Introduction à l'informatique
Partie 1 – Fonctionnement des ordinateurs

Yann Thorimbert

Préface

Ce document a été rédigé en 2024 pour servir de support de cours aux étudiants du cours d'introduction à l'informatique de l'Université de Genève.

La première partie du cours, constituée par le présent document, 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 ponctuelles sont mentionnées au fil du texte, dans les notes de bas de page.

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 langage Python endosse ce rôle. Il est néanmoins important de noter que les concepts abordés sont valables pour tous les langages de programmation. Par ailleurs, les lecteurs trop peu familiers avec la programmation ne devraient pas voir leur compréhension générale entravée par ces exemples de code.

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. Afin de comprendre les chapitres plus techniques qui suivent, il est capital d'avoir une idée claire du paysage au sein duquel s'articulent les différents concepts qui interviennent tout au long du cours. En effet, ces concepts étant interdépendants, il n'est pas aisé de les aborder les uns après les autres ; pour cette raison, nous commençons 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 franchit 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 pleine intensité. 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.

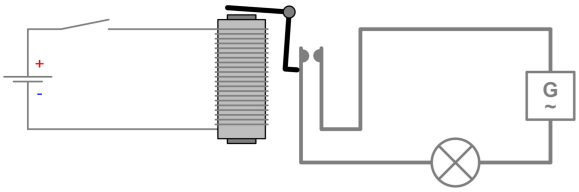


FIGURE 1.1 – Schéma d'un relais électromécanique lorsqu'aucun courant ne parvient à la bobine. Crédit : Cloullin

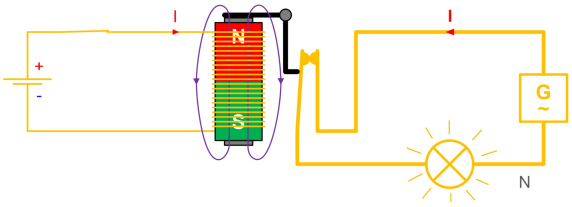


FIGURE 1.2 – Schéma d'un relais électromécanique lorsqu'un courant parvient à la bobine. Crédit : Cloullin

canisme. En l'occurence, ledit mécanisme répète le signal d'origine avec une nouvelle intensité, au sein d'un circuit électrique voisin. Après leur utilisation au sein des réseaux télégraphiques, ces relais ont également été utilisés dans les réseaux téléphoniques. Cependant, dans ce dernier cas, leur efficacité laissait à désirer. 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 une charge qu'elle contient. Plus la grille est chargée, 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 l'observation de la décoloration du verre des ampoules à incandescence. Au sein de ces dernières, le filament est chauffé à 2000 °C environ ; or, à cette température, si de l'oxygène est présent, une réaction de combustion peut apparaître. Par conséquent, les concepteurs des ampoules retirent l'air au sein des bulbes (et donc l'oxygène avec lui). On s'est alors aperçu d'une décoloration asynétrique et systématique du verre de ces ampoules à vide ; c'est ainsi que l'on s'est aperçu qu'un flux d'électrons était associé au filament chauffé par courant 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.

À partir de cette observation, on a construit des « ampoules à électrons » composées de la cathode chaude et de l'anode froide, ainsi que d'un simple vide les séparant. Avec un

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

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 (irrégulièrement « continu »). Ces « routes à sens unique pour électrons », comme elles ont été baptisées par leur inventeur John Fleming en 1904, portent le nom de **diode**[3].

Comme on va le voir, les diodes ont joué un rôle fondamental dans l'histoire de l'informatique et, par conséquent, dans l'histoire moderne de l'humanité. La figure 1.3 montre le symbole de la diode dans les circuits électriques et électroniques[4].

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, De Forest eu l'idée d'ajouter la grille chargée, 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[5]. 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.

1.2.2 Seconde implication : implémentation de la logique binaire

Si la conséquence immédiate de l'invention des diodes fut de nature technologique, une autre implication fut, dans un premier temps, de nature conceptuelle. Elle a pour

3. 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.

4. 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'occurence.

5. 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.

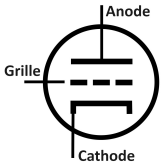


FIGURE 1.4 – Symbole de la triode au sein des circuits électroniques.

origine l'algèbre de Boole (ou logique binaire), un domaine des mathématiques qui permet d'exprimer des propositions de logique, et donc des raisonnements, sous forme algébrique. Pour ce faire, on utilise fréquemment des **tables de vérité**, qui servent à résumer tout raisonnement de façon exhaustive, et dont nous reparlerons plus tard. Ce qu'il faut ici retenir est qu'en algèbre de Boole, on choisit de travailler avec deux valeurs, arbitrairement dénotées 0 et 1, ou *faux* et *vrai*. Par exemple, 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 est 1 si le véhicule est en infraction, 0 sinon.

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[6], qui s'intéressait à l'algèbre de Boole vers 1938, eut l'idée d'une **équivalence entre circuits électroniques et tables de vérités**. Cette idée est fondamentale pour la suite de l'histoire de l'informatique, car elle permet de passer du monde des concepts mathématiques à celui de la réalité physique. Autrement dit : les triodes permettent d'implémenter dans le monde réel 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 véritable début 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 résoudre un problème grâce à un ordinateur 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 suite d'instructions et de conditions suivantes :

6. Aussi connu pour de nombreux autres travaux, notamment en théorie de l'information, et à l'origine du concept d'entropie cher aux physiciens également.

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 relais. Notamment, il a besoin d'une mémoire indépendante des données concernées par la table de vérité, afin de se souvenir des instructions à exécuter. 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 permet d'appliquer un algorithme sur des données, à supposer qu'elle ait assez de temps et de mémoire pour cela.

Nous dirons d'une machine que c'est un ordinateur électronique si elle utilise des composants électroniques pour effectuer les calculs 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 nous penchons ici sur la première à avoir vraisemblablement rempli tous les critères.

1.4 Le premier ordinateur complet

La première machine électronique à être considérée comme un ordinateur programmable capable d'effectuer en théorie n'importe quel calcul qui peut être décrit algorithmiquement est l'ENIAC[7], construite en 1946. Cette machine, qui pesait 30 tonnes et occupait 140 mètres carrés, a initialement été construite pour des applications militaires. Elle était

7. Acronyme de Electronic Numerical Integrator And Computer ; en effet, il s'agissait initialement d'une machine aidant à intégrer les équations du mouvement de problèmes de balistique.

composée de dizaines de milliers de tubes à vide et permettait en pratique d'effectuer environ un millier d'opérations décimales par seconde[8]. C'est entre autres « grâce » à cette capacité de calcul que la bombe H a pu être mise au point efficacement par les USA. C'était une machine extrêmement coûteuse, qui nécessitait une équipe de plusieurs personnes pour fonctionner. Il a été rapporté que sa plus longue utilisation sans interruption était de 116 heures. La figure 1.5 ci-dessous illustre une partie de l'ordinateur.

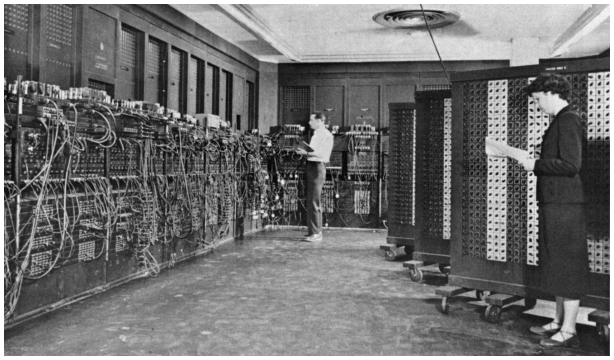


FIGURE 1.5 – Une programmeuse et un programmeur au travail sur ENIAC.

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 ! C'est le début de l'architecture de von Neumann, qui est encore utilisée aujourd'hui, et dont nous reparlerons dans le chapitre 5. Par ailleurs, contrairement à son prédecesseur, l'EDVAC travaille avec une représentation binaire des données (cf. 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 purement commerciales[9], le Z3 était une machine programmable basée sur le système de numération binaire. 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 code les nombres en binaire. Cependant, l'ABC 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) - L'IBM ASCC Mark I de l'université de Harvard était une machine programmable électromécanique, tout comme le Zuse 3. Elle était réputée fiable

8. En théorie, ENIAC pouvait effectuer 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.

9. Contrairement à nombre de ses contemporains, Konrad Zuse ne publiait pas le fruit de ses recherches au sein de journaux scientifiques, d'où son relatif manque de célébrité.

et opérait de façon autonome, nécessitant très peu d'interventions humaines. Le programme et les données étaient séparés sur des bandes perforées distinctes.

- Colossus (1943) - Rendue célèbre pour son rôle – secret à l'époque – consistant à déchiffrer les messages codés de l'armée allemande (*via* l'codeur Enigma notamment), 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.

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'est souvent le cas à l'époque. Cependant, au cours de la conception de cette machine, Babbage a eu 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. Cependant, de nombreuses réflexions sont déjà menées au sujet de l'algorithmique et de la programmation, notamment avec la collaboration d'Ada Lovelace, qui est souvent considérée comme la première programmeuse de l'histoire [10].

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

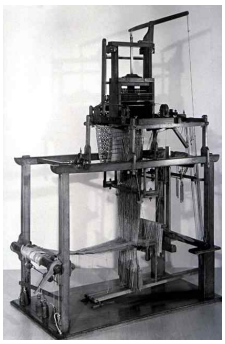


FIGURE 1.7 – Un métier Jacquard ancien, dispositif ayant utilisé des cartes perforées à un niveau industriel dès le début du dix-neuvième siècle.

10. Un célèbre langage de programmation nommé Ada s'appelle ainsi en son hommage.

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 [11]. La figure 1.8 ci-dessous illustre la différence de taille entre ces deux types de relais au moment de la popularisation des transistors.

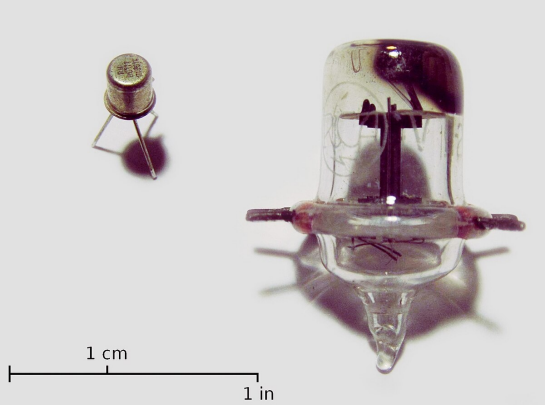


FIGURE 1.8 – Un transistor (à gauche) comparé à un tube à vide (à droite) dans les années 1950-1960.

Les transistors sont des composants électroniques qui, tout comme les triodes, permettent de contrôler le courant entre deux électrodes en fonction d'un courant de commande appliqué à une troisième électrode. 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 énormément pour des raisons physiques et techniques. La figure 1.9 illustre le symbole d'un transistor dans les circuits électroniques.

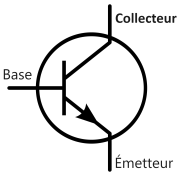


FIGURE 1.9 – Symbole du transistor au sein des circuits électroniques.

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

11. 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 de désigner les récepteurs radios par le terme « transistor ».

La figure 1.10 illustre la famille des relais. 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 (transistors) sont les plus efficaces pour cette tâche en terme de rapidité, fiabilité et miniaturisation. C'est pourquoi ils sont les plus utilisés dans les ordinateurs modernes. Mais si, demain, une nouvelle technologie permettait de réaliser la même fonction de relais 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.

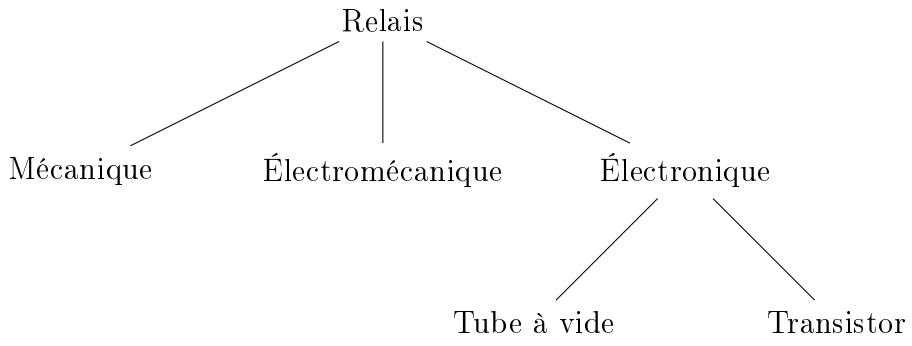


FIGURE 1.10 – Famille des relais. Seuls les types de composants discutés dans ce document sont représentés.

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 s'avèrent hautement miniaturisables. En effet, les câblages utilisés dans les décennies précédentes vont vite s'avérer fastidieux en comparaison de « pistes » conductrices gravées sur une plaque isolante, suivant une idée proposée en 1958 par un ingénieur de Texas Instruments. Cette technique dite des **circuits intégrés** permet de produire des combinaisons de composants à la fois plus fiables et davantages miniaturisables. Il est commun de désigner ces circuits intégrés par leur nom anglais de **chipsets**, ou encore **puces électroniques**. Un exemple de circuit intégré est montré sur la figure 1.11.

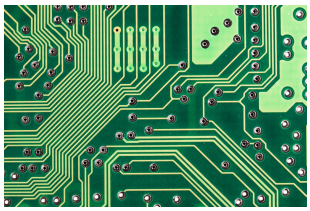


FIGURE 1.11 – Partie d'un circuit intégré où l'on peut voir des pistes conductrices gravées sur une plaque isolante.

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 spécifiques 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. On nommera ce type de circuit intégré un **microprocesseur** et, par conséquent, les ordinateurs les utilisant 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 poussé de recherche académique, industrielle ou militaire. La figure 1.12 illustre l'Altair 8800, un des premiers ordinateurs personnels à utiliser un microprocesseur.

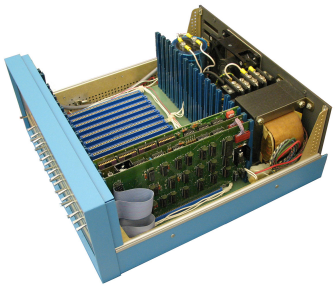


FIGURE 1.12 – Intérieur de l'Altair 8800, souvent considéré comme le premier micro-ordinateur grand public.

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 la façon dont ces composants sont combinés.

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

- Les dispositifs non-fondamentaux (technologies des écrans, des réseaux, des mémoires persistantes, cartes graphiques...)
- Les technologies liées à l'utilisation des ordinateurs (batteries pour ordinateurs portables, ...)
- La performance des ordinateurs 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 la 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 (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, à son tour, a permis d'engendrer un système de télécommunication particulièrement performant : l'internet.

Les premiers réseaux 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. Cependant, la façon de communiquer via les lignes téléphoniques, où la voix des interlocuteurs est transmise en continu au travers d'un chemin physique constant (technique nommée « commutation de circuits »), 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 est de découper ces 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.

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 proprement dites. 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 les paquets puissent être traités, il faut nécessairement qu'un ensemble de règles régisse la façon dont ils sont acheminés d'un point à un autre, la façon d'interpréter les en-têtes, etc. Cet ensemble de règles porte le nom de **protocole**. Le protocole le plus connu est le protocole IP (Internet Protocol), qui est à la base de l'internet. Il est souvent associé à un autre protocole, le protocole TCP (Transmission Control Protocol), qui permet de garantir que les paquets arrivent à destination dans l'ordre et sans erreur. Les deux pris ensemble forment le protocole TCP/IP, mis en place dès les années 1980 au sein d'ARPANET. Grâce à cela, on peut abstraire la technologie de transmission sous-jacente, qu'il s'agisse de lignes téléphoniques, de fibres optiques, de satellites, etc ; ce qui compte, c'est que les destinataires et les expéditeurs respectent des règles communes. Cela ouvre la porte à la communication entre machines issues de réseaux quelconques, et donc la réalisation de communications inter-réseaux. 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, laboratoires de recherche, etc. Dès les années 1980, le protocole SMTP permet de réguler les emails, mais jusqu'en 1990 l'internet reste un outil dévolu aux experts.

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 locale, utilisée pour être discriminée au sein du réseau local (LAN, Local Area Networ) 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 se nomme le **routage** de l'information.

World Wide Web et popularisation de l'internet

La dernière décennie du siècle, cependant, voit la mise en place du protocole HTTP, qui est central dans le développement du World Wide Web, un ensemble de règles qui permettent de grandement faciliter l'utilisation de l'internet. Le protocole HTTP (HyperText Transfer Protocol) régit l'envoi de données relatives à un type de document spécifique (page web) regroupés en ensembles appelés « site web », qui sont ensuite affichées par des navigateurs web. Ces derniers, tels que Chrome, Firefox ou Safari par exemple, sont des logiciels qui gèrent à la place de l'utilisateur les technicités de l'utilisation des protocoles, tout en proposant une visualisation des données hypertext ainsi qu'un moyen de naviguer entre elles.

C'est l'arrivée du 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. Les 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. Les navigateurs web, tels que Chrome, Firefox, Safari, etc, 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 allait de pair avec les pages web, comme le JavaScript notamment (cf. chapitre 6.1). Ces codes sont destinés à 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 est 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 qui les demandent. Les clients formulent ces demandent 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 le protocole le plus couramment utilisé pour les requêtes et les réponses dans le cadre du web, mais le modèle client-serveur est utilisé dans bien d'autres services que le web.

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, tandis que le client peut 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 est particulière en regard de la programmation de logiciels destinés à être exécutés localement.

Concernant le web spécifiquement, un ensemble de serveurs nommés « DNS » (Domain Name Service) est utilisé pour traduire les noms de domaines (comme `www.google.com`) en adresses IP, qui sont les adresses numériques des serveurs. En effet, les ordinateurs ne peuvent communiquer qu'avec des adresses IP, 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 DNS est donc un serveur qui permet de faire la correspondance entre les deux. L'alias textuel qui correspond à une page web et un protocole donnés se nomme l'URL (Uniform Resource Locator), et elle comprend 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 est une catégorisation des 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, responsbale du chemin (routage) 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 des 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 réseaux à haut débit. Ces lignes, souvent sous-marines, relient les différents continents de la planète entre eux. Les données sont transmises sous forme de lumière, et sont 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 faite au sein d'un réseau local ou 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 relais 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 « chiffres » répartis sur un nombre fini de « cases » de la mémoire de l'ordinateur. Ces quantités sont dites **digitales** [12], ce qui désigne le fait que l'information soit découpée en paliers discrets, 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 (car il faudrait une précision infinie pour exprimer le résultat exact), la mesure d'une quantité digitale livre un résultat exact.

