Département d'Informatique
Institut Universitaire de Technologie
Université de la Méditerranée

Eléments d'Architecture des Ordinateurs

Vincent RISCH

version 10.05, *chapitre 6 incomplet*

Conditions d'utilisation de ce document

[1] `http ://creativecommons.org/licenses/by-nc-nd/2.0/fr/`

Table des matières

Introduction

S'il fallait tenter de résumer brièvement par ses usages ce qu'est un ordinateur, le plus simple serait sans doute de le désigner comme un « bidule-automatisé-à-tout-faire-ou-presque ». Cette universalité avérée confère aux ordinateurs une part décisive dans le fonctionnement de bien des systèmes, de la programmation d'un grille-pain à la simulation de mondes virtuels, en passant par le pesage du linge dans une machine à laver, la correction automatique de trajectoire d'une automobile, l'estimation prévisionnelle d'un vote, le calcul d'un modèle météorologique, ou le séquençage du génome, pour ne citer que ces exemples... Pour autant, ce phénomène dont l'accélération a été remarquable au cours des cinquante dernières années, résulte d'une évolution historique de longue haleine conjugant à la fois découvertes scientifiques, contraintes économiques et développements technologiques, à la croisée de différentes branches des mathématiques et de la physique. De ce point de vue, il est intéressant d'envisager l'histoire de l'informatique sous l'angle d'une sorte de théorie de l'évolution des mécanismes de calcul : le développement de nouvelles idées scientifiques conjuguées aux évolutions économiques marquant une époque donnée forment un environnement favorisant ou contraignant l'émergence de nouvelles formes de machines.

Puisqu'il est question d'« automatisme », il est naturel de faire remonter *a posteriori* l'origine de l'informatique aux premiers dispositifs permettant d'effectuer une tâche sans intervention humaine. A cet égard, de nombreux mécanismes sont potentiellement en mesure de revendiquer le statut d'*ordinausore*... Et c'est sans aucun doute d'abord du côté du *décompte*, du *dénombrement*, et de la *mesure* que se font sentir la nécessité et la possibilité d'une mécanisation.

- La capacité de compter, et dès l'ère paléolitique, la nécessité de dénombrer (par exemple le nombre de têtes d'un troupeau), donne naissance aux premières techniques de numération : d'abord le système décimal issu, comme le signale avec humour [Tisserant03] de la première calculatrice de poche connue (la main), le système binaire connu en Chine trois mille ans avant J.C. (sous l'expression du Yin et du Yang) et dont on trouve trace sur les figures magiques de l'empereur Fou-Hi, puis la numération positionnelle accompagnée du zéro, développée en Inde environ trois cents ans avant J.C. et diffusée grâce à la traduction vers 820 après J.C. des ouvrages d'Al-Khuwarizmi, mathématicien de Bagdad. Parallèlement aux progrés de la numération, la mise au point de bouliers (en Asie) et abaques (bassin méditerranéen) facilitent et accélérent les calculs.
- La mesure précise du temps est par ailleurs l'un des premiers problèmes qui se soit prêté à une tentative d'automatisation. L'utilisation du cadran solaire, en quelque sorte énorme horloge naturelle, finira par céder la place à la conception d'horloges artificielles dont un exemple remarquable est la *clepsydre*, inventée par Ctésibios au troisième siècle avant J.C. : de l'eau s'écoule dans un réservoir contenant un flotteur qui pointe sur des graduations qui correspondent aux heures de la journée et de la nuit ; un pointeau flottant, intercalé entre le cylindre et la canalisation d'arrivée d'eau, assure la régularité du système de telle sorte

que d'une part chaque fraction mesurée soit égale à la suivante, et que d'autre part, deux clepsydres placées dans des conditions initiales similaires pointent à tout moment sur la même graduation. Des horloges mécaniques sont réalisées dès le dixième siècle, et au douzième siècle apparaissent en Hollande des carillons dont la mélodie peut être choisie, choix préfigurant la notion de programme. Au seizième siècle, les jacquemarts sont en mesure de marquer les heures. Au dix-septième siècle, Galilée (1564–1642) invente l'horloge à pendule qui améliore la régularité du mouvement, tandis que Huygens (1629–1695) conçoit un peu plus tard l'horloge à balancier.

– Les progrés techniques de l'horlogerie permettent d'envisager son utilisation dans le cadre d'autres champs d'application : le dix-huitième siècle voit l'avènement des *automates mécaniques*, dont un exemple célèbre est le joueur de flûte de Vaucanson, capable d'exécuter jusqu'à douze airs différents. Conçu à partir d'un dispositif pneumatique et mécanique, il comporte un cylindre à cames qui permet le mouvement des doigts suivant un ordre déterminé et qui conditionne les positions des lèvres pour obtenir une variation dans le débit produit par la soufflerie. Le programme est inscrit sur le cylindre qui contrôle les mouvements, le changement de cylindre permettant l'exécution d'airs différents. Le développement de l'industrialisation au début du dix-neuvième siècle, et la nécessité d'une automatisation des procédés de tissage, conduisent à la conception de métiers à tisser utilisant un codage binaire pour la reproduction de motifs à partir de cartes perforées (d'abord des planchettes en bois percées de trous pour le métier de Falcon en 1728, puis des cartes en carton perforées articulées pour le métier de Jacquard en 1850), procédé qui sera repris ultérieurement pour le codage de l'information sur les premiers calculateurs électroniques.
– Les progrés réalisés dans la fabrication des automates conjugués aux progrés mathématiques permettent à partir du quinzième siècle d'envisager la fabrication d'une forme particulière d'automate, destinée au calcul arithmétique. La machine à multiplier introduite par Napier (1550–1617), inventeur des logarithmes, constitue une transition entre le boulier et les premières machines à calculer mécaniques. La première machine capable d'effectuer additions, soustractions, et multiplications, est mise au point en 1623 par Wilhelm Schickard (1592–1635) à l'attention de Kepler ; cette *horloge calculante* est destinée au calcul des éphémérides. Additions et soustractions y sont réalisées grâce à un système de roues dentées, complétées par un procédé de Napier pour les multiplications. Elle permet de mémoriser des calculs intermédiaires, et une clochette prévient du dépassement de capacité de la machine. Elle disparaît en 1624, en raison de la guerre de trente ans qui ravage l'Allemagne du sud à cette période, et ne sera reconstituée qu'en 1960, à partir des plans d'origine. La machine à calculer la plus connue à cette époque est la *pascaline* due à Blaise Pascal (1623–1662) : il la conçoit en 1642 afin de décharger son père, collecteur des impôts, de calculs fastidieux. Basée également sur un système de roues dentées, elle effectue des additions et soustractions à six chiffres, et convertit différentes monnaies de l'époque. Elle est aussi capable de réaliser des multiplications par répétition des additions, comme le montrera Moreland en 1666. La pascaline est considérée comme le premier additionneur à retenue automatique. Gottfried Wilhelm Leibniz (1646–1716) construit en 1673 une machine plus perfectionnée, capable d'effectuer les quatre opérations et même d'extraire des racines carrées, le tout par une série d'additions sous la dépendance d'un compteur. Leibniz réinvente aussi le système binaire, et l'applique à l'étude des figures de Fou-Hi. La première commercialisation d'une machine à calculer, baptisée *arithmomètre*, due à Charles Xavier Thomas de Colmar (1785–1870) et conçue sur le principe de la machine de Leibniz,

intervient en 1821 : mille cinq cents exemplaires sont vendus en trente ans.

– La technologie des automates au début de l'ère industrielle est telle qu'elle semble pouvoir repousser toujours plus loin les capacités des machines. C'est dans ce contexte que Charles Babbage (1791–1871) tente de rapprocher machine à calculer et système de commande de Jacquard en vue de réaliser un mécanisme permettant l'exécution en séquence de plusieurs opérations arithmétiques, et donc la réalisation de calculs complexes. Il tente d'abord de réaliser une *machine à différence* destinée au calcul de tables de navigation ou de tir, mais face aux difficultés techniques et faute de moyens, le projet partiellement réalisé, est abandonné après une dizaine d'années de travail. Babbage a alors l'idée d'une machine plus générale, la *machine analytique*, permettant d'enchaîner l'exécution d'opérations arithmétiques en fonction d'instructions données par l'utilisateur. La conception de la machine analytique préfigure l'architecture générale des ordinateurs modernes : on y trouve une unité de traitement (le « moulin ») commandée par une unité de contrôle, une mémoire (le « magasin »), une unité d'entrée constituée de cylindres inspirés des jacquemarts et pouvant recevoir des cartes perforées (opérations et nombres), une unité de sortie pouvant tracer des courbes et perforer un résultat sur des cartes. Babbage sera aidé dans la conception de sa machine par Ada Lovelace (fille du poète Lord Byron) qui peut être considérée comme la première programmeuse de l'Histoire : c'est elle qui défini le principe de l'enchaînement d'itérations successives pour la réalisation d'une opération, baptisé *algorithme* en hommage au mathématicien Al-Khuwarizmi. La machine de Babbage est certainement le projet d'automate le plus ambitieux du dix-neuvième siècle, mais il ne sera pas réalisé du vivant de Babbage (un prototype sera partiellement réalisé par le fils de Babbage en 1888, avant qu'un nouvel exemplaire soit totalement reconstruit dans les années quatre-vingt à partir des plans d'origine). Au moins deux raisons expliquent l'échec de Babbage dans la tentative de réalisation effective de sa machine. D'une part la complexité du projet s'avère trop importante au regard de la technologie de l'époque. D'autre part, la machine qui traite de l'information sous forme digitale, va à contre-courant de la conception analogique du traitement de l'information en vigueur à l'époque, qui fait suite aux travaux de Joseph Fourier (1768–1830) sur la représentation des fonctions continues à partir de sommes trigonométriques.

– Alors que la machine de Babbage, malgré la modernité de certains des idées mises en oeuvre dans sa conception apparaît comme une impasse, de nouveaux développements scientifiques vont fournir les outils théoriques et technologiques permettant l'émergence d'une lignée de machine qui conduira à l'ordinateur. En 1854, Georges Boole (1815–1864) introduit la formulation mathématique des propositions logiques qui, appliqué au système binaire conçu par Leibniz, sera à la base du fonctionnement des ordinateurs. Les progrés accomplis en physique dans l'étude et la fabrication de l'éléctricité vont être décisifs. En 1890, pour les besoins du recensement de la population aux Etats-Unis, Hermann Hollerith (1860–1929) construit une machine électromécanique appropriée, utilisant des cartes perforées et pouvant compter et trier celles-ci afin d'établir des données statistiques. La détection ou la présence d'une perforation est réalisée au moyen d'aiguilles qui traversent les trous et ferment chacune un circuit électrique en trempant dans un godet de mercure. Cette machine est la première d'une lignée de machines mécanographiques, utilisées pour le calcul et surtout, la gestion. Hollerith fonde en 1896 la Tabulating Machine Company, qui devient en 1924 la célèbrissime I.B.M. (International Business Company). Dans le même temps, d'autres inventeurs améliorent les machines à calculer, dont William Burroughs

(1857–1898) et Door F. Felt (1862–1930). Ce dernier adapte aux machines à calculer le principe du clavier qui commence à être utilisé pour les machines à écrire. Sur la voie de Babbage, Leonardo Torres y Quevedo (1852–1936) propose la réalisation d'une version électromécanique de la machine analytique, mais faute de moyens, c'est un nouvel échec. Le rythme des innovations s'accélère. En 1930, Vannemar Bush construit au M.I.T. un *analyseur différentiel* capable de résoudre des équations différentielles rencontrées dans l'étude des circuits électriques. La même année, Georges Stibitz réalise un additionneur binaire à relais, le *modèle K*, s'appuyant sur les travaux de Boole. Néanmoins à ce moment, les ordinateurs analogiques, dont le fonctionnement est basé sur l'utilisation de quantités physiques continues telles que la tension, ou la vitesse de rotation des axes pour représenter les nombres, apparaissent comme le futur de l'informatique. Ces machines ont l'avantage, par rapport aux premières approches numériques, de pouvoir traiter des problèmes plus complexes, incluant même une certaine forme de parallélisme. Les débuts de l'électronique, à la veille et pendant la seconde guerre mondiale, vont changer la donne. De fait, chacun des progrés majeurs réalisé dans le domaine de l'électronique permettra la naissance d'une nouvelle *génération* de machines. Parallèlement, les logiciens Alan Turing et Alonzo Church définissent en 1936 chacun de leur côté des modèles théoriques équivalents qui délimitent les frontières de la calculabilité.

– La première génération d'ordinateurs voit le jour entre 1938 et 1956. Elle est liée à l'apparition des premiers tubes à vide, condensateurs et relais électroniques. En 1938, de façon étonnament solitaire, Konrad Zuse met au point une machine binaire programmable mécanique, le Z1. C'est le premier calculateur universel binaire contrôlé par programme, mais il ne fonctionnera jamais vraiment correctement, faute de crédits de développement.. En 1939, il perfectionne sa machine en remplaçant une partie des pièces mécaniques par des relais électromécaniques, et obtient ainsi le Z2. Le Z3 et le Z4 sont construits en 1941 et seront utilisés pour des calculs aéronautiques. Le Z4 a une mémoire de 512 mots de 32 bits. En 1939, John Atanasoff et Clifford Berry réalisent un additionneur binaire à 16 bits, utilisant pour la première fois la technologie des tubes à vide. Pendant la seconde guerre mondiale, Max Newman et ses collègues, parmi lesquels Alan Turing, concoivent *Colossus*, première machine programmable totalement électronique, composée uniquement de tubes à vide, et destinée au décryptage des codes de communication militaires allemands. L'existence de cette machine sera tenue secrète jusque dans les années 1970, et il apparaît que les travaux de l'équipe de Newman seront décisifs pour la victoire des alliés. En 1941, John Atanasoff et Clifford Berry construisent le premier ordinateur binaire à lampes, l'*ABC* (Atanasoff-Berry Computer). Parmi les derniers calculateurs électromécaniques, le *Mark 1*, développé en 1944 conjointement par I.B.M. et l'Université de Harward sous la direction de Howard H. Aiken (1900–1973) est une énorme machine inspirée de la machine de Babbage. Elle pèse cinq tonnes, mesure 16,60 mètres de long et 2,60 mètres de haut, occupant une surface de trente-sept mètres carrés, et consommant 25kW. Elle comporte 3000 relais et 760000 pièces mécaniques. Pour la petite histoire, en 1945, un insecte qui coince un relais provoquant un dysfonctionnement, est à l'origine du mot « bug », utilisé aujourd'hui universellement pour désigner une erreur qui s'est glissée dans un programme. En 1946, Prespert Eckert John Mauchly achèvent l'*ENIAC* (Electronic Numerical Integrator And Computer), souvent considéré comme le premier ordinateur électronique universel. Il est composé de 17468 tubes à vide, pèse 30 tonnes, occupe 72 mètres carrés, consomme 160 kW, est cadencé à 100 kHz et est capable d'exécuter 100000 additions ou 357 multiplications par secondes.

