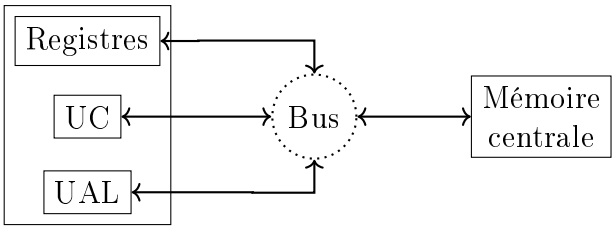


FIGURE 5.5 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que leur interconnexion *via* le bus système. Le cercle symbolise le fait que le flux d'information passe par un bus, mais il ne correspond pas à un élément isolé, le bus étant un ensemble de connexions (fils).

Cette dernière, ainsi que les registres et l'UAL, doivent recevoir et envoyer des données par le bus. Il faut garder en tête que les instructions elles-mêmes sont des données ; l'unité de contrôle, qui décompose les instructions, doit donc en recevoir. La figure 5.6 illustre le flux de données dans l'architecture de von Neumann.

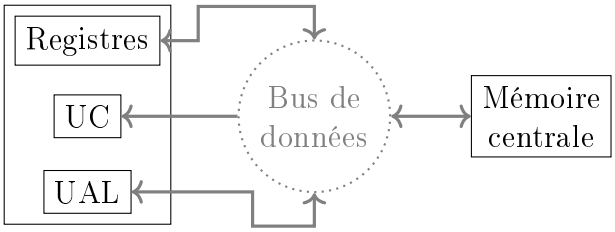


FIGURE 5.6 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que le flux de signaux transitant par le bus de données. Le cercle symbolise le fait que le flux d'information passe par un même bus, mais il ne correspond pas à un élément isolé, le bus étant un ensemble de connexions (fils).

5.2.2 Flux d'adresses

Les signaux d'adresse, quant à eux, sont unidirectionnels et dirigés vers la mémoire centrale. En effet, dès que cette dernière reçoit une adresse ainsi que le signal de lecture, elle doit renvoyer la donnée correspondante. Si à l'inverse elle reçoit une adresse assortie d'un signal d'écriture et d'une donnée, elle écrit cette donnée à l'adresse demandée. En aucun cas elle n'envoie d'adresse au sein du bus. Les registres sont souvent amenés à contenir de telles adresses, en attendant de l'envoyer dans le bus. Ces valeurs entrent dans les registres en tant que données produites par l'UAL, mais en sortent en tant qu'adresses (par exemple, celle de la prochaine instruction). L'UAL également est impliquée en raison des cas où une nouvelle adresse doit être calculée à partir d'une autre. À nouveau, dans ce cas, l'UAL reçoit deux valeurs d'input qui sont des données, mais la valeur de sortie, elle, est une adresse. La figure 5.7 illustre le flux d'adresses dans l'architecture de von Neumann.

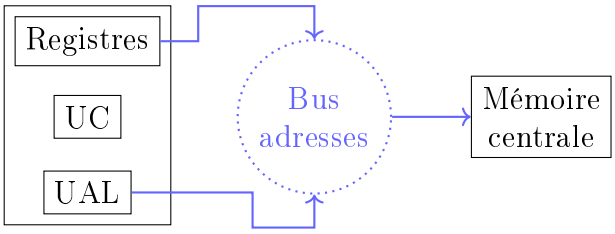


FIGURE 5.7 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que le flux de signaux transitant par le bus d'adresses. Le cercle symbolise le fait que le flux d'information passe par un même bus, mais il ne correspond pas à un élément isolé, le bus étant un ensemble de connexions (fils).

5.2.3 Flux de contrôle

Pour savoir si une donnée doit être lue ou écrite, la mémoire centrale consulte le signal reçu. Ce signal est un signal de contrôle, qui est donc unidirectionnel pour la mémoire. Les registres sont également concernés par ce signal, car ils reçoivent des indications quant à la temporalité des de la lecture et de l'écriture des données. L'UAL, enfin, consulte le signal de contrôle pour connaître le type d'opération qu'elle doit effectuer sur les inputs reçus ; par ailleurs, elle est capable d'envoyer un signal dans le bus de contrôle afin de spécifier le résultat d'un branchement qui conditionne les prochaines instructions. Ainsi, et pour terminer, l'unité de contrôle émet évidemment des signaux en concordance avec les instructions à exécuter, mais doit également être à l'écoute du résultat des branchements précités. La figure 5.8 illustre le flux de contrôle dans l'architecture de von Neumann.

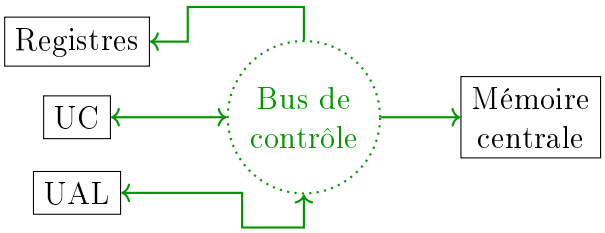


FIGURE 5.8 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que le flux de signaux transitant par le bus de contrôle. Le cercle symbolise le fait que le flux d'information passe par un même bus, mais il ne correspond pas à un élément isolé, le bus étant un ensemble de connexions (fils).

5.2.4 Flux d'information et périphériques

Les périphériques sont également concernés par le flux de l'information via le bus système. Lorsqu'un périphérique a besoin de l'attention du processeur, il envoie un signal d'interruption. Ce signal ne transmet pas directement les adresses ou les données, mais alerte plutôt le processeur qu'il doit gérer une certaine tâche liée à ce périphérique. Suite à cette interruption, le processeur peut lire ou écrire des données vers ou depuis le périphérique, en utilisant les adresses gérées par le contrôleur de ce dernier, qui sert d'intermédiaire ; par souci de simplicité, nous ne représentons pas ce contrôleur dans les schémas ni n'en discutons au-delà de cette sous-section.

Dans certains cas, les périphériques peuvent utiliser des bus spécifiques qui ne sont pas directement connectés au bus principal du processeur. Ces bus peuvent inclure des bus USB, par exemple, qui ont leurs propres protocoles de communication.

La figure 5.9 résume le flux d'information dans l'architecture de von Neumann, en incluant tout ce qui a été dit plus haut ainsi que les périphériques.

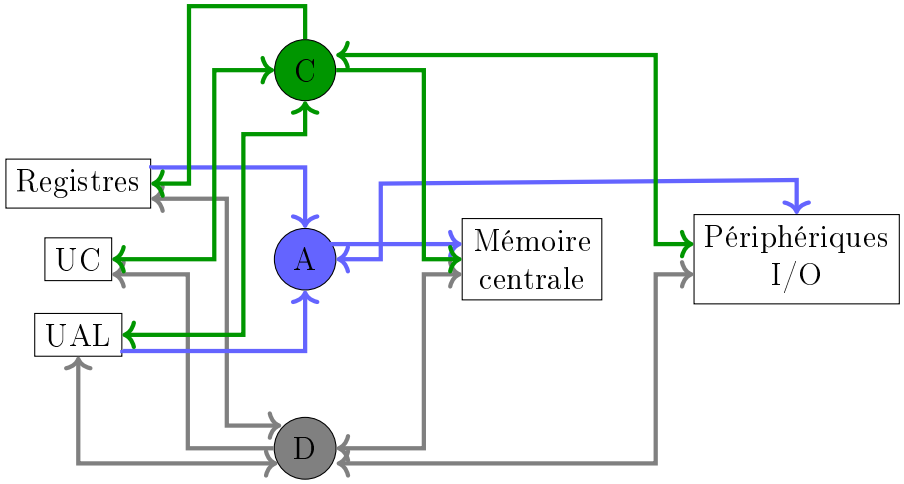


FIGURE 5.9 – Schéma abstrait de la base de l'architecture de von Neumann, où l'on a fait figurer le flux d'information. Le symbole C représente les signaux de contrôle, A les signaux d'adresse et D les signaux de données. Ces symboles ne correspondent pas à des éléments matériels mais illustrent le fait que les trois types de bus sont à distinguer. Par souci de visibilité, tous les types de périphériques ont été regroupés au sein d'un seul bloc.

5.3 Le cycle Fetch-Decode-Execute

Comme discuté au sein de la section 4.5, une caractéristique importante des processeurs modernes est le fait que leurs circuits traitent l'information de façon synchrone, c'est-à-dire qu'une certaine cadence est respectée par tous les composants afin de réaliser les instructions du programme. En particulier, trois étapes importantes, qui se succèdent en boucle de façon séquentielle, sont nécessaires pour exécuter une instruction. Ces étapes sont nommées **fetch**, **decode** et **execute** et doivent s'exécuter dans cet ordre pour une instruction données. C'est l'horloge du processeur qui déclenche le passage d'une étape à l'autre.

5.3.1 Fetch

L'étape fetch (« aller chercher » en français) correspond à la lecture de la prochaine instruction au sein de la mémoire centrale. Le compteur ordinal, dont nous avons parlé plus haut, est le registre qui contient l'adresse de la prochaine instruction à lire. Cette adresse est envoyée dans le bus d'adresse, et la mémoire centrale renvoie l'instruction correspondante dans le bus de données. Cette instruction est alors stockée dans le registre d'instruction.

Lorsque les instructions du programme s'exécutent séquentiellement, le compteur ordinal est simplement incrémenté à chaque cycle [6] afin de passer d'une instruction à l'autre. Lorsque le programme comporte un « saut » inconditionnel, le compteur ordinal peut être incrémenté d'une autre valeur. Enfin, si le programme comporte des branchements, le compteur ordinal est mis à jour en fonction du résultat de l'opération effectuée par l'UAL, et un saut est effectué à l'adresse calculée.

5.3.2 Decode

Durant cette seconde étape, centrée sur l'unité de contrôle, l'instruction contenue dans le registre d'instruction est interprétée. Cela signifie que l'unité de contrôle détermine le type d'opération à effectuer, les registres concernés, etc. Ce peut être un calcul à effectuer, une donnée à charger dans un registre, un saut inconditionnel à imposer au compteur ordinal, etc. Pour ce faire, l'unité de contrôle décompose d'abord l'instruction, qui est fractionnée en plusieurs groupes de tailles prédéfinies pour un type d'instruction donnée : un groupe contenant l'identifiant de l'instruction, un second pour l'identifiant du registre qui contient l'adresse d'une donnée à charger, un troisième pour un décalage à ajouter pour calculer l'adresse en question, et un quatrième pour spécifier le registre où stocker le résultat. Tous les groupes ne sont pas pertinents pour toutes les instructions. Une fois l'instruction décomposée, l'unité de contrôle envoie des signaux appropriés dans le bus de contrôle, qui seront reçus par les composants concernés. Dans le prochain chapitre, nous reviendrons sur cette étape dans la section dédiée aux langages assembleurs.

5.3.3 Execute

Dans cette dernière étape, l'opération liée à l'étape précédente est effectuée. La nature de cette opération peut prendre beaucoup de formes, dont par exemple :

- Un calcul arithmétique ou logique, effectué par l'UAL.
- Un accès à la mémoire centrale, pour lire ou écrire des données.
- Un saut inconditionnel, qui modifie le compteur ordinal.

La figure 5.10 illustre le flux d'information lors de l'exécution d'une instruction consistant à charger une donnée depuis la mémoire centrale.

Il faut noter que dans certains cas, l'exécution d'une instruction peut nécessiter plusieurs cycles d'horloge. Par exemple, un accès à la mémoire centrale peut prendre plusieurs cycles selon les contextes. Dans ce cas, les processeurs modernes ont plusieurs mécanismes pour gérer l'attente sans mettre tout le système en pause. Voici les stratégies principales utilisées, dont les détails sortent du cadre de ce cours :

- Exécution dans le désordre – Dans les architectures plus avancées, les processeurs peuvent exécuter des instructions hors de l'ordre initial de leur arrivée, lorsque c'est possible. Si une instruction doit attendre des données de la mémoire, le processeur

6. En l'occurrence, il est incrémenté du nombre d'octets sur lequel est codée une instruction, typiquement 4 octets dans une architecture MIPS (voir plus loin).

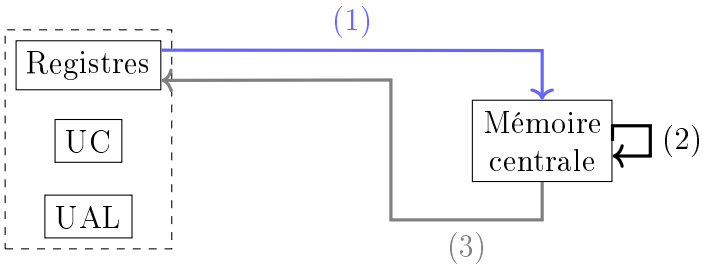


FIGURE 5.10 – Chargement d'une donnée depuis la mémoire centrale. (1) L'adresse de la donnée à charger est envoyée dans le bus d'adresse. (2) La donnée est récupérée au sein de la mémoire centrale. (3) La donnée est envoyée dans le bus de données et stockée dans un registre.

peut passer à d'autres instructions qui n'ont pas besoin d'attendre, puis revenir à l'instruction en attente une fois que les données sont disponibles.

- Techniques de prédiction et pipelining – Certains processeurs utilisent la prédiction de branchements et la spéculation pour anticiper les chemins d'exécution des programmes et commencer à exécuter des instructions avant même que l'on soit certain qu'il s'agisse des bonnes instructions et des bonnes données. Si la spéculation s'avère mauvaise, de l'énergie a été gaspillée ; si elle s'avère exacte, alors du temps d'exécution a été gagné. Cette technique est utilisée conjointement avec le concept de pipeline, qui est mis en place sur la plupart des processeurs modernes. Le principe est que différentes étapes de l'exécution d'une instruction sont traitées simultanément dans différentes parties des processeurs. Par exemple l'UAL peut tout à fait effectuer l'exécution d'un calcul pendant que l'étape fetch du cycle suivant commence. Une certaine superposition des étapes de cycles différents est donc envisageable, et le pipelining peut être plus ou moins intensif selon les architectures.
- Multithreading – Les systèmes d'exploitation et processeurs modernes peuvent également gérer plusieurs « fils d'instructions » à la fois, pour un programme donné. Si un fil est bloqué en attendant des données de la mémoire, un autre bout du programme, parallèle à ce dernier, peut utiliser les ressources du processeur afin d'éviter de perdre du temps. Le multithreading ne doit pas être confondu avec le multitasking décrit dans le chapitre 6.2.

En plus des stratégies qui viennent d'être citées, nous allons discuter des technologies de cache, qui permettent également de diminuer les temps d'attente du processeur sur le reste des éléments.

5.4 Hiérarchie de mémoires

Sur le principe, un ordinateur pourrait se passer de registres en lisant et écrivant toutes les données directement dans la mémoire centrale. En raison du gain de performance et de simplicité drastique que procurent les registres (cf. table 5.2), ces derniers, qui font partie intégrante du processeur, sont capitaux.

À vrai dire, les ordinateurs modernes utilisent même d'autres types de mémoire, à

mi-chemin entre les registres et la mémoire centrale, pour améliorer les performances en stockant certaines valeurs à proximité du processeur. En effet, l'accès à la mémoire centrale est environ 400 fois plus lent que l'accès aux registres, mais ces derniers sont nécessairement en nombre restreint. D'un autre côté, bien que cette **latence** de communication avec la mémoire centrale soit élevée, la quantité de données que l'on peut échanger avec elle est bien trop grande pour pouvoir être traitée directement par le processeur ou encore être rangée dans les registres. Une solution hybride, consistant donc en l'utilisation de **mémoires caches**, fournit des accès plus rapides qu'à la mémoire centrale, mais plus lents qu'aux registres. Ainsi, les données en provenance de la mémoire centrale peuvent être entreposées dans l'attente de leur éventuelle utilisation. Au cours de l'exécution d'un programme, toute donnée manipulée par le processeur transite forcément par la mémoire centrale et par les registres ; il est par ailleurs extrêmement probable, mais pas nécessaire dans l'absolu, qu'elle soit lue à un moment ou un autre au sein de la mémoire cache.

En général, la mémoires cache est même hiérarchisée en plusieurs niveaux : de la mémoire cache L1 (Level 1), la plus rapide mais la plus petite, à la mémoire cache L3, plus lente mais plus grande. Ces mémoires sont réalisées à l'aide de transistors et placées à proximité du CPU. Le tableau 5.2 donne un aperçu des latences pour différents types de mémoire, ainsi que leurs capacités approximatives.

TABLE 5.2 – Latence approximative pour différents types de mémoire. Pour comparaison, un cycle d'horloge de CPU cadencé à 3 GHz vaut $1/(3 \cdot 10^9) \approx 0.3$ ns. Les capacités approximatives de ces différents types de mémoire sont également indiquées. Données de latence compilées par [2] d'après [1].

Type d'accès	Latence	Latence [Cycles CPU]	Capacité
Registre	0.3 ns	1	1 mot
Cache L1	0.9 ns	3	$\sim$ 10 kB
Cache L2	2.8 ns	9	$\sim$ 500 kB
Cache L3	12.9 ns	43	$\sim$ 4 MB
Mémoire centrale	120 ns	400	$\sim$ 10 GB
Mémoire flash SSD	$\sim$ 100 μs	160'000	$\sim$ 1 TB
Disque dur magnétique	1-10 ms	3'000'000	$\sim$ 1 TB

L'efficacité des mémoires cache dépend de la propension des instructions exécutées à répondre aux heuristiques de localité temporelle et spatiale. La première stipule que si une donnée est accédée, il est probable qu'elle le soit à nouveau dans un futur proche. La seconde stipule que si une donnée est accédée, il est probable que les données voisines le soient également (on revient là au concept de tableau de valeurs précédemment mentionné). Ces heuristiques sont souvent vérifiées dans la pratique, ce qui rend les caches très efficaces dans la plupart des cas de figures classiques du programmeur, sans que ce dernier n'ait beaucoup à se soucier de la façon dont il code les instructions. L'utilisation des boucles sur un tableau est un cas de figure parlant : les variables utilisées au sein de la boucle sont amenées à être réutilisées souvent au cours des différent cycles, tandis que les valeurs du tableau sont souvent accédées de manière séquentielle.