Dans les deux prochains chapitres, dévolus au codage de l'information, 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 leur manipulation via des 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, 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 des calculs sur ordinateur peuvent être **reproductibles**, tandis que des calculs analogique ne peuvent l'être, par essence.

1.7.2 Le cas des calculateurs analogiques

Pour terminer, notons que certains calculateurs analogiques ont existé et font toujours l'objet de recherches. 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 1.13 ci-dessous illustre une reconstitution de cette machine. Plus récemment, des calculateurs

12. 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 1.13 – Une proposition de reconstitution du mécanisme d'Anticythère en 2007. Auteur : I. Mogi, licence CC BY 2.5

analogiques ont été utilisés pour résoudre des équations différentielles[13], des problèmes de contrôle, ou encore pour simuler des systèmes physiques. Ces calculateurs ont put se montrer plus rapides que leurs homologues numériques, mais sont également plus coûteux et moins flexibles. Ils sont donc utilisés dans des cas spécifiques où la rapidité est un facteur critique, mais où la flexibilité n'est pas nécessaire.

13. On notera 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.

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 divers, telles que des nombres, des lettres, des images, des vidéos, des sons, etc. En réalité, cette question cache deux problèmes de natures différentes, que nous allons aborder dans ce chapitre :

1. Comment **exprimer** un nombre non-binaire (base 2) au sein de la base 10 et *vice-versa* ?
2. Comment **coder** une donnée non-numérique (par exemple, une lettre ou un son) en une donnée numérique et *vice-versa* ?

2.1 Systèmes de numération

Au cours des millénaires, plusieurs façons d'écrire les nombres ont été développées à travers 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 échelonnées différents 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

Les systèmes de numération non positionnels 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 additionnels ; pour cette même raison, la position de chaque symbole au sein du nombre n'a pas d'importance [1]. Ces « symboles » qui servent à exprimer un **nombre** sont appelés des **chiffres**. Cette terminologie a une importance pour désigner sans ambiguité ce dont nous parlons [2], ce qui est particulièrement utile dans le cas des systèmes de numération positionnels que nous verrons plus loin.

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 letttre (« a » par exemple), certaines quantités peuvent être exprimés à l'aide d'un seul chiffre.

Par exemple, 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 dans n'importe quel ordre. Il est important de voir qu'en plus de cet ordre quelconque, les chiffres à 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.

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 systèmes. 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 bien supérieurs, sont devenus les systèmes de numération les plus utilisés dans le monde moderne.

L'étude des systèmes de numération dans le cas général sort du cadre de ce cours. Par exemple, il est possible de s'intéresser aux systèmes de numération à base non-entière.

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.

2. Par exemple, nous allons souvent devoir dire des choses comme « quel est le nombre de chiffres au sein de ce nombre ? ». Pour que de telles phrases aient du sens, il faut être rigoureux au niveau du vocabulaire.

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 chiffres qui le composent. Quand on exprime un entier, la colonne la plus à droite porte le numéro 0. Chaque chiffre du nombre est un entier compris entre 0 et 9. 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 $a_{k-1} \neq 0$.

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 $42.03 = 4 \cdot 10^1 + 2 \cdot 10^0 + 0 \cdot 10^{-1} + 3 \cdot 10^{-2}$. 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 » au quotidien.

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 $a_{k-1} \neq 0$.

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.

Base 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 a une valeur qui dépend de sa position i dans le nombre. Le chiffre en position i a une 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 est un entier compris entre la valeur nulle et $b-1$. Un nombre n en base b s'exprime alors de la façon suivante :

$$n_b = \sum_{i=0}^{k-1} a_i \cdot b^i, \tag{2.4}$$

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 $a_{k-1} \neq 0$.

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.

Enfin, les nombres non entiers s'expriment de façon analogue à ce qui a été vu pour le système décimal, c'est-à-dire en étendant la combinaison des termes $a_i \cdot b^i$ à des valeurs de i négatives :

$$n_b = \sum_{i=-v}^{k-1} a_i \cdot b^i, \tag{2.5}$$

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

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. Diviser le nombre par b et noter le reste de la division entière (notée $\lfloor n \div b \rfloor$ et désignant l'arrondi vers le bas).
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.

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 systématique de conversion du nombre 547_{10} vers la base 2 :

- 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 notation positionnel nous pousse à 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. 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, dans le cas des entiers, on cherche à chaque étape i un nombre a tel que $a \cdot b^i \leq n$, ce qui garantit que $a \leq n/b^i$.

Autrement dit, toujours pour les entiers, il existe une étape i telle que $n/b^i < 1$, ce qui clôt le processus de conversion. Pour la partie fractionnaire en revanche, on cherche à chaque étape i un nombre a tel que $a \cdot b^{-i} \leq n$. Par conséquent, $a \leq n/b^{-i}$. Ceci est problématique car, b^{-i} étant inférieur à l'unité, nous ne pouvons avoir de garantie que le développement des chiffres après la virgule se termine.

Ainsi, la conversion de la base décimale vers une base quelconque dépend le plus souvent d'un niveau de précision arbitraire que nous souhaitons. 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 celui 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.

2.1.4 Additions en binaire

La méthode manuscrite classique pour l'addition de nombres décimaux s'applique de la même façon pour les nombres binaires (et, en fait, 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'une retenue est également présente) 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, comme dans la méthode décimale. En l'occurrence, le résultat 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 n'ont pas 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 la valeur du nombre.

2.1.5 Soustractions en binaire

La méthode manuscrite classique pour la soustraction de nombres décimaux implique un nombre d'opérations et une complexité bien 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. 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 :

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

ce qui signifie que le plus petit chiffre non nul possible, ici noté a_0, participe à la somme *via* le facteur 2^{0+m} et 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 de 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. Pour résumer, 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. En effet, 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 :

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

Encore une fois, l'analogie avec toute autre base entière est possible. Par exemple $4781 \div 10^3 = 4$. Diviser un nombre binaire par 2^m revient donc à décaler tous ses chiffres de m colonnes vers la droite, en perdant les chiffres qui dépassent la colonne des unités.

Il y a ici une nuance à apporter malgré tout. Dans ce cas de figure, 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].

2.2.1 Bits, octets et mots

En binaire, les chiffres sont souvent appelés des **bits**[4]. Pour des raisons pratiques et technologiques, au sein des ordinateurs les bits sont communément regroupés par paquets de 8 nommés **octets** en français ou **bytes** en anglais. Enfin, un **mot** est un groupe de bits d'une taille qui est spécifique à un type de processeur. Le tableau 2.1 ci-dessous résume les termes que nous venons de voir. Comme nous le verrons plus loin, les ordinateurs

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	B	8 bits
byte	B	8 bits
mot de longueur k		k bits

travaillent avec des mots entiers plutôt qu'avec des bits isolés. Ces mots sont en général des multiples d'un octet. Comme cette quantité est une constante pour un type d'ordinateur donné, on dit de cet ordinateur qu'il a une **architecture** utilisant des mots de n bits.

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 ».

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.

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. Il est à noter que ces quantités d'informations sont souvent précédées d'un préfixe indiquant une puissance de 1000 (kilo (k), méga (M), giga (G), téra (T), péta (P), etc.) pour désigner des quantités plus grandes [5].

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 représentent un nombre. Inversément, on parlera de « bit de **poids faible** » pour désigner le bit le plus à droite.

2.2.2 Nombre de configurations représentables avec k bits

Un 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 différentes. Par exemple, un octet peut prendre $2^8 = 256$ valeurs différentes. Ces valeurs sont numérotées de 0 à $2^k - 1$. Nous ne parlons pas ici de la valeur de la séquence si on l'interprète via un système de numération positionnel, mais bien du nombre de **configurations différentes** que la séquence peut prendre.

Comme nous venons de le signaler, si en revanche 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. Par exemple, un octet interprété comme un entier naturel peut prendre 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 [6], 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 ? Comme cela ne change rien au nombre de chiffres nécessaires, nous pouvons supposer que n est une puissance de b. Un tel nombre s'exprimerait sous la forme $n = b^k$ et, par conséquent, le nombre de chiffres nécessaires vaut :

$$k = \log_b n. \tag{2.12}$$

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 :

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

5. Il n'est pas rare de trouver des domaines ou des auteurs travaillant avec des puissances de 1024 plutôt que de 1000, ce que nous ne faisons pas dans ce cours.

6. 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) jusqu'après la 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 deux chiffres en base 16. C'est pourquoi la base 16, appelée système **hexadécimal**, est couramment utilisée en informatique pour représenter 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.

Etats binaires et nombres

Une suite de bits au sein d'une séquence 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 était représentée comme un couple de chiffres hexadécimaux, et chaque couleur étant 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 :

$$n_2 = \underbrace{X_3 X_2 X_1 X_0}_{=X_{16}} \underbrace{Y_3 Y_2 Y_1 Y_0}_{=Y_{16}}, \tag{2.21}$$

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

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

Au sein d'un ordinateur, les entiers naturels (nombres nuls ou positifs et sans chiffres après la virgule) peuvent être représentés de façon directe en binaire. Comme discuté au sein de la section 2.2.2, le seul point à prendre en compte est le nombre de bits utilisés pour les représenter. Par exemple, une séquence de 8 bits peut représenter $2^8 = 256$ valeurs

différentes (de 0 à 255 si l'on choisit arbitrairement de faire corresponde 00000000 à zéro). Si l'on souhaite représenter davantage de valeurs différentes, il est nécessaire d'utiliser des mots plus longs. Par exemple, une séquence de 16 bits peut représenter $2^{16} = 65536$ valeurs différentes, de 0 à 65535 si l'on choisit arbitrairement de commencer à zéro.

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 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, lorsque nous définirons la fonction $cod(n)$, nous ferons référence à la valeur de la séquence de bits codant le nombre n si elle est interprétée comme un entier naturel. De même, nous pourrons parler de $dec(c)$, qui est la fonction retournant l'entier relatif associé à la séquence de bits c si cette dernière est interprétée comme un entier naturel.

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 tel que C, les types de variables correspondent à des alias 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 2.2 résume les tailles minimales dans ce langage. De nombreux langages populaires dérivant du C, tels que C++ ou Python, ont également hérité de la nomenclature de certains de ces types de variables.

TABLE 2.2 – Résumé des différentes tailles minimales des variables en C. En pratique, ces tailles peuvent être supérieures et dépendent de l'architecture utilisée. 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.

Alias	`char`	`short`	`int`	`long`	`long long`
Taille minimale (octets)	1	2	2	4	8
Taille habituelle (octets)	1	2	4	4	8

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

Il est important de comprendre que l'architecture de la machine détermine le nombre de bits sur lesquels les variables sont réellement codées ; les alias sont des conventions qui fixent un nombre minimum de bits mais ne garantissent pas la taille exacte. Par exemple, un `int` doit pouvoir représenter au moins 2^{16} valeurs différentes, ce qui implique qu'il est techniquement codable sur 2 octets ; or, dans la plupart des cas pratiques, il est codé sur 4 octets.

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. Conceptuellement, 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 additione par exemple 150 et 150, toujours sur un octet ? 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**. Il faut donc gérer les débordements (overflow en anglais) d'une façon ou d'une autre. Une manière habituelle de réguler ces débordements de façon arbitraire est d'ignorer le bit qui déborde. Dans notre cas, cela signifie que la valeur lue dans le registre après l'addition de 150 et 150 est $\not{1}00101100_2 \rightarrow 00101100_2 = 44$. Comme on le verra dans le chapitre 4.4.2 sur les circuits logiques, il est tout à fait faisable au niveau des circuits électroniques de détecter les débordements ; cependant, tous les langages de programmation ne permettent pas de le faire aisément.

Il faut noter la nature cyclique ou **périodique** de l'arithmétique sur des mots de taille finie. Notons que, dans certains cas, ce comporement peut être souhaitable. Par exemple, si l'on représente une donnée temporelle où l'entier 255 représente la dernière seconde avant le tour d'horloge complet, il est souhaitable que l'addition de 1 à 255 donne 0.

Le bit qui déborde d'un octet étant toujours le neuvième, l'ignorer revient toujours à soustraire $2^8 = 256$ 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. En général, si l'on additionne 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.

Pour terminer, notons que 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 manipuler les données de façon à libérer l'espace nécessaire. Mais, les mots ayant des tailles fixes au sein d'une architecture donnée, cette méthode n'exploite pas pleinement les capacités matérielles de l'ordinateur, en plus d'introduire un traitement supplémentaire des nombres impliqués. Les opérations sur des nombres de taille fixe sont donc plus rapides que celles sur des nombres de taille variable. C'est pourquoi, dans des cas où la vitesse d'exécution est critique, il est toujours usuel de travailler avec des mots de taille fixe.

2.4 Représentation 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.

En C et dans d'autres langages, les tailles de variables discutées dans le tableau 2.2 peuvent être couplées avec l'alias `unsigned` afin de les interpréter comme des entiers naturels et non pas relatifs (ce qui constitue le cas par défaut). Ainsi, un `short` est un entier signé sur au moins 2 octets, tandis qu'un `unsigned short` est un entier naturel sur au moins deux octets. Chacun devrait donc en théorie pouvoir représenter 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 :

s	b_6	b_5	b_4	b_3	b_2	b_1	b_0

où s est le bit de **signe** (fixons $s = 0$ pour le signe positif et $s = 1$ pour le signe négatif[7]) et b_6 à b_0 sont les bits de la **norme** du nombre. 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 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

$$cod(n) = sgn(n) \cdot 2^{k-1} + cod(|n|), \tag{2.22}$$

puisque 2^k est l'entier naturel codé comme la suite de bits $100\ldots0$ de longueur k, tandis que le codage de $|n|$ est réalisé sur les $k-1$ bits de droite. Dans la dernière expression, $sgn(n)$ représente le signe de n. Nous insistons sur le fait que $cod(n)$ représente la valeur à interpréter du codage de n si ce dernier était un entier naturel.

7. Notons que le signe peut toujours être de la forme $(-1)^s$ pour un certain s binaire. C'est une raison de plus d'utiliser 0 pour coder un signe positif, puisque $(-1)^0 = 1$, et d'utiliser 1 pour coder un signe négatif puisque $(-1)^1 = -1$.

Notons que le nombre de valeurs différentes exprimables ne diffère pas de celui des entiers naturels. Dans les deux cas, il s'agit de 2^k valeurs différentes 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.

Une conséquence de la méthode de codage signe-norme est que le zéro peut être représenté de deux façons différentes : $+0$ et -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 si l'on veut définir un algorithme pour additionner des nombres avec le codage signe-norme, il est nécessaire de distinguer les cas selon le signe des nombres impliqués. La méthode utilisée jusqu'ici (addition en colonnes avec retenue) n'est plus valable telle quelle avec le codage signe-norme.

L'une des raisons de ce problème tient au fait que ce codage implique un « changement de sens de l'addition », selon que les nombres soient négatifs ou positifs. 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. Ceci est peu désirable dans la mesure où les algorithmes correspondant aux opérations de base telles que l'addition ou la soustraction, et donc les circuits électroniques qui les implémentent, doivent en tenir compte. À la place, pour résumer, nous voulons un codage tel que :

$$cod(-n) + cod(n) = cod(0). \tag{2.23}$$

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 équilibrée entre les nombres positifs codables et les nombres négatifs codables.

De façon générale, un entier relatif n codé sur k bits est représenté par une séquence de bits équivalent au codage d'un entier naturel $cod(n)$ grâce à la transformation suivante :

$$cod(n) = n + 2^{k-1}. \tag{2.24}$$

Autrement dit, il faut retrancher 2^{k-1} de *l'entier naturel* codé en mémoire pour obtenir *l'entier relatif à interpréter*.

La relation 2.24 permet de représenter davantage de valeurs négatives si le nombre de bits k est pair. 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 :

$$cod(n) = n + 2^{k-1} - 1. \tag{2.25}$$

Il est possible d'exprimer la relation inverse, qui permet d'obtenir l'entier relatif à interpréter à partir de l'entier naturel c code :

$$dec(c) = c - 2^{k-1} + 1. \tag{2.26}$$

Par exemple, si l'on utilise le codage 2.25 pour coder sur 3 bits la valeur $n = 2$ en tant qu'entier *relatif*, il faut écrire une séquence de bits qui correspond à l'entier *naturel* donné par $cod(2) = 2 + 2^{3-1} - 1 = 5$. La séquence de bits est donc 101. Inversément, pour décoder la séquence de bits correspondant à l'entier naturel $n = 101_2 = 5$ en tant qu'entier relatif, il faut effectuer le calcul $dec(5) = 5 - 2^{3-1} + 1 = 2$.

Le nombre 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 2.3 ci-dessous permet de comparer les deux transformations possible sur un exemple où $k = 3$ bits.

TABLE 2.3 – Comparaison du décodage de séquences de 3 bits interprétés comme des entiers relatifs utilisant les biais des équations 2.24 et 2.25. La séquence de bits interprétée en tant qu'entier naturel est donnée en seconde ligne pour faciliter la lecture.

Séquence de bits	000	001	010	011	100	101	110	111
n	0	1	2	3	4	5	6	7
$n - 2^{k-1}$	-4	-3	-2	-1	0	1	2	3
$n - 2^{k-1} + 1$	-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 représenté de façon unique. Par la suite, nous allons considérer le biais $n-2^{k-1}+1$ favorisant les nombres positifs.

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. Le résultat de cette addition est l'entier relatif 0, qui est 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 $(-3) + 2^{3-1} - 1 = 0 = 000_2$. Leur addition donne donc $110_2 \neq cod(0)$. Encore une fois, il faudrait donc appliquer des modifications supplémentaires pour décoder correctement le résultat. 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.

Bien que le codage par biais soit utilisé pour d'autres applications (comme nous le verrons en section 2.5.2 pour les nombres réels), il n'est pas couramment utilisé pour les entiers relatifs.

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 qui :

1. Ne comporte pas de redondances (pas respecté par le codage signe-norme).
2. Est telle que $cod(-n) + cod(n) = cod(0)$ (respecté ni par le codage signe-norme, ni par celui du biais).

Le tableau 2.4 ci-dessous propose un correspondance entre entiers relatifs et séquence de bits qui correspond aux critères énoncés plus haut.

TABLE 2.4 – Comparaison de l'interprétation de séquences de 3 bits en codage signe-norme, en codage par biais et en complément à 2.

Séquence de bits	100	101	110	111	000	001	010	011
Signe-norme	-0	-1	-2	-3	0	1	2	3
Biais	1	2	3	4	-3	-2	-1	0
Complément à 2	-4	-3	-2	-1	0	1	2	3

Commençons par noter que, à l'instar du codage signe-norme, le codage en complément à deux contient un bit de signe (à zéro pour la valeur nulle, cependant). On s'aperçoit par ailleurs que le codage d'un nombre négatif en complément à 2 est obtenu en inversant tous les bits de sa valeur absolue, puis en ajoutant une unité. Par exemple, pour obtenir le codage de -2 sur 3 bits, on commence par le codage binaire de 2, qui est 010, puis on inverse les bits pour obtenir 101, et on ajoute 1 pour obtenir 110.