Différentes machines vont suivre : le *Manchester Mark 1* en 1949, l'*EDVAC*, puis l'*EDSAC*, qui contrairement à l'*ENIAC* ne posséde qu'une unité de calcul et tourne à 0,5MHz. En l'occurence, l'architecture des ordinateurs actuels dérive globalement de ces trois dernières machines. En 1952, *IBM* produit son premier ordinateur pour la défense américaine, l'*IBM 701*, puis en 1953, un ordinateur scientifique, l'*IBM 650*, et en 1955 le premier ordinateur commercial implémentant l'arithmétique des nombres à virgule flottante, l'*IBM 704*.

– L'invention du transistor par William Shockley, John Bardeen, et Walter Brattain en 1947, donne naissance à la seconde génération d'ordinateurs, dans lesquels les lampes, fragiles et encombrantes, sont remplacés par des composants plus petits et plus fiables permettant une diminution sensible de la taille et de la consommation. Le premier ordinateur utilisant des transistors est le *TRADIC* en 1955. L'apparition du circuit intégré (inventé par Jack St. Clair Kilby en 1958) va permettre de poursuivre la miniaturisation des composants, et susciter la troisième génération d'ordinateurs. On y trouve des machines telles que l'*IBM 360*, le *PDP 8* de *Digital Equipment Corporation*, l'*HP-2115* d'*Hewlett-Packard*. Enfin, l'invention du microprocesseur en 1971, suscite la quatrième génération d'ordinateurs et permet d'augmenter encore le degré de miniaturisation, jusqu'à l'apparition des premiers micro-ordinateurs, véritable révolution assurant à chacun un accés personnel à l'informatique. L'*INTEL 4004*, premier microprocesseur à 4 bits est en 1971 le premier circuit intégré incorporant unité de calcul, mémoire, gestion des entrées-sorties, et comporte 2300 transistors. Le premier micro-ordinateur est le *Micral N*, construit en 1973 par une petite entreprise française, *R2E*. En 1975, sort l'*Altair*, utilisant un processeur Intel 8080, en 1976 apparait l'*Apple I*, puis en 1977, l'*Apple II*. A partir, de 1978, *IBM* commence à s'intéresser au marché de la micro-informatique, et sort en 1981 le premier *IBM PC*. Suivront encore le *Commodore 64* et l'*Amiga*, de *Commodore International*, et les premiers clones compatibles PC lancés par *Compaq*. En 1983, *Apple* propose *Lisa*, premier micro-ordinateur doté d'une interface graphique, et précurseur du *Macintosh*, tandis que le PC compatible s'impose au grand public au travers de constructeurs comme *Hewlett-Packard*, *Compaq*, *Dell*, ou *NEC*.

Si l'évolution actuelle des machines en termes de performances est prévisible jusqu'à un certain terme (en particulier par la Loi de Moore), il est difficile de prévoir le futur de l'architecture des ordinateurs au delà du seuil connu d'intégration des composants. Les recherches menées actuellement ouvrent des pistes dans le domaine de la physique quantique, tandis qu'en bio-informatique des tentatives de codage et de manipulation de l'information sur les quatres bases fondant l'ADN ont été réalisées avec un certain succés. Il n'est pas impossible que les ordinateurs du futurs bénéficient d'une nouvelle révolution scientifique qui modifiera encore une fois le cours de leur évolution. Les chapitres suivants se contentent d'explorer succintement les concepts fondamentaux communs au standard des architectures actuelles. Par ailleurs, le lecteur intéressé par l'histoire de l'informatique et des ordinateurs peut se référer notamment à [Breton90], ainsi qu'à l'excellente page de l'encyclopédie en ligne [Wikipedia], dont est issue la majeure partie des informations de cette courte introduction.

Chapitre 1

Information et calcul

S'il est établi que l'informatique repose sur l'utilisation d'ordinateurs[1], *a contrario* il est évident que le rôle et le fonctionnement d'un ordinateur n'ont de sens qu'au regard de la notion de traitement de l'information. Ainsi les descriptions les plus généralistes d'un ordinateur s'accordent toutes sur le concept minimal de *machine conçue pour permettre le traitement de l'information.* L'imbrication des deux notions, informatique–ordinateur, amène donc directement aux questions :

- Qu'est-ce que l'information ?
- De quel traitement sagit-il ?

C'est à ces deux questions que tente de répondre de façon extrêmement sommaire le présent chapitre.

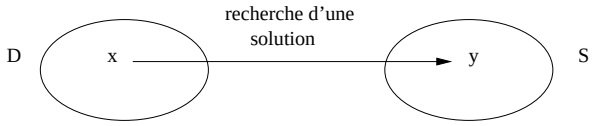
1.1 Traitement de l'information

En l'occurence, et pour aborder la première de ces questions, nous adoptons ici en première approche la définition proposée dans [MeyerBaudoin80] : on appelle *information* tout critère de choix parmi les éléments d'un ensemble, c'est-à-dire tout critère permettant de restreindre la taille d'un ensemble où l'on recherche la réponse à une certaine question.

Par exemple, sur des questions aussi diverses que le calcul de la résistance d'un pont en béton armé, le choix d'une orbite optimum pour un satellite de télécommunication, la traduction d'un texte d'une langue à une autre, des informations pertinentes seront au choix et respectivement la donnée du coefficient de résistance du béton, la longueur, la hauteur du pont, la densité du trafic de passage, ou la hauteur et le type de l'orbite, la masse du satellite, son autonomie, ou encore les lexiques respectifs des langues considérées, les règles de grammaire et de conjugaison et leurs exceptions...

A ce stade, apprendre que la couleur du pont est rouge, que le satellite porte l'inscription « made in France », ou que le texte à traduire contient 57 fois la lettre « e » n'apporte pas d'information puisqu'aucun moyen ne nous est ainsi fourni de réduire l'ensemble de recherche. La recherche d'une solution, c'est à dire le traitement de l'information par l'être humain, revient donc essentiellement en l'application d'un ensemble de méthodes permettant une *réduction* de la quantité d'information à partir des données de départ.

[1] Quoique ?

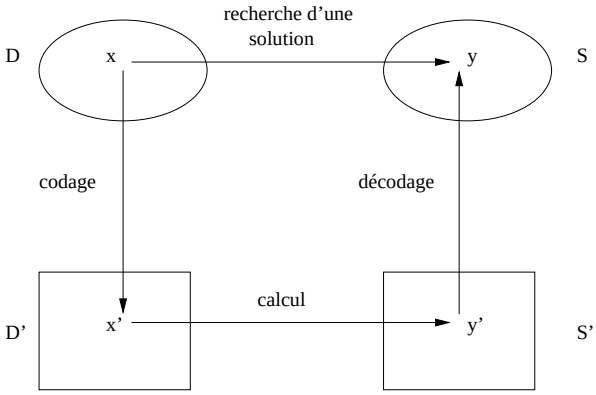


Nous sommes en mesure d'aborder la deuxième question : sur un ordinateur, « machine conçue pour permettre le traitement de l'information », il sagit alors de *simuler* d'une façon ou d'une autre ces méthodes de réduction de l'information. Pour ce faire, il faut disposer

- d'une représentation physique de l'ensemble des données D, soit D' ;
- d'une représentation physique de l'ensemble des solutions S, soit S' ;
- d'une représentation physique des méthodes de réduction de l'information, opérant sur D' et fournissant un résultat S' ; c'est à dire un *calcul*.

Pour pouvoir lancer le traitement physique et interpréter son résultat, on doit en outre disposer de deux codes de représentation :

- l'un pour représenter une donnée de D (abstraite) par un élément de D' (physique) ; c'est un *codage* ;
- l'autre pour interpréter un élément de S', résultat d'un calcul, comme la représentation d'une solution appartenant à S ; c'est un *décodage*.



y = décodage(calcul(codage(x)))

Un *programme* de traitement de l'information sera donc la donnée d'ensembles *physiquement représentables*[2] D' et S' conçus respectivement comme le codage et le décodage de D et S, d'un calcul opérant sur D' et S', tels que

$$solution(x) = décodage(calcul(codage(x)))$$

Dans la mesure où l'on montre que jusqu'à un certain point les opérations de codage et de décodage se résument elles même à un calcul, un ordinateur apparaît donc essentiellement comme un dispositif de calcul évolué. Mais qu'est ce qu'un calcul ?

[2] c'est-à-dire, entre autres, finis...

1.2 La machine de Turing

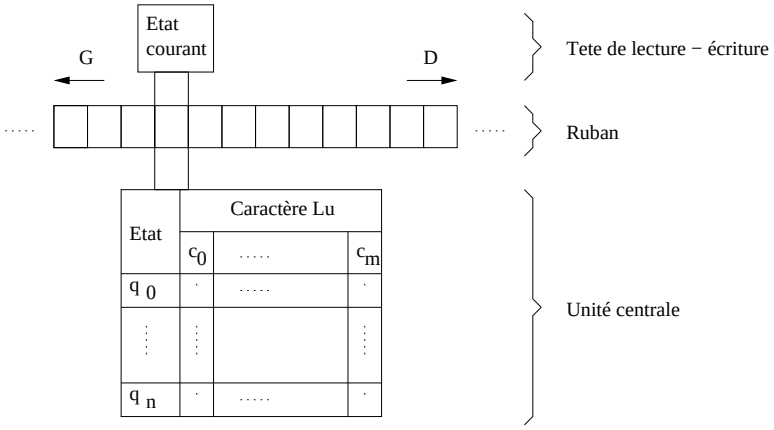
C'est Alonzo Church et Alan Turing, qui en 1936 et de façon indépendante apportent chacun avec leurs propres outils théoriques une réponse à cette question. Church propose un paradigme formel appelé Lambda calcul et montre que toute fonction récursive, ni plus ni moins, est représentable au sein de ce calcul. De son côté, Turing, bien avant la construction des premières calculatrices électroniques introduit un modèle théorique original appelé depuis lors *machine de Turing*. Turing montre l'équivalence de son modèle avec le lambda calcul, ce qui permet d'obtenir trois modèles de la calculabilité strictement équivalents :

- La théorie des fonctions récursives ;
- Le lamba calcul ;
- La machine de Turing.

De ces trois modèles, la machine de Turing est sans nul doute le plus intuitif, et c'est ce modèle que nous présentons ici.

Définition 1.1 *Une* machine de Turing *se compose précisément*

- *d'une* unité centrale*, laquelle peut prendre un certain nombre d'états internes en nombre fini ;*
- *d'un* ruban *de longueur infinie sur lequel sont inscrites les données de départ à traiter (des symboles pris dans un ensemble fini déterminé) et sur lequel la machine peut écrire de nouveaux symboles ;*
- *d'une* tête de lecture–écriture *qui opère sur un symbole du ruban à la fois, et qui, en fonction de l'état courant de l'unité centrale, peut remplacer le symbole lu par un nouveau symbole et éventuellement se déplacer d'une case vers la gauche ou vers la droite sur le ruban.*



La machine fonctionne par pas discontinus et instantanés. Dans une situation déterminée par un état interne et un symbole lu sur le ruban, elle évolue en changeant d'état et en agissant sur le ruban ; des règles de transition (autrement dit, un *programme*) déterminent cette évolution. Plus précisément :

Définition 1.2 *Un* programme *est complètement déterminé par :*

- *La donnée d'un ensemble fini* Γ *de symboles, incluant un sous-ensemble* $\Sigma \subseteq \Gamma$ *de symboles d'entrée, ainsi qu'un symbole spécial appelé « blanc », noté* b*, avec* $b \in \Gamma \setminus \Sigma$*.*
- *Un ensemble fini* Q *d'états, incluant un état spécial de départ* q_0 *et au moins un état final* q_F*.*

– *Une fonction de transition* $\delta : (Q \setminus \{q_F\}) \times \Gamma \to Q \times \Gamma \times \{G, D, \}$ *matérialisé par un ensemble de règles de la forme*

état courant, symbole lu $\to$ *nouvel état, symbole écrit, mouvement du ruban.*

L'*entrée* d'une machine de Turing est une séquence de symboles pris dans Σ telle que chaque symbole composant la séquence est placé un par un sur le ruban qui contient des caractères b partout ailleurs. Le *calcul* commence dans l'état initial q_0 et se termine dans l'état final q_F. On peut alternativement considérer deux états finaux q_Y et q_N tels que q_Y est atteint si la réponse à un programme de calcul est « oui », q_N est atteint si la réponse à ce programme de calcul est « non » (par exemple, telle séquence d'entrée possède-t'elle telle propriété ?).

Exemple 1.1 *Comment effectuer une addition sur une machine de Turing ? Une manière de faire (pas la seule...) est de coder les nombres à additionner sous forme de séquences de* 1 *successifs, par exemple* 2 *s'écrivant* 11, 3 *s'écrivant* 111, *et ainsi de suite... On considère donc* $\Sigma = \Gamma = \{1\}$, *et le caractère « blanc »* b. *L'idée est alors d'écrire un programme capable de concaténer les deux séquences à additionner en une seule qui en est l'union. Par exemple de la somme des séquences* 11 *et* 111 *résulte la séquence* 11111. *Un tel programme est, par exemple :*

$$\begin{aligned} q_0, b &\to q_0, b, D \\ q_0, 1 &\to q_1, b, D \\ q_1, b &\to q_2, 1, D \\ q_1, 1 &\to q_1, 1, D \\ q_2, b &\to q_F, b \\ q_2, 1 &\to q_F, 1 \end{aligned}$$

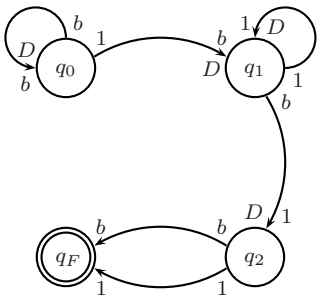
Cet ensemble de règles est une matérialisation de la fonction de transition : à gauche de la flèche se trouvent l'état courant de la machine et le symbole lu par la tête de lecture–écriture ; à droite de la flèche se trouvent l'état dans lequel passe la machine après lecture, le nouveau symbole à écrire sous la tête de lecture–écriture, et enfin le mouvement effectué par la tête sur le ruban (d'une case vers la gauche ou vers la droite, ou mouvement nul). L'exécution du programme commence dans l'état q_0 *et se termine dans l'état* q_F. *Des représentations équivalentes de ce programme existent :*

– *Sous forme d'une* table de transition *:*

Symbole lu / *Etat courant*	b	1
q_0	q_0, b, D	q_0, b, D
q_1	$q_2, 1, D$	$q_1, 1, D$
q_2	q_F, b	$q_F, 1$

Chacune des entrées de la table, en ligne (états) et en colonne (symboles lus sur le ruban), définit une situation correspondant à une des têtes de règle envisagées dans le programme. Les cellules de la table sont associées aux transitions définies par chacune des queues de règles correspondantes, et déterminent les transitions vers de nouveaux états ainsi que les opérations effectuées sur le ruban.