Notons pour finir que, bien qu'il puisse parfois être utile de réfléchir à la manière dont

les données sont stockées en mémoire pour optimiser l'accès à ces dernières, il est rare que cela soit nécessaire dans un premier temps, d'autres aspects de la performance étant souvent plus critiques au début du développement d'un programme, dont notamment la nature des algorithmes et structures de données utilisées.

5.5 Technologies et types de mémoires

Dans cette section, nous parcourons brièvement les technologies utilisées dans les ordinateurs modernes pour réaliser les différents types de mémoire de l'ordinateur, hors dispositifs auxiliaires (ceux-ci ont déjà été discutés en section 5.1.1).

5.5.1 Registres

Les registres sont en général réalisés de la manière présentée en section 4.5.5, à l'aide de bascules. Un décodeur (que l'on peut envisager comme l'inverse d'un multiplexeur) est utilisé conjointement aux bancs de registre pour sélectionner le registre à lire ou écrire à partir de son adresse.

5.5.2 Mémoire cache

Tout comme les registres, la mémoire cache est en général faite de circuits logiques tels que ceux présentés dans la section 4.5.5. Ce type de dispositifs à base de circuits à bascules est souvent désigné par le terme « static RAM » (SRAM), par opposition aux circuits mis en oeuvre de nos jours dans la mémoire centrale, désignés eux par le terme de « dynamic RAM » (DRAM), dont nous parlons ci-dessous.

5.5.3 Mémoire centrale

Bien que nous n'en traitons pas plus en détails, nous mentionnons que des circuits utilisant des condensateurs pour stocker des charges électriques peuvent être utilisés pour réaliser la mémoire centrale. L'avantage de l'utilisation de condensateurs est que les circuits qui en découlent sont plus simples ; en particulier, la densité de mémoire que l'on peut obtenir est plus élevée que pour les circuits à base de bascules, à prix égal. C'est la raison pour laquelle ils sont choisis pour la mémoire centrale, bien plus vaste que les registres ou la mémoire cache. En revanche, l'inconvénient de cette technologie est que les condensateurs se déchargent quand ils sont lus, d'une part, ainsi que de façon naturelle avec le temps d'autre part (même s'ils ne sont pas lus), ce qui nécessite de les rafraîchir régulièrement. Cette complexité de traitement supplémentaire est le prix à payer pour l'utilisation des condensateurs. Par ailleurs, avec ces dispositifs de type DRAM, le temps d'accès est plus grand que pour la SRAM, d'où le fait que cette dernière soit privilégiée pour les registres et la mémoire cache.

Enfin, notons que la structure de la mémoire centrale n'est pas linéaire, bien que les différents octets soient, eux, adressés de façon séquentielle. En réalité, afin de réduire la complexité des lignes correspondant aux différentes adresses, la mémoire centrale est disposée en lignes et en colonnes, et les adresses permettent de retrouver la colonne et la ligne où se trouvent les données adressées. L'étude du fonctionnement détaillé de l'accès aux données sort du cadre de ce cours, mais nous retenons que, d'une façon où d'une autre, il faut pouvoir passer d'une séquence de bits codant une adresse à une ligne et une colonne de la mémoire centrale. Comme nous l'avons déjà dit, des démultiplexeurs sont alors utilisés et le signal d'adresse est d'autant plus lent à traverser le démultiplexeur que le nombre de bits de l'adresse est long. Par ailleurs, la taille des mots de l'architecture a un impact sur la taille maximale de la mémoire centrale. Par exemple, dans une architecture 32 bits ne possédant pas de mécanisme pour étendre les adresses, une adresse ne peut être plus longue que 32 bits, ce qui limite la taille utile de la mémoire centrale à 2^{32} octets, soit un peu plus de 4 Go.

5.6 Jeux d'instructions

Pour clore ce chapitre, il faut remarquer qu'une caractéristique importante des processeurs consiste en l'ensemble des instructions qu'ils sont capables d'effectuer. Ces **jeux d'instructions** peuvent être rangés dans différentes catégories, dont les deux plus courantes sont :

- **RISC** (Reduced Instruction Set Computing) – Les processeurs RISC sont conçus pour exécuter un petit nombre d'instructions simples. Dans leur forme classique, les instructions y sont de longueur fixe et effectuent des opérations de base. Les processeurs ARM [7] sont des exemples de processeurs RISC.
- **CISC** (Complex Instruction Set Computing) – Les processeurs CISC sont capables d'effectuer un grand nombre d'instructions différentes. Les processeurs x86 d'Intel et AMD sont des exemples de processeurs CISC. Leurs instructions sont en général de taille variable. Dans ce cas, certaines instructions qui doivent être décomposées au sein d'un processeur RISC (telles que le calcul d'une racine carrée ou l'écriture au sein de plusieurs registres) peuvent ici exister de façon native.

Les architectures CISC ont été développées pour simplifier la programmation, en permettant aux programmeurs d'écrire des instructions plus complexes en moins de lignes de code, ce qui était particulièrement important à l'époque où les mémoires étaient de taille si limitée que l'espace occupé par le code lui-même était crucial (cela tend nettement à être moins le cas aujourd'hui). Lorsque les technologies ont évolué, les architectures RISC ont gagné en popularité de par leur simplicité et leur consommation énergétique potentiellement moindre. Il se trouve néanmoins que dans les processeurs modernes, cette distinction ne permet pas à elle seule de prédire la performance, la consommation ou la densité du code, qui dépendent aussi fortement de l'implémentation matérielle et de la compilation du code (cf. « compilation », chapitre 6.1.2).

7. Signifiant « Advanced RISC Machine », l'architecture ARM est aujourd'hui très présente au sein des smartphones, tablettes et autres systèmes pour lesquels la consommation d'électricité est importante.

L'architecture de type RISC nommée « MIPS » [8] est d'une certaine importance historique et pédagogique. En effet, elle a été beaucoup utilisée dans les années 90 et de nombreux textes dans la littératures s'en servent comme référence pour expliquer les concepts de base des architectures de processeurs. Dans le prochain chapitre, nous allons donc nous intéresser à la nature et à la représentation des instructions discutées ci-dessus en prenant l'architecture MIPS comme exemple.

Notons pour terminer que ce que l'on désigne communément par le « nombre de bits de l'architecture » (par exemple, une « architecture 64 bits ») désigne la taille des mots que le processeur manipule le plus naturellement. C'est donc une taille de référence, à laquelle correspond en général la taille des registres généraux, des entiers natifs manipulés par l'UAL ou encore des pointeurs au sein des programmes. Certaines systèmes d'exploitation tels que Windows sont souvent désignés en précisant pour quel taille de référence ils ont été compilés. De même, lorsqu'on désirer installer un logiciel sur une machine, il n'est pas rare de devoir encore aujourd'hui sélectionner manuellement entre une version compilée pour une architecture 32 ou 64 bits.

8. Ce qui signifie « microprocessor without interlocked pipeline stages » ; le concept de pipelining a été discuté dans la section 5.3.3 L'architecture MIPS et ses extensions a été utilisée notamment dans les célèbres consoles de jeu Nintendo 64 ainsi que les Playstation 1 et 2.

Chapitre 6

Fonctionnement logiciel des ordinateurs

Dans le chapitre précédent, nous avons discuté du parcours des données au sein des différents composants d'un ordinateur implémentant l'architecture de von Neumann. En particulier, nous avons vu que les instructions d'un programme, chargées au sein de la mémoire centrale en vue de leur exécution, sont appelées à tour de rôle au sein du processeur afin d'être interprétées par l'unité de contrôle, certains sauts pouvant êtres effectué en fonction de la nature de l'instruction et du résultat de calculs précédents. Après leur interprétation par l'unité de contrôle, des opérations sont effectuées par l'unité arithmétique et logique, et des données peuvent en retour être lues ou écrites en mémoire centrale.

Dans ce chapitre, nous allons nous intéresser davantage à la nature de ces instructions et à leurs conséquences sur la gestion de la mémoire centrale, et moins au détail du flux de l'information au sein de la machine. Nous aborderons également la question des langages de programmation et de leurs principales caractéristiques, ainsi que celle des systèmes d'exploitation, qui forment la couche logicielle fondamentale sur laquelle les autres logiciels s'appuient pour interagir avec le matériel informatique.

Comme certains termes reviennent à plusieurs reprises dans ce chapitre, nous en donnons ici une description succincte.

Par « programme », nous entendons une suite d'instructions abstraites destinées à être exécutées par une machine. C'est souvent ce que l'on désigne de façon informelle par « code ».

Le terme « processus » (*process* en anglais) désigne un programme en cours d'exécution. L'état d'un processus a un instant donné est caractérisé par la valeur des données en mémoire (y compris les données relatives aux instructions du programme lui-même). Un processus peut être interrompu pour laisser la place à un autre processus, et reprendre son exécution plus tard. Nous discuterons plus en détail de ces concepts dans la section sur les systèmes d'exploitation.

Le terme « logiciel » (*software* en anglais) désigne une entité plus vaste qu'un programme : il s'agit d'un ou plusieurs programmes (potentiellement en interaction entre eux) ainsi que des données sur lesquelles ces programmes portent. Un logiciel est déjà prêt à être exécuté par une machine donnée (cf. section sur le processus de compilation).

6.1 Langages de programmation

Nous savons que les instructions, ainsi que toute donnée stockée dans la mémoire de la machine, sont *in fine* codées en binaire avant d'être envoyées au processeur. Cependant, il est évident que l'écriture des instructions destinées à la machine, activité nommée « programmation », est fastidieuse et peu pratique pour l'être humain si elle doit se faire directement en binaire. C'est pourquoi des langages de programmation ont été développés. Ils proposent, dans le même temps, des mécanismes d'abstraction permettant de mieux gérer la complexité des programmes, sans devoir s'astreindre à un découpage « atomique » des instructions directement interprétables par l'unité de contrôle. Enfin, les différents langages permettent, à des degrés divers, d'abstraire les instructions de telle sorte que leur formulation devienne indépendante de l'architecture de l'ordinateur. Afin de désigner à quel point un langage de programmation est proche du langage machine (directement interprétable par le matériel), on parle de **niveau d'abstraction** du langage.

6.1.1 Les couches d'abstraction

Nous dirons d'un langage proche de la machine que c'est un langage « bas niveau ». Le langage le plus bas niveau que l'on puisse imaginer, alors, consiste à spécifier directement la valeur des séquences de bits codant les instructions envoyées au processeur, en s'astreignant aux règles d'interprétation que l'unité de contrôle. Un tel langage est appelé langage **machine**. Une séquence de bits codant une instruction est de la taille d'un mot de l'architecture dans l'exemple de l'architecture MIPS choisi ici, mais peut être variable dans le cas général.

Le premier degré d'abstraction où le code peut aisément être lu par un humain est nommé **assembleur**. Ce dernier consiste essentiellement en une transcription des instructions machine en symboles plus explicites, ainsi qu'en certains raccourcis conceptuels. Les instructions écrites en langage assembleur sont ensuite traduites en langage machine par un programme appelé **assembleur**. Le code machine étant propre à une architecture donnée, le code assembleur, qui en est très proche, l'est également. Un exemple de code assembleur MIPS (cf. section 5.6) et de sa correspondance en langage machine (ainsi qu'en C et en Python) est donné dans le tableau 6.1.

Au-dessus du langage assembleur, la distinction entre langages de haut niveau et de bas niveaux continue d'être relative ; du point de vue d'un code en Python, le langage C est bas niveau, tandis que du point de vue d'un code assembleur, le langage C est de haut niveau. Il existe cependant une différence fondamentale entre des langages tels que C et Python qui réside dans la façon dont le code du programme, nommé **code source**, est transformé en code machine. Nous discutons plus bas de cette différence, en section 6.1.2.

Le tableau 6.1 permet de comparer la formulation d'une instruction dans des langages de différents niveaux d'abstration. Notons que, par souci de lisibilité, le tableau incorpore un exemple pour lequel une seule instruction machine et assembleur correspond à une seule ligne de code en C et en Python. Cependant, dans bien des cas, il faut plusieurs instructions en assembleur pour correspondre à une ligne de code de plus haut niveau. L'annexe I donne un exemple de traduction d'une ligne de code simple en C et en langage

TABLE 6.1 – Illustration d'une instruction de soustraction dans des langages de différents niveaux d'abstraction. Ici, les exemples pour l'assembleur et le code machine sont donnés pour l'architecture MIPS mentionnée dans le chapitre précédent.

Type de langage	Exemple de langage	Instruction de soustraction
Naturel	Français	« Stocker dans `t0` la différence entre `s1` et `s2`. »
Interprété	Python	`t0 = s1 - s2`
Compilé	C	`t0 = s1 - s2;`
Assembleur	MIPS	`sub $t0, $s1, $s2`
Machine	MIPS	`000000 10001 10010 01000 00000 100010`

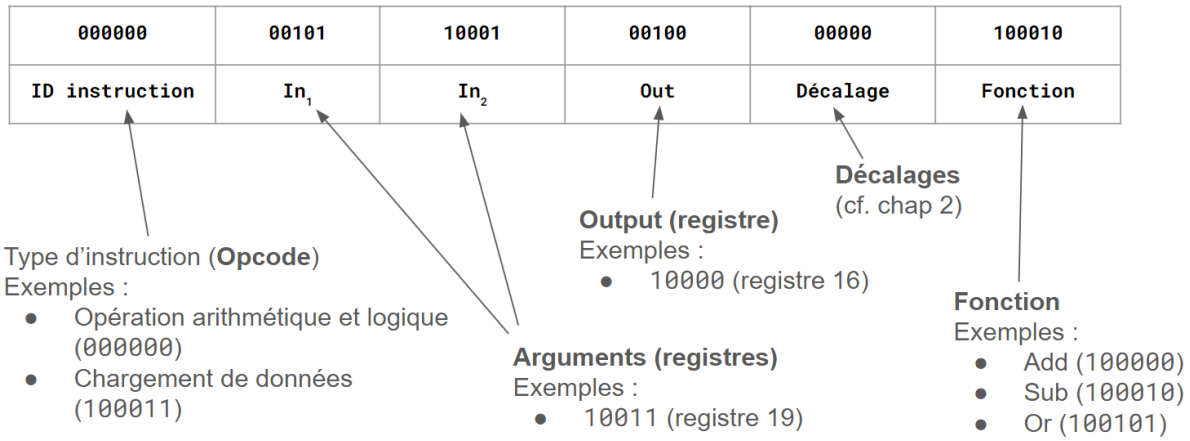


FIGURE 6.1 – Exemple d'instruction de soustraction en langage machine dans le cas de l'architecture MIPS. D'autres exemples sont également donnés pour chaque champ de bits.

assembleur MIPS.

Dans les langages de plus haut niveau, la soustraction de deux quantités et leur stockage au sein d'un emplacement de la mémoire se formule de façon assez intuitive. En revanche, afin de coller au langage machine, la façon de décrire cette opération en langage assembleur contraint à la formuler de la façon suivante, où « mnémonique » désigne le nom de code d'une instruction :

`<mnémonique> <addresse résultat> <addresse opérande 1> <addresse opérande 2>`.

On s'aperçoit en regardant la table 6.1 que cette instruction en assembleur permet déjà de faciliter le travail du programmeur, puisque son équivalent en langage machine inclut six champs de bits, correspondant pour certains à un aspect de l'instruction qui est implicitement compris au sein du mnémonique « sub ». En l'occurrence, le premier champ de bits de l'instruction machine spécifie le type d'instruction (ici, opération entre registres), tandis que le dernier spécifie l'opération à effectuer (ici, une soustraction). Enfin, l'avant-dernier champ de bits spécifie un décalage (ici nul) utilisé dans d'autres cas de figure (cf annexe I pour un exemple). Finalement, les trois champs de bits du milieu spécifient les registres qui contiennent les valeurs à soustraire ainsi que l'emplacement où stocker le résultat. La figure 6.1 illustre la structure d'une instruction machine dans le cas de l'architecture MIPS pour une instruction de soustraction.

6.1.2 Compilation et interprétation

Compilation et interprétation désignent des façons d'exécuter les instructions d'un programme, c'est-à-dire de faire en sorte que le processeur traite l'information de la manière spécifiée par le programmeur au sein d'un code informatique.

Dans ce document, nous ne traitons en détail ni des compilateurs, ni des interpréteurs, qui sont des sujets complexes sortant du cadre de ce cours. Cependant, nous donnons ici un aperçu des principes qui sous-tendent ces deux méthodes d'exécution du code.

Langages compilés

Tout comme l'assembleur transforme le code assembleur en code machine, un **compilateur** transforme le code source d'un programme écrit dans un langage de plus haut niveau en code machine[1]. Cette opération s'appelle la **compilation**. Des langages tels que le C, C++ ou Rust sont compilés.