On peut aisément se convaincre que le fait d'inverser les bits d'un entier naturel n revient à calculer $2^k - 1 - n$, où l'on se souvient qu'un entier naturel formé de k bits prend $2^k - 1$ comme valeur maximale. Lorsqu'on ajoute 1 à cette valeur, on obtient donc $2^k - n$.

Ainsi, en suivant la méthode décrite plus haut consistant à inverser les bits avant d'ajouter une unité, le codage d'un nombre négatif n sur k bits revient à coder la valeur de $2^k - |n|$ comme s'il s'agissait d'un entier naturel :

$$cod(n) = 2^k - |n|. \tag{2.27}$$

Reprenons l'exemple de $n = -2$ sur $k = 3$ bits. Cela donne : $cod(-2) = 2^3 - |-2| = 8 - 2 = 6 = 110_2$. Notons que le codage de zéro est également correctement décrit par cette méthode, à condition d'ignorer le bit de débordement.

Le chemin inverse est également possible. Pour obtenir la valeur d'un nombre négatif codé en complément à 2, il suffit d'inverser les bits et d'ajouter 1 à l'entier naturel c codé :

$$dec(c) = c - 2^k. \tag{2.28}$$

Par exemple, pour obtenir la valeur associée à la séquence de bits 110, on inverse les bits pour obtenir 001, puis on ajoute 1 pour obtenir 010, qui est bien la valeur de -2. De même, on peut adopter l'approche plus calculatoire discutée précédemment : la séquence de bits qui code le nombre, interprété en tant qu'entier naturel, possède la valeur $110_2 = 6$, et donc $dec(6) = 6 - 2^3 = -2$.

Le terme de « complément à deux » vient du fait que le complément à b d'un nombre de k bits est ce qu'il faut ajouter à ce nombre pour atteindre b^k. Le complément à deux du nombre $n = 3$ codé sur $k = 4$ bits est donc la valeur c telle que $3 + c = 2^4 \iff c = 16 - 3 = 13 = 1101_2$. C'est la raison pour laquelle, de façon générale, le complément à b d'un nombre n sur k bits est donc tel que $n + c = b^k \iff c = b^k - n$. En informatique binaire, seuls le complément à 1 (qui revient à une inversion des bits) et le complément à 2 sont communément utilisés.

Le problème de l'addition d'un nombre à son opposé est bien résolu :

$$cod(n) + cod(-n) = n + 2^k - n = 2^k, \tag{2.29}$$

ce qui correspond à une séquence de $k + 1$ bits, dont le bit de poids fort (qui déborde) vaut 1 et tous les autres valent 0 – autrement dit, c'est le codage du nombre zéro.

Par ailleurs, 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 valeure 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 ; le tableau 2.4 est donc à comprendre comme se refermant sur lui-même (ce qui sort par la droite entre par la gauche et *vice versa*).

2.5 Représentation 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. Une différence fondamentale, cependant, sépare la partie entière de la partie qui suit la virgule. En effet, comme discuté en section 2.1.3, la partie fractionnaire d'un nombre fini exprimé dans une base donnée n'est pas nécessairement finie dans une autre base, dans le cas général. Par exemple, pour exprimer le nombre exact 42.301 en base 7, il faut se fixer un degré de précision arbitraire, sans quoi le développement des chiffres après la virgule peut potentiellement être infini. Il en va de même pour la valeur 0.1 en base 2, qui est une série infinie de chiffres après la virgule.

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 laquel 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, quel que soit la place dont on dispose et quel 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, qui 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 :

$$n = s \cdot \sum_{i=0}^{k-1} a_i 2^i + \sum_{i=1}^{\ell} f_i 2^{-i}, \tag{2.30}$$

où s est le signe et où la somme impliquant les quantités a_i représente la partie entière du nombre et celle impliquant les quantités f_i représente sa partie fractionnaire (cf. section 2.1.3). Ici, un nombre donné est donc 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).

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_4	f_3	f_2	f_1

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 du fait qu'une représentation en virgule fixe implique une précision constante pour tous les nombres, ainsi qu'en raison d'autres raisons pratiques, la représentation en virgule fixe a été utilisée sur beaucoup de calculatrices par le passé.

Cependant, le défaut de la méthode est évident : en pratique, il est très utile de pouvoir coder des nombres dont le nombre de chiffres après la virgule sont différents les uns des autres. Pour cette raison, la représentation en virgule flottante, présentée 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 2.30 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. Avec cette représentation, on pourrait dans le même temps représenter le nombre 653.7 à l'aide de l'entier 6537 et de la mise à l'échelle 10^{-1}. 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 de chaque 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 :

s	a_3	a_2	a_1	a_0	f_3	f_2	f_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 $3.141 = 3141 \cdot 10^{-3}$. Cela est bien entendu vrai en base 2 également [8]. Par exemple : $1.1011_2 = 11011_2 \cdot 2^{-4}$. Ainsi, de façon générale, un nombre réel n peut être exprimé comme suit :

$$n = s \cdot m \cdot b^E, \tag{2.31}$$

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 -3.127 en base 10, le signe est -1, la mantisse est 3127 et l'exposant est -3. Pour le nombre -1.1011 en base 2, le signe est -1, la mantisse est 11011 et l'exposant est -100_2, c'est-à-dire -4 en décimal.

On reconnaît au sein de l'équation 2.31 la structure d'un nombre exprimé en 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.

Avant de continuer, réecrivons l'équation 2.31 en base 2 et en exprimant le signe s sous la forme $(-1)^s$, comme évoqué précédemment pour les entiers codés en signe-norme. Nous avons alors :

$$n = (-1)^s \cdot m \cdot 2^E. \tag{2.32}$$

Le fait de contraindre $1 \leq m < b$ signifie, pour le binaire, que le premier chiffre de la mantisse est forcément 1. Cela permet de l'omettre dans la représentation de la mantisse en mémoire. Ainsi, de façon implicite, une mantisse codée par m correspond à une valeur de mantisse égale à $1 + m$.

8. 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. Pour pouvoir représenter des nombres plus petits que la base, E doit être négatif. Pour pouvoir représenter nombres plus grands que la base, E doit 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 une méthode de codage différente, dont nous parleons plus bas. Il est important de garder en tête que les raisons qui ont mené à ce choix ne sont pas fondamentales mais techniques ; l'important est bien de pouvoir coder E comme un entier relatif.

La méthode de codage par biais correspondant à la transformation 2.25 est choisie pour représenter l'exposant. On parle alors d'**exposant biaisé**. Ainsi, si la valeur de l'exposant est E, son codage E_{biais} est donné par :

$$E_{biais} = 2^{k-1} + E - 1 \Longleftrightarrow E = E_{biais} - 2^{k-1} + 1, \tag{2.33}$$

Par exemple, si l'exposant vaut $E = 12$, alors son codage sur 8 bits est $E_{biais} = 2^{8-1} + 12 - 1 = 139 = 10001011_2$. Lorsqu'on fixe le nombre de bits sur lesquels l'exposant est codé, on fixe également le biais. Sur un octet, le décalage vaut $2^{8-1} - 1 = 127$.

Pour résumer toutes les règles précédemment mentionnées, un nombre n est décodé de la façon suivante à partir du bit de signe s, des bits m_i de la mantisse et de la valeur E_{biais} de l'exposant :

$$n = (-1)^s \cdot \left(1 + \sum_{i=1}^{k_m} m_i 2^{-i}\right) \cdot 2^{E_{biais}+1-2^{k_e-1}}, \tag{2.34}$$

où k_m est le nombre de bits de la mantisse et k_e le nombre de bits de l'exposant.

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.

Formats de réels

Comme on vient de le voir, le format de codage possède comme paramètre le nombre de bits dévolus aux 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.
- 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.

Prenons maintenant un exemple de codage d'un nombre réel sur 32 bits. 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 $E_{biais} = 10000010_2 = 130$ 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 en s'amplifiant et affecter le résultat final de façon très significative.

Prenons l'exemple du nombre $0.3 = 3/10$. En base 10, son développement fractionnaire est fini[9] car $0.3 = 3 \cdot 10^{-1}$. En base 2 cependant, où le nombre doit être exprimé sous la forme $0.3 = (1 + m) \cdot 2^E$, 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 décimal 0.3 est

$$\underbrace{0}_{s}\ \underbrace{01111101}_{E_{biais}}\ \underbrace{00110011001100110011010}_{m}, \tag{2.35}$$

et le nombre approximé qui en découle vaut donc $n = 0.30000001192092895507812 5$.

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 A un exemple de code illustrant ce phénomène. En particulier, on montre que si l'on additionne 0.3 (codé en simple précision) à lui-même un million de fois, le résultat vaudra 299546.6875 au lieu des 300'000 attendus, soit une erreur relative de plus de 0.15 %. Si l'incrément est de 0.2 au lieu de 0.3, l'erreur relative grimpe à quasiment 1%. On peut créer une erreur 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 le code donné en annexe A pour s'en convaincre.

9. En base b, les nombres qui peuvent s'exprimer sous la forme d'une fraction irréductible dont le dénominateur est une puissance de b sont nécessairement ceux dont le développement fractionnaire est fini. En effet, comme on l'a vu, une multiplication par b^k revient à décaler la virgule de k chiffres. Les nombres décimaux dont le développement fractionnaire en base 2 est fini sont ceux qui s'expriment sous la forme d'une fraction simplifiée du type $n = \frac{k}{2^m}$, où $k, m \in \mathbb{N}$.

Nombres dénormalisés et nombres spéciaux

Il est important de noter qu'au sein de la norme IEEE 754, tous les nombres pour lesquels l'exposant est nul mais pas la mantisse sont des nombres dits **dénormalisés**[10]. En effet, cette plage de nombres rompt avec la convention utilisée plus haut pour représentée les nombres au format « notation scientifique ». Ces nombres dénormalisés sont des nombres très petits. Pour les représenter, on utilise à la place une technique similaire à celle des nombres en virgule fixe, c'est-à-dire que l'exposant est fixé à une valeur spécifique (par exemple, $E = -126$ en simple précision) et la mantisse est codée telle qu'elle est, sans le bit implicite de poids fort valant 1. Ainsi, un nombre dénormalisé est un nombre qui est représenté avec une précision moindre que les autres nombres, mais qui permet 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 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 spéciales de l'exposant et de la mantisse pour les nombres spéciaux en virgule flottante. Ici, les valeurs sont données pour le format simple précision.

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

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 $\infty[$
overflow		underflow		overflow
$-\infty$	normalisés	0 et dénormalisés	normalisés	$+\infty$

Les valeurs spéciales $+\infty$ et $-\infty$ se comprennent aisément de par leur similarité avec

10. Si l'exposant est nul, le nombre devrait valoir 1 quelle que soit la mantisse. Ainsi, il est tentant d'exploiter ce « gaspillage » de valeurs, ce qui donne lieu aux nombres dénormalisés.

les cas de débordement d'un entier. Ici, lorsque la valeur de l'exposant E n'est pas exprimable à l'aide des bits qui lui sont dévolus parce qu'elle est trop élevée, on parle d'**overflow** du nombre. Dans ce cas, la valeur du nombre est remplacée par $+\infty$ ou $-\infty$ en fonction du signe du nombre.

De même, lorsque la valeur de l'exposant est trop faible pour être exprimée, on parle d'**underflow**. Dans ce cas, la valeur du nombre est soit remplacée par un nombre dénormalisé, soit remplacée par le zéro exact, dépendemment des options de compilation. 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 $E_{biais} = 11111110_2 = 254$, 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 $E_{biais} = 00000001_2 = 1$, 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.

Chapitre 3

Codage de l'information – Textes, images et sons

3.1 Représentation des caractères

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.

La représentation des caractères au sein d'un texte est un exemple important de ce type de correspondance. C'est l'objet de cette section.

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 un octet pour coder chaque caractère ; cependant, 1 bit étant

utilisé pour contrôler les erreurs, 7 bits restent disponibles pour coder les caractères. Ainsi, $2^7 = 128$ caractères différents peuvent être représentés en ASCII. La table 3.1 ci-dessous présente les caractères ASCII les plus courants.

$b_7 b_6 b_5$ → / Bits $b_4 b_3 b_2 b_1$ ↓ / Column → / Row ↓	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	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 41 en hexadécimal.

Bit de parité

Le bit qui n'est pas utilisé pour coder les caractères est souvent utilisé pour la parité, c'est-à-dire pour vérifier que le message n'a pas été corrompu lors de sa transmission. 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 remplaçant le bit de parité, discuté plus haut, par un bit supplémentaire pour coder des caractères. 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 150'000 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. Certains codages d'Unicode font ne permettent d'ailleurs de coder qu'un sous-ensemble des caractères prévus par le standard Unicode.

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. Bien qu'il existe des façons fixes de coder des parties (ou tout) du standard Unicode, 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 du système d'exploitation Windows, utilise deux octets par caractère (et peut donc encode de U+0000 à U+FFFF), ce qui permet 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 l'occurence, en UCS-2, le point de code interprété comme entier coïcinde avec le codage lui-même.

Cependant, cette simplicité a un coût : le gaspillage de mémoire. En effet, un texte en

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).

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 les caractères les plus fréquents, mais au prix d'une complexité accrue de l'algorithme d'interprétation du codage. En effet, dans ce cas, on ne peut pas directement accéder à un caractère en mémoire en temps constant : il faut parcourir les caractères 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.

Le codage UTF-8 constitue un exemple commun de codage à taille variable, et compose de nos jours la grande majorité des pages web, par exemple. Les caractères y sont codés sur 1, 2, ou 3 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 et Latin-1, de sorte que les points de code des caractères ASCII et Latin-1 sont les mêmes en Unicode.

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 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 une telle erreur :

```
1  original_string = "Ce codage n'est pas réussi !"
2
3  # Encodage de la chaîne en utf-8
4  encoded_string = original_string.encode('utf-8')
5
6  # Décodage de la chaîne encodée
7  decoded_string = encoded_string.decode('latin-1')
8
9  # Affichage du résultat
10 print(decoded_string) #affiche "Ce codage reste Ã  vÃ©rifier !"
```

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 Représentation des sons

D'un point de vue physique, le son est une onde transversale qui se propage au sein d'un milieu donné (nous considérerons l'air à partir d'ici). Cette vibration 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 micro inversé : il convertit les variations de tension électrique en vibrations d'une membrane, et donc de l'air ambiant.

À l'échelle humaine, on peut faire l'excellente approximation que le nombre de niveaux de pression acoustique de l'air est infini. 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 approximant son amplitude à certains moments, ce qui résulterait en une séquence finie de mesures d'une précision limitée, tel qu'illustré au centre de la figure 3.2. Cette séquence de nombre serait celle que nous pourrions stocker de façon digitale au sein de la mémoire d'un ordinateur. Lorsque la séquence de nombre 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.

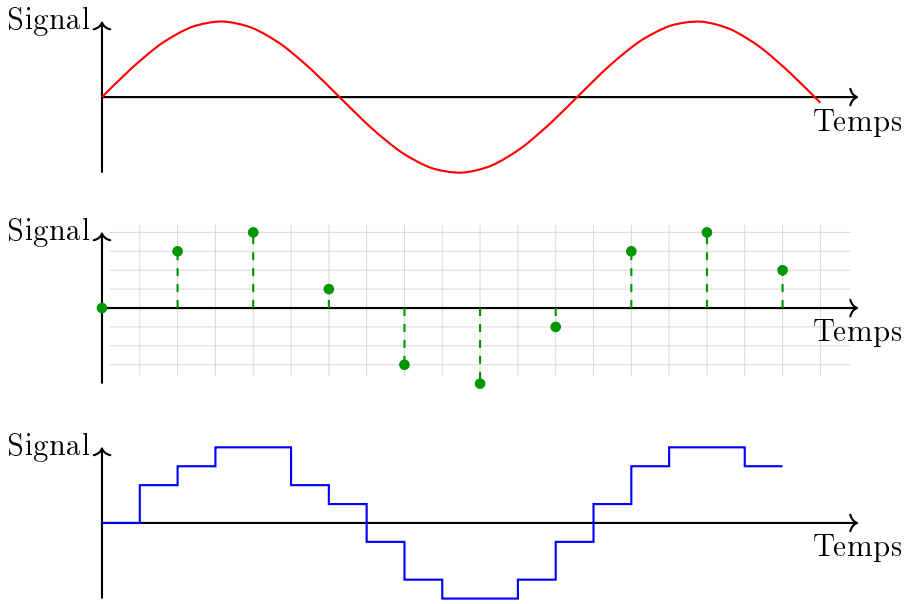


FIGURE 3.2 – Signal sonore analogique (en haut), discrétisé (au milieu) et analogique reconstruit (en bas). C'est la version du milieu, discrétisée en une séquence finie de nombres d'une précision finie (représentée par les points de la grille), qui est stockée au sein de la mémoire d'un ordinateur.

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.

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. Les algorithmes de compression audio sont un sujet vaste qui sort du cadre de ce cours. Nous verrons que le

même principe de compression est utilisé pour les images.

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 **bitmaps** (encore nommé image matricielle ou raster) et les formats **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 de l'image. 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 Apparté 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.

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.

3.3.3 Principe du codage RGB

Avant d'aller plus loin, notons brièvement que, d'un point de vue biologique, les couleurs peuvent être décomposées en trois composantes de base : le rouge, le vert et le bleu (ou toute autre triplet de couleurs dont aucune ne peut être obtenue à partir des deux autres). Ces trois couleurs sont appelées **couleurs primaires** et sont suffisantes pour

coder toutes les couleurs visibles par l'oeil humain. En effet, l'oeil humain possède trois types de cônes, chacun sensible à une couleur primaire. En mélangeant ces trois couleurs primaires, on peut obtenir toutes les autres couleurs. C'est le principe de la synthèse additive des couleurs. [2]

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, tout comme on peut décider que le nombre représenté par le codage de la couleur du pixel correspond à l'intensité de sa couleur. Par exemple, si l'on décide que le nombre 0 correspond au noir et le nombre 255 au blanc, alors un pixel dont la valeur est 128 sera de couleur grise, et tout pixel de l'image sera d'une nuance plus ou moins extrême de gris.

Il se trouve que l'oeil humain est capable de distinguer environ 10 millions de couleurs différentes, ce qui est bien plus que les 256 couleurs que l'on peut coder avec un octet. En supposant que ces 10 millions de couleurs 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'un dégradé de couleurs soit perçu comme continu :

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

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. C'est un heureux hasard, car 24 est justement divisble par 3, qui est le nombre de couleurs primaires pour l'humain. C'est pourquoi les images non-compressées sont souvent codées sur 3 octets par pixel, soit 1 octet pour chaque composante de couleur (rouge, vert, bleu), ce qui permet de coder environ dix-sept millions de couleurs, avec 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. 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.