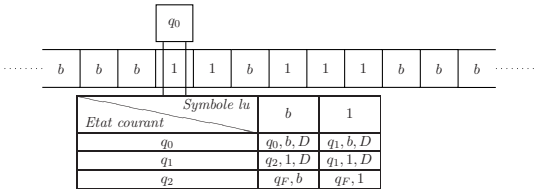
– *Sous forme de* diagramme d'états finis *:*



A chacun des états est associé un noeud du graphe, les transitions entre états ainsi que les opérations effectuées sur le ruban étant matérialisées par les arcs orientés reliant les noeuds. L'état final est signalé comme un noeud particulier.

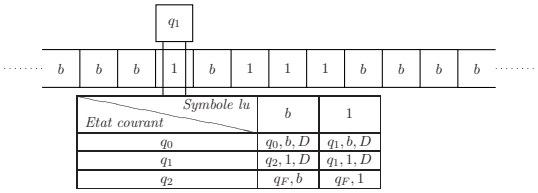
La trace de l'exécution du programme sur l'addition de 11 *et* 111 *est :*

Etape 1



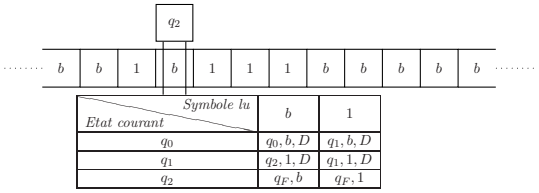
Etat courant \ *Symbole lu*	b	1
q_0	q_0, b, D	q_1, b, D
q_1	$q_2, 1, D$	$q_1, 1, D$
q_2	q_F, b	$q_F, 1$

Etape 2



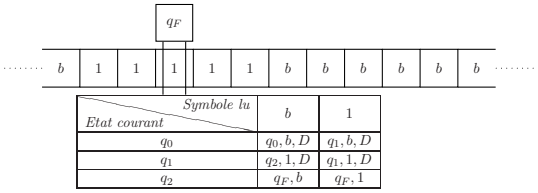
Etat courant \ *Symbole lu*	b	1
q_0	q_0, b, D	q_1, b, D
q_1	$q_2, 1, D$	$q_1, 1, D$
q_2	q_F, b	$q_F, 1$

Etape 3



Etat courant \ *Symbole lu*	b	1
q_0	q_0, b, D	q_1, b, D
q_1	$q_2, 1, D$	$q_1, 1, D$
q_2	q_F, b	$q_F, 1$

Etape 4



Etat courant \ *Symbole lu*	b	1
q_0	q_0, b, D	q_1, b, D
q_1	$q_2, 1, D$	$q_1, 1, D$
q_2	q_F, b	$q_F, 1$

Un calcul étant précisément ce qui peut être accompli sur une machine de Turing, la machine de Turing représente la plus petite architecture possible pour un ordinateur, et peut donc être vue sous son aspect fonctionnel comme l'intersection de tous les ordinateurs possibles. En délimitant précisément les frontières de la calculabilité, elle est théoriquement capable de simuler[3] tout ordinateur passé, présent ou futur. En fait et malgré sa simplicité, elle représente sous son aspect matériel bien plus qu'un ordinateur, car elle dispose à travers son ruban d'une capacité de mémorisation infinie.

La théorie des machines de Turing et l'étude de la calculabilité vont bien au-delà de cette présentation succinte[4]. Sans aller plus loin dans cette voie, nous illustrerons néanmoins à divers

[3] si nécessaire en un temps *très* long...

[4] Une étude plus approfondie peut être menée, par exemple, au travers les ouvrages de référence de Minsky [Minsky67] ou de Lewis et Papadimitriou [LewisPapadimitrou81]

moments le fonctionnement de certains éléments réels d'architecture par leur équivalent sous forme de machine de Turing.

Ayant abordé la calculabilité et donc pressenti la fonction fondamentale d'un ordinateur, la question est maintenant de savoir sous quelle forme réaliser effectivement une machine à calculer, ce à quoi s'attachent les chapitres qui suivent...

Chapitre 2

Numération et codage

Les machines à calculer auquelles on s'intéresse sont des dispositifs physiques dont le fonctionnement repose sur l'emploi du courant électrique. Cette simple remarque, aussi évidente qu'elle paraisse, est lourde de conséquences. En effet, l'information la plus simple que l'on puisse représenter sur un tel dispositif est (schématiquement) « le courant passe » ou « le courant ne passe pas[1] ». Autrement dit, l'ordinateur traite de l'information digitale, le support de l'information étant un système à deux états d'équilibre 0 et 1 correspondant aux informations élémentaires du système, appelées *bits* (contraction des mots anglais « binary » et « digit ». Toute information est alors représentée par une combinaison de bits. A n bits correspondent 2^n configurations binaires possibles différentes (le nombre d'applications d'un ensemble de cardinal n dans l'ensemble $\{0,1\}$, de cardinal 2). Un *mot* est une séquence $q_{n-1}\dots q_0$ de n bits, q_{n-1} étant le bit de *poids fort*, et q_0 le bit de *poids faible*. En particulier, un mot de 8 bits est appelé *octet*. Enfin, nous appelons désormais *codage* l'opération qui établit une correspondance biunivoque (appelée *code*) entre les informations à représenter et les configurations binaires possibles. L'ensemble des développements de ce chapitre s'appuient sur les résultats théoriques donnés en annexe B.

2.1 Le codage des caractères

De nombreux standards existent. Le plus simple et le plus répandu est le code ASCII (American Standard Code for Information Interchange). Il permet de représenter sur un octet[2] des données alphanumériques, dont les caractères latins, les chiffres décimaux, et d'autres informations comme, par exemple, le retour chariot (une table complète du codage ASCII est donnée en annexe A). Le code ASCII a évolué vers le standard ISO 8859–1 (Latin 1), qui utilise huit bits pour représenter entre autres les caractères accentués. Un autre standard très répandu est Unicode, qui utilise deux octets pour coder aussi des jeux de caractères non latins, cyrilliques, hébreu, asiatiques.

2.2 Le codage des entiers naturels

Un résultat élémentaire de numération sur les entiers nous dit que tout entier a s'exprime comme la décomposition polynômiale d'une suite de coefficients entiers dans une base b donnée

[1] En réalité ce sont des tensions électriques servant de seuils qui sont mesurées.
[2] En fait sept bits plus un bit de parité.

(cf. Théorème B.2, annexe B). Puisque nous ne disposons que des deux symboles 0 et 1, la base 2 s'impose comme le seul système de numération adapté à la représentation interne des entiers naturels. Autrement dit, avec $b = 2$, le théorème B.2, annexe B, devient :

Corollaire 2.1 (Théorème B.2) *Pour tout $a \in \mathbb{N}$, supérieur ou égal à 2, il existe une séquence unique $q_0 \dots q_{n-1}$ telle que :*

$$\begin{aligned} a &= q_{n-1}.2^{n-1} + \dots + q_2.2^2 + q_1.2 + q_0 \\ &= \sum_{i=0}^{n-1} q_i.2^i \end{aligned}$$

avec $q_{n-1} = 1$ et $q_i \in \{0, 1\}$ pour $0 \leq i \leq n-2$.

q_0, coefficient de plus petit indice, est le bit de *poids faible* ; q_{n-1}, coefficient de plus grand indice, est le bit de *poids fort*.

Exemple 2.1 *L'entier 43_{10} se décompose en base 2 :*

$$43_{10} = 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0$$

Autrement dit, 43_{10} s'écrit en binaire 101011_2.

Théorème 2.1 *Sur n bits, on peut représenter les entiers naturels a tels que $0 \leq a \leq 2^n - 1$.*

Preuve : Le plus grand entier représentable sur k bits est :

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Q.E.D.

Ainsi, sur un octet on peut représenter tous les entiers naturels de 0 à 255. Deux octets permettent de représenter tous les entiers naturels de 0 à 32 767. Un mot de 32 bits permet de compter de 0 à 4 294 967 295. On a aussi le résultat suivant[3] :

Corollaire 2.2 *Pour tout entier naturel a, il existe un entier n strictement supérieur à 0, tel que $2^{n-1} \leq a < 2^n$.*

Si l'utilisation du code binaire est proche de la réalité physique, il s'agit d'une représentation malcommode à utiliser pour l'être humain : la longueur du codage et le manque de discrimination dû à l'utilisation de deux caractères seulement est facilement source d'erreurs dans la communication. Aussi une représentation compacte, plus aisée à manipuler, mais permettant un passage immédiat au code binaire est évidemment bienvenue : le système de numération héxadécimale représente le meilleur choix pour tout système informatique qui utilise des mots dont le nombre de bits est divisible par 4, ce qui est le cas de la quasi-totalité des ordinateurs actuellement sur le marché (mots de 8, 16, 32, et 64 bits).

[3] Ce résultat se généralise à une base quelconque.

Représentation hexadécimale

Le système de numération héxadécimale requiert 16 chiffres : les 10 premiers chiffres sont empruntés au système décimal et les 6 derniers sont les lettres A à F de l'alphabet latin. La table suivante fournit l'équivalent de chaque chiffre héxadécimal dans le système de numération décimal et le code de numération binaire sur quatre bits (appelé *Décimal Codé Binaire* ou encore code *DCB*) :

Décimal	DCB	Héxadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

En hexadécimal, $b = 16$, et le théorème B.2 devient :

Corollaire 2.3 (Théorème B.2) *Pour tout $a \in \mathbb{N}$, supérieur ou égal à 16, il existe une séquence unique $q_0 \dots q_{n-1}$ telle que :*

$$\begin{aligned} a &= q_{n-1}.16^{n-1} + \ldots + q_2.16^2 + q_1.16 + q_0 \\ &= \sum_{i=0}^{n-1} q_i.16^i \end{aligned}$$

avec $q_{n-1} \in \{1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ *et*
$q_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ *pour* $0 \leq i \leq n-2$.

Exemple 2.2 *L'entier 43_{10} se décompose en base 16 :*

$$43 = 2.16^1 + B.16^0$$

Autrement dit, 43_{10} s'écrit en hexadécimal $2B_{16}$

2.3 Conversion d'un entier naturel

La conversion d'un nombre exprimé en décimal en une base b quelconque peut être réalisée par l'une des deux méthodes suivantes :

– la méthode des *divisions successives* ;
– la méthode des *soustractions successives*.

La mise en œuvre de chacune de ces deux méthodes n'est qu'une variation à partir du résultat du théorème B.1. La méthode des divisions successives calque la démonstration du théorème B.2. La méthode des soustractions successives repose, elle, sur le corollaire B.1.

Méthode des divisions successives

L'idée est de diviser par b l'entier a dont on veut obtenir la représentation dans la base b, puis de diviser le quotient obtenu par b, et de répéter l'opération jusqu'à ce que le quotient obtenu soit nul. La suite des restes des divisions successives correspond à la représentation de a dans la base b. En effet :

$$\begin{aligned} a &= q_{n-1}.b^{n-1} + q_{n-2}.b^{n-2} + \ldots + q_2.b^2 + q_1.b^1 + q_0 \\ &= \underbrace{(q_{n-1}.b^{n-2} + q_{n-2}.b^{n-3} + \ldots + q_2.2^1 + q_1)}_{a_1 \text{ (quotient)}}.b + \underbrace{q_0}_{\text{(reste)}} \end{aligned}$$

La dernière égalité dit que q_0, c'est à dire le chiffre le moins significatif de la représentation en base b de l'entier a, est le reste obtenu par suite de la division euclidienne de a par b. De la même façon, considérons le quotient a_1, obtenu lors de la première division. On a :

$$\begin{aligned} a_1 &= q_{n-1}.b^{n-2} + q_{n-2}.b^{n-3} + \ldots + q_2.b^1 + q_1 \\ &= \underbrace{(q_{n-1}.b^{n-3} + q_{n-2}.b^{n-4} + \ldots + q_2)}_{a_2 \text{ (quotient)}}.b + \underbrace{q_1}_{\text{(reste)}} \end{aligned}$$

où q_1, c'est à dire le prochain chiffre dans la représentation en base b, est donné par le reste de la division du quotient a_1 par b. En poursuivant les divisions, on suit l'algorithme de conversion de la représentation décimale d'un entier en base b. Plus précisément :

1. l'entier décimal a est divisé par b, ce qui donne pour quotient a_1 et pour reste q_0 ;
2. si $a_1 > 0$, il est divisé par b, ce qui donne pour quotient a_2 et pour reste q_1 ;
3. l'opération de division mentionnée ci-dessus est renouvelée sur chacun des quotients obtenus jusqu'à obtenir un quotient $a_m = 0$;
4. les restes obtenus successivement sont les chiffres de la représentation en base b de l'entier a, avec q_0 comme chiffre le moins significatif, q_1 comme chiffre suivant, et ainsi de suite.

Autrement dit, en base b, a s'écrit $q_{n-1} \ldots q_1\ q_0$ (avec $0 < q_{n-1} < b$ et $0 \leq q_i < b, 0 \leq i \leq n-2$).