Les avantages des langages compilés sur les langages de type assembleur sont les suivants :

- L'étape de compilation permet d'autoriser à l'utilisateur des raccourcis et **abstractions** qui ne sont pas disponibles en assembleur. Par exemple, l'écriture de la soustraction au sein du tableau 6.1 est plus concise en C qu'en assembleur, parce que dans le cas du C le compilateur prend en charge des éléments qui sont cachés à l'utilisateur, tels que la gestion des registres. Un autre exemple est le mécanisme des boucles, qui permet au programmeur d'abstraire les sauts d'instructions de façon logique, en les liant à des conditions, sans devoir spécifier l'adresse des instructions à atteindre en fonction des conditions vérifiées.
- Les langages compilés sont souvent conçus pour être **portables** entre différentes plateformes matérielles, ce qui signifie que le même code source peut être compilé et exécuté sur différents types de systèmes sans modification. En revanche, l'assembleur est spécifique à une architecture de processeur particulière, il faut donc qu'un compilateur soit disponible pour la machine cible.
- Les compilateurs modernes sont capables d'effectuer des **optimisations** complexes qui seraient difficiles et fastidieuses à effectuer manuellement en assembleur. Ces optimisations peuvent améliorer la performance du code sans effort supplémentaire de la part du programmeur. Dans de rare cas cependant, les compilateurs ne sont pas en mesure de repérer des optimisations qui seraient à la portée d'un programmeur expérimenté en assembleur.
- Les langages compilés permettent souvent des **vérifications** quant à la cohérence du code lors de la compilation, ce qui peut réduire le nombre d'erreurs se produisant au moment de l'exécution.

Bien entendu, ce qui est gagné en abstraction avec les langages compilés est perdu en contrôle. En effet, le programmeur n'a plus la main sur tous les détails de l'exécution du programme et doit se fier au compilateur pour générer un code machine efficace. Par ailleurs, le compilateur peut, selon l'envergure du code à compiler, mettre un temps non négligeable à produire le code machine, ce qui peut ralentir le développement du code.

Comme dit plus haut, un programme donné doit être compilé pour chaque architecture sur laquelle il doit être exécuté. Par exemple, un jeu-vidéo destiné à être exécuté sur PC et sur

1. Il est également possible de configurer un compilateur pour qu'il produise du code assembleur en lieu et place du code machine.

téléphone ou tablette doit être compilé séparément pour le jeu d'instructions de l'architecture x86 (CISC) ainsi que pour celui de l'architecture ARM (RISC).

Langages interprétés

L'idée d'un langage interprété est que le code source est en fait l'input d'un logiciel spécial [2] qui, lui, se charge d'appeler des fonctions prédéfinies pour exécuter les instructions du code source. Ce logiciel s'appelle un **interpréteur**.

Une variante consiste à traduire le code source en une forme appelée **bytecode**, qui est la version réellement interpretée par l'interpréteur. La génération du bytecode est automatique et se fait avant l'exécution du programme. Notons que ce bytecode constitue un langage intermédiaire dans lequel est converti le code source. On pourrait imaginer un interpréteur capable d'interpréter directement du code source, mais il existe de nombreux avantages à passer par un bytecode. En effet, afin de simplifier les interpréteurs, ces derniers sont conçus pour accepter en entrée un code formaté d'une façon bien précise, que l'on ne veut pas contraintre le programmeur à respecter au sein du code source. Afin d'éviter que cette restructuration du code n'ait à être effectuée à chaque exécution, l'utilisation d'un bytecode s'impose. Par ailleurs, un certain nombre d'optimisations peuvent être effectuées sur le bytecode, ce qui permet d'améliorer la performance du programme, bien que de façon moins significative qu'avec un langage compilé. Python est un exemple célèbre de langage interprété utilisant un bytecode. L'interpréteur standard de Python est écrit dans le langage C. Enfin, notons qu'il est courant d'employer le terme « script » pour se référer à un programme interprété plutôt que compilé.

Compilation Just-In-Time

Des formes hybrides, à mi-chemin entre la compilation et l'interprétation, sont envisageables. Dans ces langages hybrides, l'interpréteur convertit et optimise les parties les plus critiques du bytecode en instructions machine au moment même de l'exécution (Just-In-Time compilation, abrégé JIT). Il faut noter qu'étant donné le temps de compilation potentiellement long, cette méthode est utilisée conjointement avec une méthode d'interprétation « pure » en attendant que la compilation JIT se termine et que le code machine généré prenne le relais. C'est pourquoi, dans le cas de Java ou de C# par exemple, on parle de langages semi-interprétés ou hybrides. Ainsi, bien qu'*in fine* le code source soit transformé en code machine, cette étape est en général invisible pour l'utilisateur du programme.

Comparaison entre langages compilés et interprétés

La majorité du temps, les langages compilés sont plus performants à l'exécution que les langages interprétés, car le code machine est directement exécuté par le processeur, sans pas-

2. En réalité, on peut imaginer des interpréteurs matériels, bien que ce soit rarement ce que l'on cherche à désigner en utilisant ce terme ; un processeur est pour ainsi dire un interpréteur de langage machine.

ser par des couches d'interprétation supplémentaires [3]. Néanmoins, les langages interprétés sont populaires en raison des avantages qu'ils procurent typiquement :

- La portabilité des langages interprétés est en général plus aisée du point de vue du programmeur (si tant est qu'un interpréteur existe pour la machine cible), parce que le code source ne doit pas être compilé pour l'architecture cible.
- La gestion de la mémoire est le plus souvent gérée automatiquement par les interpréteurs, ce qui peut simplifier la tâche du programmeur. Cette gestion automatique de la mémoire se fait grâce à des mécanismes tels que le « ramasse-miettes » (*garbage collector* en anglais).
- Certains types de bugs sont plus faciles à repérer dans un langage interprété, notamment en raison des débogueurs capables d'exécuter le code ligne par ligne et de fournir des informations sur l'état du système à tout moment de l'exécution. Par ailleurs, les messages d'erreur sont souvent plus explicites au sein des langages interprétés. Cependant, les langages compilés offrent de nos jours également des outils de débogage puissants. Enfin, notons certains langages interprétés tolèrent des pratiques de programmation moins rigoureuses (cf. typage dynamique ci-dessous, par exemple), ce qui peut mener à des erreurs plus difficiles à repérer.
- Les langages interprétés ont souvent une syntaxe qui peut rendre leur apprentissage moins intimidant et fastidieux pour les débutants.
- À l'instar de Python, les langages interprétés proposent parfois un typage dynamique, ce qui signifie que le type des données (entier, flottant, chaîne de caractères, etc.) n'est pas spécifié dans le code source, mais est déterminé au moment de l'exécution, et peut donc être amené à changer au cours de cette dernière. Cela peut sembler simplifier la tâche du programmeur débutant, mais peut aussi mener à des erreurs difficiles à repérer. À de nombreux égards, la spécification explicite du type des données peut être vue comme un avantage plutôt qu'un inconvénient.

Tous les aspects mentionnés ci-dessus ne sont pas strictement inhérents aux langages interprétés, mais constituent plutôt une tendance globale. On peut en général envisager, ne serait-ce qu'en théorie, des mécanismes pour intégrer la plupart des avantages des langages interprétés au sein d'un langage compilé.

Pour conclure, il est important de remarquer qu'un langage de programmation n'est, en soi, ni compilé, ni interprété. En effet, un langage de programmation est un ensemble de règles syntaxiques et sémantiques qui permettent de formuler des instructions destinées à être exécutées par un ordinateur. Cependant, la façon de gérer l'exécution du code est la plupart du temps standardisée et étroitement liée au langage, d'où les termes de « langage compilé » et « langage interprété », qui sont à strictement parler des abus de langage. En théorie, rien n'empêcherait de concevoir un compilateur de code Python, ni un interpréteur de code C.

Le tableau 6.2 ci-dessous mentionne quelques langages de programmation populaires ainsi que leur mode d'exécution le plus commun.

3. Notons que cette différence de performance, dans bien des cas, n'est pas pertinente. Par exemple, pour une application de messagerie, l'essentiel des potentielles « lenteurs » dont l'utilisateur fait l'expérience provient du réseau lui-même, tandis que la partie du traitement de l'information qui concerne la saisie et l'affichage du texte est largement trop simple pour qu'une différence due au mode d'exécution soit appréciable.

TABLE 6.2 – Exemples de langages de programmation ainsi que leur mode d'exécution usuel. À noter que la classification entre langages hybrides et interprétés est sujette à discussion, en particulier pour des langages qui ne permettent qu'optionnellement ou partiellement des mécanismes de compilation Just-In-Time tels que Lua, Erlang ou Octave. Il faut donc garder en tête que ce tableau est une classification simplifiée.

Mode d'exécution	Exemples de langages
Compilée	C, C++, Go, Rust, Haskell, Lisp, Fortran, Pascal, Swift, Zig
Hybride	Java, C#, Matlab, Julia, Erlang, Lua
Interprétée	Python, JavaScript, Ruby, Perl, PHP, Octave, Mathematica, Maple

6.2 Systèmes d'exploitation

6.2.1 Origines et rôle général

Dès les années 50, il devint évident que l'insertion physique (*via* des cartes perforées) des différentes parties des programmes constituait une tâche répétitive qu'il fallait automatiser ; une solution a donc été de regrouper les différentes parties d'un programme en lots (*batchs* en anglais) et de laisser l'ordinateur gérer l'exécution séquentielle de chacun. Pour ce faire, il fallait qu'un programme spécial demeure au sein de l'ordinateur – quel que soit le programme qui intéresse réellement l'utilisateur en fin de compte – afin d'automatiser la gestion des batchs. Ce programme, à exécuter avant tout autre, était appelé « moniteur de programme », précurseur des systèmes d'exploitation modernes.

D'autres problèmes encore se posaient à l'époque :

1. La façon d'intéragir avec les périphériques constitue une spécificité non triviale que le programmeur, déjà en prise avec des considérations propres au problème de base qu'il aborde, doit gérer.
2. Les périphériques sont souvent beaucoup plus lents que le processeur (par exemple une imprimante utilisée pour afficher les résultats), qui doit attendre que les données soient prêtes avant de les traiter. Le processeur est donc souvent inactif, ce qui est une perte de temps.
3. Les pertes de temps du point précédent sont d'autant plus regrettables que les machines sont en général partagées par de multiples utilisateurs (souvenons-nous que nous parlons d'une époque où les ordinateurs personnels n'existaient pas).

Une solution à ces problèmes est de développer un programme spécial que nous nommerons **système d'exploitation** (operating system en anglais, abrégé **OS**) qui gère les problèmes mentionnés plus haut de la façon suivante, en les cachant au programmeur :

1. L'interaction avec les périphériques est gérée grâce à des codes qui servent de **pilotes** de périphériques (*drivers* en anglais), permettant de séparer la logique de code servant à la communication avec les périphériques de la logique de code propre aux tâches spécifiques que le programmeur désire implémenter. Ce dernier n'a plus à se soucier des détails relatifs aux périphériques, et peut se concentrer sur la logique de son propre programme.
2. Si le processeur doit attendre une donnée, il peut exécuter un autre programme durant ce laps de temps (soit du même utilisateur, soit d'un autre utilisateur), ce qui permet de maximiser l'utilisation du processeur. C'est ce qu'on appelle une exécution **multitâche** (*multitask* en anglais). L'OS doit alors gérer la mémoire de façon à ce que les programmes

n'entrent pas en collision, ce qui entraîne toutes sortes de complications que nous discutons brièvement plus loin.

Rapidement, les systèmes d'exploitation sont devenus incontournables dans la plupart des situations, tant ils permettent d'abstraire des détails techniques complexes mais répétitifs et très éloignés des problèmes que le programmeur cherche à résoudre. En plus des aspects historiques mentionnés ci-dessus, à savoir les pilotes et la gestion de la mémoire, la plupart des OS ont rapidement inclu un système de gestion des permissions et des utilisateurs, des systèmes de gestion des fichiers, de la sécurité, des réseaux, etc. L'ensemble de ces composants essentiels de l'OS forment son **noyau**[4] : un logiciel « maître » qui gère tous les autres logiciels ainsi que leurs interactions avec le matériel. Comme mentionné dans le chapitre 5.1.1, le noyau de l'OS est chargé en mémoire dès que possible au démarrage de l'ordinateur, et est supposé y rester.

Il faut noter que le noyau possède un espace mémoire qui lui est réservé, et qui est protégé des autres programmes de l'utilisateur, chacun ayant son propre espace mémoire. Pour accéder au noyau de la façon prévue à cette effet, les programmes doivent passer par des **appels système** (*system calls* en anglais), qui sont des fonctions prédéfinies pour différents cas de figure. Les appels système sont souvent utilisés pour effectuer des opérations qui nécessitent des privilèges particuliers, tels que la lecture ou l'écriture dans des fichiers, la gestion de la mémoire centrale, des processus, etc. Au niveau de la programmation, les fonctions du système sont regroupées au sein d'une **interface de programmation** (API en anglais) qui permet au programmeur d'accéder aux fonctionnalités du système d'exploitation de façon standardisée.

Les OS modernes viennent en général avec des logiciels secondaires qui servent d'outils pour l'utilisateur, tels que des explorateurs de fichiers, des éditeurs de texte, etc. Ces derniers ne constituent pas une partie essentielle de l'OS. Il convient donc de distinguer le noyau de l'OS de son interface et de ses **logiciels applicatifs** qui sont des programmes « ordinaires ». Mentionnons tout de même que parmi ces logiciels applicatifs, l'interface graphique (**GUI** pour *graphical user interface*) d'un OS est souvent étroitement associée à ce dernier dans la culture populaire. Cependant, il est tout à fait possible en principe d'utiliser un OS sans interface graphique autre qu'une console, ou encore de changer l'interface graphique d'un OS donné sans toucher au noyau.

Ainsi, l'OS désigne donc un logiciel spécial qui occupe un rôle très bas niveau, au plus proche du matériel, et qui possède des privilèges lui permettant de gérer l'utilisation des ressources par les autres logiciels, dits « applicatifs ». Les OS les plus célèbres incluent Windows, macOS, Android, iOS, ainsi que la famille des systèmes BSD et des distributions Linux telles qu'Ubuntu. Il faut noter que macOS et iOS reposent sur Darwin, un système dérivé de BSD et de Mach, historiquement lié à la famille UNIX, tandis qu'Android repose sur le noyau Linux[5]. Enfin, ce que l'on nomme ci-dessus les « distributions » Linux désignent des OS complets qui se servent du même noyau (Linux) mais qui incluent en plus un ensemble de programmes (système de gestion de paquets, interface graphique, logiciels applicatifs, etc.) qui rendent l'OS utilisable en pratique. En fait, en plus du noyau Linux, ces distributions incluent le plus souvent des composants issus du projet GNU (« GNU's Not UNIX », créé dans les années 80), qui vise à développer des logiciels libres et open-source, *a contrario* des logiciels propriétaires auxquels sa philosophie s'oppose. On peut notamment citer le compilateur GCC ou encore le langage de script Bash, qui est un

4. Il semblerait que le terme de *kernel* ait été employé, surtout dans le contexte d'UNIX, de façon à constituer une analogie avec le cœur d'une noix (cerneau) ; ainsi, le *shell* (coquille), qui est également le nom donné à l'interface système, correspond à la couche la plus externe de ce dernier, qui constitue l'interface avec l'extérieur, et isole le noyau. Cette analogie est importante car elle est réutilisée dans d'autres domaines de l'informatique, où le terme d'interface apparaît souvent.

5. Linux est un noyau libre et open-source ; les systèmes BSD sont, quant à eux, des systèmes d'exploitation complets, également libres et open-source, historiquement liés à la famille UNIX. Linux aussi bien que les systèmes BSD modernes ont fait leur apparition au début des années 1990.

shell pour UNIX. GNU et le noyau Linux sont si souvent utilisés conjointement pour former des OS cohérents et complets que l'on parle parfois de systèmes GNU/Linux pour désigner ces derniers. Pour terminer ce tour d'horizon, de nombreux systèmes de type UNIX implémentent tout ou partie de la norme POSIX, qui définit notamment des interfaces d'appels système et d'environnement utilisateur ; cela concerne notamment de nombreuses distributions Linux, les systèmes BSD, ainsi que macOS.

6.2.2 Gestion de la mémoire et de l'exécution des programmes

Ainsi que mentionné dans la section précédente, l'OS doit gérer la mémoire centrale en l'allouant judicieusement aux différents programmes en cours d'exécution, tant pour des raisons de sécurité que d'efficacité. Un programme en cours d'exécution est appelé un **processus** et est caractérisés par son état, c'est-à-dire toutes les données à stocker afin de pouvoir reprendre l'exécution du programme à un point précis s'il s'arrêtait. Ces données incluent par exemple les valeurs de chaque registre, les instructions du programme lui-même, ou encore les données manipulées par le programme. De ce fait, on peut voir l'OS dans son entièreté comme l'intermédiaire de communication entre le hardware et les processus.

Mémoire virtuelle

Pour traiter efficacement tous les problèmes soulevés dans les sous-sections qui précède, un mécanisme d'abstraction des adresses de la mémoire centrale s'avère très utile. Cette abstraction consiste à différencier l'emplacement réel des données en mémoire du nombre (adresse) qu'on utilise pour désigner cet emplacement, et se nomme **mémoire virtuelle**. Grâce à ce mécanisme, chaque programme peut être écrit « comme s'il était seul » sur la machine, en adressant les données à partir d'une adresse « zéro » de façon continue, alors même que l'adresse physique de ces données peut être fractionnée dans la mémoire centrale en raison de l'exécution d'autres programmes[6]. De même, cela permet si besoin de déplacer l'entièreté des données relatives à un processus sur une mémoire auxiliaire, par exemple lorsqu'un processus est en pause et que l'on manque d'espace en mémoire centrale pour exécuter un autre programme. L'OS doit donc déterminer un segment de la mémoire à attribuer aux différents programmes en cours d'exécution[7].