Il faut noter qu'il est également courant de coder les couleurs sur 4 octets, en ajoutant un octet supplémentaire pour coder l'opacité du pixel sur 256 niveaux possibles. On parle alors de codage RGBA (pour Red, Green, Blue, Alpha).

3.3.4 Autres codages de couleurs

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 par `(255,0,0)` et hexadécimal par `#FF0000`, où l'utilisation du dièse est une convention pour indiquer que l'on parle de couleurs. La table 3.1 ci-dessous donne quelques exemples de couleurs importantes et leur codage en RGB et en hexadécimal.

2. Il faut insister sur le fait que cette décomposition a pour origine la biologie humaine et n'est pas une propriété universelle des couleurs, qui correspondent à des longueurs d'ondes de la lumière.

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 B 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.

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.

3.3.5 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. Tout comme le 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, à l'instar du format WAV 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. L'indication « mixte » signifie que le format existe en plusieurs versions et peut être utilisé avec ou sans pertes.

Format image	Format audio	Compression	Perte d'information
PPM, BMP	WAV	Non	Non
PNG, GIF	FLAC	Oui	Non
JPEG	MP3, AAC, OGG	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 toute information pouvait être représentée de manière binaire aussi bien que décimale ou autre.

Dans ce chapitre, nous allons nous intéresser à la façon dont les calculs logiques peuvent être effectués au travers de circuits logiques binaires. Il est important de comprendre qu'ici, nous nous situons 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 (électromécanique, tube à vide, transistor) 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 (binaire) de rouler à plus de 50 km/h sera dénoté A, tandis que le fait (binaire) de se situer hors localité sera dénoté B. Enfin, le résultat (binaire) d'être en infraction sera dénoté R. Quelle que soit la proposition A, B ou C, nous noterons 1 si elle est vraie et 0 sinon. La table de vérité correspondante est donnée dans le tableau 4.1 ci-dessous.

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. Dans notre cas, 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

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 égalementet 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

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 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.

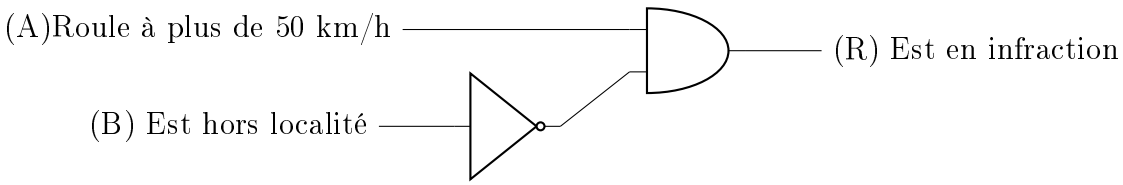


FIGURE 4.1 – Circuit logique pour le problème de l'excès de vitesse.

Ainsi, 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 qui implémentent les portes logiques que nous avons définies, ce que nous discutons 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 ambigue dans le langage naturel, 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* et *NOR* et *XOR*. Chacune de ces portes logiques possède un symbole (voir figure 4.2) ainsi qu'une table de vérité associée (voir tables 4.2 à 4.7), qui indique comment les valeurs d'entrée (inputs) sont combinées pour donner le résultat en sortie (output).

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

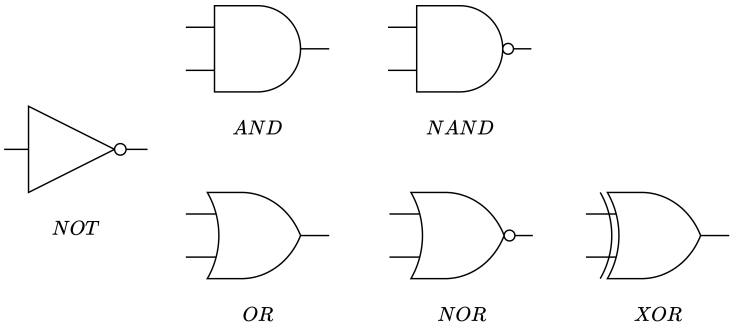
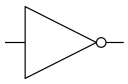
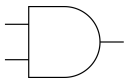
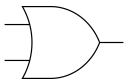


FIGURE 4.2 – Symboles pour les six types de portes logiques utilisées dans ce cours. Chaque porte est située au-dessus du texte indiquant son nom.

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.

Fonction	Algèbre	Logique	Programmation	Diagrammes logiques
NOT	!	$\neg$ ou $\sim$	!	
AND	$\cdot$	$\wedge$	&& ou *and*	
OR	$+$	$\vee$	$\|\|$ ou *or*	

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

4.2.2 De transistors à portes logiques

Bien que cela ne soit pas le sujet principal de ce chapitre, nous donnons ici bref un aperçu de la façon dont les 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'une petite tension de contrôle (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.

Dans la littérature, on trouve fréquemment des situations 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'occurence, le transistor de la figure 4.3 permet de réaliser une porte logique *NOT*, où l'on lit l'output de la fonction logique au niveau du collecteur du transistor, et où l'input correspond à la tension appliquée à la base.

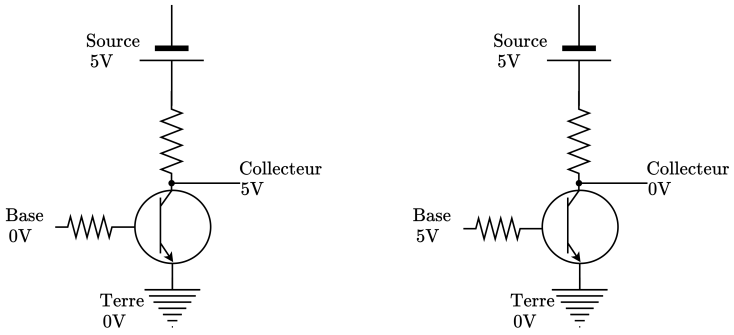


FIGURE 4.3 – Transistor réalisant la fonction logique *NOT* au sein d'un circuit électronique. À gauche, la base n'activant pas le transistor, aucun courant ne s'établit dans le circuit et une tension de 5V est mesurée en output au niveau du collecteur. À droite, la base activant le transistor, un courant 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 successivement l'un, puis l'autre des transistors pour pouvoir s'installer, on dit bien qu'il doit francher l'un *et* l'autre ; inversément, 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.

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.

3. En théorie, ces résistances ne sont pas nécessaires, et elle ne constituent pas ici l'objet étudié ; nous les représentons car, en pratique, elle sont nécessaires pour contrôler le courant subi par les différents éléments du transistor.

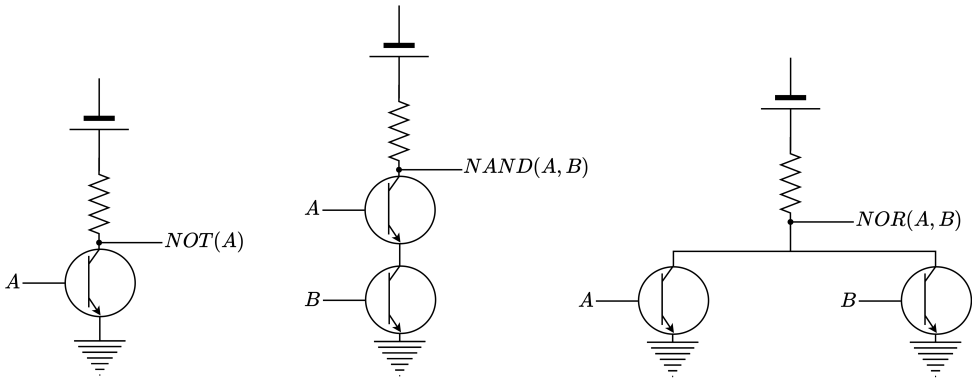


FIGURE 4.4 – Fonctions logiques *NOT*, *NAND* et *NOR* réalisées grâce à des transistors au sein d'un circuit électronique. Par souci de lisibilité, les légendes pour la source d'énergie et la terre sont omises ainsi que les résistances sur les lignes d'input.

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.

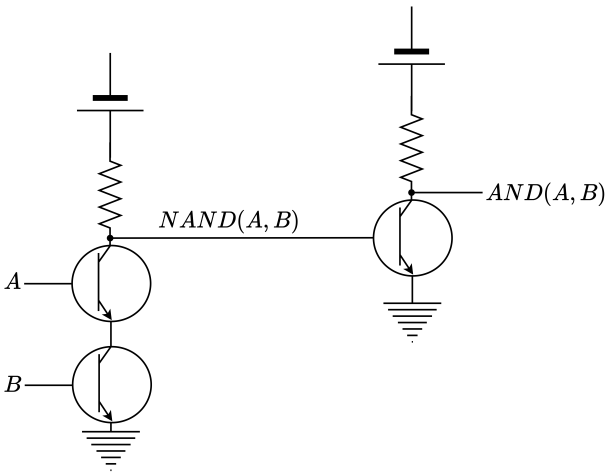


FIGURE 4.5 – Fonction logique *AND* réalisée grâce à des transistors au sein d'un circuit électronique. Ici, la combinaison d'une porte *NAND* et d'une porte *NOT* a servi à réaliser le circuit. Par souci de lisibilité, les légendes pour la source d'énergie et la terre sont omises ainsi que les résistances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

Avec ces combinaisons de transistors, nous sommes maintenants parés pour passer à l'étape suivante : considérer la combinaison de plusieurs portes logiques afin de construire un circuit dit « textbfcombinatoire ». 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 ci-dessus 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, nous avons besoin d'un cadre mathématique. En effet, toutes les tables de vérités ne sont pas aussi intuitives à retranscire 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.

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 :

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

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)*	$x + y$	$1 + 1 = 1$

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

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

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.

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 et de robustesse peuvent en résulter.

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

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

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. Mais qu'en est-il des portes et des fonctions logiques ? Existe-t-il une porte qui permettrait, en se combinant à elle-même de façon répétée, de modéliser toutes les fonctions logiques ?

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

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

tout comme on peut exprimer OR de la façon suivante en utilisant la version AND du théorème de De Morgan [6] :

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

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 :

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

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.

On peut effectuer le même raisonnement avec OR :

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

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 $NAND$. Comme nous le verrons plus loin dans ce chapitre, ces portes s'avéreront également universelles pour les circuits dits « séquentiels », qui sont des circuits combinatoires auxquels on ajoute une notion de temps. Ainsi, l'ensemble des circuits formant un ordinateur peuvent être réalisés à l'aide d'un unique type de porte logique composée de deux transistors !

5. À interpréter comme : nous avons x et y si nous n'avons pas (l'inverse de x ou l'inverse de y).
6. À interpréter comme : nous avons x ou y si nous n'avons pas (l'inverse de x et l'inverse de y).

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. Ici, la variable n est donnée pour facilité la lecture, mais elle ne constitue pas une entrée de la table de vérité dont les seules entrées sont x, y, z et dont l'output est P.

n	z	y	x	P
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

On peut lister toutes les combinaisons des inputs x, y et z qui donnent un output égal à 1. Ces input sont 010, 011, 101 et 111. Autrement dit :

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

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 (second 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 (premier terme de l'expression finale). Grâce à l'application de règles du calcul booléen, nous avons donc réussi à condenser en deux phrases ce que la table de vérité semblait dire en huit phrases.

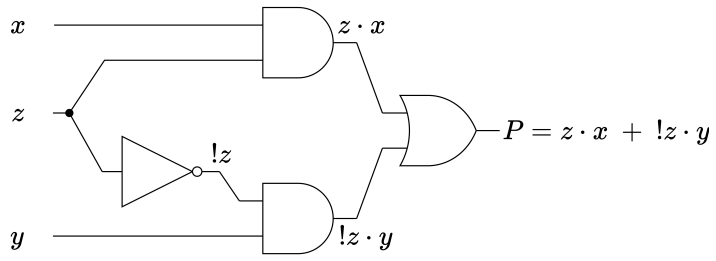


FIGURE 4.6 – Fonction P relative à la table 4.12 réalisée grâce à des portes logiques au sein d'un circuit combinatoire. L'expression d'output des portes est indiqué à proximité de ces dernières.

Pour terminer, remarquons que la « traduction » mathématique directe de la table de vérité exprimée par l'équation 4.12, qui revêt la forme d'une somme de produits, est appelée une **forme canonique**.

4.3.5 Méthode 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.
3. Nous avons écrit la fonction logique sous forme de somme de produits correspondant à la liste susmentionnée.
4. 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 que au minimum l'un d'entre eux doit être vrai pour que le tout soit vrai. On peut également utiliser la méthode des **maxtermes**, qui consiste à 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. Comme c'est un produit, au maximum un seul de ses facteurs doit être faux pour que le tout soit faux.

Appliquons la méthode des maxtermes au problème du nombre premier à trois bits (fonction P). On peut lister toutes les combinaisons des inputs x, y et z qui donnent un output égal à 0. Ces input sont 000, 001, 100 et 110. Autrement dit :

$$!P = !z!y!x + !z!yx + z!y!x + zy!x \tag{4.15}$$
$$= !z!y(!x + x) + z!x(!y + y) \tag{4.16}$$
$$= !z!y + z!x, \tag{4.17}$$

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 $P = !(!P)$:

$$P = !(!z!y + z!x) \tag{4.18}$$
$$= !(!z!y) \cdot !(z!x) \tag{4.19}$$
$$= (!!z + !!y) \cdot (!z + !!x) \tag{4.20}$$
$$= (z + y) \cdot (!z + x), \tag{4.21}$$

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 trouvé avec les mintermes. En l'occurence, on a besoin du même nombre de portes logiques dans chacun des deux circuits logiques [7].

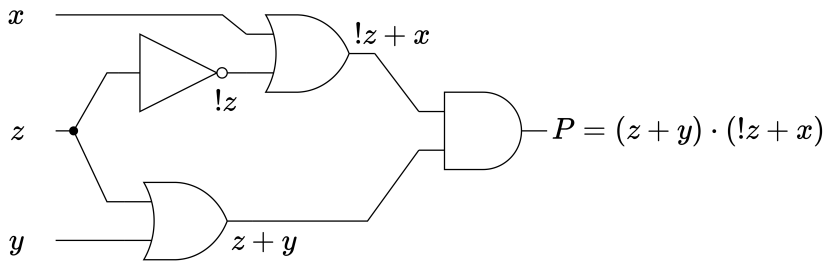


FIGURE 4.7 – Fonction P relative à la table 4.12 réalisée grâce à des portes logiques au sein d'un circuit combinatoire. L'expression pour P a été trouvée en utilisant la méthode des maxtermes. L'output des portes est indiqué à proximité de ces dernières.

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, tandis que l'on préférera les mintermes dans le cas contraire.

4.3.6 La méthode de Karnaugh

Bien qu'il n'existe pas de méthode générale 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 de ces méthodes. Elle consiste à regrouper les mintermes ou les maxtermes de la table de vérité en blocs de 1, 2, 4 ou 8 éléments, 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 un tableau 1D. Pour appliquer la méthode de Karnaugh, en revanche, nous allons disposer les inputs dans un tableau en 2D, de telle sorte que :

7. 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).

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 n'importe quelle autre, 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[8]. 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

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

Pour les tables de vérités 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

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

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.

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

Il est courant de décrire un algorithme en termes d'une condition de sélection nommée 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. Un multiplexeur est un circuit logique qui permet de choisir entre plusieurs entrées pour les transmettre à une seule sortie. Il est représenté par un symbole comme celui de la figure 4.8 ci-dessous.

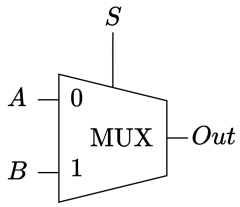


FIGURE 4.8 – Symbole d'un multiplexeur. Si S est nul, la sortie vaut A, sinon elle vaut B.

Il est important de comprendre que le multiplexeur renvoie une **ligne de donnée** quelle que soit la valeur de cette dernière. En effet, dans la phrase d'exemple donnée plus haut, la valeur de sortie est soit A soit B, peu importe la valeur de B et de C. Le multiplexeur, s'il était décrit par un code, serait de la forme :

```
if A == 0:
    return A
else:
    return B
```

Dans le cas où plus de deux choix sont possible, 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 :

$$k = 2^n \iff n = \lceil \log_2(k) \rceil, \tag{4.22}$$

où n est le nombre de bits de contrôle et k le nombre de voies du multiplexeur. Un arrondi vers l'entier supérieur est effectué puisque le nombre de bits doit être entier. La figure 4.9 illustre le symbole d'un multiplexeur à voies multiples au sein d'un circuit électronique.

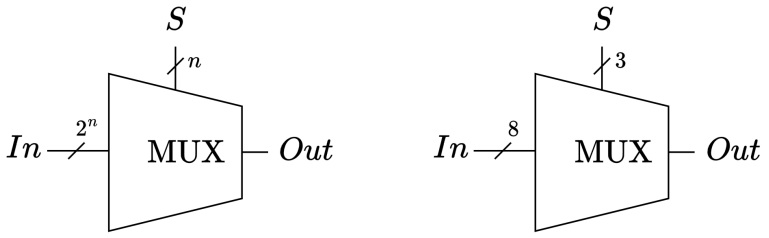


FIGURE 4.9 – Symbole d'un multiplexeur à voies multiples. (gauche) Multiplexeur à 2^n voies. (droite) Multiplexeur à 8 voies.

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

Il se trouve que la table de vérité d'un multiplexeur à 2 voies est exactement la même que celle qui décrit les bits d'écriture des nombres premiers de 3 bits 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 une discussion de la réalisation électronique d'un multiplexeur à 4 voies au sein des sections 4.3.4 et 4.3.5, 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 du multiplexeur.

Notons qu'un dispositif effectuant le travail inverse, à savoir prenant un seul mot d'entrée et le transmettant à l'une des 2^n sorties possibles désignées par S (n bits), est appelé un **démultiplexeur**.

Pour terminer, notons qu'en règle général, 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 S est divisée par 2 à chaque étage de l'arbre en direction de la racine, et décalée de 2 entre des multiplexeurs d'un étage donné. Le nombre de feuilles de l'arbre étant égal à k, le nombre d'étages est alors $\log_2(k)$.

4.4.2 Additionneur

Au sein d'un processeur, l'additionneur est un circuit logique faisant partie de l'unité d'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 nommées a_i et b_i, qui représentent les bits de poids i des deux nombres a et b en input de l'additionneur. Les outputs de l'additionneur sont les bits s_i de la somme s et le bit R de la retenue de sortie.

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

Malheureusment, 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 donner lieu à une retenue à reporter sur la colonne qui voisine (de poids plus 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 de l'additionneur, 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.