Exemple 2.3 *Conversion du nombre* 43_{10} *en binaire :*

$$\begin{aligned} 43 &= 21.2 + 1 \\ 21 &= 10.2 + 1 \\ 10 &= 5.2 + 0 \\ 5 &= 2.2 + 1 \\ 2 &= 1.2 + 0 \\ 1 &= 0.2 + 1 \end{aligned}$$

$43_{10} = 2.(2.(2.(2.(2.0 + 1) + 0) + 1) + 0) + 1$ *donc* 43_{10} *s'écrit en binaire* 101011_2.

Méthode des soustractions successives

La méthode des soustractions successives est souvent plus rapide que la méthode des divisions successives, pour peu que soit connue la suite des puissances successives de la base b dans laquelle est effectuée la conversion. Cette remarque la rend plus particulièrement adaptée à la conversion en base 2. On rappelle (généralisation du corollaire B.1) que tout entier naturel a peut être encadré par deux puissances successives de b (par exemple en binaire, il existe $k \in \mathbb{N}^*$ tel que $2^{k-1} \leq a < 2^k$). Le principe de la méthode consiste à soustraire à a la plus grande puissance de b immédiatement inférieure à a, puis à répéter l'opération avec le résultat obtenu jusqu'à ce que le résultat de l'opération soit nul. Plus précisément :

1. La puissance de b inférieure à a la plus proche (par exemple b^m) est soustraite de a, ce qui donne pour résultat s_m ;
2. si $s_m > 0$, la puissance de b inférieure à s_m la plus proche (par exemple $b^p, p < m$)est soustraite de s_m, ce qui donne pour résultat s_p ;
3. l'opération de soustraction mentionnée ci-dessus est renouvelée sur chacun des s_i obtenus $(0 \leq i \leq m)$ jusqu'à obtenir un certain $i = z$ tel que $s_z = 0$;
4. les restes obtenus par soustractions successives sont les chiffres de la représentation en base b de l'entier a, avec s_m comme chiffre le plus significatif, s_p comme chiffre suivant, et ainsi de suite jusqu'à s_z le chiffre le moins significatif.

Autrement dit, en base b, a s'écrit $s_m\ s_p \dots s_z$.

Exemple 2.4 *Conversion du nombre* 43_{10} *en binaire :*

$$\begin{array}{rcccl} 2^5 & \leq & 43 & < & 2^6\,, \\ 2^3 & \leq & 11 & < & 2^4\,, \\ 2^1 & \leq & 3 & < & 2^2\,, \\ 2^0 & \leq & 1 & < & 2^1\,, \end{array} \qquad \begin{array}{rcl} 43 & - & 2^5 = 11 \\ 11 & - & 2^3 = 3 \\ 3 & - & 2^1 = 1 \\ 1 & - & 2^0 = 0 \end{array}$$

$$\begin{aligned} 43 &= 2^5 + 2^3 + 2^1 + 2^0 \\ &= 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 \end{aligned}$$

autrement dit 43_{10} *s'écrit en binaire* 101011_2.

Conversion binaire–hexadécimal

Les 16 chiffres du code hexadécimal se codent en binaire sur 4 bits. En utilisant cette correspondance on peut convertir de l'hexadécimal en binaire, à chaque chiffre héxadécimal correspond une combinaison binaire de 4 bits. Inversement, en découpant la combinaison binaire à convertir, par paquets de 4 bits à partir du poids le plus faible et en associant à ce paquet le chiffre hexadécimal correspondant, on peut convertir du binaire en héxadécimal.

Exemple 2.5 198_{10} *s'écrit* 11000110_2 *en binaire,* $\underbrace{1100}_{C}\ \underbrace{0110}_{6}$ *et donc* $C6_{16}$ *en hexadécimal.*
43_{10} *s'écrit* 101011_2 *en binaire,* $\underbrace{0010}_{2}\ \underbrace{1011}_{B}$ *et donc* $2B_{16}$ *en hexadécimal.*

2.4 0pérations binaires

Les opérations arithmétiques élémentaires d'un seul bit sont simples et suivent les tables présentées ci-dessous. Certaines situations impliquent la génération d'une *retenue* et son report vers le bit de poids supérieur le plus proche. La division binaire, semblable à la division décimale, s'effectue par une succession de comparaisons, de soustractions et de multiplications. Enfin, on notera les opérations particulières que sont le *décalage logique à gauche* et le *décalage logique à droite*.

Table d'addition binaire

$0+0$	$=$		0
$0+1$	$=$		1
$1+0$	$=$		1
$1+1$	$=$	$\underbrace{1}_{retenue}$	0

Exemple 2.6

$$\begin{array}{cccccc} & {}^{1}1 & {}^{1}1 & {}^{1}0 & 1 \\ + & & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 \end{array}$$

Table de soustraction binaire

$0-0$	$=$		0
$1-0$	$=$		1
$1-1$	$=$		0
$0-1$	$=$	$\underbrace{1}_{retenue}$	1

Exemple 2.7

$$\begin{array}{cccccc} & 1 & 0 & 1 & 1 \\ - & {}^{1} & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

Table de multiplication binaire

0×0	$=$	0
0×1	$=$	0
1×0	$=$	0
1×1	$=$	1

Exemple 2.8

$$\begin{array}{cccccccc} & & & 1 & 0 & 1 & 1 \\ \times & & & & 1 & 0 & 1 \\ \hline & & & 1 & 0 & 1 & 1 \\ & & {}^{1}0 & 0 & 0 & 0 & \\ & 1 & 0 & 1 & 1 & & \\ \hline & 1 & 1 & 0 & 1 & 1 & 1 \end{array}$$

Table de division binaire

$0 \div 0$	:	indéfini
$0 \div 1$	=	0
$1 \div 0$	:	indéfini
$1 \div 1$	=	1

Exemple 2.9

$$\begin{array}{cccc|ccc} 1 & 0 & 1 & 0 & 1 & 0 & \\ \cline{5-7} 1 & 0 & & & 1 & 0 & 1 \\ \cline{1-2} 0 & 0 & 1 & & & & \\ & 0 & 0 & & & & \\ \cline{2-3} & 0 & 1 & 0 & & & \\ & & 1 & 0 & & & \\ \cline{3-4} & & 0 & 0 & & & \end{array}$$

Opérations de décalages

Les deux opérations suivantes de décalage sont définies : *sll* (Shift Logical Left, ou décalage logique à gauche), *slr* (Shift Logical Right, ou décalage logique à droite). Etant donnée la représentation d'un entier sur n bits $(q_{n-1}\, q_{n-2} \ldots q_0)$, on a :

$$\begin{aligned} sll(q_{n-1}\, q_{n-2} \ldots q_0) &= q_{n-2} \ldots q_0\, 0 \\ slr(q_{n-1}\, q_{n-2} \ldots q_0) &= 0\, q_{n-1} \ldots q_1 \end{aligned}$$

L'interprétation arithmétique des décalages est donnée par le théorème suivant :

Théorème 2.2 *Etant donnée $q_{n-1}\, q_{n-2} \ldots q_0$ la représentation sur n bits d'un entier naturel a, on a :*

$$\begin{aligned} sll(q_{n-1}\, q_{n-2} \ldots q_0) &= 2 \times a \\ slr(q_{n-1}\, q_{n-2} \ldots q_0) &= a \div 2 \end{aligned}$$

Preuve : Immédiate, en effectuant les multiplications $q_{n-1}\, q_{n-2} \ldots q_0 \times 2$ et $0\, q_{n-1} \ldots q_1 \times 2$. *Q.E.D.*

Dépassement de capacité

L'addition est l'opération fondamentale effectuée par l'UAL (Unité Arithmétique et Logique) d'un ordinateur : toutes les autres opérations, soustraction d'entiers naturels, addition et soustraction d'entiers relatifs, multiplication, division devront être conçues pour exploiter efficacement l'additionneur naturel. Or toute addition effectuée par l'UAL l'est sur un nombre fixe de bits correspondant par convention à la taille maximum du type d'entier représentable (par exemple, souvent 16 bits pour un entier, 32 bits pour un entier long...). Lorsque l'addition UAL

produit une retenue au delà du $n^{\text{ème}}$ bit de poids le plus fort autorisé par la représentation, il se produit un *dépassement de capacité* : le résultat de la somme des deux entiers considérés n'est pas représentable sur la taille de mot choisie, et le résultat produit est erroné. Par exemple, sur quatre bits, l'addition UAL de 0111 et 1100 donne 0011 avec retenue 1 : en effet, $7 + 12 = 19$ qui ne peut être représenté sur quatre bits. Toutefois la détection d'un dépassement de capacité sur l'addition de deux entiers naturels est immédiate : notons par $\oplus$ l'addition telle qu'effectuée conventionnellement par l'UAL sur une taille fixe de mots de n bits ; a_2 et b_2 étant les représentations binaires des entiers naturels a et b, on a :

$$a_2 + b_2 = a_2 \oplus b_2 + (R.2^n)_2 \quad \text{où } R \in \{0, 1\}$$

Si l'addition de deux entiers naturels produit un dépassement de capacité, $R = 1$, sinon $R = 0$. Nous verrons que ce critère n'est plus aussi simple si l'on considère la somme d'entiers relatifs.

2.5 Représentation des nombres signés

On a vu que n bits permettent de parcourir un ensemble de 2^n configurations binaires permettant de représenter tout entier naturel compris entre 0 et $2^n - 1$. La représentation des entiers signés implique de diviser cet ensemble de configurations par 2, car on veut pouvoir représenter autant d'entiers positifs que d'entiers négatifs, ce qui nécessite d'utiliser une moitié des configurations binaires disponibles pour les entiers positifs et l'autre moitié pour les entiers négatifs : 2^{n-1} configurations seront donc dévolues à la représention des entiers négatifs et 2^{n-1} configurations resteront disponibles pour la représentation des entiers positifs.

$$\underbrace{-2^{n-1} \longleftrightarrow 0 \longleftrightarrow 2^{n-1} - 1}_{2^n \textit{ valeurs distinctes}}$$

Autrement dit, $n - 1$ bits seront utilisés pour coder les entiers positifs comme les entiers négatifs, la distinction sur le signe découlant donc du positionnement à 1 ou 0 du $n^{\text{ème}}$ bit de poids fort, appelé dès lors *bit de signe* :

$$\underbrace{q^{n-1}}_{\textit{bit de signe}} \qquad \underbrace{q^{n-2} \ldots q^1\ q^0}_{\textit{valeur absolue } (n-1 \textit{ bits})}$$

Afin de rester cohérent avec la représentation des entiers naturels donnée précédemment, on décide conventionnellement de positionner le bit de signe à 0 pour les entiers positifs et à 1 pour les entiers négatifs. La question qui reste alors à résoudre est celle de l'ordre à adopter sur les 2^{n-1} configurations binaires utilisées pour coder les entiers négatifs (bit de signe est à 1). Trois solutions sont habituellement envisagées :

- le codage en *signe et valeur absolue*
- le codage en *complément à* 1
- le codage en *complément à* 2

Codage en signe et valeur absolue

Il s'agit du codage le plus intuitif. L'idée est d'associer une seule configuration binaire à la *valeur absolue* de chaque entier, la distinction entre la représentation négative et la représentation

positive de cet entier étant assurée par le bit de signe. Par exemple, sur quatre bits, on codera $+1$ par 0001 et -1 par 1001. L'inconvénient de ce codage est qu'il implique deux représentations de zéro : une représentation positive donnée par 0000 et une représentation négative donnée par 1000. Il y a donc gaspillage d'une configuration binaire. Mais surtout, les opérations algorithmiques utilisées dans la mise en œuvre de l'addition et de la soustraction sont différentes. Malgré sa simplicité apparente, il s'agit à tout point de vue d'un codage peu économique, mais dont on verra une application dans le cadre du cadre du codage des nombres flottants.

Codage en complément à 1

Dans le codage en complément à 1, le changement de signe s'effectue en inversant les 0 et les 1 : toute occurence de 0 devient 1, et toute occurence de 1 devient 0. Par exemple, sur quatre bits, $+1$ étant codé par 0001, -1 est codé 1110. On constate, en procédant ainsi qu'une moitié des configurations binaires disponibles est effectivement dévolue à la représentation des entiers négatifs et que le bit de poids fort est bien le bit de signe. Par contre, tout comme dans le codage en signe et valeur absolue, deux représentations distinctes de zéro coexistent : par exemple sur quatre bits, un zéro positif codé 0000, et un zéro négatif codé 1111. Il en découle le même type de difficulté dans la mise en œuvre des opérations arithmétiques élémentaires.

Codage en complément à 2

L'idée à la base du complément à deux est précisément l'obtention d'une représentation des entiers négatifs qui permette un codage uniforme des opérations d'addition et de soustraction : un entier négatif est tel qu'additionné à l'entier positif correspondant, on obtienne bien un zéro unique, le même que celui obtenu lors de la soustraction de cet entier par lui même. Autrement dit, la représentation de l'entier négatif en question doit pouvoir être obtenue en soustrayant la représentation positive de cet entier à zéro. Par exemple sur quatre bits, $+1$ étant codé 0001, on aura :

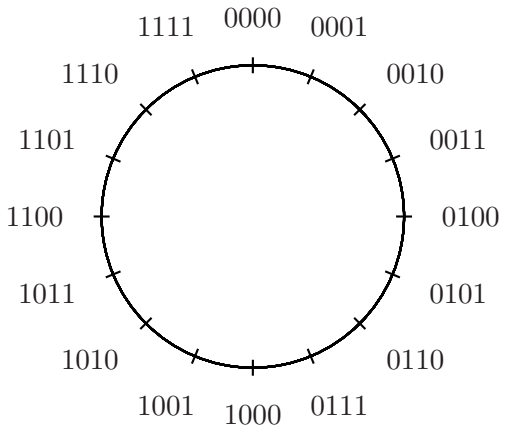
$$\begin{array}{rcccc} & 0 & 0 & 0 & 1 \\ - & 0 & 0 & 0 & 1 \\ \hline & 0 & 0 & 0 & 0 \end{array}$$

Le codage de -1 est alors normalement obtenu par la soustraction $0 - 1$ en binaire :

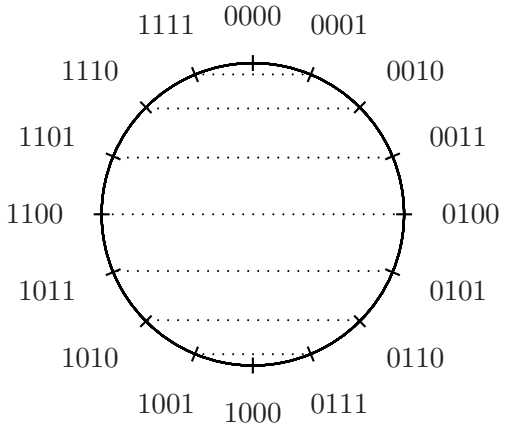
$$\begin{array}{rcccc} & 0 & 0 & 0 & 0 \\ - & {}^{1}0 & {}^{1}0 & {}^{1}0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 \end{array}$$

Il est facile de constater à partir de cet exemple que la soustraction de tout entier positif à zéro génère une retenue qui se propage au-delà du bit de poids le plus fort, correspondant à un dépassement de capacité. En l'occurence, c'est bien la génération de cette retenue qui permet de diviser l'ensemble des configurations binaires en deux et permet de constituer le bit de poids fort en bit de signe, abstraction faite du dépassement de capacité. Si on ignore le dépassement de capacité, on obtient bien une représentation des entiers négatifs compatible avec les comportements attendus de l'addition et de la soustraction. Or, du point de vue arithmétique, ignorer un dépassement de capacité sur le $n + 1^{\text{ème}}$ bit revient à considérer un codage des entiers modulo 2^n. Autrement dit, on représente 2^n configurations binaires distinctes sur un cercle, chacune représentant une des classes d'équivalence de la relation de congruence modulo 2^n (cf. annexe B, paragraphe B.4).