De la même façon qu'il gère les différents espaces alloués en mémoire aux différents programmes, l'OS gère via l'**ordonnanceur** (*scheduler*) l'alternance temporelle avec laquelle le processeur exécute les instructions issues des différents processus. L'annexe J propose une synthèse comparative entre multitasking, multithreading (cf. chapitre 5.4) et parallélisme (cf. chapitre 5.1.2), afin d'éviter les confusions.

6. Notons que des circuits électroniques spécialement dévolus à la conversion entre mémoire physique et mémoire virtuelle sont utilisés, afin de ne pas déléguer à un logiciel cette tâche critique du point de vue de la sécurité et de la performance. Ces circuits font partie d'un élément nommé MMU pour *memory management unit*

7. D'ailleurs, si un processus tente d'accéder à une zone de la mémoire hors de son segment, l'OS se charge d'éliminer le processus, donnant lieu à une erreur de segmentation (*segmentation fault*).

Ordonnancement des processus

À l'origine, un ordonnancement dit « coopératif » pouvait être envisagé entre les processus : dans un tel cas, chaque processus rend la main à l'OS automatiquement à chaque fois qu'il invoque ce dernier *via* un appel système, qui implique typiquement des instructions telles que des allocations de la mémoire ou des écritures dans des fichiers prenant plusieurs cycles d'horloge à être exécutées. Cependant, un code malveillant ou mal écrit pourrait ne jamais rendre la main à l'OS, ce qui rendrait l'ordinateur inutilisable. C'est pourquoi la plupart des OS modernes utilisent un ordonnancement dit « préemptif », où l'OS peut interrompre un processus à tout moment pour donner la main à un autre. L'ordonnancement préemptif est rendu possible par l'horloge de l'ordinateur, qui permet de mesurer le temps écoulé et de déterminer si un processus donné a eu un laps de temps suffisant durant les derniers instants. Dans tous les cas, une telle interruption de processus fait partie du fonctionnement usuel de l'ordinateur et n'est normalement pas perçue par l'utilisateur. Ce changement de processus est nommée « context switch » en anglais. La valeur des registres du processus en question sont copiés dans la mémoire centrale (voire sur une mémoire auxiliaire, faute de mieux), et le processus est mis en attente jusqu'à ce que l'ordonnanceur lui redonne la main. Notons que dans le cas d'un ordinateur multicœur, chaque cœur peut exécuter un processus différent, ce qui permet d'effectuer un véritable multitasking parallèle.

Pile et tas

Au moment où un programme est exécuté, il se voit alloué un espace mémoire qui lui est propre, et qui est divisé en trois parties : les instructions (le **code** lui-même [8]), la **pile** (stack en anglais) et le **tas** (heap en anglais). En réalité, l'espace mémoire dévolu à un processus comprend également d'autres zones non traitées dans ce document.

Le tas sert essentiellement à stocker certaines valeurs qui doivent exister en dehors de toute fonction ou dont la taille est imprévisible. Il est géré par le programmeur qui, dans les langages bas-niveau doit la plupart du temps explicitement utiliser un mécanisme prévu par le langage (comme par exemple `malloc` en C) pour réserver de l'espace mémoire pour y stocker des données, et libérer cet espace une fois qu'il n'est plus nécessaire.

La gestion de la pile, quant à elle, est normalement effectuée automatiquement même dans un langage bas-niveau. Alors que les variables peuvent être récupérées dans un ordre arbitraire au sein du tas, dans la pile seule la dernière valeur ajoutée peut être récupérée (« dépilée ») à un instant donné. Cette logique de pile est souvent nommée LIFO, pour *Last In First Out*. Elle sert à stocker des données de taille fixe. Son premier avantage est d'être plus rapide que le tas, puisque la récupération des données est plus simple. Son second avantage est qu'elle offre une gestion optimale de la mémoire, sans laisser de « trous » entre les données stockées, contrairement à ce qui peut arriver dans le tas (dans ce cas, on parle de « fragmentation » de la mémoire).

Notons qu'à chaque fois qu'une fonction est appelée au sein d'un programme, une « sous-pile » est ajoutée sur la pile, et la référence au début de la sous-pile est gardée en mémoire afin de revenir à l'état précédent l'appel de la fonction lorsque celle-ci se termine, en y ajoutant éventuellement la valeur de retour de la fonction. Voici un exemple :

8. C'est d'ailleurs là une différence fondamentale entre un programme compilé et un programme interprété. Dans le second cas, la partie du segment dévolue au code est la même quel que soit le script qui est interprété, ce dernier étant simplement une donnée d'input du véritable logiciel : l'interpréteur !

```
1  int f(int x) {
2      return x + 2;
3  }
4  int g(int x) {
5      return x * x;
6  }
7  int main() {
8       int A = 3;
9       int result = f(g(A));
10 }
```

Si l'on veut exécuter ce code à la main et calculer la valeur de `result`, on commence par vouloir évaluer la fonction `f` en ligne 9. Ce résultat sera un entier, du fait de la définition de la fonction `f` (ligne 1). On peut donc, après avoir jeté un oeil à la définition de `f`, noter sur une feuille de papier que le résultat est un entier qui vaut `x + 2`. Cependant, `x` est inconnu. On doit maintenant remplacer la valeur de `x` par un appel à `g`, qui encore une fois retourne une valeur de taille connue (ligne 4). On peut donc noter une seconde ligne, au-dessus de la première, qui indique que `x` est un entier qui vaut `y * y`. Le problème est maintenant reporté sur la valeur de `y`. Celle-ci, toujours d'après la ligne 9, vaut `A`. On peut enfin remplacer `y` par `A`, qui est une variable locale de la fonction `main` (ligne 8). La pile des appels n'ira pas plus haut, car on peut maintenant commencer à « dépiler » les valeurs, en commençant par la dernière, pour remonter jusqu'à la première. On obtient alors successivement, pour chaque ligne, que `y=A=3` (cas trivial), puis `x=y*y=3*3=9`, puis `result=x+2=9+2=11`. En faisant cela, on peut effacer au fur et à mesure les lignes de la pile qui sont utilisées, et simplement retenir la conclusion `result=11`.

Ainsi, la pile augmente de taille à chaque fois qu'une fonction est appelée, et réduit de taille à chaque fois qu'une valeur de retour est atteinte. Elle permet de façon naturelle de ne pas perdre la trace des variables locales et des valeurs de retour au fil des appels de fonction. Au sein de l'architecture MIPS, il existe des registres spéciaux destinés à accueillir les adresses de début et de fin de la pile. Il faut noter que la pile n'est pas adaptée pour stocker des valeurs qui doivent perdurer en-dehors de la fonction, car ces valeurs seraient effacées dès que la fonction se termine. Pour cette raison, c'est le tas qui est utilisé pour stocker de telles valeurs. De même, la pile ne peut accueillir des données qui sont amenées à changer de taille durant l'exécution, car ce faisant elles effaceraient les données qui les suivent. Pour ces raisons, c'est à nouveau le tas qui est utilisé pour stocker des données de taille variable, telles que des tableaux dynamiques ou des structures de données dynamiques comme des arbres ou des listes chaînées (cf. partie du cours sur les structures de données).

Notons pour terminer que, du fait que la pile contienne des adresses permettant de garder le fil des instructions à suivre, elle est souvent la cible de tentatives de piratage informatique, qui visent à modifier ces adresses pour exécuter du code malveillant. C'est pourquoi les OS modernes incluent des mécanismes de protection de la pile, dont l'étude sort du cadre de ce cours.

6.3 Performance d'un ordinateur

Tant dans les domaines du calcul scientifique que dans celui de l'informatique de tous les jours, la performance d'un trio algorithme-programme-ordinateur constitue souvent une grandeur importante. En effet, comme nous le verrons dans la partie du cours dévolue à l'algorithmique et à la programmation, certains problèmes sont intrinsèquement difficiles à résoudre, et il est donc

important de disposer d'ordinateurs performants pour les résoudre en un temps raisonnable. Nous nous intéressons ici uniquement aux éléments relatifs à l'architecture et au système d'exploitation qui influent sur la performance d'un ordinateur, et délaissons à la partie algorithmique du cours les considérations d'ordre mathématique et logique.

6.3.1 Mesures de performance

Le **temps d'exécution** d'un programme donné dépend de nombreux facteurs, dont les principaux sont les suivants :

- La **fréquence d'horloge** du processeur, qui détermine le nombre de cycles fetch-decode-execute (cf. section 5.3) que le processeur peut effectuer par unité de temps. Cette fréquence est mesurée en Hertz ($1\ \text{Hz} = 1\ \text{s}^{-1}$).
- Le **nombre de fils d'exécution parallèles** participant au programme considéré (cf. section 6.2 et annexe J), ainsi que leur capacité à participer efficacement au programme sans se gêner mutuellement par des accès concurrents aux ressources de la machine.
- Le coût des différents **accès à la mémoire** (cf. section 5.4), qui dépend de la hiérarchie des caches, de la taille des caches, de la fréquence des accès, etc.
- Le temps d'attente pour les **périphériques** (par exemple, lecture d'un fichier écrit sur le disque dur), qui peut être très long par rapport à l'échelle de temps d'autres éléments du programme (cf. tableau 5.2).
- L'efficacité du **compilateur** ou de l'**interpréteur** à générer du code machine. On a vu par exemple qu'un programme en Python a tendance a être plus lent à exécuter qu'un programme similaire en C, du fait de la nature interprétée de Python.
- Le fait que le programme en cours d'exécution n'est pas le seul à être exécuté sur l'ordinateur, et que d'**autres processus** peuvent occuper des ressources (mémoire, processeur, etc.) qui seraient autrement pleinement disponibles.

Il faut noter que la façon de formuler le code du programme a un fort impact sur certains des points considérés ci-dessus.

Nous allons ici nous préoccuper essentiellement du premier des points mentionnés. La fréquence d'horloge d'un processeur moderne est, depuis le milieu des années 2000, de l'ordre du Giga Hertz, c'est-à-dire 10^9 cycles par seconde. Cela signifie que la durée d'un cycle d'horloge est de l'ordre de la nanoseconde.

Comme nous l'avons vu précédemment, un programme peut être décomposé en une suite d'instructions « atomiques ». Si un programme est composé de n de ces instructions et que son temps d'exécution vaut T sur un processeur avec une fréquence d'horloge f, la relation $T = n/f$ n'est pas pour autant vérifiée, car rien ne dit qu'exactement une instruction est exécutée par cycle d'horloge. Comme on l'a vu dans les chapitres précédents, certaines instructions mettent de nombreux cycles a être exécutées, tandis que des techniques permettent de paralléliser des instructions de sorte à en réaliser potentiellement plusieurs par cycle. On peut donc écrire la relation :

$$T = \frac{n \cdot c}{f}, \tag{6.1}$$

où c s'interprète comme le **nombre moyen de cycles d'horloge par instruction**.

Outre le temps d'exécution, le nombre d'opérations en virgule flottante par seconde (**FLOPS** pour *floating point operations per second*) est souvent mentionné comme une mesure de la performance d'un ordinateur. Encore une fois, cette mesure ne donne qu'une idée approximative de la

performance d'une machine, car elle ne prend pas en compte tous les autres facteurs mentionnés plus haut. Par ailleurs, le nombre d'opérations en virgule flottante par seconde dépend du type d'opération effectuée, et peut varier de façon significative en fonction de la nature des calculs. Les processeurs modernes effectuent quelques opérations par cycle d'horloge, et peuvent donc en théorie atteindre des performances de l'ordre de plusieurs dizaines de GFLOPS.

En pratique, et surtout au sein de la communauté scientifiques, un certains nombres de tests (« benchmarks » en anglais) sont utilisés à titre d'étalon de la performance d'un ordinateur. Ces benchmarks sont des programmes standardisés qui effectuent des opérations courantes, telles que des opérations matricielles, des opérations d'entrée-sortie, etc.

6.3.2 « Loi » de Moore

Depuis les débuts de l'informatique jusqu'au début des années 2000, la fréquence d'horloge des processeurs a augmenté de façon exponentielle (phénomène lié à une conjecture connue sous le nom de **loi de Moore** [9]). Cependant, certaines limites physiques ont été atteintes par la suite. En effet, la miniaturisation des transistors et autres composants a, dans une forte mesure, permis ce gain de performance ; or, un degré de miniaturisation tel a été atteint que des problématiques de gestion de la chaleur et d'effets quantiques ne permettent plus, à l'heure actuelle, de miniaturiser significativement les processeurs davantage. En revanche, d'autres méthodes pour augmenter la performance des ordinateurs ont été développées, ayant principalement trait à l'architecture des processeurs et à la façon dont les programmes sont exécutés, notamment en exploitant la possible parallélisation des calculs. Si les calculs sont aisément parallélisables, cela signifie que les différentes unités de calcul de la machine peuvent collaborer pour mener à bien une tâche plus rapidement sur des données de taille fixe, ou alors de traiter de plus grandes quantités d'information en un temps donné ; si en revanche les calculs ne sont pas aisément parallélisables, il n'en reste pas moins que l'on peut effectuer plus de tâches indépendantes en parallèle. Les cartes graphiques sont un exemple de spécialisation de composants pour effectuer des calculs de façon hautement parallèle.

Ainsi, les graphiques montrant l'évolution de la fréquence d'horloge des processeurs en fonction du temps exhibe un changement de pente depuis les années 2000, tandis que ceux qui concernent l'évolution d'autres mesures de performance des ordinateurs en fonction du temps peuvent continuer de montrer une croissance exponentielle. En particulier, le nombre de transistors par puce a continué de croître de façon exponentielle grâce à l'adoption d'architecture multicoeurs. Les figures 6.2 et 6.3 montrent respectivement l'évolution de la fréquence des processeurs et du nombre de transistors par puce de processeur en fonction du temps.

9. Plus qu'une loi, il s'agit d'une conjecture émise par G. Moore dans les années soixante, estimant que le nombre de transistors au sein d'un microprocesseur d'un prix donné double tous les deux ans.

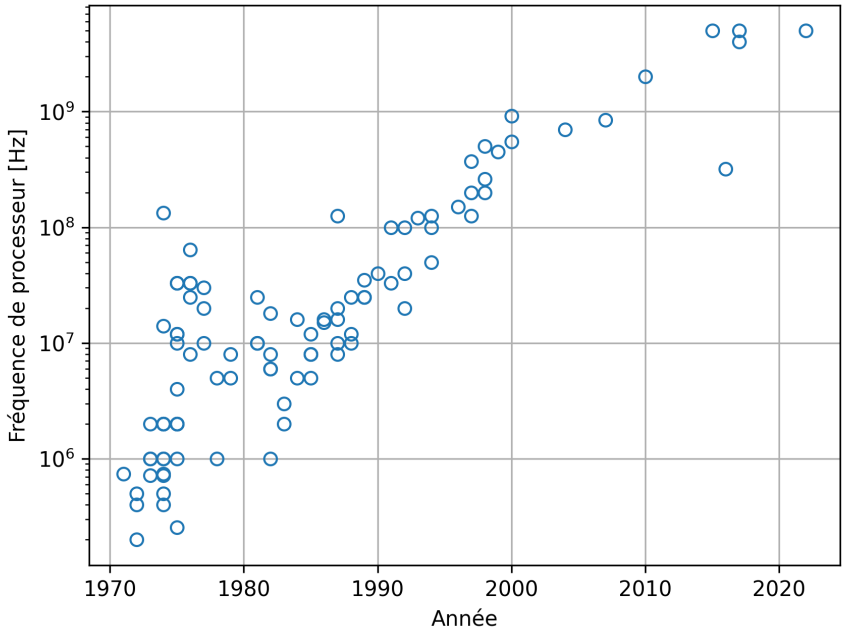


FIGURE 6.2 – Évolution temporelle de la fréquence des processeurs. Celle-ci est environ multipliée par 100 tous les 10 ans jusque dans les années 2000. Données issues de `https://en.wikipedia.org/wiki/Microprocessor_chronology`.

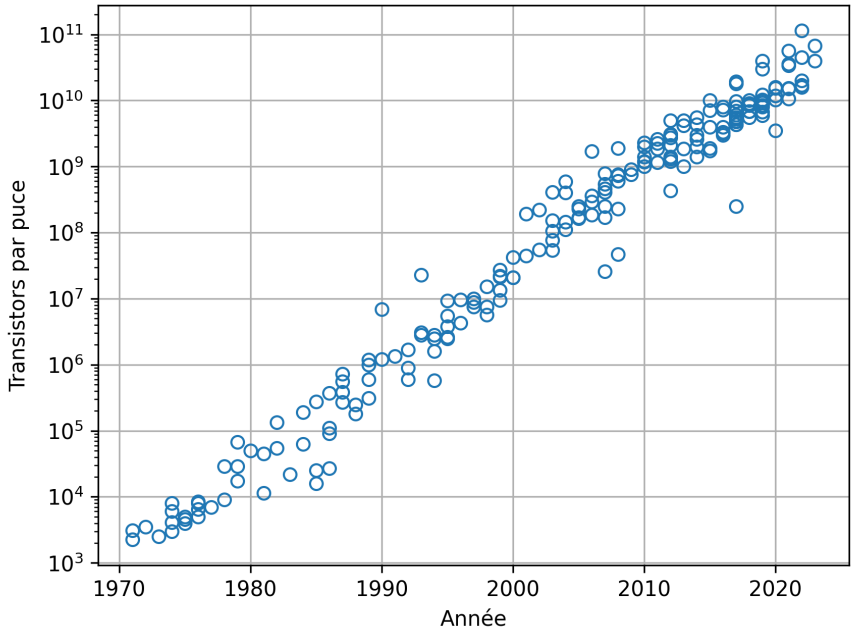


FIGURE 6.3 – Évolution temporelle du nombre de transistors par puce de processeur (y compris utilisation de processeurs multi-coeurs). Ce nombre est environ multiplié par 10 chaque décennie. Données issues de `https://en.wikipedia.org/wiki/Transistor_count`.