Ces deux conditions s'expriment donc ainsi :

$$R = ab + r(a \oplus b). \tag{4.23}$$

La condition pour que s vale 1, en revanche, est :

- Soit r vaut 0 et uniquement l'un des deux bits a et b vaut 1 ;
- Soit r vaut 1 et $a = b$, peu importe la valeur de a et b. Cette dernière condition, $a = b$, revient à dire que $a \oplus b = 0 \Longleftrightarrow !(a \oplus b) = 1$.

Ces deux conditions pour s s'expriment donc ainsi :

$$\begin{aligned} s =& !r(a \oplus b) + r!(a \oplus b) && (4.24)\\ =& \, r \oplus (a \oplus b), && (4.25) \end{aligned}$$

où l'on a utilisé la définition de XOR : $x \oplus y \equiv x!y + !xy$.

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 4.23 et 4.25. Le circuit logique correspondant est montré sur la figure 4.10 ci-dessous.

Finalement, pour parvenir à notre objectif d'additioner 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 4.10 par le symbole donné dans la figure 4.11 ci-dessous.

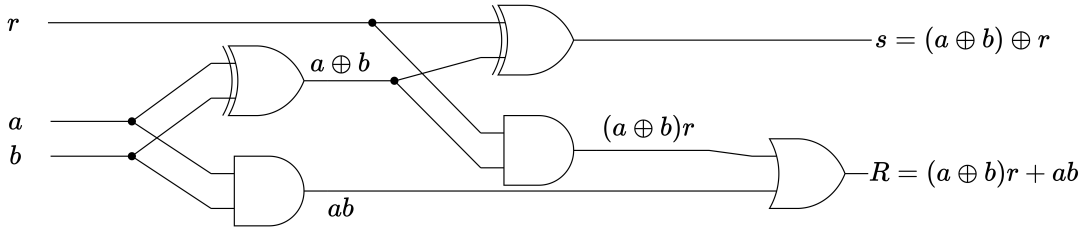


FIGURE 4.10 – Additionneur (cf. table 4.15) réalisé grâce à des portes logiques au sein d'un circuit combinatoire.

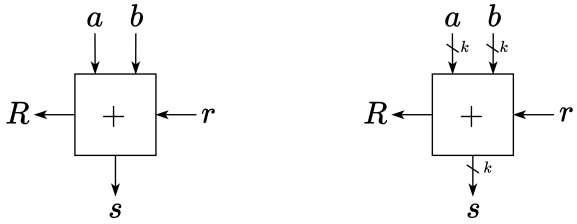


FIGURE 4.11 – Symboles d'un additionneur complet pour des mots de 1 bit (à gauche) et de k bits (à droite). Ici, le sens de l'information est donné par les flèches de façon informelle (les flèches ne sont pas à indiquer au sein d'un véritable diagramme).

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 4.12 ci-dessous. Il faut noter que cette façon de procéder

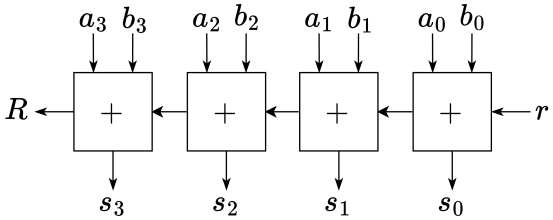


FIGURE 4.12 – Combinaisons de plusieurs additionneurs afin de réaliser l'addition de deux mots binaires de 4 bits. Pour cela, il faut faire en sorte que $r_i = R_{i-1}$ pour toutes les colonnes qui ont deux voisins. La retenue finale R signale donc un débordement.

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. La retenue finale R, quant à elle, signale un débordement du résultat (c'est-à-dire que le résultat final de l'addition ne peut être exprimés avec k bits) : 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 choisi dans le chapitre 2.4 pour décider d'un bon codage des entiers relatifs. 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 état égal $a + not(b) + 1$. Le circuit permettant de réaliser une soustraction et réutilisant l'additionneur est présenté dans la figure 4.13 ci-dessous.

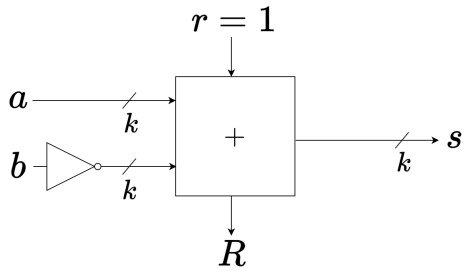


FIGURE 4.13 – Circuit permettant de réaliser une soustration entre deux entiers a et b.

4.5 Circuits logiques séquentiels

Nous avons dans ce chapitre 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.

Nous avons vu comment réaliser des circuits logiques combinatoires qui permettent, pour des inputs donnés, de calculer les outputs correspondants. Comme discuté ci-dessus, ce lien entre tension électrique d'output et tension électrique d'input n'est pas instantané. 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.

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. 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 temps écoulé entre deux pulsations soit supérieur à la durée de stabilisation de la tension électrique en sortie des portes logiques. La figure 4.14 ci-dessous montre un exemple de signal possible. Chaque période complète du signal est appelée un **cycle**.

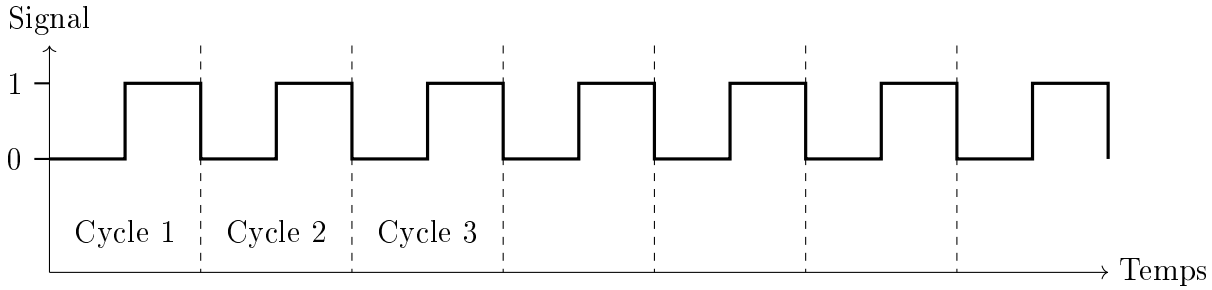


FIGURE 4.14 – Signal d'horloge idéalisé, ici sous forme d'onde carrée.

Au sein de la figure 4.15 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

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.

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*.

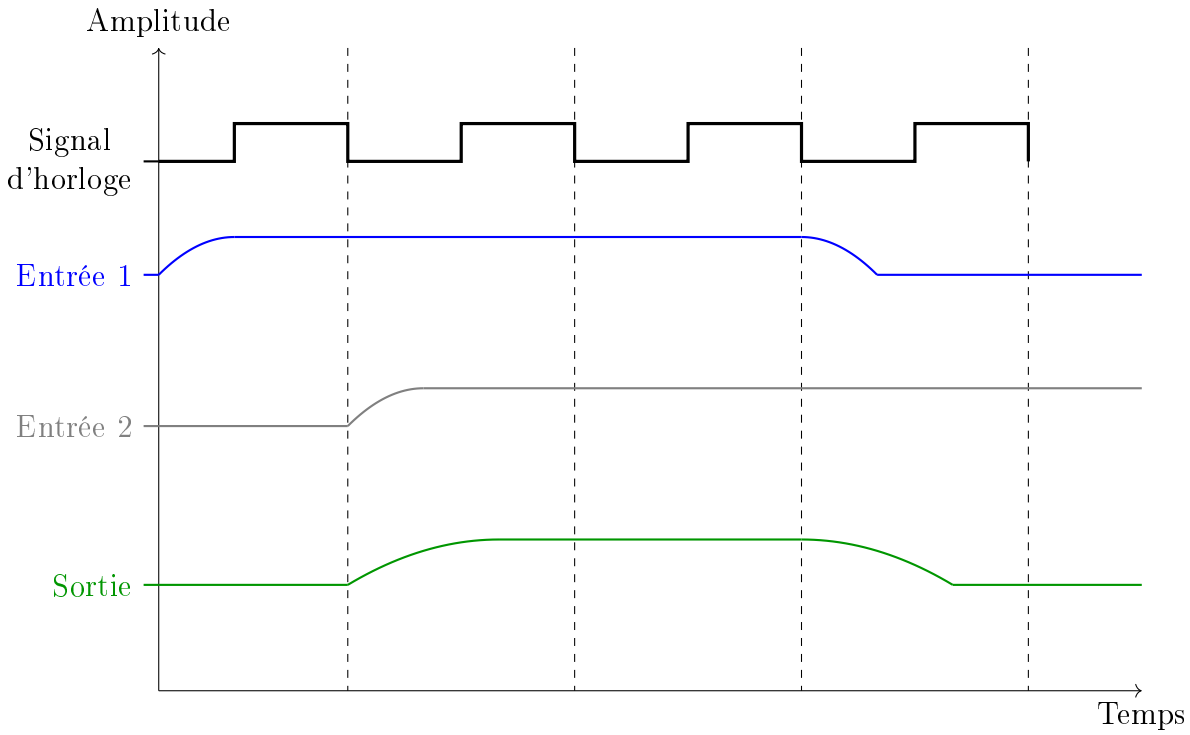


FIGURE 4.15 – Évolution schématique de la tension aux entrées et à la sortie d'un circuit pour la porte logique *AND* au sein d'un circuit séquentiel. La première entrée prend la valeur 1 durant le premier cycle, tandis que la seconde entrée prend la valeur 1 durant le second cycle, ainsi que la sortie, mais sur une durée supérieure. 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 incohérente avec la logique du *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 (ou tout autre intervalle régulier donné par l'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 :

```
i = 0
while i < 7:
    i = i + 1
    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 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 la notion 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.

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
else:
    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 en-

10. On entend par « état » l'ensemble des valeurs des signaux du circuit à la fin du cycle. Le terme « état » renvoie au fait que nous considérons la valeur que peuvent prendres ces bits en fonction du temps, un peu comme on emploie le mot « variable » au lieu de « nombre » pour désigner une quantité susceptible de prendre différentes valeurs selon les cas de figures qui nous intéressent.

trée (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.

TABLE 4.16 – Table de vérité d'une bascule, où l'on décidé arbitrairement de l'encodage 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 :

$$Q_{t+1} = !S!RQ_t + S!R = !R(!SQ_t + S) = !R(S + Q_t) \tag{4.26}$$

On peut manipuler cette dernière expression de manière à la reformuler en termes de la fonction *NOR* uniquement [11], ce qui est remarquable étant donné que cette dernière, avec *NAND*, est universelle (cf. section 4.3.3).

$$Q_{t+1} = !R(S + Q_t) = !R!!(S + Q_t) = !(R + !(S + Q_t)) = NOR(R, NOR(S, Q_t)), \tag{4.27}$$

où le théorème de De Morgan a été utilisée dans l'avant-dernière égalité. Une telle expression logique correspond au circuit suivant, en figure 4.16.

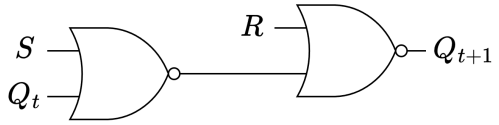


FIGURE 4.16 – Circuit logique correspondant à l'expression logique $NOR(R, NOR(S, Q_t))$ obtenue pour caractériser une bascule *NOR* réalisant la table de vérité 4.16.

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.

Le fait que Q_t sert 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 4.17 ci-dessous. Pour le reste de ce document, nous visualiserons le circuit en utilisant la représentaiton de droite sur la figure 4.17, 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

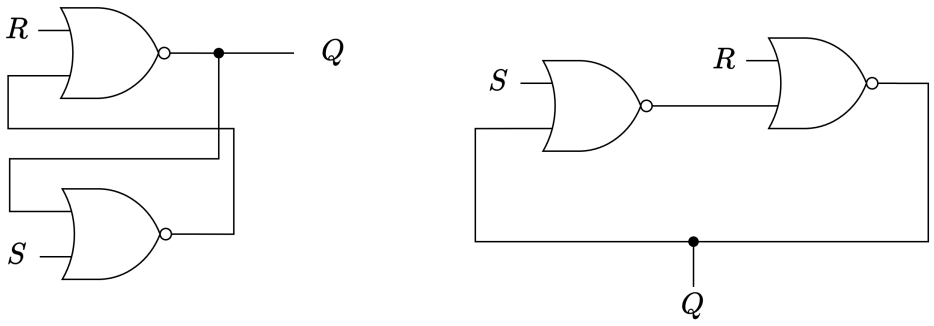


FIGURE 4.17 – Circuit logique correspondant à la bascule *NOR* réalisant la table de vérité 4.16 ; (gauche) visualisation courante au sein de la littérature, (droite) visualisation utilisée dans ce document.

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 4.18 ci-dessous illustre les quatre cas de figures possibles pour une bascule *NOR*.

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 bien plus faible que la période d'oscillation du signal d'horloge ; du point de vue de l'horloge, la bascule agit en quelque sorte instantanément sur les données, ce qui ne permet pas la réalisation d'un circuit séquentiel à proprement parler, puisque la notion de différence temporelle entre les entrées et les sorties des bascules est perdue.

Ce que l'on cherche à faire est en quelque sorte un « synchronisateur » à ajouter aux circuits combinatoires. Cette mémoire à bascule, que l'on détaille plus bas, est nommé **Delay Flip-Flop** (DFF) et reçoit une entrée particulière en plus d'un bit In : il s'agit du signal d'horloge Clk. Le symbole du DFF au sein des circuits est représenté sur la figure 4.19 ci-dessous. La figure 4.20, quant à elle, illustre comment un DFF s'intègre au sein d'un circuit combinatoire pour en faire un circuit séquentiel.

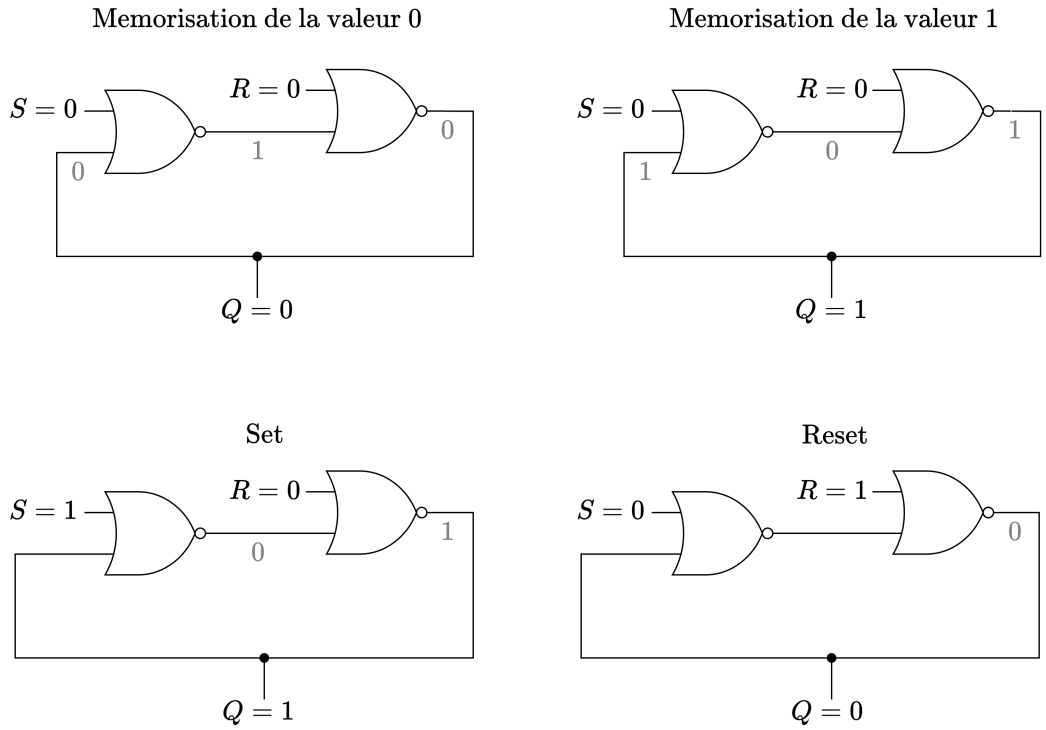


FIGURE 4.18 – Quatre cas de figures valides pour une bascule *NOR*. La ligne du haut représente les cas où la valeur Q est préservée d'un cycle à l'autre. Sur la ligne du bas à gauche, $S = 1$ et $R = 0$ (« Set ») et la nouvelle valeur en mémoire est nécessairement 1, quelle que soit la valeur précédente ; 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.

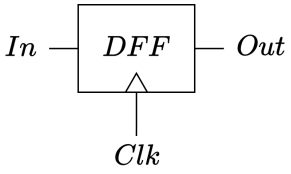


FIGURE 4.19 – Représentation d'un DFF au sein d'un circuit électronique. En général, l'entrée *Clk* est implicite, mais ce n'est pas le cas ici.

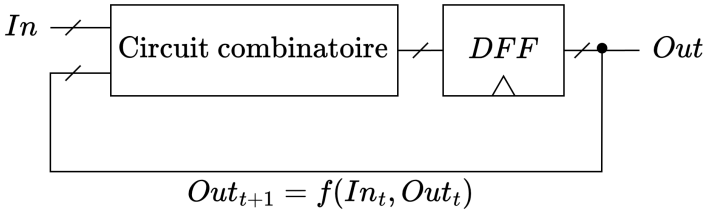


FIGURE 4.20 – Ajout d'un DFF à un circuit combinatoire quelconque de telle sorte que le circuit devienne un circuit séquentiel, c'est-à-dire pour lequel un état au cycle t dépend de l'état au cycle $t - 1$.

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

Cette idée est également résumée dans la figure 4.21 ci-dessous, où l'on utilise les notations Clk pour le signal d'horloge (« Clock ») et In pour la valeur d'entrée.

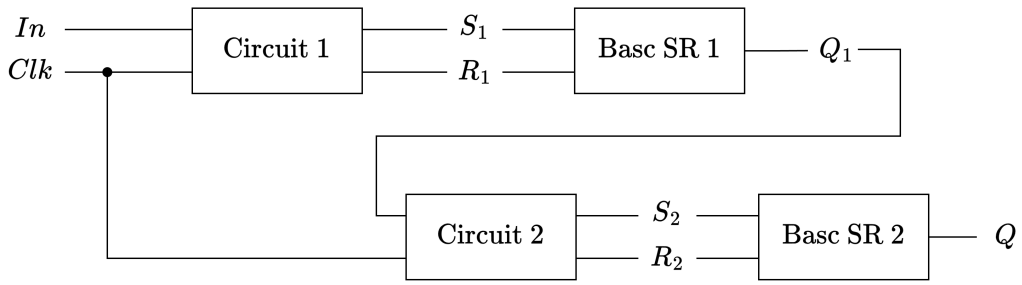


FIGURE 4.21 – Principe de fonctionnement d'un DFF composé de deux bascules, où Circuit 1 et Circuit 2, responsables du pilotage des valeurs S et R de chaque bascule, ne sont pas détaillés. Basc SR correspond à une bascule Set-Reset.

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 la fin de chaque 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.