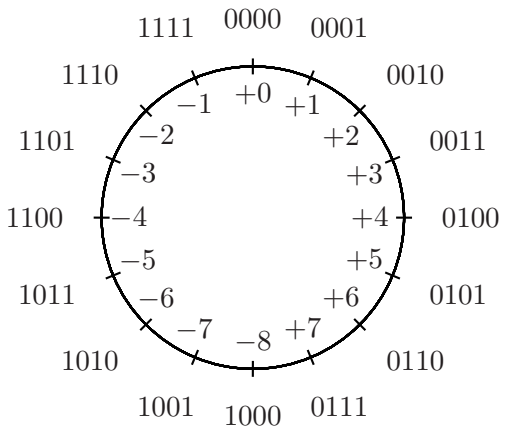
Par exemple sur quatre bits, c'est-à-dire pour $n = 4$:



Une moitié des configurations accessibles sert à représenter les entiers positifs (de 0 à $2^{n-1} - 1$), l'autre moitié les entiers négatifs (de -2^{n-1} à -1). Puisque la relation de congruence est compatible avec l'addition, toute configuration x accessible sur le cercle est telle qu'additionnée avec la configuration représentant sa valeur opposée, $\bar{x}$, on obtienne 0 modulo 2^n : $x + \bar{x} \equiv 0 \pmod{2^n}$. Par exemple : $0001 + 1111 \equiv 0 \pmod{2^n}$ et par conséquent 1111 est la représentation sur quatre bits de -1. Sur le cercle modulo 2^n, il est aisé de constater que chaque paire de valeurs opposées se trouve sur une même ligne horizontale :



Sur quatre bits, on obtient donc le codage suivant des entiers relatifs sur le cercle :



Le théorème suivant permet le calcul de la représentation en complément à deux d'un entier naturel à partir de sa représentation en complément à 1 :

Théorème 2.3 *Etant donné un entier naturel* $a \in [-2^{n-1}+1, 2^{n-1}-1]$, *la représentation de* $-a$ *en complément à* 2 *est obtenue à partir du complément à* 1 *de* a *auquel on ajoute* 1.

Preuve : Supposons $a > 0$ (sinon il suffit d'inverser les rôles de a et $-a$). On a

$$a = \sum_{i=0}^{n-1} q_i.2^i$$

Le nombre a' obtenu en ajoutant 1 au complément à 1 de a est :

$$a' = 1 + \sum_{i=0}^{n-1} \bar{q}_i.2^i$$

en notant $\bar{q}_i$ le conjugué du bit q_i, et en remarquant que l'opération ne produit pas de retenue pour $a > 0$. Par conséquent

$$a + a' = 1 + \sum_{i=0}^{n-1} (q_i + \bar{q}_i).2^i = 2^n$$

donc a' est la représentation de $-a$. *Q.E.D.*

Exemple 2.10

a	:		*0*	*0*	*0*	*1*
Complément à 1	:		*1*	*1*	*1*	*0*
ajoût de 1	:	+				*1*
-a	:		*1*	*1*	*1*	*1*

Extension de signe

Théorème 2.4 *Etant donné un entier relatif a représenté sur n bits par $q_{n-1}\, q_{n-2} \dots q_0$, a est représenté sur m $(m > n)$ bits par $q_{n-1} \dots q_{n-1}\, q_{n-2} \dots q_0$.*

Preuve : La proposition est évidente si $a \geq 0$. Si $a < 0$, les deux nombres ont le même complément à 2. *Q.E.D.*

Cette opération est appelée *extension de signe* : le bit de poids fort q_{n-1} est recopié sur les $m - n$ bits manquants. Par exemple, 7 est représenté par 0111 sur quatre bits, et par 00000111 sur un octet ; -2 est représenté par 1110 sur quatre bits, et par 11111110 sur un octet.

Dépassement de capacité

Si la représentation en complément à deux permet d'unifier addition et soustraction des entiers relatifs, elle ne résoud pas pour autant la question du dépassement de capacité qui peut se produire au sein de l'UAL. La question est de savoir si l'addition UAL est compatible avec la représentation en complément à deux, ou plus précisément, si l'addition binaire est isomorphe à l'addition UAL : *Comp2* désignant la fonction qui à un entier relatif associe sa représentation en complément à 2, a-t'on

$$Comp2(a + b) \stackrel{?}{=} Comp2(a) \oplus Comp2(b)$$

On sait d'ores et déjà que cela n'est pas toujours vrai pour les entiers positifs. Cela ne l'est pas non plus pour les entiers négatifs : par exemple sur quatre bits, l'addition UAL de -7 (1001) et -6 (1010) donne 0011, soit 3 et non -13. De même, l'addition UAL de 7 (0111) et 2 (0010) donne 1001, qui représente -7 et non 9 : on voit que le critère de détection d'un dépassement de capacité sur l'addition des entiers naturels (génération d'un bit de retenue sur le $n^{\text{ème}}$ bit) n'est pas applicable tel quel aux entiers relatifs. Pour l'addition des relatifs, le théorème suivant [GermainEtiemble] fournit un critère de correction qui généralise le critère obtenu pour l'addition des entiers naturels.

Théorème 2.5 *L'addition UAL de deux entiers relatifs a et b fournit toujours un résultat correct si a et b ne sont pas de mêmes signes. Si a et b sont de même signe, le résultat est correct si le bit de signe est égal à la retenue.*

Preuve : Il s'agit de vérifier que

$$Comp2(a + b) = Comp2(a) \oplus Comp2(b) \tag{2.1}$$

si et seulement si les conditions du théorème sont vérifiées. On distingue trois cas :

Cas 1 : $(a \geq 0, b < 0)$. Alors $-2^{n-1} \leq a + b < 2^{n-1}$ car $0 \leq a < 2^{n-1}$ et $-2^{n-1} \leq b < 0$; donc $a_2 + b_2$ est représentable. On a $Comp2(a) = a_2, Comp2(b) = (2^n + b)_2 = (2^n)_2 + b_2$ d'où

$$Comp2(a) \oplus Comp2(b) = a_2 + b_2 + (2^n)_2 - (R.2^n)_2$$

où R est le dépassement de capacité généré par l'addition binaire $a_2 + b_2$.

– Il y a retenue si $a + b + 2^n \geq 2^n$, autrement dit, si $a + b \geq 0$. D'une part $Comp2(a+b) = (a+b)_2 = a_2 + b_2$, car $a + b \geq 0$ est représentable. D'autre part, $Comp2(a) \oplus Comp2(b) = a_2 + b_2$ puisque $R = 1$. Par conséquent $Comp2(a + b) = Comp2(a) \oplus Comp2(b)$.

– Il n'y a pas retenue si $a+b+2^n < 2^n$, autrement dit, si $a+b < 0$. Dans ce cas $Comp2(a+b) = a_2+b_2+(2^n)_2$, car $a+b<0$ est représentable. On a aussi $Comp2(a) \oplus Comp2(b) = a_2+b_2+(2^n)_2$ car $R=0$. Par conséquent $Comp2(a+b) = Comp2(a) \oplus Comp2(b)$.

Cas 2 : ($a \geq 0, b \geq 0$). Alors $0 \leq a+b < 2^n$; par conséquent $a+b$ est représentable si $0 \leq a+b < 2^{n-1}$ et non représentable sinon. On a $Comp2(a) = a_2, Comp2(b) = b_2$. L'addition UAL ne produit jamais de retenue : les bits de poids fort sont tous deux à 0, d'où

$$\begin{aligned} Comp2(a) \oplus Comp2(b) &= a_2 + b_2 - (R.2^n)_2 \\ &= a_2 + b_2 \end{aligned}$$

puisque $R=0$. D'autre part $Comp2(a+b) = a_2+b_2$ si $a+b$ est représentable, puisque $a+b \geq 0$. Donc l'équation (2.1) est vraie si et seulement si $a+b \leq 2^{n-1}$, autrement dit si le bit de poids fort est égal à 0, comme la retenue.

Cas 3 : ($a < 0, b < 0$). Alors $-2^n \leq a+b < 0$; par conséquent $a+b$ est représentable si $2^{n-1} \leq a+b < 0$ et non représentable sinon. On a $Comp2(a) = a_2 + 2_2^n, Comp2(b) = b_2 + (2^n)_2$. L'addition UAL produit toujours une retenue, les bits de poids fort étant tous deux à 1, d'où

$$\begin{aligned} Comp2(a) \oplus Comp2(b) &= a_2 + (2^n)_2 + b_2 + (2^n)_2 - (R.2^n)_2 \\ &= a_2 + b_2 + (2^n)_2 \end{aligned}$$

puisque $R=1$. D'autre part $Comp2(a+b) = a_2+b_2+(2^n)_2$ si $a+b$ est représentable, puisque $a+b<0$. Donc l'équation (2.1) est vraie si et seulement si $a+b > -2^{n-1}$, donc si $2^n + a = b > 2^{n-1}$, autrement dit si le bit de poids fort est égal à 1, comme la retenue.

Q.E.D.

2.6 Représentation des nombres fractionnaires

Tout nombre fractionnaire s'écrit comme la somme d'une partie entière (supérieure ou égale à 1) et d'une partie fractionnaire (comprise entre 0 et 1). Or, le principe de la décomposition polynômiale d'un entier en base b s'étend de façon immédiate à la partie fractionnaire d'un nombre, exprimée comme une somme de puissances *négatives* de b (puisque par définition la partie fractionnaire est inférieure à 1). Autrement dit :

$$\begin{aligned} a &= q_{n-1}.b^{n-1} + q_{n-2}.b^{n-2} + \ldots + q_1.b^1 + q_0.b^0 \\ &\quad + q_{-1}.b^{-1} + q_{-2}.b^{-2} + \ldots + q_{-m}.b^{-m} \\ &= \underbrace{\sum_{i=0}^{n-1} q_i.b^i}_{\textit{partie entière}} + \underbrace{\sum_{j=1}^{m} q_{-j}.b^{-j}}_{\textit{partie fractionnaire}} \end{aligned}$$

avec q_i et $q_j \in \{0, \ldots, b-1\}$ pour $0 \leq i \leq n-1$ et $1 \leq j \leq m$.

Exemple 2.11

$$\begin{aligned}
(\frac{3}{4})_{10} &= 0.10^0 + 7.10^{-1} + 5.10^{-2} \\
-25,76_{10} &= -2.10^1 + -5.10^0 + 7.10^{-1} + 6.10^{-2} \\
(\frac{1}{2})_5 &= 0.5^0 + 2.5^{-1} + 2.5^{-2} + 2.5^{-3} + \ldots \\
&\approx 0.5^0 + 2.5^{-1} + 2.5^{-2} + 2.5^{-3} \quad \textit{à trois décimales après la virgule} \\
0,01101_2 &= 0.2^0 + 0.2^{-1} + 1.2^{-2} + 1.2^{-3} + 0.2^{-4} + 1.2^{-5} \\
\pi_{10} &= 3.10^0 + 1.10^{-1} + 4.10^{-2} + \ldots \\
&\approx 3.10^0 + 1.10^{-1} + 4.10^{-2} \quad \textit{à deux décimales après la virgule}
\end{aligned}$$

L'ensemble des nombres fractionnaires est donc un sous-ensemble des rationnels. Dit autrement, un nombre fractionnaire désigne un nombre rationnel (le quotient de deux entiers naturels) quand le développement polynômial de ce nombre est fini, ou dans le cas contraire, son approximation à partir d'un développement fini. Il permet donc aussi l'approximation finie d'un nombre irrationnel.

La conversion d'un nombre fractionnaire s'effectue par la conversion de la partie entière, puis par la conversion de la partie fractionnaire. La conversion des nombres entiers a été vue précédemment, on note f, la partie fractionnaire :

$$f = q_{-1}.b^{-1} + q_{-2}.b^{-2} + \ldots + q_{-m}.b^{-m}$$

en multipliant par b, on obtient :

$$b.f = \underbrace{q_{-1}.b^0}_{\textit{partie entière}} + \underbrace{q_{-2}.b^{-1} + \ldots + q_{-m}.b^{-m+1}}_{\textit{partie fractionnnaire}}$$

on a donc déterminé q_{-1}. Notons maintenant f' la partie fractionnaire de f :

$$f' = q_{-2}.b^{-1} + \ldots + q_{-m}.b^{-m+1}$$

en multipliant par b, on obtient :

$$b.f' = \underbrace{q_{-2}.b^0}_{\textit{partie entière}} + \underbrace{q_{-3}.b^{-1} + \ldots + q_{-m}.b^{-m+2}}_{\textit{partie fractionnaire}}$$

on a donc déterminé q_{-2} et on répète ainsi l'opération jusqu'à la détermination de q_{-m}. La représentation de f en base b est $q_{-1}\ q_{-2}, \ldots, q_{-m}$.

Exemple 2.12 *Conversion de* $0,75$ *en binaire : la partie entière de* $0,75$ *est* 0*, la partie fractionnaire est* $0,75$ *:*

$$\begin{aligned}
0,75 \times 2 &= 1,50 \rightarrow q_{-1} = 1 \\
0,50 \times 2 &= 1,00 \rightarrow q_{-2} = 1 \\
0,00 \times 2 &= 0,00 \rightarrow q_{-3} = 0
\end{aligned}$$

$0,75$ *s' écrit donc en binaire* $0,110_2$.