Exercices

Chapitre 2 : Codage des nombres

Systèmes de numération et écriture des entiers naturels

Exercice 2.0 | Conversion d'entiers naturels

Convertissez les nombres suivants en binaire :

1. 42
2. 123
3. 256
4. 1024

Exercice 2.1 | Conversion d'entiers naturels

Convertissez les nombres suivants en décimal :

1. 1010_2
2. 1101_2
3. 1111_2
4. 10000_2

Exercice 2.2 | Conversion d'entiers naturels

Convertissez les nombres suivants en hexadécimal :

1. 42
2. 123
3. 256
4. 1024
5. 1026

Exercice 2.3 | Conversion d'entiers naturels

Convertissez les nombres hexadécimaux suivants en décimal et en binaire :

1. $3A_{16}$
2. $7B_{16}$
3. FF_{16}
4. 400_{16}
5. $5A0B_{16}$

Exercice 2.4 | Conversion d'entiers naturels

1. Comment représenter 42 en base 7 ?
2. Comment représenter 43 en base 5 ?
3. Comment représenter 41 en base 9 ?
4. Comment représenter 42 en base 42 ?
5. Comment représenter 42_7 en base 10 ?
6. Comment représenter 0 en base n ?
7. Comment représenter 1 en base n ?
8. Comment représenter n en base n ?

Exercice 2.5 | Conversion d'entiers naturels

1. Quel est l'entier maximal que l'on peut exprimer avec k chiffres en base b ?
2. De combien de bits a-t-on besoin (au plus) pour écrire en binaire un entier décimal à k chiffres ?
3. De combien de chiffres hexadécimaux a-t-on besoin (au plus) pour écrire un nombre décimal à k chiffres ?

Exercice 2.6 | Codage de communes

Sachant qu'il existe 2'148 communes en Suisse et en supposant que chacune possède un code postal distinct, est-il adéquat d'utiliser un code postal de 4 chiffres décimaux pour encoder les communes suisses ? Aurait-on pu faire mieux avec le système décimal ?
En binaire, combien de bits sont nécessaires pour encoder toutes les communes ?
Et en hexadécimal ?

Codage des entiers relatifs

Exercice 2.7 | Codage d'entiers relatifs en complément à 2

Codez les nombres suivants en binaire sur un octet, en utilisant la convention du complément à 2 :

1. 42
2. -42
3. 0
4. 123
5. -123

Exercice 2.8 | Décodage d'entiers relatifs en complément à 2

Interprétez les états binaires suivants en nombres décimaux, sachant qu'ils ont été codés en utilisant la convention du complément à 2 :

1. 100
2. 0010
3. 1101
4. 1111
5. 10000

Exercice 2.9 | Addition d'entiers relatifs en complément à 2

Ci-dessous, des additions et soustractions entre nombres sont données en base 10. Effectuez ces opérations en binaire avec la méthode du complément à 2. Donnez la réponse finale après gestion du débordement : indiquez l'état binaire correspondant à la réponse ainsi que son interprétation en tant qu'entier relatif codé en complément à 2.

1. 7 + 7 (sur 5 bits)
2. 15 + 15 (sur 5 bits)
3. 7 - 15 (sur 5 bits)
4. (-2) - 15 (sur 5 bits)

Exercice 2.10 | Codage d'entiers relatifs avec la méthode du biais

Codez les nombres suivants en binaire avec $cod_{\mathbb{Z}B^+_{(8)}}$:

1. 42
2. 128
3. -127
4. -38

Exercice 2.11 | Décodage d'entiers relatifs avec la méthode du biais

Décodez les états binaires suivants à l'aide de $dec_{\mathbb{Z}B^+}$:

1. 101
2. 101111
3. 001111
4. 0000

Exercice 2.12 | Représentation des entiers relatifs pour une sonde spatiale

Une sonde spatiale destinée à fonctionner avec très peu d'énergie à disposition doit mesurer des températures à l'autre bout du système solaire. Les physiciens de la mission pouvant se contenter de nombres entiers, le résultat d'une mesure n'est donc pas un nombre réel. Sachant que la température minimale à mesurer est de -273°C et la température maximale de 200°C :

- Donnez le nombre de bits à utiliser pour pouvoir coder la plage de températures en complément à 2.
- Proposez un autre codage spécifique à cette sonde (pas nécessairement l'un de ceux vus en cours !) qui permette de coder les températures mesurées.

Exercice 2.13 | Exercice hors programme - Conversion par concaténation

Prouvez que si une base B et une base b sont telles que $B = b^k$ pour un certain entier k, alors la conversion d'un nombre de B en b peut se faire par concaténation des chiffres de B.
Cet exercice n'est pas corrigé.

Exercice 2.14 | Exercice hors programme - Programmation

Ecrivez un code dans le langage de votre choix qui implémente les fonctions $cod_{\mathbb{Z}_{(k)}}$ et $dec_{\mathbb{Z}_{(k)}}$ pour le complément à 2. La première de ces fonctions doit se nommer `cod_c2(k, n)` et retourne un string représentant l'état binaire codant le nombre n sur k bits. La seconde fonction doit se nommer `dec_c2(k, b)` et retourne l'entier correspondant à l'état binaire b codé en complément à 2 sur k bits.
Cet exercice n'est pas corrigé.

Codage des réels

Exercice 2.15 | Décodage de réel

Quel est le nombre réel en simple précision correspondant à la suite de bits suivante, où l'on a introduit des espaces pour des questions de lisibilité : 0 01111110 10000000000000000000000 ?

Exercice 2.16 | Décodage de réel

Quel est le nombre réel en simple précision correspondant à la suite de bits suivante, où l'on a introduit des espaces pour des questions de lisibilité : 1 00111110 10010000000000000000000 ?

Exercice 2.17 | Ecriture de nombre décimal en notation positionnelle

Comment se note le nombre décimal 6.25 en base 2 ? Attention : cet exercice ne porte sur aucun type de codage informatique, mais uniquement sur les systèmes de numération positionnels.

Exercice 2.18 | Ecriture de nombre décimal en notation positionnelle

Comment coder le nombre -6.25 en simple précision ?

Exercice 2.19 | Erreur d'arrondi

Supposons un format de virgule flottante fictif dans lequel la mantisse n'est codée qu'avec 5 bits (en commençant par 2^{-1} comme avec le standard IEEE754).
Avec ce format, quel est le nombre réel le plus proche de 0.1 que l'on peut coder ? Quelle est l'erreur d'arrondi ?

Chapitre 3 : Codage des médias

Exercice 3.0 |

Proposez une convention pour coder les lettres de l'alphabet suivant : A,B,C,x,y,z. Utilisez le moins de bits possible.

Exercice 3.1 |

Quel poids le texte suivant occupe-t-il en mémoire, sachant qu'on a créé un codage exprès pour qu'il pèse le moins possible ? `Bonjour tout le monde !`

Exercice 3.2 |

Un artiste a fait un dessin contenant cinq couleurs différentes. Sur combien de bits au minimum doit-on coder les couleurs de son illustration pour la représenter fidèlement sur un ordinateur ? Dans ce problème, le codage n'est pas un codage RGB.

Exercice 3.3 |

Une photo est codée au format RGB, où chaque couleur primaire est codée sur un octet. Sachant que la taille de l'image est de 1024 pixels de large et 800 pixels de haut, estimez

l'espace mémoire nécessaire pour la stocker. Donnez votre réponse en MB.

Exercice 3.4 |

L'codage RGB d'une photo permet d'exprimer 262'144 couleurs. Sachant que chaque couleur primaire est codée sur autant de bits que les autres, trouvez le nombre de bits dévolu à chaque couleur primaire.

Exercice 3.5 |

Un artiste fait des photos en nuances de rouge uniquement. Il veut pouvoir représenter au moins 1'024 nuances de rouges en tout. Proposez un codage de la couleur de ses photos de façon à minimiser le poids des images dans la mémoire.

Exercice 3.6 |

Dans une image RGBA, un quatrième octet est ajouté à la représentation de chaque pixel afin de spécifier l'opacité de la couleur qui y est représentée. Si la taille d'une image au format RGBA est de 500×500 pixels, quel espace occupe-t-elle dans la mémoire ?

Exercice 3.7 |

En utilisant un codage sur un nombre de bits minimum, quel est l'espace en mémoire occupé par l'image ci-dessous, composée de trois couleurs différentes ?

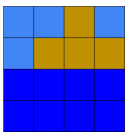


FIGURE 3.4 – *NB* : pour délimiter les pixels, on a dessiné une grille, mais celle-ci ne fait pas partie de l'image !

Exercice 3.8 |

Supposons que les images d'un film soient codées au format RGB avec 1 octet par couleur primaire. Le film est en haute résolution (aussi nommée « 4K ») : chaque image du film fait 3840 pixels de large et 2160 pixels de haut. De plus, le film est tourné en 30 images par seconde. La durée du film est de trois heures.

1. Quel est l'espace occupé par le film dans la mémoire ? Donnez votre réponse en GB (gigabytes ou gigaoctets), arrondi à deux décimales. *NB* : le préfixe "giga" signifie 10^9.
2. Pour comparaison, quelle est la mémoire persistante de votre téléphone ? Qu'en concluez-vous sur l'codage des films ?

Exercice 3.9 |

Un éditeur d'image n'accepte que le format hexadécimal pour les couleurs RGB.

1. Comment exprimer la couleur bleue dans ce format ?
2. Comment exprimer la couleur jaune dans ce format ?
3. Comment exprimer la couleur blanche dans ce format ?

Exercice 3.10 |

Convertissez les couleurs (format RGB, 1 octet par couleur primaire) d'un format à l'autre dans le tableau ci-dessous.

Exercice 3.11 |

Une image de 2000 pixels × 2000 pixels occupe un espace mémoire de 4 MB. Combien de couleurs différentes peuvent en théorie figurer sur cette image ? Attention : le codage n'est pas forcément au format RGB

Exercice 3.12 |

Une image bitmap codée en RGB avec un octet par couleur primaire occupe 3 MB en mémoire.

1. Donnez le nombre de pixels de l'image.
2. Combien de couleurs différentes possibles le codage utilisé permet-il ?
3. À présent, on veut réduire l'espace mémoire occupé par l'image en réduisant le nombre de couleurs. On veut que l'image n'occupe pas plus de 500 kB dans la mémoire. Sur combien de bits chaque couleur primaire doit-elle être codée maintenant ?
4. À la place du point précédent, on décide plutôt de réduire la définition de l'image (le nombre de pixels). Quelle doit être la définition utilisée pour que l'image n'occupe pas plus de 500 kB dans la mémoire, si on utilise un octet par couleur primaire ?

Chapitre 4 : Circuits logiques et algèbre de Boole

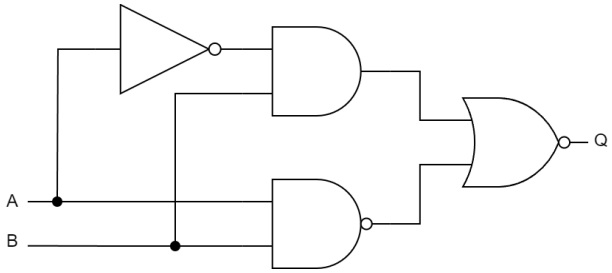
Exercice 4.0 | Expressions booléenennes

Simplifiez les expressions booléennes suivantes lorsque c'est possible :

1. $AB+!AB+BC$
2. $(A+!B)(A+B)$
3. $!(A+B)(A+!C)$
4. $AB+(!A!B)+(B!C)$
5. $(A+B)(!A+C)(B+C)$

Exercice 4.1 | De circuit à expression logique

Quelle est l'expression logique Q réalisée par le circuit ci-dessous ?

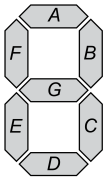


Exercice 4.2 | Multiplexeur

Dessinez un circuit logique combinatoire pour un multiplexeur à 2 voies.

Exercice 4.3 | Affichage à segments

L'affichage digital LCD (cristaux liquides) utilisant 7 segments, comme dans l'image de gauche ci-dessous, est très commun dans toutes sortes d'appareils électroniques : horloges, calculatrices, etc.



Pour réaliser ceci, on prédéfinit des segments comme sur l'image de droite ci-dessus, notés de A à G, puis chaque segment est allumé ou éteint pour former un chiffre. Par exemple, le chiffre 6 est formé en allumant tous les segments sauf le B.
Dans cet exercice, nous allons uniquement considérer la valeur que doit prendre le segment E en fonction de la valeur à afficher. Voici la liste des tâches à réaliser :

1. Donnez la table de vérité pour le segment E (1 pour « allumé », 0 pour « éteint ») en fonction de la valeur à afficher. Les inputs de la table de vérité sont donc les nombres de 0 à 9 exprimés sous forme binaire. On décide que les inputs ne faisant pas sens (par exemple 1111_2) doivent avoir une valeur d'output de 0.
2. Proposez une expression booléenne simplifiée pour le segment E en utilisant les mintermes, les maxtermes ou la méthode de Karnaugh.
3. Proposez un circuit logique correspondant à l'expression trouvée au point précédent en utilisant les portes logiques de votre choix parmi celles vues en cours.

Exercice 4.4 | Porte universelle

Donnez le diagramme logique d'un circuit qui réalise la table de vérité de P (table 4.12) en utilisant un seul type de porte logique.

Corrigés

Correction — Chapitre 2 : Codage des nombres

Correction — Systèmes de numération et écriture des entiers naturels

Exercice 2.0 | Conversion d'entiers naturels

Convertissez les nombres suivants en binaire :

1. $42 = 101010_2$
2. $123 = 1111011_2$
3. $256 = 100000000_2$
4. $1024 = 10000000000_2$

Exercice 2.1 | Conversion d'entiers naturels

Convertissez les nombres suivants en décimal :

1. $1010_2 = 10$
2. $1101_2 = 13$
3. $1111_2 = 15$
4. $10000_2 = 16$

Exercice 2.2 | Conversion d'entiers naturels

Convertissez les nombres suivants en hexadécimal :

1. $42 = 2A_{16}$
2. $123 = 7B_{16}$
3. $256 = 100_{16}$
4. $1024 = 400_{16}$
5. $1026 = 402_{16}$

Exercice 2.3 | Conversion d'entiers naturels

Convertissez les nombres hexadécimaux suivants en décimal et en binaire :
NB : pour le binaire, il est plus facile de convertir chiffre par chiffre depuis la base 16 que de passer par la base décimale.

1. $3A_{16} = 58_{10} = 111010_2$
2. $7B_{16} = 123_{10} = 1111011_2$
3. $FF_{16} = 255_{10} = 11111111_2$
4. $400_{16} = 1024_{10} = 10000000000_2$
5. $5A0B_{16} = 23051_{10} = 101101000001011_2$

Exercice 2.4 | Conversion d'entiers naturels

1. Comment représenter 42 en base 7 ? $42_{10} = 60_7$
2. Comment représenter 43 en base 5 ? $43_{10} = 133_5$
3. Comment représenter 41 en base 9 ? $41_{10} = 45_9$
4. Comment représenter 42 en base 42 ? $42_{42} = 10_{42}$
5. Comment représenter 42_7 en base 10 ? $42_7 = 30_{10}$
6. Comment représenter 0 en base n ? $0_{10} = 0_n$
7. Comment représenter 1 en base n ? $1_{10} = 1_n$
8. Comment représenter n en base n ? $n_{10} = 10_n$

Exercice 2.5 | Conversion d'entiers naturels

1. Quel est l'entier maximal que l'on peut exprimer avec k chiffres en base b ? En utilisant le même argument que dans le cours, on trouve $b^k - 1$
2. De combien de bits a-t-on besoin (au plus) pour écrire en binaire un entier décimal à k chiffres ? Le plus grand entier que l'on peut écrire avec k chiffres en décimal est $10^k - 1$. Il faut donc au plus $\lceil \log_2(10^k - 1) \rceil$ bits.
3. De combien de chiffres hexadécimaux a-t-on besoin (au plus) pour écrire un nombre décimal à k chiffres ? Le plus grand entier que l'on peut écrire avec k chiffres en décimal est $10^k - 1$. Notons que $\log_{16}(n) = \frac{\log_2(n)}{\log_2(16)} = \frac{\log_2(n)}{4}$. Il faut donc au plus $\lceil \frac{1}{4} \log_2(10^k - 1) \rceil$ chiffres hexadécimaux.

Exercice 2.6 | Codage de communes

Sachant qu'il existe 2'148 communes en Suisse et en supposant que chacune possède un code postal distinct, est-il adéquat d'utiliser un code postal de 4 chiffres décimaux pour encoder les communes suisses ? Aurait-on pu faire mieux avec le système décimal ?
En binaire, combien de bits sont nécessaires pour encoder toutes les communes ?
Et en hexadécimal ?
Décimal : on ne peut pas faire mieux que 4 chiffres. En effet, avec 3 chiffres on peut coder seulement 1000 communes différentes, tandis qu'avec 4 chiffres on peut en coder 10'000.
En binaire, on trouve par tâtonnement (où à l'aide du logarithme) qu'il faudrait 12 bits.
En hexadécimal, on trouve par tâtonnement (où à l'aide du logarithme) qu'il faudrait 3

chiffres.