```
#Circuit 1
if Clk == 0:
    if In:
        S1 = 1
        R1 = 0
    else:
        S1 = 0
        R1 = 1
else:
    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

TABLE 4.17 – Table de vérité des circuits 1 et 2 au sein du Delay Flip-Flop.

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

expressions logiques suivantes pour le circuit 1 :

$$\begin{aligned} S_1 &= !Clk \cdot !In \\ R_1 &= !Clk \ \cdot \ In \end{aligned} \tag{4.28}$$

et pour le circuit 2 :

$$\begin{aligned} S_2 &= Clk \cdot In \\ R_2 &= Clk \cdot !In \end{aligned} \tag{4.29}$$

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 au cycle $t+1$ à la fin de ce cycle.

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 [13], on voit que :

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

$$b'_0 = NOT(b_0) \tag{4.30}$$

$$b'_1 = XOR(b_1, b_0) \tag{4.31}$$

$$b'_2 = XOR(b_2, AND(b_1, b_0)) \tag{4.32}$$

La figure 4.22 ci-dessous représente le circuit correspondant aux expressions séquentielles obtenues.

13. Dans le cas général du compteur périodique à n bits, on aurait $b'_{n+1} = XOR(b_n, AND(b_{n-1}, ..., b_0))$.

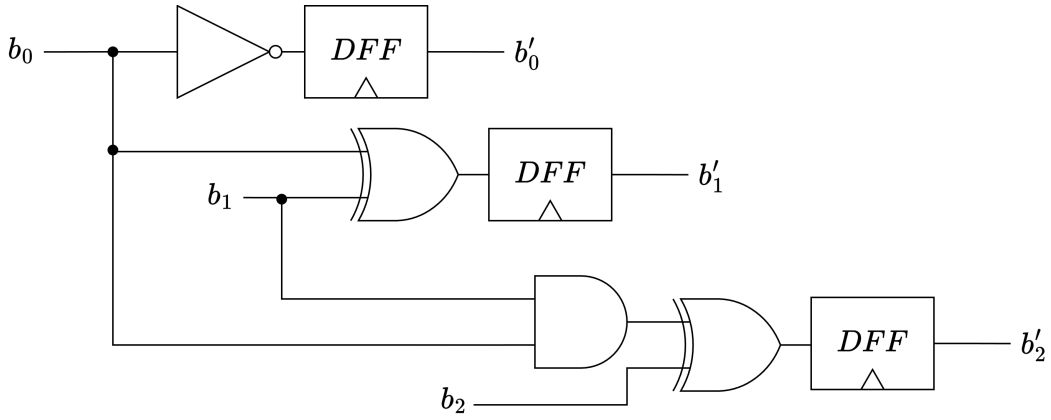


FIGURE 4.22 – Compteur périodique à 3 bits réalisé à l'aide de DFF et de portes logiques. Par souci de lisibilité, les b'_i sont représentés en sortie et les b_i en entrée, mais ils sont en réalités reliés entre eux (b' étant l'état qui suit b).

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. 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
```

Il serait possible d'écrire les 16 lignes de la table de vérité du registre à 1 bit, 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 4.23 ci-dessous illustre le circuit séquentiel correspondant à un registre à 1 bit.

Pour réaliser des registres à plusieurs bits, on doit donc regrouper plusieurs registres 1 bit et envoyer la même valeur de `Load` à chacun, comme suggéré dans la figure 4.24 ci-dessous. Pour réaliser un registre à n bits en utilisant la notation vue plus haut pour les lignes à plusieurs bits, on regroupe des registres à 1 bit comme indiqué dans la figure 4.25 ci-dessous.

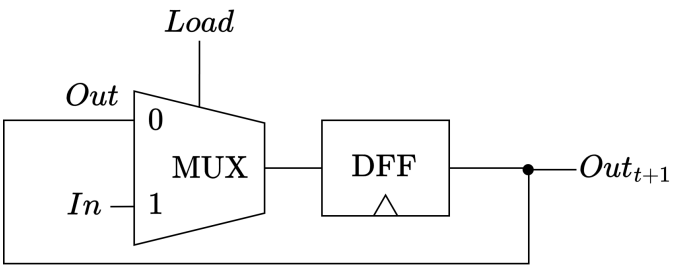


FIGURE 4.23 – Registre à 1 bit réalisé à l'aide de DFF et de portes logiques.

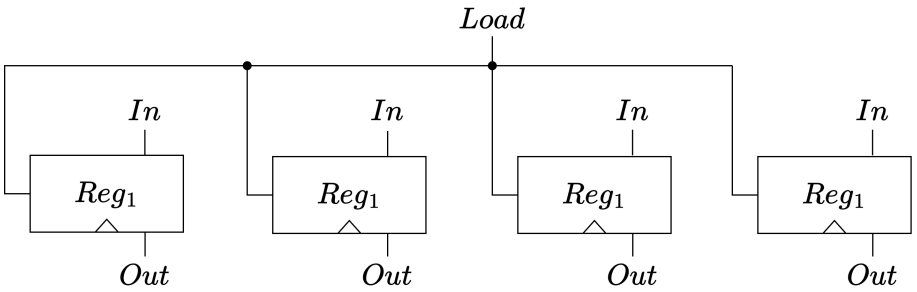


FIGURE 4.24 – Branchements de plusieurs registres 1 bit pour réaliser un registre à 4 bits.

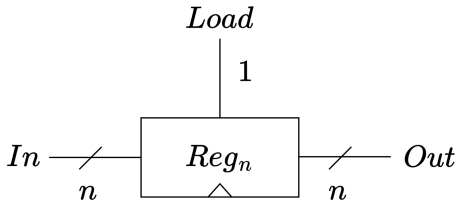


FIGURE 4.25 – Branchements de plusieurs registres 1 bit pour réaliser un registre à n bits.

Chapitre 5

Architecture des ordinateurs

Dès les années cinquante, les informaticiens ont constaté 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 matériels.

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 dédiée était substantiel.

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 illustrée sur la figure 5.1 ci-dessous, où les périphériques d'entrée (par exemple clavier, souris, réseau, ...) et de sortie (par exemple écran, haut-parleur, imprimante, réseau, ...) sont représentés, ainsi que les périphériques de stockage auxiliaire (par exemple disque dur, clé USB, carte 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 de ces éléments pour

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.

interagir avec l'ordinateur.

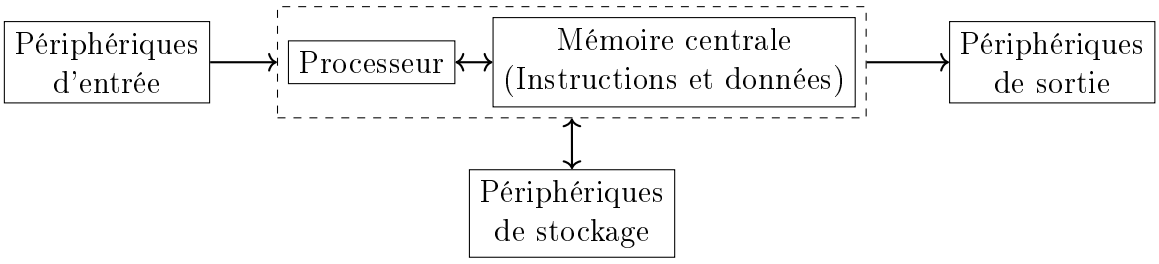


FIGURE 5.1 – Base de l'architecture de von Neumann.

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 néanmoins 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**). 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 précédentes. 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.

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.

Puisque le contenu de la mémoire centrale est effacé après une mise hors tension, il est important qu'un ordinateur possède une petite quantité de mémoire persistante (en l'occurence, une ROM comprise 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 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 [2], 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 l'UEFI (Unified Extensible Firmware Interface).

Au sein de la mémoire centrale, les données sont accédées mot par mot (cf. chapitre 2.2.1). Ainsi, quand bien même on ne serait intéressé que par un octet au sein de la mémoire centrale, ce sont bien 4 octets (dans le cas d'une architecture 32 bits) ou 8 octets (dans le cas d'une architecture 64 bits) qui seront lus et déposés dans les registres (cf. section 5.1.2). Le nombre représentant chaque adresse en mémoire suit celui de l'adresse précédente : on parle d'« adressage continu ». Nous verrons plus loin dans ce cours que cette propriété peut être exploitée par les personnes qui programment les logiciels 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 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

La mémoire auxiliaire de la machine, bien qu'ayant un rôle relativement secondaire durant l'exécution d'un programme sur un ordinateur à programme stocké, est importante en pratique pour stocker le programme lorsque la machine est hors tension (et d'autres informations capitales pour le bon fonctionnement de la machine, comme discuté plus haut). C'est pourquoi nous en traitons ici également. Il existe plusieurs familles de technologies permettant de stocker de l'information binaire de façon persistante. Voici une proposition de classification :

- Les supports de type **mécanique**, où l'information est codée grâce à une modifica-

2. Ce type de programme bas-niveau intégré au matériel est nommé « firmware » en anglais

tion 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).

- 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*. Exemples : CD, DVD, Blu-Ray.
- Les supports de type **flash**, où des transistors spéciaux, munis d'un second input en plus de la base, 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.

Nombre des exemples qui viennent d'être cités ne sont plus utilisés 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.

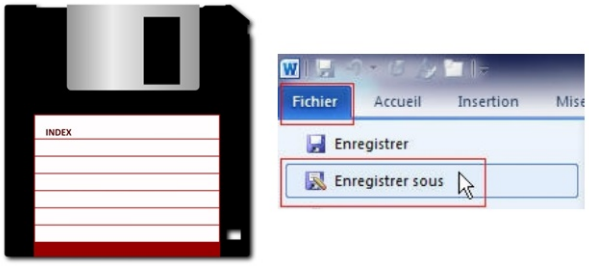


FIGURE 5.2 – (gauche) Dessin d'une disquette 3.5 pouces. (droite) Représentation schématique d'une disquette utilisée dans le logiciel Micrososft Word comme icône de sauvegarde.

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	Quelques TB
Flash	Clé USB	Centaines de Gb à 2 TB

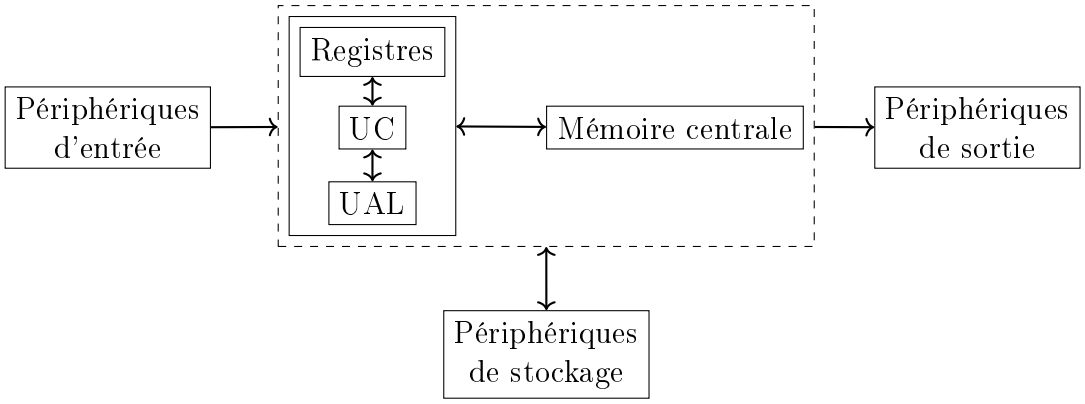


FIGURE 5.3 – Base de l'architecture de von Neumann, où l'on a détaillé les composants du processeur. UC signifie unité de contrôle, UAL signifie unité arithmétique et logique.

5.1.2 Le processeur

Le processeur lui-même 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[3]. 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 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.

3. 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.

Les registres

Les registres contiennent les valeurs intermédiaires des calculs effectués par le processeur ou des adresses d'instructions à lire ainsi que de données à lire ou écrire. Par ailleurs, deux registres spéciaux, sur lesquels nous reviendrons plus loin, sont 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 mémoires très rapides au sein même du processeur. Ils sont réalisés grâce aux DFF, 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. D'autres problèmes de nature électrique apparaissent aussi, puisque tous les registres sont connectés aux même lignes d'entrée. Si le nombre de registres est élevé, les signaux appliqués à ces lignes prennent alors plus de temps pour adopter une valeur stable.

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 4.23) 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.

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, et 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. 4). Les opérations arithmétiques sont les opérations de base de l'arithmétique, à savoir l'addition, la soustraction, la mul-

tiplication et la division. Les opérations logiques correspondent aux opérations vues dans le chapitre 4 telles que *AND*, *OR*, *NOR*, etc. L'UAL prend toujours deux mots en entrée pour en produire un seul en sortie ; c'est pourquoi elle est souvent représentée dans les schémas par le symbole indiqué en figure 5.4, qui reflète cette fusion de l'information.

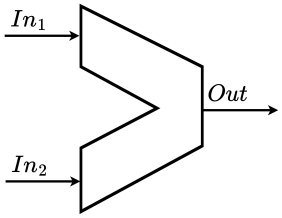


FIGURE 5.4 – Symbole de l'UAL dans les schémas de circuits, symbolisant le fait qu'elle prend deux mots en entrée (gauche) pour en produire un en sortie (droite). Des lignes de contrôle, non représentées ici, sont également nécessaires pour spécifier à l'UAL le type d'opération à effectuer.

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.

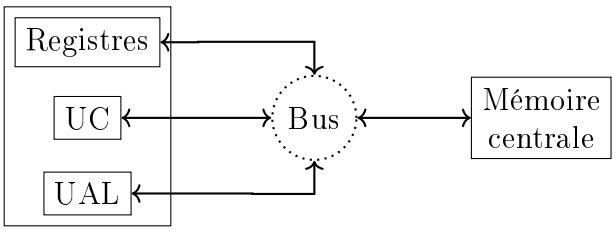


FIGURE 5.5 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que leur interconnexion *via* le bus système. Le cercle symbolise le fait que le flux d'information passe par un bus, mais il ne correspond pas à un élément isolé, le bus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

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. 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 doit donc recevoir des données. La figure 5.6 illustre le flux de données dans l'architecture de von Neumann.

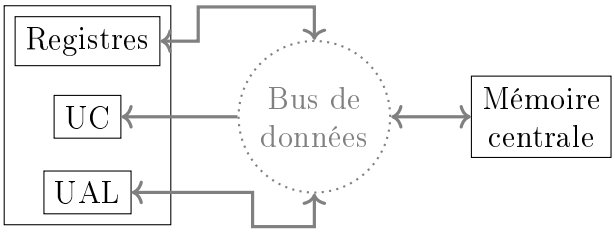


FIGURE 5.6 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que le flux de signaux transitant par le bus de données. Le cercle symbolise le fait que le flux d'information passe par un même bus, mais il ne correspond pas à un élément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

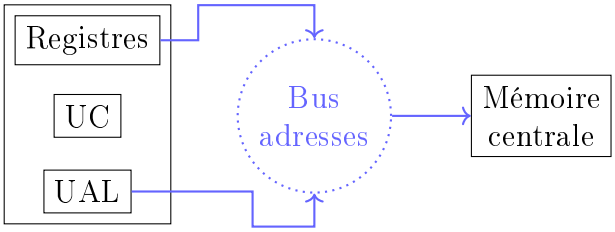


FIGURE 5.7 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que le flux de signaux transitant par le bus d'adresses. Le cercle symbolise le fait que le flux d'information passe par un même bus, mais il ne correspond pas à un élément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

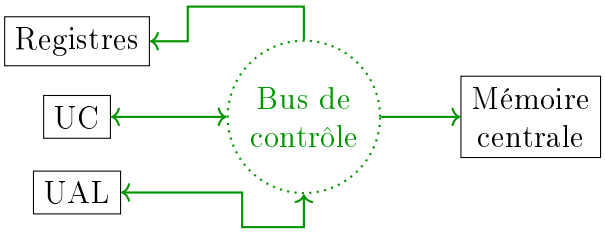


FIGURE 5.8 – Base de l'architecture de von Neumann (sans périphériques), où l'on a fait figurer les composants du processeur ainsi que le flux de signaux transitant par le bus de contrôle. Le cercle symbolise le fait que le flux d'information passe par un même bus, mais il ne correspond pas à un élément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

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, y compris la gestion des adresses pour le transfert de données.

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.

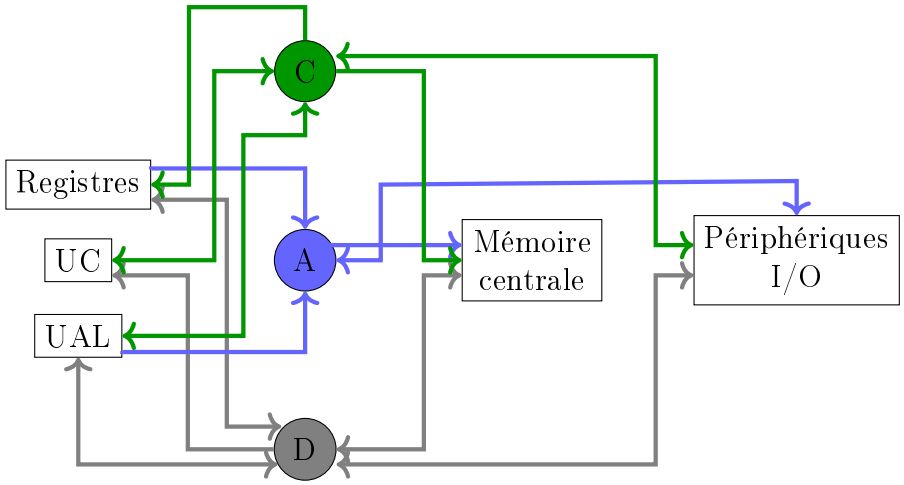


FIGURE 5.9 – Schéma abstrait de la base de l'architecture de von Neumann, où l'on a fait figurer le flux d'information. Le symbole C représente les signaux de contrôle, A les signaux d'adresse et D les signaux de données. Ces symboles ne correspondent pas à des éléments matériels mais illustrent le fait que les trois types de bus sont séparés physiquement. 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 ») 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 [4] 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,

4. En l'occurence, il est incrémenté du nombre d'octets sur lequel est encodée une instruction, typiquement 4 octets dans une architecture MIPS.

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 : 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.

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.

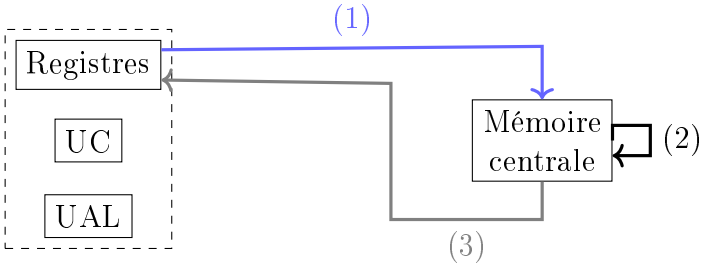


FIGURE 5.10 – Chargement d'une donnée depuis la mémoire centrale. (1) L'adresse de la donnée à charger est envoyée dans le bus d'adresse. (2) La donnée est récupérée au sein de la mémoire centrale. (3) La donnée est envoyée dans le bus de données et stockée dans un registre.

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 peut passer à d'autres instructions qui n'ont pas besoin d'attendre, puis revenir à l'instruction bloquée une fois que les données sont disponibles.
- Techniques de prédiction et pipelining – Certains processeurs utilisent la prédiction de branchements et la spéculation pour deviner les chemins d'exécution des programmes et commencer à exécuter des instructions avant que toutes les données ne soient confirmées comme nécessaires. Si la spéculation s'avère mauvaise, de l'énergie a été gaspillée, mais pas du temps ; 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. En effet, 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'attentes 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 (voire dans une mémoire auxiliaire persistante, qui ne serait plus auxiliaire en l'occurence). En raison du gain de performance drastique que procurent les registres (cf. table 5.2), cette solution n'est pas à retenir, et les registres, 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 grande, la quantité de données que l'on peut échanger avec elle est bien trop grande pour pouvoir être traitée directement par le processeur ou 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. Pour résumer, au cours de l'exécution d'un programme, toute donnée transite forcément par la mémoire centrale et par les registres ; il 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 réalité, 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. En particulier, l'utilisation de tableaux de valeurs, couplée à la nature continue de l'adressage au sein de la mémoire, est un cas typique où les caches sont très efficaces. L'utilisation des boucles 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. Ainsi, bien que pour un programmeur 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 cours du développement d'un programme.

5.5 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. Les instructions sont de longueur fixe et effectuent des opérations de base. Les processeurs ARM [5] 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, ce qui tend à être moins le cas aujourd'hui. Lorsque les technologies ont évolué, les architectures RISC ont gagné en popularité de par leur simplicité. Elles permettent par ailleurs une meilleure gestion de la consommation énergétique du processeur, bien qu'elles tendent toujours à faire une utilisation moins optimisée de la mémoire que les architectures CISC. D'une manière générale, un programme donné peut être formulé d'une façon qui favorise plutôt le nombre total d'instructions envoyées à la machine ou plutôt la vitesse d'exécution du code (grâce à une subdivision des instructions faisant bon usage de la mémoire et du flux d'informations). D'ailleurs, la façon dont les programmes sont traduits en code interprétable par la machine (cf. « compilation », chapitre 6.1.2) peut souvent être paramétrisée pour favoriser l'une ou l'autre de ces approches.

L'architecture de type RISC nommée « MIPS » [6] 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.

5. 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.

6. 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

Programmes et logiciels

Au cours du 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, stockées dans une mémoire persistante puis chargée au sein de la mémoire vive au moment 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 dépendemment 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.

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 donc 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.

Par « logiciel », nous entendons un programme existant sous une forme déjà exécutable par la machine de l'utilisateur.

6.1 Langages de programmation

Nous savons que les instructions, ainsi que toute donnée stockée dans la mémoire de la machine, sont codées en binaire. 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 composant les mots machines représentant les instructions envoyées au processeur, en s'astreignant aux règles d'interprétation que l'unité de contrôle utilise pour déchiffrer ces mots. Un tel langage est appelé langage **machine**.

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 et compactes, 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 5.5) 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.

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 de type RISC.

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`

Notons que, par souci de lisibilité, le tableau 6.1 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 C donne un exemple de traduction d'une ligne de code simple en C et en langage 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 la façon suivante : *<mnémonique>, <emplacement du stockage du résultat> <emplacement de la première opérande>, <emplacement de la seconde opérande>*. On s'aperçoit en regardant la table 6.1 que cette instruction en assembleur permet déjà de faciliter la vie du programmeur, puisque son équivalent en langage machine inclut six champs de bits, correspondant parfois à un aspect de l'instruction qui est implicitement compris au sein du mnémonique « sub ». En l'occurence, 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 C 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.

6.1.2 Compilation et interprétation

Dans ce cours, 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 ou le C++ 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 permettent au programmeur d'abstraire les sauts d'instructions de façon logique, en les liant à des conditions, sans devoir spécifier l'adressse des instructions à atteindre.
- 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.
- Les compilateurs modernes pour les langages de haut niveau sont capables d'effectuer des **optimisations** complexes qui seraient difficiles et fastidieuses à faire

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

manuellement en assembleur. Ces optimisations peuvent améliorer la performance du code sans effort supplémentaire de la part du développeur. Cependant, dans certains cas, 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, dans certains grands projets, mettre un certain temps non négligeable à produire le code machine, ce qui peut ralentir le développement d'un projet.

Notons qu'un programme donné doit être compilé pour chaque architecture matérielle sur laquelle il doit être exécuté. Par exemple, un jeu-vidéo destiné à être exécuté sur PC et sur téléphone 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 [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 (on parle alors de compilation Ahead-of-Time (AOT)). 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.

Il est courant de rencontrer l'emploi du terme « script » pour se référer de façon informelle à des codes courts, souvent écris dans des langages interprétés, qui sont souvent destinés à automatiser des tâches peu critiques du point de vue de la performance. Cependant, le terme n'a pas de signification rigoureuse et nous lui préférons le terme « programme ».

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.

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 que, 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 langage semi-interprété 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 rapides que les langages interprétés, car le code machine est directement exécuté par le processeur, sans passer par des couches d'interprétation supplémentaire. Néanmoins, les langages interprétés ont gagné en popularité en raison des avantages qu'ils procurent :

- 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.
- La gestion de la mémoire est le plus souvent gérée automatiquement par les interpréteurs, ce qui peut simplifier la tâche du programmeur, qui ne se soucie que de la logique de son programme, 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, 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 plus aisé 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 simplifier la tâche du programmeur débutant, mais peut aussi mener à des erreurs difficiles à repérer. À de nombreux égards, la spécification du type des données peut être vue comme un avantage plutôt qu'un inconvénient, et ce sujet est l'objet de nombreux débats parmi les spécialistes.

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.

En effet, il est important de remarquer qu'un langage de programmation n'est pas, en soi, compilé ou 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.

TABLE 6.2 – Exemples de langages de programmation ainsi que leur mode d'exécution standard. 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.

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

6.2 Systèmes d'exploitation

Dès les années 50, il devint évident que l'insertion physique des différentes parties des programmes constituait une tâche répétitive qu'il fallait automatiser ; une solution a donc été de regrouper les différentes parties du programmes 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 – afin d'automatiser la gestion des batchs. Ce programme, à exécuter avant tout autre, est appelé « moniteur de programme », précurseur des systèmes d'exploitation modernes.

Par la suite deux problèmes se posèrent en lien avec les périphériques (par exemple une imprimante pour afficher les résultats) :

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 code, doit gérer.
2. Les périphériques sont souvent beaucoup plus lents que le processeur, 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.

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 les logiques de code servant à la communication avec les périphériques. Le programmeur n'a plus à se soucier de ces détails, 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 dans ce laps de temps, ce qui permet de maximiser l'utilisation du processeur. C'est ce qu'on appelle une exécution **multitâche** (multitasking en anglais). Dans ce dernier cas, l'OS doit gérer la mémoire de façon à ce que les programmes n'entrent pas en collision, notamment en employant le concept de **mémoire virtuelle**, grâce auquel chaque programme peut être programmé « comme s'il était seul » sur la machine, en adressant les données à partir de l'adresse 0 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.
3. 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 programmes en cours d'exécution – ces séquences d'instructions sont appelées **processus**. L'annexe D propose une synthèse comparative entre multitasking, multithreading (cf. chapitre 5.4) et parallélisme (cf. chapitre 5.1.2), afin d'éviter les confusions.

Rapidement, les systèmes d'exploitation sont devenus incontournables dans la plupart des ordinateurs, 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** [3] : 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 et disposant d'un autre 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 des processus, etc. Au niveau de la programmation, les fonctions du systèmes 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. Sous Windows, ce sont des librairies logicielles (fichiers .dll), tandis

3. Il semblerait que le terme de *kernel* ait été employé, surtout dans le contexte d'Unix, de façon analogique 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.

que sous Unix, ce sont des fichiers .so, que l'on peut voir comme des bibliothèques de fonctions appelées par les logiciels applicatifs.

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 qui s'appuient sur le noyau pour fonctionner. Mentionnons tout de même que parmi ces logiciels applicatifs, l'interface graphique (graphical user interface, GUI) d'un OS est souvent étroitement liée à ce dernier. Cependant, il est tout à fait possible d'utiliser un OS sans interface graphique autre qu'une console.

À strictement parler, 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 ». Cependant, dans le langage courant, il arrive de rencontrer le terme « OS » employé pour désigner l'ensemble des logiciels fournis avec l'ordinateur, y compris les logiciels applicatifs. Les OS les plus célèbres incluent Windows, MacOS, Android, iOS ainsi que la famille des distributions de BSD et de Linux telles qu'Ubuntu. MacOS, Android et iOS descendent du système Unix, qui est l'un des premiers OS à avoir été développé, tandis que BSD et Linux sont des OS libres et open-source s'inspirante d'Unix et développés dans les années 80 et 90.

6.2.1 Pile et tas

Nous avons précédemment mentionné le rôle de l'OS dans la gestion de la mémoire attribuée aux différents logiciels applicatifs. Nous creusons ici un peu plus la façon dont la mémoire centrale est utilisée par ces derniers. Néanmoins, la description qui suit est une simplification généralisatrice de la réalité, étant donné que le but n'est que de donner au lecteur une idée générale de la façon dont la mémoire est attribue aux applications.

Au moment où un programme est exécuté, il se voit alloué un espace mémoire qui lui est propre, et qui est divisé en trois segments : les instructions (le code lui-même), la **pile** (stack en anglais) et le **tas** (heap en anglais).

Le tas sert essentiellement à stocker les valeurs qui doivent exister en dehors de toute fonction. Le tas est géré par le programmeur, qui doit la plupart du temps explicitement demander de l'espace mémoire pour stocker des données, et libérer cet espace une fois qu'il n'est plus nécessaire.

La gestion de la pile est normalement formulée automatiquement au moment de la compilation (ou le ramasse-miettes de l'interpréteur). Alors que les variables peuvent être récupérée dans un ordre arbitraire dans le tas, au sein de la pile seul le dernier objet ajouté peut être récupéré en premier. Cette logique de pile est souvent nommée LIFO, pour Last In First Out. Elle sert à stocker des données de taille fixe, telles que des variables locales, des adresses de retour, etc. 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.

Pour comprendre l'utilité de la pile, considérons le code C suivant :