Exemple 2.13 *Conversion de* $1,57$ *en binaire : la partie entière de* $1,57$ *est* 1*, la partie fractionnaire est* $0,57$ *:*

$$\begin{aligned} 0,57 \times 2 &= 1,14 \rightarrow q_{-1} = 1 \\ 0,14 \times 2 &= 0,28 \rightarrow q_{-2} = 0 \\ 0,28 \times 2 &= 0,56 \rightarrow q_{-3} = 0 \\ 0,56 \times 2 &= 1,12 \rightarrow q_{-4} = 1 \\ 0,12 \times 2 &= 0,24 \rightarrow q_{-5} = 0 \end{aligned}$$

$1,57$ *s' écrit en binaire* $1,10010_2$.

Il existe deux formats de codage des nombres fractionnaires : la représentation en *virgule fixe* et la représentation en *virgule flottante*.

Représentation en virgule fixe

En virgule fixe, la virgule n'est pas représentée, elle est virtuelle : c'est le programmeur qui décide du nombre de chiffres avant et après la virgule. S'il s'agit d'un codage très naturel d'un nombre fractionnaire, il présente cependant un inconvénient : il faut penser à conserver un certain nombre de chiffres avant et après la virgule faute de quoi un dépassement de capacité risque de se produire. Mais la nécessité de conserver ces chiffres, peut d'une part amener à une approximation assez mauvaise du nombre à représenter, et d'autre part ralentit le temps d'exécution des différentes opérations.

Exemple 2.14 *Soient* $a = 4328$ *et* $b = 3612$ *deux nombres entiers de* 4 *chiffres, en virgule fixe pour pouvoir calculer correctement leur produit, il faut penser à conserver au moins* 8 *chiffres avant la virgule, car le produit,* $a \times b = 15632736$*, nécessite* 8 *chiffres.*

Représentation en virgule flottante

En virgule flottante, le système de représentation est indépendant du nombre de chiffres significatifs.

$$x = \underbrace{s}_{signe} \quad \underbrace{m}_{mantisse} \quad \underbrace{b^e}_{base\ et\ exposant}$$

La mantisse m est exprimée dans la base b. La précision est le nombre de chiffres de la mantisse, l'exposant est un nombre entier signé. En machine, $b = 2$. Un nombre décimal a une infinité de représentations mantisse–exposant :

Exemple 2.15 *Par exemple, en notation décimale*

$$3,14 = 3,14 \times 10^0 = 0,314 \times 10^1 = 314 \times 10^{-2} \ldots$$

Clairement, la représentation des nombres fractionnaires pose un problème de précision. L'ordinateur travaillant sur des représentations finies, plusieurs difficultés se présentent :

1. D'une part, on est souvent amené à *approximer* la valeur d'un nombre. Par exemple, que ce soit en base 10 ou en base 2, 1/3 n'est pas représentable précisément, ni en virgule fixe, ni en virgule flottante, puisque l'ensemble des nombres fractionnaires représentables en machine (appelés usuellement, mais à tort, *réels* dans le jargon informatique) est un sous-ensemble propre des rationnels. La question est alors d'arriver à minimiser l'erreur issue de l'arrondi, c'est-à-dire minimiser l'*erreur absolue* ($=$ *valeur approchée* $-$ *valeur réelle*) et l'*erreur relative* ($= \frac{\textit{erreur absolue}}{|\textit{valeur réelle}|}$) qui en résulte, correspondant au pourcentage de déviation de l'arrondi par rapport à la valeur exacte.
2. D'autre part, et on le sait déjà, il n'est pas possible de représenter en machine des nombres arbitrairement grands ou petits en valeur absolue. Or, la longueur d'un mot de type donné étant fixée par l'architecture, le choix du nombre de positions binaires attribuées à l'exposant d'une part, et à la mantisse d'autre part, représente un compromis : si l'on attribue plus de positions à l'exposant, on augmente la plage de représentation, mais la précision se trouve diminuée. Inversement, si l'on attribue plus de positions à la mantisse, la précision se trouve augmentée, mais la plage de représentation est plus restreinte.
3. Enfin, et puisqu'il sagit de nombres, il est nécessaire de disposer d'un format unique permettant d'assurer leur comparaison. Sachant que les algorithmes de comparaison procèdent (naturellement) à une comparaison chiffre à chiffre (en binaire bit à bit) à partir des poids forts, comment par exemple comparer efficacement $31,4 \times 10^{-1}$ et $4,13 \times 10^{1}$? On constate sans difficulté sur cet exemple que l'utilisation d'un format de représentation unique (par exemple $3,14 \times 10^{0}$ et $41,3 \times 10^{0}$) rend cette comparaison immédiatement plus aisée. Le choix d'une valeur unique d'exposant, qui facilite ici la lecture par un être humain, n'est toutefois pas la solution la plus adéquate à une représentation en machine. Dans ce dernier cas, on préfère utiliser un cas particulier de *représentation scientifique* (un seul chiffre avant la virgule dans la mantisse), appelé *représentation normalisée* (un seul chiffre, différent de 0, avant la virgule dans la mantisse). Par exemple, $4,13 \times 10^{1}$ est en représentation normalisée (ce qui n'est pas le cas pour $41,3 \times 10^{0}$). La normalisation produit à la fois une représentation unique et une répartition équilibrée des nombres. A la suite de [Goldberg91], considérons à titre d'illustration l'exemple (simpliste) de normalisation de nombres flottants exprimés en base 2 sur 3 bits de mantisse avec un exposant sur 2 bits pouvant prendre les valeurs successives $-1, 0, 1, 2$ (correspondant à un codage en excès à 3, dont le principe est exposé un peu plus loin). En considérant uniquement des nombres positifs, les mantisses disponibles après normalisation sont :

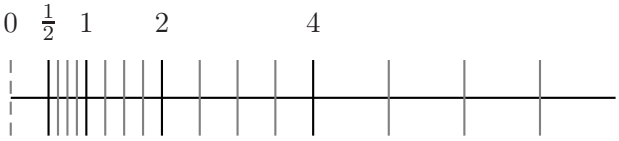
$$\begin{aligned} 1,00_2 &= 1_{10} \\ 1,01_2 &= 1,25_{10} \\ 1,10_2 &= 1,5_{10} \\ 1,11_2 &= 1,75_{10} \end{aligned}$$

chacune étant multipliée avec l'une des quatre valeurs possibles de base–exposant parmi $2^{-1}, 2^{0}, 2^{1}, 2^{2}$, soient seize nombres normalisés au total. Les valeurs successives obtenues

sont :

$$
\begin{array}{rcccccccc}
1.2^{-1},\ 1,25.2^{-1},\ 1,5.2^{-1},\ 1.75.2^{-1} & \text{soient} & 0,5 & \overset{0,125}{\longleftrightarrow} & 0,625 & \overset{0,125}{\longleftrightarrow} & 0,75 & \overset{0,125}{\longleftrightarrow} & 0,875 \\
1.2^{0},\ 1,25.2^{0},\ 1,5.2^{0},\ 1.75.2^{0} & \text{soient} & 1 & \overset{0,25}{\longleftrightarrow} & 1,25 & \overset{0,25}{\longleftrightarrow} & 1,5 & \overset{0,25}{\longleftrightarrow} & 1,75 \\
1.2^{1},\ 1,25.2^{1},\ 1,5.2^{1},\ 1.75.2^{1} & \text{soient} & 2 & \overset{0,5}{\longleftrightarrow} & 2,5 & \overset{0,5}{\longleftrightarrow} & 3 & \overset{0,5}{\longleftrightarrow} & 3,5 \\
1.2^{2},\ 1,25.2^{2},\ 1,5.2^{2},\ 1.75.2^{2} & \text{soient} & 4 & \overset{1}{\longleftrightarrow} & 5 & \overset{1}{\longleftrightarrow} & 6 & \overset{1}{\longleftrightarrow} & 7
\end{array}
$$

La figure suivante décrit les nombres représentés :



Les barres foncées correspondent au cas où la mantisse vaut 1. Dans chaque intervalle, l'espacement est double du précédent : la différence entre deux nombres consécutifs n'est pas constante, mais l'erreur relative créée par un arrondi est, elle, constante. Remarquons encore qu'en base 2, et en représentation normalisée, le seul chiffre différent de zéro avant la virgule étant nécessairement 1, on peut considérer qu'il est inutile de gaspiller un bit pour l'indiquer. La mantisse comporte donc souvent conventionnellement un 1 implicite, et dans l'exemple précédent, deux bits suffisent alors à la représentation de la mantisse (ou inversement, trois bits permettent de gagner en précison). En laissant momentanément de côté la question de la représentation de zéro (non normalisé par définition), peut-on se contenter des seules représentations normalisées ? Dans l'exemple traité ci-dessus, un « vide » important apparaît entre zéro et le plus petit nombre normalisé accessible, en l'occurence 1/2. Cela signifie que toute valeur prise dans la plage $[0, 1/2[$, trop petite pour être représentée, est approximée par 0. Bien sûr, l'exemple utilisé décrit ici une situation plutôt grossière, et il suffit d'augmenter le nombre de bits alloués à l'exposant pour gagner en plage de représentation : par exemple, en passant l'exposant de deux à trois bits, il devient possible de représenter des nombres sous forme normalisée entre 1/4 et 8. On peut alors légitimement penser qu'il suffit d'élargir la plage de représentation liée à l'exposant autant que nécessaire pour approximer de façon plus « satisfaisante » les nombres proches de zéro. Pour autant, aussi loin qu'on aille, il est aisé de constater qu'un déséquilibre subsiste sur l'erreur relative d'un arrondi, selon qu'il est effectué dans l'intervalle des nombres normalisés, ou dans l'intervalle qui sépare zéro du premier nombre normalisé accessible. Une conséquence grave découlant de cette situation est que la soustraction en machine de deux nombres normalisés peut donner pour résultat zéro alors que ces nombres sont distincts. Par exemple, $x = 1,01_2.2^{-1}$ et $y = 1,00_2.2^{-1}$ sont deux nombres normalisés distincts, dont la différence $x - y = 0,625 - 0,5 = 0,125 = \frac{1}{8}$. La soustraction effectuée en machine est $x \ominus y = 0$ parce que la différence $x - y$, trop petite pour être représentée ici par un nombre normalisé, est arrondie à zéro. On a donc $x \ominus y = 0$ alors que $x \neq y$! Autrement dit, deux codes identiques du point de vue mathématique peuvent produire des résultats différents : par exemple [Goldberg91], le fragment de code `if not(x=y) then z=1/(x-y)` risque de conduire à une division par zéro inattendue, alors que `if not(x-y=0)`

`then z=1/(x-y)` ne produit pas d'erreur. Il est donc essentiel de préserver la propriété

$$(x = y) \Leftrightarrow (x - y = 0) \tag{2.2}$$

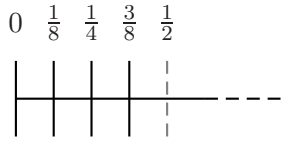
La solution consiste à autoriser l'utilisation de valeurs dénormalisées entre zéro et le premier nombre normalisé accessible de façon à remplir le « vide » existant entre ces deux valeurs. Dans notre exemple, cela revient à placer l'ensemble des nombres dénormalisés disponibles, soient *a priori* seize nombres compris entre 0 (inclus) et 1/2 (exclus). En réalité, sagissant de nombres dénormalisés, l'unicité de la représentation acquise à la normalisation n'est pas conservée, et sur les seize représentations possibles, seules quatre ne correspondent pas à des représentations de nombres déjà obtenus. Plus précisément, à partir des quatres valeurs disponibles de mantisse

$$\begin{aligned} 0,00_2 &= 0_{10} \\ 0,01_2 &= 0,25_{10} \\ 0,10_2 &= 0,5_{10} \\ 0,11_2 &= 0,75_{10} \end{aligned}$$

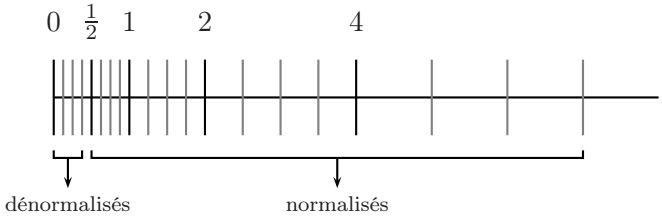
et des quatre valeurs possibles de base–exposant parmi $2^{-1}, 2^0, 2^1, 2^2$, on obtient les valeurs successives suivantes :

$$0.2^{-1},\ 0,25.2^{-1},\ 0,5.2^{-1},\ 0.75.2^{-1} \quad \text{soient} \quad 0 \overset{0,125}{\longleftrightarrow} \tfrac{1}{8} \overset{0,125}{\longleftrightarrow} \tfrac{1}{4} \overset{0,125}{\longleftrightarrow} \tfrac{1}{4} + \tfrac{1}{8} = \tfrac{3}{8}$$

Les douze représentations dénormalisées restantes ($0.2^0,\ 0,25.2^0,\ 0,5.2^0,\ 0.75.2^0,\ 0.2^1,\ 0,25.2^1,\ 0,5.2^1,\ 0.75.2^1,\ 0.2^2,\ 0,25.2^2,\ 0,5.2^2,\ 0.75.2^2$) correspondent à des valeurs déjà obtenues. Quatre nombres dénormalisés (dont zéro) sont donc à placer :



et on obtient finalement la répartition suivante :



Il est aisé de vérifier la validité de la propriété 2.2 pour x et y normalisés en montrant que la plus petite différence que l'on peut obtenir entre x et y est égale au plus petit nombre dénormalisé différent de zéro constructible par cette méthode. Si l'approche utilisée garantit bien la validité de la propriété 2.2 sur les nombres normalisés, elle suscite aussi quelques remarques... En premier lieu, l'utilisation d'un 1 implicite pour le bit de partie entière des

nombres normalisés implique de distinguer les nombres dénormalisés par un moyen qui ne fasse pas appel à la partie entière (en l'occurence zéro pour un nombre dénormalisé). Une possibilité est par exemple d'utiliser une valeur spécifique d'exposant. En second lieu, si la propriété 2.2 est vérifiée pour les nombres normalisés, par contre la différence de deux nombres dénormalisés peut elle-même être trop petite pour être représentée, même de façon dénormalisée, et peut donc générer un arrondi à zéro. Il est donc important de tenir compte de la plus petite valeur représentable, et de la façon dont on procède à un arrondi.