Correction — Codage des entiers relatifs

Exercice 2.7 | Codage d'entiers relatifs en complément à 2

Codez les nombres suivants en binaire sur un octet, en utilisant la convention du complément à 2 :

1. 42 : $cod_{\mathbb{Z}_{(8)}}(42) = 00101010$
2. -42 : $cod_{\mathbb{Z}_{(8)}}(-42) = 11010110$
3. 0 : $cod_{\mathbb{Z}_{(8)}}(0) = 00000000$
4. 123 : $cod_{\mathbb{Z}_{(8)}}(123) = 01111011$
5. -123 : $cod_{\mathbb{Z}_{(8)}}(-123) = 10000101$

Exercice 2.8 | Décodage d'entiers relatifs en complément à 2

Interprétez les états binaires suivants en nombres décimaux, sachant qu'ils ont été codés en utilisant la convention du complément à 2 :

1. 100 : $dec_{\mathbb{Z}_{(3)}}(100) = dec_{\mathbb{N}_{(3)}}(100) - 2^3 = 4 - 8 = -4$
2. 0010 : $dec_{\mathbb{Z}_{(4)}}(0010) = dec_{\mathbb{N}_{(4)}}(0010) = 2$
3. 1101 : $dec_{\mathbb{Z}_{(4)}}(1101) = dec_{\mathbb{N}_{(4)}}(1101) - 2^4 = 13 - 16 = -3$
4. 1111 : $dec_{\mathbb{Z}_{(4)}}(1111) = dec_{\mathbb{N}_{(4)}}(1111) - 2^4 = 15 - 16 = -1$
5. 10000 : $dec_{\mathbb{Z}_{(5)}}(10000) = dec_{\mathbb{N}_{(5)}}(10000) - 2^5 = 16 - 32 = -16$

Exercice 2.9 | Addition d'entiers relatifs en complément à 2

Ci-dessous, des additions et soustractions entre nombres sont données en base 10. Effectuez ces opérations en binaire avec la méthode du complément à 2. Donnez la réponse finale après gestion du débordement : indiquez l'état binaire correspondant à la réponse ainsi que son interprétation en tant qu'entier relatif codé en complément à 2.

1. 7 + 7 (sur 5 bits) : Ici, on a $cod_{\mathbb{Z}_{(5)}}(7) = 00111$. Par conséquent, $cod_{\mathbb{Z}_{(5)}}(7) + cod_{\mathbb{Z}_{(5)}}(7) = 01110$, ce qui correspond à la valeur décimale 14.
2. 15 + 15 (sur 5 bits) : Ici, on a $cod_{\mathbb{Z}_{(5)}}(15) = 01111$. L'état binaire correspondant au résultat est $cod_{\mathbb{Z}_{(5)}}(15) + cod_{\mathbb{Z}_{(5)}}(15) = 11110$. Par conséquent, $dec_{\mathbb{Z}_{(5)}}(11110) = dec_{\mathbb{N}_{(5)}}(11110) - 2^5 = 30 - 32 = -2$ en décimal. On a débordé de la représentation externe maximale en complément à 2 sur 5 bits (qui en l'occurrence vaut $2^{5-1} - 1 = 15$).
3. 7 - 15 (sur 5 bits) : Ici, on a $cod_{\mathbb{Z}_{(5)}}(7) = 00111$ et $cod_{\mathbb{Z}_{(5)}}(-15) = 10001$. L'état binaire correspondant au résultat est $cod_{\mathbb{Z}_{(5)}}(7) + cod_{\mathbb{Z}_{(5)}}(-15) = 11000$. Par conséquent, $dec_{\mathbb{Z}_{(5)}}(11000) = dec_{\mathbb{N}_{(5)}}(11000) - 2^5 = 24 - 32 = -8$ en décimal.
4. (-2) - 15 (sur 5 bits) : Ici, on a $cod_{\mathbb{Z}_{(5)}}(-2) = 11110$ et $cod_{\mathbb{Z}_{(5)}}(-15) = 10001$. L'état binaire correspondant au résultat est $cod_{\mathbb{Z}_{(5)}}(-2) + cod_{\mathbb{Z}_{(8)}}(-15) = 01111$.

Par conséquent, $dec_{\mathbb{Z}_{(5)}}(01111) = dec_{\mathbb{N}_{(5)}}(01111) = 15$ en décimal. Il y a eu un débordement en représentation interne (on a dû ignorer le bit de retenue final).

Exercice 2.10 | Codage d'entiers relatifs avec la méthode du biais

Codez les nombres suivants en binaire avec $cod_{\mathbb{Z}B^+_{(8)}}$:

1. 42 : $cod_{\mathbb{Z}B^+_{(8)}}(42) = cod_{\mathbb{N}_{(8)}}(42 + 2^{8-1} - 1) = cod_{\mathbb{N}_{(8)}}(169) = 10101001$
2. 128 : $cod_{\mathbb{Z}B^+_{(8)}}(128) = cod_{\mathbb{N}_{(8)}}(128 + 2^{8-1} - 1) = cod_{\mathbb{N}_{(8)}}(255) = 11111111$
3. -127 : $cod_{\mathbb{Z}B^+_{(8)}}(-127) = cod_{\mathbb{N}_{(8)}}(-127 + 2^{8-1} - 1) = cod_{\mathbb{N}_{(8)}}(0) = 00000000$
4. -38 : $cod_{\mathbb{Z}B^+_{(8)}}(-38) = cod_{\mathbb{N}_{(8)}}(-38 + 2^{8-1} - 1) = cod_{\mathbb{N}_{(8)}}(89) = 01011001$

Exercice 2.11 | Décodage d'entiers relatifs avec la méthode du biais

Décodez les états binaires suivants à l'aide de $dec_{\mathbb{Z}B^+}$:

1. 101 : $dec_{\mathbb{Z}B^+_{(3)}}(101) = dec_{\mathbb{N}_{(3)}}(101) - 2^{3-1} + 1 = 2$
2. 101111 : $dec_{\mathbb{Z}B^+_{(6)}}(101111) = dec_{\mathbb{N}_{(6)}}(101111) - 2^{6-1} + 1 = 16$
3. 001111 : $dec_{\mathbb{Z}B^+_{(6)}}(001111) = dec_{\mathbb{N}_{(6)}}(001111) - 2^{6-1} + 1 = -16$
4. 0000 : $dec_{\mathbb{Z}B^+_{(4)}}(0000) = dec_{\mathbb{N}_{(4)}}(0000) - 2^{4-1} + 1 = -7$

Exercice 2.12 | Représentation des entiers relatifs pour une sonde spatiale

Une sonde spatiale destinée à fonctionner avec très peu d'énergie à disposition doit mesurer des températures à l'autre bout du système solaire. Les physiciens de la mission pouvant se contenter de nombres entiers, le résultat d'une mesure n'est donc pas un nombre réel. Sachant que la température minimale à mesurer est de -273°C et la température maximale de 200°C :

- Donnez le nombre de bits à utiliser pour pouvoir coder la plage de températures en complément à 2. En complément à deux sur k bits, on a :

  $$\mathbb{Z}_{(k)} = \{x \in \mathbb{Z} | -2^{k-1} \leq x < 2^{k-1}\}$$

  On trouve que $k = 9$ est la plus petite valeur telle que $-273 \in \mathbb{Z}_{(k)}$ et $200 \in \mathbb{Z}_{(k)}$.
- Proposez un autre codage spécifique à cette sonde (pas nécessairement l'un de ceux vus en cours !) qui permette de coder les températures mesurées. On observe qu'il y a 474 valeurs à coder. On peut donc utiliser un codage sur $\lceil \log_2(473) \rceil = 9$ bits. Le codage d'un nombre n peut par exemple s'écrire comme un biais :

  $$cod(n) = cod_{\mathbb{N}_{(9)}}(n + 273),$$

  et le décodage d'un état binaire b serait alors :

  $$dec(b) = dec_{\mathbb{N}_{(9)}}(b) - 273.$$

  Cela correspond approximativement à une représentation de la température en degrés Kelvin.

Exercice 2.13 | Exercice hors programme - Conversion par concaténation

Prouvez que si une base B et une base b sont telles que $B = b^k$ pour un certain entier k, alors la conversion d'un nombre de B en b peut se faire par concaténation des chiffres de B.
Cet exercice n'est pas corrigé.

Exercice 2.14 | Exercice hors programme - Programmation

Ecrivez un code dans le langage de votre choix qui implémente les fonctions $cod_{\mathbb{Z}_{(k)}}$ et $dec_{\mathbb{Z}_{(k)}}$ pour le complément à 2. La première de ces fonctions doit se nommer `cod_c2(k, n)` et retourne un string représentant l'état binaire codant le nombre n sur k bits. La seconde fonction doit se nommer `dec_c2(k, b)` et retourne l'entier correspondant à l'état binaire b codé en complément à 2 sur k bits.
Cet exercice n'est pas corrigé.

Correction — Codage des réels

Exercice 2.15 | Décodage de réel

Quel est le nombre réel en simple précision correspondant à la suite de bits suivante, où l'on a introduit des espaces pour des questions de lisibilité : 0 01111110 10000000000000000000000 ? Le signe vaut $s = (-1)^0 = 1$. L'exposant vaut $dec_{\mathbb{N}_{(8)}}(01111110) - 127 = 126 - 127 = -1$. La mantisse vaut $m = 1 + 2^{-1} = 1.5$. Le nombre réel correspondant est donc $n = +1.5 \cdot 2^{-1} = 0.75$.

Exercice 2.16 | Décodage de réel

Quel est le nombre réel en simple précision correspondant à la suite de bits suivante, où l'on a introduit des espaces pour des questions de lisibilité : 1 00111110 10010000000000000000000 ? Le signe vaut $s = (-1)^1 = -1$. L'exposant vaut $dec_{\mathbb{N}_{(8)}}(00111110) - 127 = 62 - 127 = -65$. La mantisse vaut $m = 1 + 2^{-1} + 2^{-4} = 1.5625$. Le nombre réel correspondant est donc $n = -1.5625 \cdot 2^{-65} \approx -4.235 \cdot 10^{-20}$.

Exercice 2.17 | Ecriture de nombre décimal en notation positionnelle

Comment se note le nombre décimal 6.25 en base 2 ? Attention : cet exercice ne porte sur aucun type de codage informatique, mais uniquement sur les systèmes de numération positionnels. $6.25 = 6 + 0.25 = 2^2 + 2^1 + 2^{-2} = 110.01_2$.

Exercice 2.18 | Ecriture de nombre décimal en notation positionnelle

Comment coder le nombre -6.25 en simple précision ? On a vu plus haut que $6.25 = 110.01_2$. Ici, le signe est négatif, donc le bit de signe vaut $b_s = 1$ afin que $(-1)^{b_s} = -1$.
Par ailleurs, on voit que l'exposant doit valoir 2 : en effet, $6.25 = 110.01_2 = 1.1001_2 \cdot 2^2$. On doit donc coder l'exposant biaisé 129, de sorte que $129 - 127 = 2$. Les 8 bits codant

l'exposant sont donc 10000001.
Enfin, avec le même raisonnement que pour l'exposant, on trouve que la mantisse est 1.1001_2, il faut donc coder les bits 1001 puis combler de zéros jusqu'à arriver à 23 bits : 10010000000000000000000.

Exercice 2.19 | Erreur d'arrondi

Supposons un format de virgule flottante fictif dans lequel la mantisse n'est codée qu'avec 5 bits (en commençant par 2^{-1} comme avec le standard IEEE754).
Avec ce format, quel est le nombre réel le plus proche de 0.1 que l'on peut coder ? Quelle est l'erreur d'arrondi ? Par tâtonnement, on parvient à : $0.1 \approx 1.10011_2 \cdot 2^{-4} = 0.099609375$. En supposant que le format fictif permette de coder correctement l'exposant, on a donc une erreur d'arrondi de $0.1 - 0.099609375 \approx 0.00039$.

Correction — Chapitre 3 : Codage des médias

Exercice 3.0 |

Proposez une convention pour coder les lettres de l'alphabet suivant : A,B,C,x,y,z. Utilisez le moins de bits possible.

code	valeur codée
000	A
001	B
010	C
011	x
100	y
101	z

Exercice 3.1 |

Quel poids le texte suivant occupe-t-il en mémoire, sachant qu'on a créé un codage exprès pour qu'il pèse le moins possible ? `Bonjour tout le monde !`
Le texte contient les caractères suivants : t ; j ; u ; r ; d ; B ; l ; e ; o ; m ; n ; <espace> ; ! Cela fait 13 caractères différentes. Il faut donc 4 bits pour coder cet alphabet. Par ailleurs, la longueur totale du texte est de 23 caractères. Par conséquent, le poids du texte est de $T = 23 \cdot 4$ b $= 92$ bits, donc 12 octets.

Exercice 3.2 |

Un artiste a fait un dessin contenant cinq couleurs différentes. Sur combien de bits au minimum doit-on coder les couleurs de son illustration pour la représenter fidèlement sur un ordinateur ? Dans ce problème, le codage n'est pas un codage RGB.

Il y a cinq valeurs différentes a représenter (cinq couleurs). Une méthode possible est de les faire correspondre aux nombres de 0 à 4. On définit la convention suivante :
$0 \rightarrow$ couleur1,
$1 \rightarrow$ couleur2,
etc.
Par conséquent, on a besoin d'au minimum trois bits pour représenter ces couleurs, car le plus grand nombre exprimable avec trois bits est $111_2 = 7$. Avec seulement deux bits, le plus grand nombre est $11_2 = 3$.
Rappel : avec n bits, on peut exprimer 2^n nombres différents.

Exercice 3.3 |

Une photo est codée au format RGB, où chaque couleur primaire est codée sur un octet. Sachant que la taille de l'image est de 1024 pixels de large et 800 pixels de haut, estimez l'espace mémoire nécessaire pour la stocker. Donnez votre réponse en MB.
Le nombre de pixels de l'image vaut :

$$N = 1024 \cdot 800 = 8.192 \cdot 10^5$$

À chacun de ces pixels sont associés trois octets : un octet par couleur primaire. Par conséquent, le nombre d'octets associé à l'image toute entière est :

$$T = 3 \cdot N = 3 \cdot 8.192 \cdot 10^5 = 2.4576 \cdot 10^6 \text{ octets} = 2.4576 \text{ MB}.$$

Exercice 3.4 |

L'codage RGB d'une photo permet d'exprimer 262'144 couleurs. Sachant que chaque couleur primaire est codée sur autant de bits que les autres, trouvez le nombre de bits dévolu à chaque couleur primaire.
Appelons n le nombre de bits dévolu à chaque couleur primaire de le codage RGB. Le nombre de couleurs N codables vaut alors

$$N = 2^n \cdot 2^n \cdot 2^n = 2^{3n}$$

Si $n = 6$, on a :

$$N = 2^{3 \cdot 6} = 262'144.$$

Exercice 3.5 |

Un artiste fait des photos en nuances de rouge uniquement. Il veut pouvoir représenter au moins 1'024 nuances de rouges en tout. Proposez un codage de la couleur de ses photos de façon à minimiser le poids des images dans la mémoire.
Comme il n'y a que des nuances de rouges, un codage RGB n'est pas nécessaire (ce serait du gaspillage de mémoire). On veut donc simplement pouvoir coder 1024 couleurs différentes (le fait que ce soient des nuances de rouge n'a pas d'importance !). On cherche donc un

nombre de bits n qui est tel que

$$2^n \geq 1024.$$

On s'aperçoit que cela fonctionne si $n = 10$. Finalement, le codage proposé est le suivant : la première nuance de rouge correspond au nombre 0 codé sur 10 bits, la seconde nuance de rouge correspond au nombre 1 codé sur 10 bits, etc. La dernière nuance de rouge correspond au nombre 1023 codé sur 10 bits.

Exercice 3.6 |

Dans une image RGBA, un quatrième octet est ajouté à la représentation de chaque pixel afin de spécifier l'opacité de la couleur qui y est représentée. Si la taille d'une image au format RGBA est de 500×500 pixels, quel espace occupe-t-elle dans la mémoire ?

La raisonnement est le même que pour l'exercice 2. Cependant, 4 octets sont associés à chaque pixel au lieu de 3. Par conséquent, on obtient

$$T = 4 \cdot 500 \cdot 500 = 10^6 = 1 \text{ MB}.$$

Exercice 3.7 |

En utilisant un codage sur un nombre de bits minimum, quel est l'espace en mémoire occupé par l'image ci-dessous, composée de trois couleurs différentes ?
Il y a 3 couleurs différentes, par conséquent on peut choisir un codage sur 2 bits au minimum (bien qu'une valeur soit « gaspillée » tout de même). Par ailleurs, l'image contient 16 pixels en tout. L'espace en mémoire vaut donc

$$T = 2 \cdot 16 = 32 \text{ bits}.$$

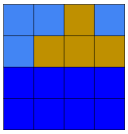


FIGURE 3.5 – *NB* : pour délimiter les pixels, on a dessiné une grille, mais celle-ci ne fait pas partie de l'image !

Exercice 3.8 |

Supposons que les images d'un film soient codées au format RGB avec 1 octet par couleur primaire. Le film est en haute résolution (aussi nommée « 4K ») : chaque image du film fait 3840 pixels de large et 2160 pixels de haut. De plus, le film est tourné en 30 images par seconde. La durée du film est de trois heures.

1. Quel est l'espace occupé par le film dans la mémoire ? Donnez votre réponse en GB (gigabytes ou gigaoctets), arrondi à deux décimales. *NB* : le préfixe "giga" signifie 10^9.
2. Pour comparaison, quelle est la mémoire persistante de votre téléphone ? Qu'en concluez-vous sur l'codage des films ?