```
int f(int x) {
   return x + 2;
}
int g(int x) {
   return x * x;
}
int main() {
    int A = 3;
    int result = f(g(A));
}
```

Si l'on veut exécuter ce code à la main et calculer la valeur de `result`, on s'aperçoit d'abord qu'il faut évaluer la fonction `f`. Ce résultat sera un entier, du fait de la définition de la fonction `f`. On peut donc 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. On peut donc noter une seconde ligne, au-dessus de la première, qui indique que `x` est un entier qui vaut `y * y`. On peut enfin remplacer `y` par `number`, qui est une variable locale de la fonction `main`. 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=3*3=9`, puis `result=x+2=9+2=11`. On peut alors effacer les lignes de la pile, qui ne servent plus à rien, et simplement retenir la conclusion `result=11`.

Ainsi, la pile augmente de taille à chaque fois qu'un appel de fonction se fait, 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 le début et la 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 est utilisé pour stocker ces valeurs.

Notons pour terminer que, du fait que la pile contient des adresse 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 et qui dépassent le 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 triplet algorithme-programme-ordinateur constitue parfois

le point central de l'attention. En effet, comme nous le verrons dans la partie du cours dévolue à l'algorithme 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.

Dans le langage de tous les jours, il est courant de se référer à la « puissance » d'un ordinateur, mais ce terme est en réalité assez vague. En effet, 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 D), ainsi que leur capacité à participer efficacement au programme sans se gêner mutuellement.
- 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** (disque dur, clavier, souris, etc.), qui peut être très long par rapport au temps d'exécution du programme.
- 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 programmes** peuvent occuper des ressources (mémoire, processeur, etc.) qui seraient autrement pleinement disponibles.

Il faut noter que la façon de formuler l'algorithme, et donc de l'implémenter, a un fort impact sur chacun des points considérés plus haut.

Nous allons ici nous préoccuper essentiellement du premier des points mentionnés. La fréquence d'horloge d'un processeur moderne est 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 instructions et que son temps d'exécution vaut T sur un processeur avec une fréquence d'horloge f, alors on peut écrire la relation :

$$T = \frac{n \cdot c}{f}, \tag{6.1}$$

où c s'interprète comme le **nombre d'instructions exécutées par cycle d'horloge**.

Outre le temps d'exécution, le nombre d'opérations en virgule flottante par seconde (floating point operations per second, **FLOPS**) 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 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 effectués. Les processeurs modernes effectuent quelques opérations par cycle d'horloge, et peuvent donc atteindre des performances de l'ordre de 10 GFLOPS.

En pratique, et surtout au sein de la communauté scientifiques, un certains nombres de « benchmarks » sont utilisés à titre d'étalon de la performance d'un ordinateur ou d'un programme. Ces benchmarks sont des programmes standardisés qui effectuent des opérations courantes, telles que des opérations matricielles, des opérations d'entrée-sortie, etc.

Depuis les débuts de l'informatique jusqu'au début des années 2000, la fréquence d'horloge des processeurs a augmenté de façon exponentielle (phénomène connu sous le nom de **loi de Moore**[4]), ce qui a impacté la performance des ordinateurs de façon significative. Cependant, certaines limites physiques ont été atteintes par la suite. En effet, la miniaturisation des 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'effet tunnel quantique 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 de nombreux calculs. Les cartes graphiques sont un exemple de spécialisation de composants pour effectuer des calculs de façon parallèle.

Ainsi, un graphique montrant l'évolution de la fréquence d'horloge des processeurs en fonction du temps exhibe un changement de pente depuis les années 2000, tandis qu'un graphique montrant l'évolution d'autres mesures de performance des ordinateurs en fonction du temps continue 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 6.1 et 6.2 montrent respectivement l'évolution de la fréquence des processeurs et du nombre de transistors par puce de processeur en fonction du temps.

4. 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.

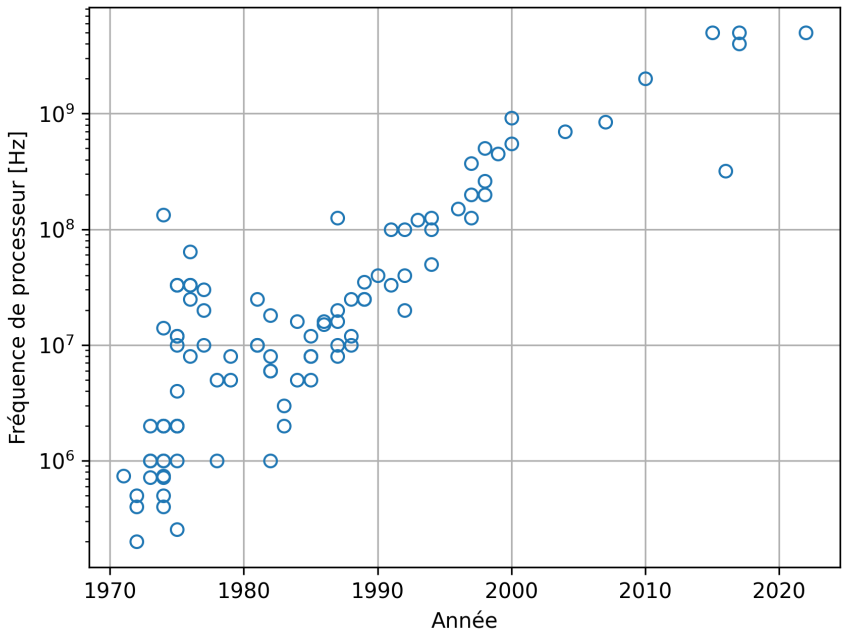


FIGURE 6.1 – Évolution temporelle de la fréquence des processeurs. Celle-ci est environ multipliée par 100 tous les 10 ans jusque dans les années 2000. Données issues de `https://en.wikipedia.org/wiki/Microprocessor_chronology`.

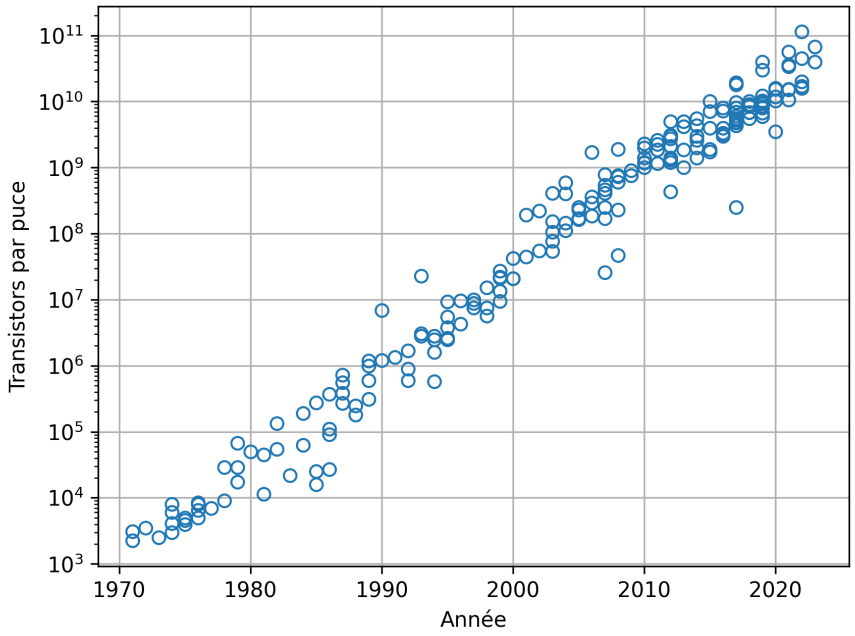


FIGURE 6.2 – Évolution temporelle du nombre de transistors par puce de processeur. Celui-ci est environ multiplié par 10 chaque décennie. Données issues de `https://en.wikipedia.org/wiki/Transistor_count`.

Chapitre 7

Exercices

Codage des nombres

Exercice 7.0 | Conversion d'entiers naturels

Convertir les nombres suivants en binaire :

1. 42
2. 123
3. 256
4. 1024

Exercice 7.1 | Conversion d'entiers naturels

Convertir les nombres suivants en décimal :

1. 1010_2
2. 1101_2
3. 1111_2
4. 10000_2

Exercice 7.2 | Conversion d'entiers naturels

Convertir les nombres suivants en hexadécimal :

1. 42
2. 123
3. 256
4. 1024

Exercice 7.3 | Conversion d'entiers naturels

Convertir les nombres hexadécimaux suivants en décimal et en binaire :

1. 2A
2. 7B
3. FF
4. 400
5. 5A0B

Exercice 7.4 | 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 1 en base n ?
6. Comment représenter n en base n ?
7. Comment représenter $n - 1$ en base n ?

Exercice 7.5 | Conversion d'entiers relatifs en complément à 2

Coder les nombres suivants en binaire, en utilisant la convention de complément à 2 et en utilisant le moins de bits possible :

1. 42
2. -42
3. 0
4. 123
5. -123

Exercice 7.6 | Conversion d'entiers relatifs en complément à 2

Convertir les nombres binaires suivants en décimal, sachant qu'ils ont été codés en utilisant la convention de complément à 2 :

1. 100
2. 0010
3. 1101
4. 1111
5. 10000

Exercice 7.7 | Addition d'entiers relatifs en complément à 2

Effectuez les additions en colonnes pour effectuer les opérations suivantes en binaire, en prenant soin d'utiliser la méthode du complément à 2 lorsque nécessaire. En cas de débordement, créez des colonnes supplémentaires si nécessaire.

1. 10 + 10
2. 7 + 7
3. 7 - 17
4. 17 - 4
5. 127 - 128
6. 65 - 120
7. 65 + 120

Exercice 7.8 | 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 ?

Exercice 7.9 | Conversion de nombres à virgule flottante en réels

Quel est le nombre réel en simple précision correspondant à la suite de bits suivante : 0 01111110 10000000000000000000000 ?

Exercice 7.10 | Conversion de nombres à virgule flottante en réels

La valeur d'un mot en mémoire est `0x404900DB`. Quel est le nombre réel correspondant en simple précision ?

Exercice 7.11 | Approximation de nombres réels en virgule flottante

...

Exercice 7.12 | Prévision d'erreur d'arrondi

Supposons le format de virgule flottante suivant : 1 bit de signe, 3 bits pour l'exposant et 4 bits pour la mantisse.

Codage des médias

Exercice 7.0

Proposez une convention pour encoder les lettres de l'alphabet suivant : A,B,C,x,y,z. Utilisez le moins de bits possible.

Exercice 7.1

Quel poids le texte suivant occupe-t-il en mémoire, sachant qu'on a créé un encodage exprès pour qu'il pèse le moins possible ? `Bonjour tout le monde !`

Exercice 7.2

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 7.3

Une photo est encodée 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 7.4

L'encodage RGB d'une photo permet d'exprimer 262'144 couleurs. Sachant que chaque couleur primaire est encodée sur autant de bits que les autres, trouvez le nombre de bits dévolu à chaque couleur primaire.

Exercice 7.5

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 encodage de la couleur de ses photos de façon à minimiser le poids des images dans la mémoire.

Exercice 7.6

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 7.7

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 ?

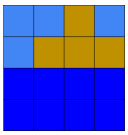


FIGURE 7.1 – *NB* : pour délimiter les pixels, on a dessiné une grille, mais celle-ci ne fait pas partie de l'image !

Exercice 7.8

Supposons que les images d'un film soient encodées 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'encodage des films ?

Exercice 7.9

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 7.10

Convertissez les couleurs (format RGB, 1 octet par couleur primaire) d'un format à l'autre dans le tableau ci-dessous.

Exercice 7.11

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 : l'encodage n'est pas forcément au format RGB

Exercice 7.12 |

Une image bitmap encodée 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 l'encodage 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 encodée 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 ?

Circuits logiques

Exercice 7.0 | 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. $(A+B)(!A+C)(B+C)$
6. $(A+AB)+(!BA)C$
7. $(A!BC)+(AB)$
8. $(A+B)(!A+!B)(A+C)$
9. $AB+!AC+B!C$
10. $(A(B+C))+(!A(C+!B))$

Exercice 7.1 | De circuit à expression logique

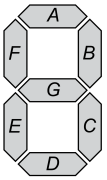
Quelle est l'expression logique réalisée par le circuit ci-dessous ? [Image]

Exercice 7.2 | Multiplexeur

Dessinnez un circuit logique combinatoire pour un multiplexeur à 2 voies.

Exercice 7.3 | 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 7.4 | 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.

Exercice 7.5 |

Donnez le diagramme d'un circuit logique réalisant un additionneur à $k = 2$ bits. Le diagramme doit être réalisé au même niveau d'abstraction que la figure 4.12 et les mêmes types de portes doivent être utilisés. Le nom de tous les bits d'input et d'output doit être indiqué sur le diagramme, c'est-à-dire r, a_1, a_0, b_1, b_0, s_0, s_1, R.

Architecture

Exercice 7.0

Au sein d'un ordinateur, le temps pris par le signal électrique entre deux éléments est en général masqué par des effets électriques qui sortent du cadre de ce cours, notamment le besoin de stabilisation du signal. Cependant, traitons ici d'un cas naïf où le signal électrique transite à la vitesse de la lumière ($c \approx 3 \cdot 10^8$ m/s) et s'installe instantanéement au sein d'un élément donné.

1. Calculez la fréquence (nombre d'aller-retour par seconde) maximale d'un processeur utilisant le cycle Fetch-Decode-Execute, sachant que la distance moyenne entre deux éléments du processeur est de 1 cm et qu'aucune information n'est à aller chercher ailleurs que dans les registres.
2. Même question, mais en considérant que l'information doit être accédée dans la mémoire centrale, située à 10 cm du processeur, 1 fois par cycle.

Exercice 7.1

Estimez l'ordre de grandeur du nombre de transistors contenus au sein d'un processeur moderne sur la base de ce qui a été dit en cours en prenant uniquement en compute l'UAL, les registres et les mémoires cache. Faites l'approximation qu'une porte logique quelconque contient environ 4 transistors, en vertu des discussions de la section 4.2.2 et 4.3.3.

Chapitre 8

Corrigés

Correction — Codage des nombres

Exercice 8.0 | Conversion d'entiers naturels

Convertir les nombres suivants en binaire :

1. 42
2. 123
3. 256
4. 1024

Exercice 8.1 | Conversion d'entiers naturels

Convertir les nombres suivants en décimal :

1. 1010_2
2. 1101_2
3. 1111_2
4. 10000_2

Exercice 8.2 | Conversion d'entiers naturels

Convertir les nombres suivants en hexadécimal :

1. 42
2. 123
3. 256
4. 1024

Exercice 8.3 | Conversion d'entiers naturels

Convertir les nombres hexadécimaux suivants en décimal et en binaire :

1. 2A
2. 7B
3. FF
4. 400
5. 5A0B

Exercice 8.4 | 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 1 en base n ?
6. Comment représenter n en base n ?
7. Comment représenter $n-1$ en base n ?

Exercice 8.5 | Conversion d'entiers relatifs en complément à 2

Coder les nombres suivants en binaire, en utilisant la convention de complément à 2 et en utilisant le moins de bits possible :

1. 42
2. -42
3. 0
4. 123
5. -123

Exercice 8.6 | Conversion d'entiers relatifs en complément à 2

Convertir les nombres binaires suivants en décimal, sachant qu'ils ont été codés en utilisant la convention de complément à 2 :

1. 100 Ici, $k = 3$ et le bit de signe est 1 donc le nombre est négatif. Interprété en tant qu'entier naturel, on a $100_2 = 4$. On calcule donc $|n| = 2^k - 4 = 2^3 - 4 = 4$. Par conséquent, $n = -4$.
   Seconde approche : $100_2 - 1 = 011_2$. Après inversion des bits, on obtient $100_2 = 4$. Le complément à 2 de 100_2 est donc 4.
2. 0010 $n = 2$
3. 1101 Ici, $k = 4$ et le bit de signe est 1 donc le nombre est négatif. Interprété en tant qu'entier naturel, on a $1101_2 = 13$. On calcule donc $|n| = 2^k - 13 =$

$2^4 - 13 = 16 - 13 = 3$. Par conséquent, $n = -3$.
Seconde approche : $1101_2 - 1 = 1100_2$. Après inversion des bits, on obtient $011_2 = 3$. Le complément à 2 de 1101 est donc -3.

4. 1111 On calcule $|n| = 2^4 - (2^4 - 1) = 1$. Par conséquent, $n = -1$.
5. 10000 On calcule $|n| = 2^5 - 2^4 = 16$. Par conséquent, $n = -16$.

Exercice 8.7 | Addition d'entiers relatifs en complément à 2

Effectuez les additions en colonnes pour effectuer les opérations suivantes en binaire, en prenant soin d'utiliser la méthode du complément à 2 lorsque nécessaire. En cas de débordement, créez des colonnes supplémentaires si nécessaire.

1. 10 + 10
2. 7 + 7
3. 7 - 17
4. 17 - 4
5. 127 - 128
6. 65 - 120
7. 65 + 120

Exercice 8.8 | 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 ?
On ne peut pas faire mieux que 4 chiffres décimaux. En effet, avec 3 chiffres on peut encoder seulement 1000 communes, tandis qu'avec 4 chiffres on peut en encoder 10'000. En binaire, on trouve par tâtonnement qu'il faudrait 12 bits. En hexadécimal, on trouve par tâtonnement qu'il faudrait 3 chiffres.

Exercice 8.9 | Conversion de nombres à virgule flottante en réels

Quel est le nombre réel en simple précision correspondant à la suite de bits suivante : 0 01111110 10000000000000000000000 ?

Exercice 8.10 | Conversion de nombres à virgule flottante en réels

La valeur d'un mot en mémoire est `0x404900DB`. Quel est le nombre réel correspondant en simple précision ?

Exercice 8.11 | Approximation de nombres réels en virgule flottante

...

Exercice 8.12 | Prévision d'erreur d'arrondi

Supposons le format de virgule flottante suivant : 1 bit de signe, 3 bits pour l'exposant et 4 bits pour la mantisse.

Correction — Codage des médias

Exercice 8.0 |

Proposez une convention pour encoder les lettres de l'alphabet suivant : A,B,C,x,y,z. Utilisez le moins de bits possible.

code	valeur encodée
000	A
001	B
010	C
011	x
100	y
101	z

Exercice 8.1 |

Quel poids le texte suivant occupe-t-il en mémoire, sachant qu'on a créé un encodage 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 encoder cet alphabet. Par ailleurs, la longueur totale du texte est de 24 caractères. Par conséquent, le poids du texte est de $T = 24 \cdot 4$ b $= 96$ b $= 12$ octets.

Exercice 8.2 |

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 8.3

Une photo est encodée 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 8.4

L'encodage RGB d'une photo permet d'exprimer 262'144 couleurs. Sachant que chaque couleur primaire est encodée 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 l'encodage RGB. Le nombre de couleurs N encodables 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 8.5

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 encodage 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 encodage RGB n'est pas nécessaire (ce serait du gaspillage de mémoire). On veut donc simplement pouvoir encoder 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, l'encodage proposé est le suivant : la première nuance de rouge correspond au nombre 0 encodé sur 10 bits, la seconde nuance de rouge correspond au nombre 1 encodé sur 10 bits, etc. La dernière nuance de rouge correspond au nombre 1023 encodé sur 10 bits.

Exercice 8.6

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 8.7

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 ?

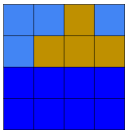


FIGURE 8.1 – *NB* : pour délimiter les pixels, on a dessiné une grille, mais celle-ci ne fait pas partie de l'image !

Il y a 3 couleurs différentes, par conséquent on peut choisir un encodage 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

$$T = 2 \cdot 16 = 32 \text{ bits}.$$

Exercice 8.8

Supposons que les images d'un film soient encodées 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'encodage 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 :

$$T = T_1 3,24 \cdot 10^5 \approx 8,07 \cdot 10^{12} \text{ octets} = 8070 \text{ GB}.$$

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 8.9

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 8.10

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 8.11

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 : l'encodage 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 8.12

Une image bitmap encodée 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 l'encodage 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 encodée 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. La condition se traduit par l'équation suivante, où l'on appelle n notre inconnue, qui est le nombre de bits par pixel :

$$n \cdot 10^6 = 500'000\ B$$

   Par conséquent :

$$n = \frac{500'000\ B}{10^6} = 0.5\ B = 4 \text{ bits}$$

   On doit donc coder chaque couleur sur 4 bits. Cela signifie qu'on ne peut plus utiliser une convention RGB avec le même nombre de bits pour chaque couleur primaire !

3. 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 — Architecture

Exercice 8.0

Au sein d'un ordinateur, le temps pris par le signal électrique entre deux éléments est en général masqué par des effets électriques qui sortent du cadre de ce cours, notamment le besoin de stabilisation du signal. Cependant, traitons ici d'un cas naïf où le signal électrique transite à la vitesse de la lumière ($c \approx 3 \cdot 10^8$ m/s) et s'installe instantanéement au sein d'un élément donné.

1. Calculez la fréquence (nombre d'aller-retour par seconde) maximale d'un processeur utilisant le cycle Fetch-Decode-Execute, sachant que la distance moyenne entre deux éléments du processeur est de 1 cm et qu'aucune information n'est à aller chercher ailleurs que dans les registres. On peut estimer que pour qu'un cycle complet puisse s'effectuer, il faut que le signal aille d'un élément à l'autre, puis retour, au moins 3 fois. La distance totale parcourue est donc de 6 cm. La distance parcourue par le signal en une seconde valant $d = 3 \cdot 10^8$ m, on trouve que le nombre de cycles effectuables chaque seconde vaut $f = (3 \cdot 10^8)/(6 \cdot 10^{-2}) \approx 5 \cdot 10^9$ Hz, c'est-à-dire 5 GHz.
2. Même question, mais en considérant que l'information doit être accédée dans la mémoire centrale, située à 10 cm du processeur, 1 fois par cycle. En appliquant le même raisonnement, on trouve que la distance parcourue par le signal pour effectuer un cycle est d'environ $2 \cdot 10$ cm $+ 4 \cdot 1$ cm. On trouve que le nombre de cycles effectuables chaque seconde dans ces conditions vaut $f = (3 \cdot 10^8)/(24 \cdot 10^{-2}) \approx 1.25$ GHz. Bien qu'aucun de ces deux cas de figure ne corresponde fidèlement à la réalité puisque des temps de latence de diverses natures puissent largement dominer la situation en réalité, on s'aperçoit que la fréquence des processeurs modernes stagne depuis bientôt deux décennies (pour d'autres raisons que le temps de transit du signal, cela dit) autour de quelques GHz, ce qui est cohérent avec ces estimations.

Exercice 8.1

Estimez l'ordre de grandeur du nombre de transistors contenus au sein d'un processeur moderne sur la base de ce qui a été dit en cours en prenant uniquement en compute l'UAL, les registres et les mémoires cache. Faites l'approximation qu'une porte logique quelconque contient environ 4 transistors, en vertu des discussions de la section 4.2.2 et 4.3.3.

- UAL : l'additionneur complet contient environ 30 transistors.
- Un registre 1 bit complet, fait d'un multiplexeur à 2 voies (environ 4 portes, donc environ 15 transistors) et d'un DFF (environ 20 portes donc environ 80 transistors), contient environ 100 transistors. Un registre 32 bits contient donc environ $32 \times 100 \approx 3000$ transistors.
- L'ensemble des registres, s'ils sont au nombre de 32 comme dans l'architecture MIPS, contiennent donc environ $32 \times 3000 \approx 100'000$ transistors. Nous avons négligé les multiplexeurs et démultiplexeurs pour l'adressage des registres. On voit que l'on peut ignorer la contribution de l'UAL.
- Comme on le pressent, c'est au final le nombre de transistors que nécessitent le niveau L3 du cache qui va déterminer l'ordre de grandeur du nombre de transistors nécessaires à la réalisation d'un processeur moderne. En effet, un cache L3 de 4 MB contient environ 10^6 mots mémoire, c'est-à-dire environ 10^9 transistors, si l'on se cale sur l'estimation d'un registre 32 bits.

Annexe A

Erreurs d'arrondi : code de démonstration

Dans le code C ci-dessous, on peut observer les erreurs d'arrondi résultant des additions successives d'un certain incrément. Il est important de noter que ces erreurs varient considérablement en fonction de l'incrément. Si ce dernier peut s'écrire sous la forme 2^{-n}, alors l'erreur d'arrondi sera nulle.