La prise en compte de toutes les questions liées à la représentation et au traitement des flottants a suscité de très longs débats entre constructeurs de machine et utilisateurs du calcul scientifique. C'est la raison pour laquelle le standart actuel, la norme IEEE 754, initiée en 1977, n'a finalement été complètement adoptée qu'en 1985. Cette norme fait d'ailleurs actuellement l'objet de nouvelles révisions, sous le nom IEEE 754r. Le standard IEEE[4] prescrit l'utilisation de la représentation normalisée en signe et valeur absolue pour la mantisse, le positionnement des bits d'exposant entre le bit de signe et les bits de la mantisse, et distingue deux formats : *simple* et *double* précision, chacun de ces deux formats pouvant être en outre *étendus*.

Simple précision (32 bits)

1	8	23
s	E (exposant)	M (mantisse)

Double précision (64 bits)

1	11	52
s	E (exposant)	M (mantisse)

Il aisé de constater que le positionnement relatif des différents champs (bit de signe, exposant, mantisse) est destiné à faciliter la comparaison de deux nombres flottants. Un algorithme de comparaison, partant donc des bits de poids forts, traitera d'abord les signes, puis les exposants, et enfin les mantisses sous forme normalisée : clairement, dès qu'un bit du premier nombre à comparer est inférieur ou supérieur au bit de même rang du second nombre à traiter, la comparaison est terminée. Sans plus de précision toutefois, deux problèmes demeurent *a priori* :

– Une difficulté surgit dans le cas de la représentation du zéro. Evidemment, la mantisse du nombre zéro doit contenir dans toutes les positions binaires correspondantes des chiffres 0. Théoriquement, son exposant peut avoir n'importe quelle valeur, mais compte-tenu de la façon dont s'expriment les nombres qui s'approchent de zéro (c'est-à-dire par des valeurs de puissances négatives de plus en plus élevées), il est cohérent que la valeur de l'exposant pour zéro soit la puissance négative la plus élevée possible. Un tel zéro est appelé *zéro propre*. Si l'exposant est donné en complément à deux, et en dehors de toute tentative de normalisation, la représentation du zéro propre en simple précision (par exemple) est alors :

$$\underbrace{0}_{signe}\ \underbrace{10000000}_{exposant}\ \underbrace{00000000000000000000000}_{mantisse}$$

Cette représentation ne contient pas le bit 0 dans toutes les positions, ce qui constitue un inconvénient au moment du test et pose un problème de compatibilité avec la représentation de zéro en virgule fixe (représentation qui ne contient que des zéros dans toutes les positions).

[4] Acronyme de *Institute of Electrical and Electronics Engineers*

– Plus grave encore, un nombre ayant un exposant négatif apparaîtra supérieur à un nombre ayant un exposant positif. Le problème provient de la comparaison bit à bit des des représentations binaires : par exemple 11000000 apparaîtra supérieur à l'exposant 01000000, alors qu'en complément à deux, le premier est négatif, et le second positif.

Pour remédier à ces problèmes, le standard IEEE utilise un exposant corrigé e, donné en *valeur par excès* d'une constante égale au nombre de positions binaires positives, moins une, affectées à l'exposant. Par exemple, en simple précision, donc sur 8 bits, la constante utilisée est $256/2-1 = 127$. Par conséquent, e représentant la valeur réelle de l'exposant en complément à deux, $e =$ interprétation de E en naturel $- 127$. L'utilisation d'un exposant corrigé permet la translation de 127 valeurs négatives vers des représentations positives considérées comme équivalentes, et on obtient la correspondance suivante :

Binaire	e (excés à 127)	E (interprétation non signée)
00000000	-127 (e_{min})	0
00000001	-126	1
...	...	...
01111111	0	127
10000000	$+1$	128
...	...	...
11111111	$+128$ (e_{max})	255

Il est facile de constater que cette approche permet bien d'obtenir une représentation homogène de zéro : l'interprétation en naturel du plus grand exposant négatif représentable sur 8 bits étant 127 (8 bits positionnés à 1), l'exposant corrigé $e = 127 - 127 = 0$, d'où une représentation flottante de zéro dont tous les bits sont positionnés à 0, mais dont l'exposant exprime bien la puissance négative la plus élevée possible.

Puisqu'en simple précision, e est l'interprétation de E en excès à 127 ($e = E_{10} - (2^8/2 - 1)$, on a $e_{min} = -127$, $e_{max} = +128$. En double précision, e est l'interprétation de E en excès à 1023 ($e = E_{10} - 2^{16}/2 - 1$), on a donc $e_{min} = -1023$, $e_{max} = +1024$.

Une question reste en suspend : en effet, le choix par le standard IEEE d'un déplacement de 127 valeurs à droite (en simple précision) n'est pas le plus naturel *a priori*. Puisque sur 256 configurations binaires disponibles en complément à deux, on compte de -128 à $+127$, il semble censé de considérer alors un déplacement de $256/2 = 128$ positions vers la droite (le nombre de positions négatives disponibles), au lieu du déplacement de $256/2-1 = 127$ positions adopté. Autrement dit, pourquoi enlever 1, c'est-à-dire prendre le parti de ne compter qu'à partir de -127 ? En l'occurence, le standard IEEE fait le choix de réserver les valeurs extrêmes de e à la représentation de valeurs exceptionnelles, dont les nombres dénormalisés. On distingue ainsi le cas $e_{min} < e < e_{max}$, associé aux nombres normalisés, le cas $e = e_{min}$, associé aux nombres dénormalisés (dont zéro), et enfin le cas $e = e_{max}$ associé à la représentation de valeurs particulières : $-\infty$, $+\infty$, d'une part et NaN(*Not a Number*) d'autre part.

– La présence des infinis permet de simuler la clôture topologique de $\mathbb{R}$: par exemple, la division d'un nombre distinct de zéro par zéro renvoie un des infinis. Le codage d'un infini est caractérisé par le bit de signe, le champs de l'exposant valant e_{max}, et la mantisse étant égale à zéro.
– La valeur particulière NaNpermet la détection d'une opération invalide : tentative de division de zéro par lui même, division d'un infini par un infini, multiplication de zéro par

un infini, soustraction de deux infinis, extraction de la racine carrée d'un nombre négatif... Toute opération dont un des opérateurs est NaNrenvoie NaN. Le codage d'un NaNest caractérisé par le fait que le champs de l'exposant vaut e_{max}, la mantisse étant différente de zéro.

L'ensemble des cas possibles est finalement répertorié dans le tableau suivant :

Cas	e (en excés)	M	valeur
Normalisé	$e_{min} < e < e_{max}$	quelconque	$(-1)^s \times 1, M \times 2^e$
Dénormalisé	$e = e_{min}$	$\neq 0$	$(-1)^s \times 0, M \times 2^{e_{min}}$
Zéro	$e = e_{min}$	0	$(-1)^s \times 0$
Infini	$e = e_{max}$	0	$(-1)^s \times \infty$
NaN	$e = e_{max}$	$\neq 0$	NaN

Exemple 2.16 *Soit le nombre* $+87,125$ *qu'on veut représenter comme un flottant en simple précision. On convertit en binaire, séparément sa partie entière et sa partie fractionnaire :*

$$\begin{array}{rcl} 87_{10} & = & 1010111_2 \\ 0,125_{10} & = & 0,001_2 \end{array}$$

d'où $87,125_{10} = 1010111,001_2$*. En effectuant la normalisation, on obtient*

$$1010111,001_2 = 1,010111001_2 \times 2^6$$

donc

- $1,010111001$ *représente la mantisse (dont le* 1 *en partie entière est à laisser implicite)*
- 110 *représente l'exposant*
- *l'exposant corrigé est* 10000101 *(*$133 = 6 + 127$*).*

Par conséquent, la représentation de $+87,125$ *comme un flottant en simple précision est :*

$$0\ 10000101\ 01011100100000000000000$$

La représentation de $-87,125$ *ne diffère que par le bit de signe :*

$$1\ 10000101\ 01011100100000000000000$$

Les tailles des champs dévolus à la représentation de la fraction et de l'exposant donnent un ordre de magnitude important. Le plus grand nombre flottant représentable en simple précision est donné par :

$$\begin{array}{rcl} (1 + 2^{-1} + 2^{-2} + \ldots + 2^{-23}) \times 2^{+127} & = & \dfrac{1 - 2^{-24}}{1 - 2^{-1}} \times 2^{+127} \\ & = & 2 \times (1 - 2^{-24}) \times 2^{+127} \\ & = & (2 - 2^{-23}) \times 2^{+127} \\ & = & 2^{128} - 2^{104} \\ & \approx & 3,402\ 823\ 466.10^{38} \end{array}$$

Le standard IEEE impose le repérage des situations suivantes :

– dépassement de précision (underflow)
– dépassement de capacité (overflow)

Le repérage et un bon traitement de ces situations exceptionnelles peut être vital dans certains cas. Le crash du vol 501 de la fusée Ariane après quarante secondes de vol en juin 1996 en est un exemple particulièrement patent : dû à un mauvais traitement d'un dépassement de capacité, il a coûté la bagatelle de 500 millions de dollars, et mis à mal un programme de développement d'une dizaine d' années pour un coût estimé à 7 billions de dollars... L'origine du problème s'est avéré être une erreur de programmation dans le système de référence inertielle : un flottant codé sur 64 bits, et donnant la vitesse horizontale de la fusée, était converti en entier signé sur 16 bits. L'entier obtenu étant plus grand que 32767 (le plus grand entier positif représentable en complément à deux sur 16 bits), la conversion échouait, déclanchant une exception non traitée. Le plus remarquable est que tous les tests logiciels avaient été passés avec succés par le programme en question, mais qu'ils avaient été effectués avec les données d'Ariane 4, fusée moins puissante et donc moins rapide qu'Ariane 5, pour laquelle la vitesse horizontale restait inférieure au maximum de 32767 (cf. [Arnold98])...

Outre le traitement de situations exceptionnelles, la norme stipule quatre modes possible d'arrondi :

– Arrondi au nombre pair le plus proche (mode par défaut)
– Arrondi à zéro
– Arrondi au nombre supérieur le plus plus proche
– Arrondi au nombre inférieur le plus plus proche

La standard IEEE 754 précise de quelle façon réaliser un certain nombre d' opérations arithmétiques sur les flottants. Les opérations les plus courantes (comparaison, addition, soustraction, multiplication, et division) sont traitées de la façon suivante :

Comparaison. Le format de codage d'un flottant (signe, exposant en valeur par excés, mantisse) permet, on l'a vu, d'effectuer la comparaison de deux flottants bit à bit à partir des poids forts.

Addition et soustraction. L'addition et la soustraction de deux flottants sont réalisées à partir d'une dénormalisation du flottant le plus petit afin de rendre les exposants égaux en minimisant une possible perte de précision. Les mantisses sont ensuite additionnées ou soustraites, avant renormalisation. Plus précisément, l'algorithme suivi est le suivant :

```
Début
  x et y, deux flottants à additionner ou
  soustraire, respectivement sous la forme :
    signe_x exp_x mant_x
    signe_y exp_y mant_y;
  Si (exp_x > exp_y)
    alors exp_G = exp_x;
          exp_p = exp_y;
          mant_G = mant_x;
          mant_p = mant_y;
    sinon exp_G = exp_y;
          exp_p = exp_x;
          mant_G = mant_y;
```

```
            mant_p = mant_x;
        diff = exp_G - exp_p;
        slr(mant_p) sur diff bits;
        mant_res = mant_G +/- mant_p;
        arrondir mant_res;
        Si (underflow ou overflow)
          alors lever une exception;
          sinon normaliser : exp_res et mant_res modifiés;
                résultat = sign_res exp_res mant_res;
    Fin
```

Par exemple, l'addition des deux flottants en simple précision $x = 1,01000\ldots0 \times 2^{-5}$ et $y = 1,11010\ldots0 \times 2^{-3}$ est effectuée de manière suivante :

- Calcul de la différence entre l'exposant le plus grand et l'exposant le plus petit : $exp_x - exp_y = -3 + 5 = 2$
- Alignement de la mantisse du plus petit nombre sur la valeur d'exposant le plus élevé d'où décalage de deux positions à droite de la mantisse de plus petit exposant, ce qui dénormalise y : $mant_y = 0,01010\ldots0$.
- Les exposants étant maintenant égaux, il est possible de procéder à l'addition des mantisses :

$$\begin{aligned} mant_{res} &= 0,01010\ldots0 + 1,11010\ldots0 \\ &= 10,0010\ldots0 \end{aligned}$$

- Arrondi par défaut (nombre pair le plus proche), et renormalisation :

$$\begin{aligned} mant_{res} &= 1,00010\ldots0 \\ exp_{res} &= -4 \end{aligned}$$

- L'addition des flottants x et y donne : $résultat = 1,00010\ldots0 \times 2^{-4}$

Remarque : L'addition flottante n'est pas associative... Etants donnés par exemple trois nombres flottants en simple précision

$$\begin{aligned} x &= 1,5 \times 10^{38} \\ y &= -1,5 \times 10^{38} \\ z &= 1 \end{aligned}$$

$$x + (y + z) \neq (x + y) + z$$

En effet, compte-tenu des limitations imposées sur la précision et des approximations nécessairement réalisées au cours des calculs, la différence de magnitudes entre x ou y d'une part, et z d'autre part, amène aux séquences de calcul différentes suivantes :

$$\begin{aligned} x + (y + z) &= 1,5 \times 10^{38} + (-1,5 \times 10^{38} + 1) \\ &\approx 1,5 \times 10^{38} + (-1,5 \times 10^{38}) \\ &= 0 \\ (x + y) + z &= (1,5 \times 10^{38} + -1,5 \times 10^{38}) + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

Multiplication. Le produit de deux flottants a pour résultat le produit des signes de chacun des flottants, suivi du produit des mantisses, et enfin de l'addition des exposants respectifs. Plus précisément, l'algorithme de multiplication est le suivant :

```
Début
  x et y, deux flottants à multiplier,
  respectivement sous la forme :
    signe_x exp_x mant_x
    signe_y exp_y mant_y;
Effectuer en parallèle 1) et 2) :
  1) traitement des mantisses et des signes :
     mant_res = mant_x.mant_y;
     signe_res = signe_x.signe_y;
  2) traitement des exposants :
     exp-res = exp_x + exp_y - valeur par excés;
arrondir mant_res;
    Si (underflow ou overflow)
      alors lever une exception;
      sinon normaliser : exp_res et mant_res modifiés;
            résultat = sign_res exp_res mant_res;
Fin
```

Par exemple, la multiplication des deux flottants en simple précision $x = 1,01000\ldots0 \times 2^{-5}$ et $y = 1,11010\ldots0 \times 2^{-3}$ est effectuée de manière suivante :

1. Produit des mantisses, des signes, et calcul parallèle de l'exposant résultant :

$$\begin{aligned} mant_x \times mant_y &= 1,01000\underbrace{0\ldots0}_{18\ bits} \times 1,11010\underbrace{0\ldots0}_{18\ bits} \\ &= 10,0010001000\underbrace{0\ldots0}_{36\ bits} \\ exp_x + exp_y + 127 &= -3 - 5 + 127 = 119 \end{aligned}$$

2. La mantisse est normalisée : on obtient $1,00100010000\underbrace{0\ldots0}_{36\ bits} \times 2$, autrement dit, et sachant que la mantisse est exprimée sur 23 bits, $1,00100010\underbrace{0\ldots0}_{16\ bits}$, avec pour exposant résultant 120.