On associe 3 octets à chaque pixel de chaque image du film. Chaque image contient $3840 \cdot 2160 = 8,2944 \cdot 10^6$ pixels. La taille d'une image vaut donc $T_1 = 3 \cdot 8,2944 \cdot 10^6 \approx 2,49 \cdot 10^7$ octets. Chaque seconde du film, 30 images d'une taille T_1 se succèdent à l'écran. Le film dure 3 heures, ce qui correspond à $3 \cdot 60 \cdot 60 = 10'800$ secondes. Comme 30 images se succèdent chaque seconde, le film contient au total $30 \cdot 10'800$ images $= 3.24 \cdot 10^5$ images. Par conséquent, au total, la taille du film vaut :

$$T = T_1 \cdot 3.24 \cdot 10^5 \approx 8,07 \cdot 10^{12} \text{ octets} = 8070 \text{ GB}.$$

La mémoire persistante d'un téléphone est généralement comprise entre 32 et 256 GB. En réalité, les films (et les images) utilisent un système de *compression* de l'information qui permet de réduire leur taille en mémoire, parfois au détriment de la qualité. Le son des films et musiques peut également être compressé, bien que nous n'en traitions pas dans cet exercice. Par exemple, le format de fichier `png` compresse les images, tandis que le format `bmp` ne le fait pas !

Exercice 3.9 |

Un éditeur d'image n'accepte que le format hexadécimal pour les couleurs RGB.

1. Comment exprimer la couleur bleue dans ce format ? Réponse : 0000FF.
2. Comment exprimer la couleur jaune dans ce format ? Réponse : FFFF00.
3. Comment exprimer la couleur blanche dans ce format ? Réponse : FFFFFF.

Exercice 3.10 |

Convertissez les couleurs (format RGB, 1 octet par couleur primaire) d'un format à l'autre dans le tableau ci-dessous.

Hexadécimal RGB	Décimal RGB
FF0000	(255, 0, 0)
7F3FFF	(127, 63, 255)
D00045	(208, 0, 69)
1010FF	(16, 16, 255)
C82020	(200, 32, 32)
40807F	(64, 128, 127)
708C28	(112, 140, 40)

Exercice 3.11 |

Une image de 2000 pixels × 2000 pixels occupe un espace mémoire de 4 MB. Combien de couleurs différentes peuvent en théorie figurer sur cette image ? Attention : le codage n'est pas forcément au format RGB

Le nombre total de pixels vaut $N = 2000 \cdot 2000 = 4 \cdot 10^6$ pixels. Si l'on divise l'espace mémoire occupé par l'image par le nombre de pixels qu'elle contient, on obtient la taille dévolue à chaque pixel individuellement. L'espace occupé par l'image vaut $4\ MB = 4 \cdot 10^6\ B$. Par conséquent, chaque pixel occupe un byte, ce qui vaut 8 bits. On peut donc représenter au maximum $2^8 = 256$ couleurs différentes sur cette image.

Exercice 3.12 |

Une image bitmap codée en RGB avec un octet par couleur primaire occupe 3 MB en mémoire.

1. Donnez le nombre de pixels de l'image.
2. Combien de couleurs différentes possibles le codage utilisé permet-il ?
3. À présent, on veut réduire l'espace mémoire occupé par l'image en réduisant le nombre de couleurs. On veut que l'image n'occupe pas plus de 500 kB dans la mémoire. Sur combien de bits chaque couleur primaire doit-elle être codée maintenant ?
4. À la place du point précédent, on décide plutôt de réduire la définition de l'image (le nombre de pixels). Quelle doit être la définition utilisée pour que l'image n'occupe pas plus de 500 kB dans la mémoire, si on utilise un octet par couleur primaire ?

1. L'espace en mémoire vaut 3 MB $= 3 \cdot 10^6$ B. Chaque pixel occupe 3 B de mémoire d'après la consigne (un octet par couleur primaire en RGB). Appelons x le nombre de pixels de l'image. On peut poser l'équation suivante :

$$x \cdot 3\ B = 3 \cdot 10^6\ B$$

   Donc $x = 10^6$ pixels.
2. Comme on vient de le voir, si l'on dispose d'un octet par couleur primaire, alors on peut exprimer $(2^8)^3 \approx 17$ millions de couleurs.
3. La condition se traduit par l'équation suivante, où l'on appelle n notre inconnue, qui est le nombre de bits par pixel :

$$n \cdot 10^6 = 500'000\ B = 4 \cdot 10^6\ b$$

   Par conséquent :

$$n = \frac{4 \cdot 10^6}{10^6} = 4 \text{ bits}$$

   On doit donc coder chaque couleur (chaque pixel) sur 4 bits. Cela signifie qu'on ne peut plus utiliser une convention RGB avec le même nombre de bits pour chaque couleur primaire !
4. La condition se traduit par l'équation suivante, où N est le nouveau nombre de pixels :

$$N \cdot 3\ B = 500'000\ B$$

   Par conséquent, la nouvelle image doit avoir au maximum $N = 500'000/3 \approx 167'000$ pixels.

Correction — Chapitre 4 : Circuits logiques et algèbre de Boole

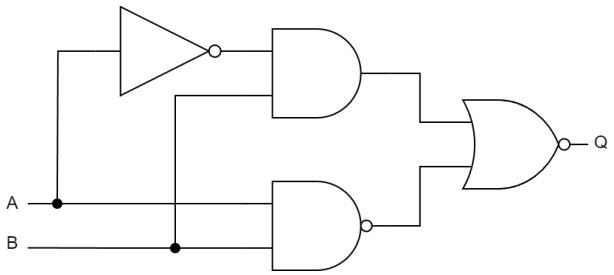
Exercice 4.0 | Expressions booléenennes

Simplifiez les expressions booléennes suivantes lorsque c'est possible :

1. $AB+!AB+BC = B(A+!A+C) = B(1+C) = B$
2. $(A+!B)(A+B) = AA+AB+A!B+\cancel{B!B} = A(1+B+!B) = A$
3. $!(A+B)(A+!C) = !A!B(A+!C) = \cancel{!A!BA}+!A!B!C = !A!B!C$
4. $AB+(!A!B)+(B!C) = B(A+!C)+(!A!B)$
5. $(A+B)(!A+C)(B+C) = (\cancel{A!A}+AC+B!A+BC)(B+C)$
   $= ACB+B!AB+BCB+ACC+B!AC+BCC$
   $= ACB+B!A+BC+AC+B!AC+BC$
   $= B(AC+!A+C+!AC+C)+AC$
   $= B(C(A+1+!A+1)+!A)+AC$
   $= B(C+!A)+AC$

Exercice 4.1 | De circuit à expression logique

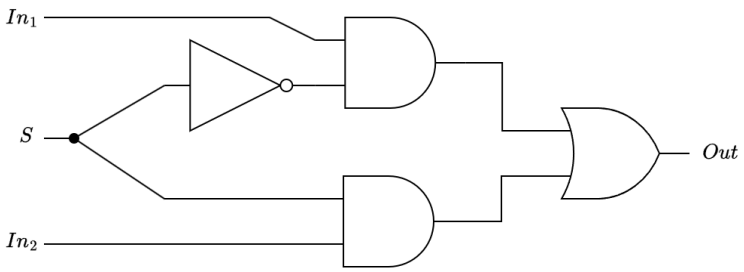
Quelle est l'expression logique Q réalisée par le circuit ci-dessous ?



$Q = !(!AB+!(AB)) = !(!AB)\cdot!!(AB) = (A+!B)\cdot AB = AB$

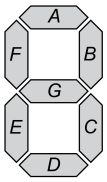
Exercice 4.2 | Multiplexeur

Dessinez un circuit logique combinatoire pour un multiplexeur à 2 voies. On a vu que $Out = In_1!S + In_2S$. Un circuit correspondant possible est le suivant :



Exercice 4.3 | Affichage à segments

L'affichage digital LCD (cristaux liquides) utilisant 7 segments, comme dans l'image de gauche ci-dessous, est très commun dans toutes sortes d'appareils électroniques : horloges, calculatrices, etc.



Pour réaliser ceci, on prédéfinit des segments comme sur l'image de droite ci-dessus, notés de A à G, puis chaque segment est allumé ou éteint pour former un chiffre. Par exemple, le chiffre 6 est formé en allumant tous les segments sauf le B.
Dans cet exercice, nous allons uniquement considérer la valeur que doit prendre le segment E en fonction de la valeur à afficher. Voici la liste des tâches à réaliser :

1. Donnez la table de vérité pour le segment E (1 pour « allumé », 0 pour « éteint ») en fonction de la valeur à afficher. Les inputs de la table de vérité sont donc les nombres de 0 à 9 exprimés sous forme binaire. On décide que les inputs ne faisant pas sens (par exemple 1111_2) doivent avoir une valeur d'output de 0.
2. Proposez une expression booléenne simplifiée pour le segment E en utilisant les mintermes, les maxtermes ou la méthode de Karnaugh.
3. Proposez un circuit logique correspondant à l'expression trouvée au point précédent en utilisant les portes logiques de votre choix parmi celles vues en cours.

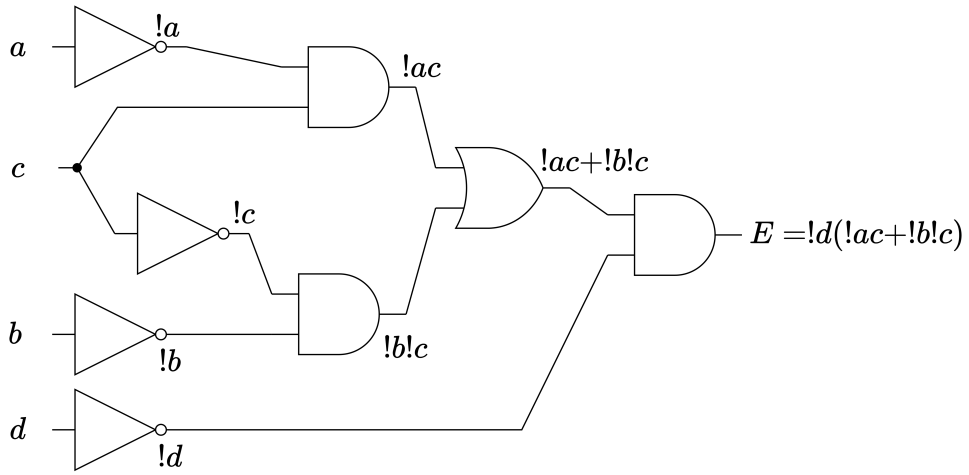
1. Nommons a, b, c, d les bits de l'input, respectivement de poids $2^3, 2^2, 2^1, 2^0$. La table de vérité est alors la suivante (en omettant les valeurs qui n'ont pas de sens, et en gardant en tête que nous appliquons la méthode des mintermes) :

a	b	c	d	E
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0

2. On peut essayer les mintermes ou les maxtermes, mais il n'est pas aisé de trouver les manipulations algébriques qui permettent de simplifier l'expression. On peut donc utiliser la méthode de Karnaugh. La table de Karnaugh est la suivante :

	$!c!d$	$!cd$	cd	$c!d$
$!a!b$	1	0	0	1
$!ab$	0	0	0	1
ab	0	0	0	0
$a!b$	1	0	0	0

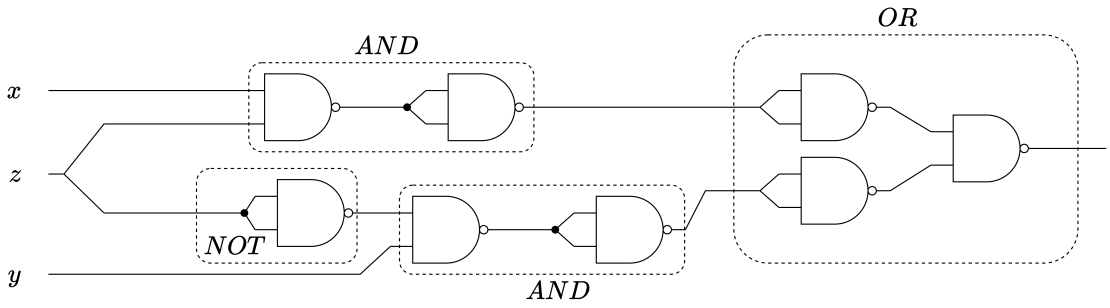
Deux groupes sont identifiés (le groupe en gris est périodique verticalement) et leur expression est $E = !b!c!d + !ac!d = !d(!b!c+!ac)$



Exercice 4.4 | Porte universelle

Donnez le diagramme logique d'un circuit qui réalise la table de vérité de P (table 4.12) en utilisant un seul type de porte logique.

Nous choisissons ici de représenter le diagramme de la figure 4.6 (lié aux mintermes) en utilisant uniquement des portes $NAND$. Ici, les relations 4.5, 4.6 and 4.8 nous sont utiles. Dans le diagramme ci-dessous, il est indiqué à proximité de chaque groupes de portes (en traitillés) la fonction logique qu'il remplit :



Annexe A

Pièges avec les entiers en C

En C, le type `size_t` est un type qui permet de représenter des tailles de variables. Le spécificateur de format `%zu` permet d'afficher des variables de type `size_t`. Le code suivant permet d'afficher les tailles de certains types de données en C.

```c
#include <stdio.h>

int main()
{
    printf("**** Entiers ****\n");
    printf("Taille d'un char: %zu octets\n", sizeof(char));
    printf("Taille d'un short: %zu octets\n", sizeof(short));
    printf("Taille d'un int: %zu octets\n", sizeof(int));
    printf("Taille d'un long: %zu octets\n", sizeof(long));
    printf("Taille d'un long long: %zu octets\n", sizeof(long long));

    printf("\n**** Non entiers ****\n");
    printf("Taille d'un float: %zu octets\n", sizeof(float));
    printf("Taille d'un double: %zu octets\n", sizeof(double));
    printf("Taille d'un long double: %zu octets\n", sizeof(long double));

    return 0;
}
```

Voici également quelques exemples quant au décodage des types numériques entiers. Il est à noter que ces exemples supposent le cas usuel rencontré sur les machines du cours : `unsigned char` codé sur 8 bits, `short` codé sur 16 bits, et représentation en complément à 2 pour les entiers signés. En toute rigueur, certains détails peuvent dépendre de l'implémentation du langage C et de la machine utilisée, mais cela sort du cadre de ce cours. Enfin, l'utilisation de spécificateurs de format (`hhd`, `hhu`, `hd`, `hu`, `hd`) permet d'afficher les variables de type entier en fonction de leur type, sans avoir à discuter davantage de conversions de type.

A.1 Dépassement d'entier non signé

```c
#include <stdio.h>
int main()
{
    unsigned char a = 255; //a est codé sur 8 bits
    printf("Valeur de 'a': %hhu\n", a);
    a = a + 1;
    printf("Nouvelle valeur de 'a': %hhu\n", a);
    return 0;
}
```

A.2 Interprétation d'un état binaire

```c
#include <stdio.h>

int main()
{
    // Même état binaire, interprété de deux façons différentes
    signed char y = -4; // en complément à 2 sur 8 bits : 11111100
    unsigned char z = (unsigned char)y; // toujours 11111100

    printf("y = %hhd\n", y);
    printf("z = %hhu\n", z);

    return 0;
}
```

Voici un second exemple :

```c
#include <stdio.h>

int main()
{
    // Exemple : on suppose short sur 16 bits
    printf("Taille d'un short : %zu octets.\n", sizeof(short));

    printf("2 - 7 comme short signé     : %hd\n", (short)(2 - 7));
    printf("2 - 7 comme short non signé : %hu\n", (unsigned short)(2 - 7));

    return 0;
}
```

A.3 Nombre sans opposé en complément à 2

Cet exemple illustre, dans le cas usuel du complément à 2 sur 8 bits, que la valeur la plus négative d'un type signé n'a pas toujours d'opposé représentable dans ce même type.

```c
#include <stdio.h>

int main()
{
    // Exemple pédagogique dépendant de l'implémentation :
    // sur les machines usuelles en complément à 2 sur 8 bits,
    // la valeur -128 n'a pas d'opposé représentable dans le même type.
    signed char x = -128;
    signed char y = -x;

    printf("x = %hhd\n", x);
    printf("y = %hhd\n", y);

    return 0;
}
```

Annexe B

Codage et décodage

Pour tout codage, on définit $cod(x)$ et $dec(x)$ comme des fonctions inverses sur leurs domaines respectifs : $cod(dec(b)) = b$ et $dec(cod(n)) = n$. Dès lors, si $cod(n) = c$, alors $dec(c) = dec(cod(n)) = n$.

Voici un exemple pour le codage d'un entier relatif en signe-norme. Pour rappel :

$$c \equiv cod_{\mathbb{Z}SN_{(k)}}(n) = \begin{cases} cod_{\mathbb{N}_{(k)}}(|n| + 2^{k-1}) & \text{si } n < 0, \\ cod_{\mathbb{N}_{(k)}}(n) & \text{sinon.} \end{cases} \tag{B.1}$$

On cherche à présent la transformation inverse $dec_{\mathbb{Z}SN_{(k)}}(c)$. Le cas où $n > 0$ est trivial ; en appliquant la définition, on obtient bien $dec_{\mathbb{Z}SN_{(k)}}(c) = dec_{\mathbb{N}_{(k)}}(c)$. Pour le cas où $n < 0$, on a :

$$cod_{\mathbb{Z}SN_{(k)}}(n) = cod_{\mathbb{N}_{(k)}}(|n| + 2^{k-1}) \tag{B.2}$$

$$\iff dec_{\mathbb{N}_{(k)}}(cod_{\mathbb{N}_{(k)}}(|n| + 2^{k-1})) = |n| + 2^{k-1} = -n + 2^{k-1} \tag{B.3}$$

$$\iff n = 2^{k-1} - dec_{\mathbb{N}_{(k)}}(c). \tag{B.4}$$

Comme par ailleurs $n = dec_{\mathbb{Z}SN_{(k)}}(cod_{\mathbb{Z}SN_{(k)}}(n)) = dec_{\mathbb{Z}SN_{(k)}}(c)$, on obtient bien la relation 2.24 du cours.