```c
#include <math.h> //pour utiliser fabs
#include <stdio.h>

int main() {
  float tot = 0.0f; // variable qui va stocker les additions successives
  int n = 1000000; // Nombre d'additions
  float increment = 0.3f; // Valeur ajoutée à chaque itération
  float expected = n * increment; // Résultat attendu

  // Additionne <increment> à <tot>, <n> fois
  for (int i = 0; i < n; i++) {
    tot += increment;
  }

  // Affiche le résultat (avec 6 chiffres après la virgule)
  printf("Somme calculée: %.6f\n", tot);
  printf("Valeur attendue: %.6f\n", expected);
  printf("Erreur d'arrondi: %.6f\n", fabs(tot - expected));

  return 0;
}
```

Nous donnons en page suivante une version équivalente en Python, où le type des variables a été explicité. Les erreurs d'arrondi ne sont pas nécessairement les mêmes qu'en C, pour les raisons discutées au sein du cours.

```
tot:float = 0.
n:int = int(1e6)
increment:float = 0.3
expected:float = n * increment

for i in range(n):
    tot += increment

print(f"expected: {expected}, tot: {tot}, diff: {expected - tot}")
```

Annexe B

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 les couleurs sont en ASCII, et qu'elles sont en 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 bleu vif a bleu sombre
0 255 0
0 187 0
0 118 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*.

```
0 50 0
# La quatrième ligne sera un dégradé du noir au blanc.
0 0 0
85 85 85
170 170 170
255 255 255
```

Annexe C

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 , 20( $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 D

Différentes formes de parallélisme

Nous comparons ici les différents termes employés tout au long du cours et 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 traiter plusieurs informations 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.

Multithreading : Se réfère 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.

Multitasking : Se réfère au fait de traiter plusieurs **programmes différents** en même temps. Les processus sont des parties d'un programme. 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.

Multiprocessing : Se réfère au fait d'utiliser plusieurs **processeurs distincts** pour traiter de l'information en parallèle. Cela peut se faire sur un seul ordinateur, ou sur plusieurs ordinateurs.

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 en général certains niveaux de cache (en général, L3) ainsi que la mémoire centrale. Les processeurs multicœurs permettent de traiter efficacement plusieurs tâches en parallèle, que ce soit des parties d'un même programme (multithreading) ou des programmes différents (multitasking).

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.