Voici un autre exemple basé sur le même principe, pour l'un des types de codage par biais :

$$cod_{\mathbb{Z}B^{+}_{(k)}}(n) = cod_{\mathbb{N}_{(k)}}(n + 2^{k-1} - 1) \equiv c. \tag{B.5}$$

$$\iff dec_{\mathbb{N}_{(k)}}(c) = dec_{\mathbb{N}_{(k)}}(cod_{\mathbb{N}_{(k)}}(n + 2^{k-1} - 1)) = n + 2^{k-1} - 1. \tag{B.6}$$

$$\iff n = dec_{\mathbb{N}_{(k)}}(c) - 2^{k-1} + 1 = dec_{\mathbb{Z}B^{+}_{(k)}}(c). \tag{B.7}$$

Annexe C

Erreurs d'arrondi : codes de démonstration

C.1 Représentation inexacte des flottants

Les nombres flottants ne sont stockés qu'avec une précision finie. Certains nombres décimaux très simples, comme 0.1 ne peuvent donc pas être représentés exactement en mémoire.

En C :

```c
#include <stdio.h>
int main()
{
float a = 0.1f;
// double a = 0.1; // a tester !
printf("Valeur : %.27f\n", a);
        return 0;
}
```

En Python :

```python
a = 0.1
print(f"{a:.27f}") #on demande d'afficher 27 décimales
```

C.2 Piège de la comparaison des flottants

Le code Python ci-dessous illustre le fait qu'un nombre en virgule flottante ne peut pas toujours être exactement codé. À la première lecture de cet exemple, on pourrait croire qu'il ne présente aucun problème particulier, et que le message de fin sera affiché. Or, ce n'est pas le cas, car l'incrément de 0.2 n'est pas exactement représenté au sein de la machine, et la variable val ne vaudra donc jamais exactement zéro malgré les soustractions successives (même s'il en est très proche).

```python
val = 1.0
while val != 0:
    val -= 0.2 # Essayez avec 0.25
print("Programme terminé.")
```

C.3 Accumulation des erreurs

Dans le code C ci-dessous, on peut observer l'écart entre deux façons de calculer une même quantité en virgule flottante : par additions successives, ou par une multiplication directe. Cet écart dépend fortement de l'incrément choisi. Lorsqu'un incrément est exactement représentable en binaire, cet écart peut être plus faible, voire nul dans certains cas, mais cela n'est pas garanti en général. Si l'incrément peut s'écrire sous la forme 2^{-n} et que le nombre de bits est suffisant pour le coder, alors l'erreur d'arrondi sera nulle.

```c
#include <stdio.h>
int main()
{
    const float delta = 0.2f; // increment (essayer 0.25f aussi!)
    const long N = 10000000; // nombre d'additions
    float a = 0.0f; // compteur
    const float theo_result = delta * N;
    printf("delta * N = %f * %ld = %f\n", delta, N, theo_result);
    for(long i=0; i<N; i++)
        a = a + delta;
    printf("delta + delta + ... (N fois) = %f\n", a);
}
```

C.4 Catastrophic cancellation : code de démonstration

Le code C ci-dessous illustre un exemple de catastrophic cancellation telle que décrite dans la section 2.5.2.

```c
#include <stdio.h>

int main() {
    float a = 1.000001f;
    float b = 1.000000f;
    float c = 0.000001f;
    float result = (a - b) / c;
    printf("Ceci devrait valoir 1: %.12f\n", result);
    return 0;
}
```

Voici une version équivalente en Python :

```
a = 1.000001
b = 1.000000
c = 0.000001
result = (a - b) / c #On pourrait croire que cela vaut 1
print(f"Ceci devrait valoir 1: {result:.12f}")
```

Annexe D

Epsilon et écart entre flottants

Comme nous l'avons vu, l'écart d'un flottant à son voisin vaut environ 2^{E-k_m}, où k_m est le nombre de bits de la mantisse et E l'exposant du nombre représenté. Ainsi, le rapport de cette quantité avec celle du nombre représenté vaut $2^{E-k_m}/2^E = 2^{-k_m}$.

Par ailleurs, il est courant de définir l'epsilon de la machine comme l'écart entre 1 et le flottant suivant. D'un point de vue pratique, on peut voir cette valeur comme celle qui, additionnée à l'unité, donne une valeur identique à ce dernier. L'approximation de l'epsilon d'une représentation en virgule flottante peut donc se faire grâce à la procédure suivante, sans même connaitre le nombre de bits dévolus à la mantisse : on débute en choisissant une valeur de n suffisamment grande, comme l'unité ; ensuite, on réduit n de moitié tant que la valeur de $1+n/2$ est distinguable de 1. En voici une implémentation en Python 3, grâce à laquelle le lecteur peut se convaincre que les flottants par défaut correspondent à la double précision :

```
import math #pour le calcul du log
n = 1
while 1 + n/2 != 1:
    n = n/2
print("Epsilon vaut environ", n)
print("Le nombre de bits de la mantisse vaut certainement", int(-math.log(n,2)))
```

Une nuance importante est à apporter : il est tentant de reformuler la condition de la boucle while comme `while n/2 != 0`, mais dans ce cas le nombre spécial zéro, dont nous avons discuté, entre en jeu. Sans les nombres spéciaux, la condition de la boucle ne serait jamais vraie puisque le zéro exact ne peut être représenté en notation scientifique en base 2 tout en utilisant le bit implicite de la mantisse ; ici, la boucle s'arrête donc en réalité au plus petit nombre dénormalisé avant zéro. Or, comme nous l'avons vu en section 2.5.2, le codage dénormalisé des nombres est un codage en virgule fixe, où l'exposant est fixé à $2^{1-E_{biais}}$, ce qui vaut 2^{-1022} pour la précision double. Le plus petit nombre dénormalisé vaut alors $0.00\ldots001_2 \cdot 2^{-1023} = 2^{-52-1022} \approx 10^{-324}$.

Annexe E

Ecritures de caractères dans un fichier

Le code ci-dessous écrit un message au sein d'un fichier. Chaque caractère est directement codé sous forme d'entier, en utilisant la fonction `fputc`.

```c
#include <stdio.h>
/*
Compilation : gcc bonjour_ascii.c -o bonjour_ascii
Execution : ./bonjour_ascii
*/
int main() {
    // On ouvre le fichier en mode binaire pour écrire.
    // Cela signifie que nous ne pouvons directement écrire
    // que des séquences d'octets.
    FILE *file = fopen("fichier.bin", "wb");
    if (file == NULL) {
        perror("Erreur lors de l'ouverture du fichier");
        return 1;
    }
    // La fonction fputc écrit un caractère dans un fichier
    // à partir de son code numérique.
    // Essayez de retrouver le message écrit grâce à une table ASCII.
    fputc(66, file);
    fputc(111, file);
    fputc(110, file);
    fputc(106, file);
    fputc(111, file);
    fputc(117, file);
    fputc(114, file);
    fclose(file);
    printf("Fini.\n");
    // Ouvrez le fichier 'fichier.bin' dans le bloc-notes maintenant.
    return 0;
}
```

Annexe F

Écriture d'un fichier MIDI

Le code ci-dessous utilise une librairie Python afin d'écrire un fichier MIDI sur le disque. Vous pouvez ensuite lire ce fichier avec un logiciel de lecture de fichiers MIDI.

```
from midiutil import MIDIFile

mf = MIDIFile(1)  # Une seule piste
mf.addTrackName(track=0, time=0, trackName="Piste 1")
mf.addTempo(track=0, time=0, tempo=120)

# Avec cette façon de faire, on peut 'dicter' le nom des notes à jouer
dict_notes = {"do": 60, "ré": 62, "mi": 64, "fa": 65,
              "sol": 67, "la": 69, "si": 71}
nom_notes = "do,do,do,ré,mi, ,ré, ,do,mi,ré,ré,do"
liste_notes_midi = [dict_notes.get(n, -1) for n in nom_notes.split(",")]

# Tandis que ci-dessous, on indique directement le code de chaque note
# liste_notes_midi = [60, 60, 60, 62, 64, -1, 62, -1, 60, 64, 62, 62, 60]

for i in range(len(liste_notes_midi)):
    note = liste_notes_midi[i]
    if note >= 0: # On a décidé de coder par -1 les silences
        mf.addNote(track=0,
                channel=0,
                pitch=note,
                time=i,
                duration=1,
                volume=100)
print("Terminé de coder les notes.")

filename = "ma_musique.mid"
with open(filename, 'wb') as outf:
    mf.writeFile(outf) # Ecriture du fichier sur le disque
```

Annexe G

Exemple de codage d'image matricielle

Le format PPM est un format d'image très simple, qui permet entre autres de stocker des images en couleurs. Nous donnons ici un exemple de codage d'une image matricielle comprenant très peu de pixels, afin de coder manuellement la couleur de chaque pixel.

Le code ci-dessous est à enregistrer dans un fichier avec l'extension `.ppm`. Pour visualiser une image ppm sous Windows, vous devrez probablement installer un logiciel tel que Gimp of IrfanView [1]. Sous Linux, vous pouvez utiliser le visualiseur d'image par défaut.

```
P3
# Le P3 signifie que l'image est décrite en ASCII,
# et que les couleurs sont au format RGB.
# On indique ensuite la taille de l'image, ici 4 colonnes et 4 lignes:
4 4
# On indique finalement la valeur maximale de chaque couleur primaire:
255

# Encodons maintenant chaque pixel !
# La première ligne sera entièrement rouge vif
255 0 0
255 0 0
255 0 0
255 0 0
# La seconde ligne sera entièrement vert vif
0 255 0
0 255 0
0 255 0
0 255 0
# La troisième ligne sera un dégradé de vert vif a vert sombre
0 255 0
0 187 0
0 118 0
0 50 0
```

1. Dans le cas d'IrfanView, il est probablement nécessaire de désactiver le *resampling*, qui lisse les couleurs de l'image et crée des dégradés indésirables dans notre cas. Pour ce faire, il faut se rendre dans le menu *view*, puis *display options*, puis il faut décocher *use resample*.

```
# La quatrième ligne sera un dégradé du noir au blanc.
0 0 0
85 85 85
170 170 170
255 255 255
```

Annexe H

Algèbre de Boole en mathématiques

Dans la littérature mathématique, on se réfère à « une algèbre de Boole » pour désigner un type de structure algébrique, c'est-à-dire un ensemble muni de deux opérations. Par raccourci de langage (comme dans ce document), on parle souvent d'« algèbre de Boole » pour désigner un cas particulier d'algèbre de Boole à deux éléments, que nous décrivons ci-dessous.

On peut définir une algèbre de Boole comme un ensemble pour lequel deux opérations binaires (souvent notées $+$ et $\cdot$) et une opération unaire (ici notée !) sont définies, et qui vérifie certaines propriétés : la commutativité et la distributivité des opérations, l'existence d'un élément neutre pour chacune des opérations, et l'existence d'un complément *via* l'opération unaire pour chaque élément. Dès lors, on retrouve également les propriétés d'associativité, d'idempotence, etc. D'autres choix de propriétés de base sont possibles.

Comme il est courant dans la littérature de voir les termes d'anneau et de corps également associés à l'algèbre de Boole, nous les résumons ici. Lorsque que l'opération $+$ représente un ou exclusif, une algèbre de Boole est un anneau particulier. Un anneau est un ensemble pour lequel deux opérations (souvent notées $+$ et $\cdot$) sont définies, et qui vérifie certaines propriétés : l'associativité des deux opérations, la commutativité de l'addition, la distributivité de la multiplication par rapport à l'addition, et l'existence d'un élément neutre pour les deux opérations. Enfin, chaque élément doit posséder un opposé : dans ce cas, l'opération d'addition doit être le ou exclusif, et non pas le ou inclusif utilisé en logique classique, de sorte que chaque élément est son propre opposé. Cet anneau correspond alors à la définition d'un « corps commutatif », dans notre cas à deux éléments.

Annexe I

Exemple de code assembleur avec accès à la mémoire centrale

Supposons que les mots de l'architecture soient d'une longueur de 32 bits, ce qui équivaut à 4 octets. Ainsi, en langage assembleur, chaque élément du tableau `a` est décalé de 4 unités de mémoire par rapport à son voisin précédent. L'élément numéro i du tableau est donc situé à l'adresse $4i$ par rapport à l'adresse de départ du tableau. Cette dernière est utilisée comme point de départ pour accéder aux éléments du tableau.

En assembleur MIPS, les mnémoniques `lw` et `sw` permettent respectivement de lire un mot de la mémoire centrale pour le déposer au sein d'un registre, et de lire un mot au sein d'un registre pour le déposer dans la mémoire centrale.

Supposons un bout de code C où un tableau `a` est déjà déclaré. On souhaite maintenant, pour l'exemple, additionner les éléments d'indices 6 et 3 et stocker le résultat dans l'élément d'indice 2. Le code C est donc du type :

```
a[2] = a [6] + a [3];
```

Pour la correspondance en assembleur MIPS, on supposera que `$s0` contient l'adresse du début du tableau. Le code assembleur correspondant au code C est alors du type :

```
lw $t0 , 24( $s0 ) #on lit a[6] en MC et on le stocke dans le registre $t0
lw $t1 , 12( $s0 ) #on lit a[3] en MC et on le stocke dans le registre $t1
add $t0 , $t0 , $t1 #on stocke $t0 + $t1 dans $t0
sw $t0 , 8( $s0 ) #on stocke le registre $t0 à l'emplacement a[2] en MC
```

Dans le code précédent, « MC » est l'acronyme de « mémoire centrale », et on a utilisé le fait que le sixième élément du tableau correspond à un décallage de $6 \times 4 = 24$ par rapport à l'adresse de départ du tableau. De même, le troisième élément correspond à un décallage de $3 \times 4 = 12$. Enfin, l'addresse du second élément correspond à un décallage de $2 \times 4 = 8$ octets.

Annexe J

Différentes formes de parallélisme

Nous comparons ici les différents termes employés tout au long du cours faisant référence à une forme ou une autre de parallélisme. Nous employons les termes anglais quand ce sont ceux qui sont le plus souvent utilisés dans les ressources en ligne en français.

Parallélisme : Forme de traitement de l'information qui consiste à exécuter plusieurs instructions en même temps, peu importe comment. Ici, « en même temps » est à comprendre avec une certaine granularité : ainsi, si deux tâches sont effectuées en alternance très rapidement, on peut dire qu'elles progressent, du moins à l'échelle humaine, de façon parallèle (même si, dans ce dernier cas, on parle plus volontiers de « concurrence »).

Processeur multicœur : Se réfère à un processeur qui contient plusieurs cœurs, c'est-à-dire des unités de calcul indépendantes **au sein de la même puce**. Les différents cœurs partagent habituellement certains niveaux de cache (en général, L3) ainsi que la mémoire centrale.

Multiprocessing : Se réfère au fait d'utiliser plusieurs **processeurs distincts** pour traiter de l'information en parallèle. Cette forme de parallélisation se rencontre fréquemment dans le domaine du calcul scientifique. Il est tout à fait envisageable d'utiliser plusieurs processeurs multicœurs : deux niveaux de parallélisme interviennent alors. Enfin, il faut noter que la façon dont les données sont partagées entre les processeurs conditionne grandement la façon de programmer et de mettre en œuvre le multiprocessing. Si les processeurs peuvent accéder à la même mémoire centrale, on parle de mémoire partagée. Si chaque processeur a sa propre mémoire centrale, on parle de mémoire distribuée (et dans ce cas, les processeurs doivent s'échanger les données grâce à une forme ou une autre de connexion).

Multitasking : Se réfère au fait de traiter plusieurs **programmes différents** en même temps. Cela peut se faire sur un seul processeur, en alternant les tâches à traiter (par exemple dans l'attente d'une réponse d'un périphérique), ou sur plusieurs processeurs. Le multitasking ne dépend que de l'OS et pas de la façon dont les programmes impliqués ont été conçus.

Multithreading : Se réfère *dans ce cours* au fait de traiter certaines parties **d'un même programme** en parallèle. Cela peut se faire sur un seul processeur, en alternant les tâches à traiter (par exemple dans l'attente d'une réponse d'un périphérique), ou sur plusieurs processeurs. Le multithreading requiert en général d'avoir été pensé et programmé en conséquence. Malheureusement, la signification de ce terme varie beaucoup en fonction des contextes et des sources.

Bibliographie

[1] Brendan Gregg. *Systems Performance : Enterprise and the Cloud*. Prentice Hall, oct 2013.

[2] Jonas Lätt. Principes de fonctionnement des ordinateurs. Cours universitaire, Université de Genève, 2023.

[3] N. Nisan and S. Schocken. *The Elements of Computing Systems : Building a Modern Computer from First Principles*. Prentice-Hall of India, 2005.

[4] David A. Patterson and John L. Hennessy. *Computer Organization and Design MIPS Edition : The Hardware/Software Interface*. Morgan Kaufmann, 5 edition, sep 2013.

[5] M. Sobieszczanski. Paul baran (1926-2011) pionnier des réseaux distribués. *Hermès, La Revue*, 2011/3(61) :221–225, 2011.