Division. La division est effectuée de manière standard à partir d'une suite de soustraction successives. L'algorithme est le suivant :

```
Début
  x et y, deux flottants à diviser,
  respectivement sous la forme :
    signe_x exp_x mant_x
    signe_y exp_y mant_y;
Soient Z_x e Z_y le nombre de zéro sur les poids forts de x et y;
Effectuer en parallèle 1) et 2) :
```

```
  1) traitement des mantisses et des signes :
     sll(mant_x) sur Z_x bits;
     sll(mant_y) sur Z_y bits;
     mant_res = mant_x/mant_y;
     signe_res = signe_x.signe_y;
  2) traitement des exposants :
     exp_res = exp_x - exp_y + valeur par excés - Z_x + Z_y;
arrondir mant_res;
    Si (cas exceptionnel)
      alors lever une exception;
      sinon normaliser : exp_res et mant_res modifiés;
            résultat = sign_res exp_res mant_res;
Fin
```

Une description précise de la norme IEEE 754, détaillant en particulier l'arithmétique étendue et la gestion des erreurs, est présentée dans [GOLDBERG91].

2.7 Autres codages : codes de Gray, codes détecteurs, codes correcteurs

De nombreux types de codages des nombres sont envisageables. Si les codes binaires étudiés dans les paragraphes précédents sont cohérents par rapport à des besoins calculatoires arithmétiques, ils sont inadaptés à d'autres applications comme par exemple le cryptage, le transport de données, la numérisation rapide à partir de capteurs...

Codes de Gray

Illustrons le problème avec l'exemple suivant : on souhaite minimiser le nombre de commutations de bits effectuées par un compteur au moment du passage d'une configuration binaire à la suivante. Avec le codage binaire des entiers sous forme de décomposition polynômiale en base 2, ce critère n'est pas respecté : par exemple, lorsqu'on passe de la valeur 7 (0111_2) à la valeur 8 (1000_2), quatre bits sont changés d'un coup. Il est pourtant possible d'assurer la commutation d'un seul bit à la fois quelque soient les entiers n et $n+1$ considérés. La famille de codes assurant une telle propriété est celle des *codes de Gray*. Un tel code fut utilisé en 1878 par l'ingénieur Emile Baudot dans la conception de son télégraphe ; toutefois c'est Frank Gray, chercheur du Bell Labs, qui en réalisa en 1953 la première étude mathématique.

Définition 2.1 *Tout ordre défini sur les 2^n configurations binaires de mots de n bits tel que le passage d'une configuration à la suivante ne change qu'un seul bit est un code de Gray*[5].

Exemple 2.17 *Un code de Gray sur* 3 *bits est* $(000, 010, 011, 001, 101, 111, 110, 100)$.

Parmi les codes de Gray, le *code binaire réfléchi* est remarquable en ce sens qu'à la propriété caractéristique d'un code de Gray, il ajoute la *propriété d'adjacence* : deux configurations binaires successives ne diffèrent que par un bit *de même rang*.

[5] De façon alternative et pour les spécialistes : Tout cycle hamiltonien sur un hypercube de dimension n détermine un code de Gray...

Exemple 2.18 *Le codage binaire réfléchi des entiers de* 0 *à* 15 *est :*

décimal	*binaire pur*	*binaire réfléchi*
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

L'algorithme suivant permet de générer du code binaire réfléchi sur n bits :

```
Début
  séquence = ‘‘0 1’’;
  rang = 1;
  Tant que rang  < n :
    réaliser une copie de la séquence;
    inverser l’ordre de la copie;
    concaténer ‘‘0’’ à la gauche de chaque élément de la séquence;
    concaténer ‘‘1’’ à la gauche de chaque élément de la copie;
    séquence = concaténation(séquence,copie);
    rang = rang + 1;
Fin
```

Le code binaire réfléchi sert notamment à la fabrication des tables de Karnaugh, qui seront vues dans le chapitre suivant.

Les codes détecteurs

Il est pratiquement impossible d'éviter l'apparition d'erreurs dans un ordinateur, erreurs de transmission, de traitement, pannes d'éléments internes ou externes.

Une méthode simple et efficace pour détecter un bit en erreur dans un caractère ou un mot est le *contrôle de parité*. Il s'agit d'ajouter aux n bits d'un mot ou d'un caractère, un bit supplémentaire appelé *bit de parité*. Dans le cas du contrôle impair de parité, on ajoute un bit de telle sorte que le nombre total de bits égaux à 1 soit impair. Dans le cas du contrôle pair de parité, on ajoute un bit de telle sorte que le nombre total de bits égaux à 1 soit pair.

Exemple 2.19 *Le code ASCII, avec un contrôle de parité pair :*

bit de parité	2^6	2^5	2^4	2^3	2^2	2^1	2^0	*caractère*
1	1	1	0	0	0	0	1	a
1	1	1	0	0	0	1	0	b
0	1	1	0	0	0	1	1	c
1	1	1	1	1	0	1	0	z
0	1	0	0	0	0	0	1	A

Le contrôle de parité permet dans le cas d'une erreur, de détecter l'erreur, c'est à dire de savoir qu'il y a eu une erreur sans pour autant la corriger. Pour cela, on attribue aux mots du code une propriété qui disparaît dans le cas d'une erreur. Le principe est le suivant : on fabrique d'abord les messages sous forme de mots binaires de même longueur et on ajoute une nouvelle composante, la somme modulo 2 des composantes. Par exemple, 10110 devient 101101 ou encore 11000 devient 110000. La dernière composante est appelé *bit de parité*, 0 si la somme des composantes du mot initial est paire, 1 dans le cas contraire. De ce fait, dans chaque nouveau mot le nombre de 1 est pair, c'est-à-dire que la somme modulo 2 de toutes les composantes est nulle. Si à la réception cette somme n'est pas nulle, c'est qu'il y a au mois une erreur. Si cette somme est nulle, et dans l'hypothèse que l'on n'a pas pu faire plus d'une erreur, alors le mot reçu est correct.

Une autre méthode est celle de la répétition. Elle consiste par exemple à envoyer deux fois de suite le même mot. Si l'on suppose qu'il y a eu au plus une erreur sur l'ensemble des deux mots (condition plus contraignante que dans le cas du bit de parité), la détection se fait en constatant que les deux mots reçus sont différents. On pourra même corriger en envoyant trois fois le même mot, mais il faudra supposer qu'il y a eu au plus une erreur sur l'ensemble des trois mots (de plus en plus contraignant). Dans ce cas il y aura necessairement un mot répété deux fois à la réception et ce sera le bon.

Sur ce dernier exemple on voit apparaître un problème important : pour corriger plusieurs erreurs et pour des mots de grandes longueurs[6], ce procédé devient rapidement onéreux en transmission car il augmente considérablement la longueur des mots à transmettre, et donc le temps de transmission. On cherchera donc, en plus de la possibilité de correction, un codage le plus économique possible en longueur des mots.

Toute la théorie du codage correcteur d'erreurs vise à généraliser de manière économique les procédés élémentaires de bit de parité et de répétition.

Les codes correcteurs

Protéger les transmissions des erreurs de façon efficace, est précisément l'objet que se fixe la théorie des codes correcteurs d'erreurs. La necessité d'utiliser plus de symboles est due à l'introduction de la *redondance* qui est l'un des maître-mots du codage correcteur d'erreurs. Comme dans le langage courant c'est la redondance qui permet de se protéger contre la perte d'information. C'est pourquoi les mots d'une langue sont suffisamment longs, ce qui permet qu'ils soient suffisamment différents les uns des autres, afin que l'on puisse les reconstituer lors d'une écoute imparfaite. Impossible de confondre « féli..tations » pour félicitations avec « congéd..ment » pour congédiement. De même, on enfonce le clou au téléphone en disant, par exemple : C comme Carole, O comme Odile, D comme Daniel, A comme Alfred, G comme Galois, E comme Europe, C.O.D.A.G.E. : codage.

[6]La longueur d'un mot est le nombre de symboles qui le constituent.

Dans le même ordre d'idées, examinons ce qu'on peut faire avec des "0" et des"1". On supposera maintenant, et dans toute la suite, que les « mots » ont toujours le même nombre de symboles (ou de lettres, si l'on préfère). Ce nombre détermine la « longueur » des mots.

Exemple 2.20

Messages	*Code*
a	*0 0 0*
b	*0 1 0*
c	*0 0 1*
d	*1 1 0*
e	*1 0 1*

Supposons que le mot envoyé soit 001 *et que le mot reçu soit* 011*, c'est à dire avec une erreur dans la deuxième position. Il est impossible de savoir quel est le message initial, car* 011 *peut très bien provenir également* 010 *avec une erreur dans la troisième position. La raison de cela est évidemment que* 001 *ressemble trop à* 010*.*

Au contraire, un bon exemple (pédagogique) est donné par le tableau ci-dessous.

Exemple 2.21 *On a représenté* 5 *mots,* C_1, C_2, C_3, C_4, C_5 *de longueur* 6*, et en dessous de chacun d'eux, tous les mots erronnés possibles qui peuvent en provenir en faisant une erreur.*

C_1	C_2	C_3	C_4	C_5
110011	*010100*	*000111*	*101101*	*011001*
010011	*110100*	*100111*	*001101*	*111001*
100011	*000100*	*010111*	*111101*	*001001*
111011	*011100*	*001111*	*100101*	*010001*
110111	*010000*	*000011*	*101001*	*011101*
110001	*010110*	*000101*	*101111*	*011011*
110010	*010101*	*000110*	*101100*	*011000*

On constate que les différents ensembles de mots erronnés sont deux à deux disjoints. Il n'y a donc pas d'ambiguité possible dans le cas d'une seule erreur et l'on peut donc corriger le mot reçu par simple comparaison avec les mots du code C_1, C_2, C_3, C_4, C_5*. Un seul d'entre eux diffère du mot reçu en une seule position.*

Pour généraliser dans la suite l'exemple précédent, on va s'intéresser à la transmission de messages qui peuvent se décrire au moyen d'un n-uple $(x_1, x_2, \ldots, x_n)$ où les x_i appartiennent à un ensemble A, appelé *alphabet*. Chaque n-uple ainsi obtenu s'appelle alors un *mot*, l'entier n s'appelle la *longueur* du mot, et l'ensemble des mots ainsi fabriqués s'appelle un *code* (sur l'alphabet A). L'alphabet le plus utilisé est l'ensemble des nombres 0 et 1.

A partir d'une telle représentation des messages, on peut tenter de décrire des propriétés concernant la possibilité de détecter et corriger des erreurs éventuelles apparues en cours de transmission. Inversement, en choisissant convenablement l'alphabet et le code, on peut aussi tenter d'améliorer la transmission à propos des erreurs. En particulier, on peut choisir, dans ce but, un alphabet et un code dotés de structures algébriques et combinatoires particulières.

Soit A un ensemble fini, non vide, et n un entier naturel non nul. A^n désigne l'ensemble des $x = (x_1, x_2, \ldots, x_n)$ avec $x_i \in A$.

Définition 2.2 *Soient x et y deux éléments de A^n. On appelle* distance de Hamming *entre x et y, le nombre de composantes pour lesquelles ces éléments diffèrent. Plus précisement si $x = (x_1, x_2, \dots x_n)$ et $y = (y_1, y_2, \dots y_n)$, alors :*

$$d(x, y) = \sharp\{i \in \{1, 2, \dots, n\} \text{ tel que } x_i \neq y_i\}$$

$\sharp$ désignant la cardinalité de l'ensemble considéré.

Définition 2.3 *Un* code *sur A de* longueur *n est un sous-ensemble C de A^n. L'ensemble A est appelé* l'alphabet*, n la* longueur *du code C et les éléments de C sont appelés les* mots *du code.*

Exemple 2.22 *Le code*

$$C = \{(0, 1, 1, 0, 1, 0), (1, 1, 1, 0, 1, 1), (1, 1, 1, 1, 1, 1), (1, 1, 0, 0, 0, 0), (0, 1, 1, 1, 0, 0)\}$$

est un code de longueur 6 sur l'alphabet $A = \{0, 1\}$.

Lorsqu'on utilise un code pour transmettre des messages, si $x = (x_1, x_2, \dots x_n)$ est un mot envoyé et $x' = (x'_1, x'_2, \dots x'_n)$ le mot reçu, éventuellement entaché d'erreurs, alors le nombre d'erreurs commises sur les composantes du mot x est $d(x, x')$, d'après la définition de la distance de Hamming. Lorsqu'on suppose qu'il n'y a pas plus de e erreurs commises[7], c'est-à-dire, $d(x, x') \leq e$, on pourra corriger ces erreurs, autrement dit retrouver x, à la condition que chaque mot erroné reçu ne puisse provenir que d'un seul mot du code.

C'est dans une note interne de 1948, qu'Hamming donne le moyen de construire des codes corrigeant une erreur ayant le meilleur rendement possible $R = k/n$, k étant le nombre de symboles du message à transmettre. L'alphabet $\{0, 1\}$ est délibérément choisi afin d'utiliser le contrôle de parité. Le calcul d'un contrôle de parité partiel, appelé syndrome, est effectué à partir de certains symboles du message à coder et de certains symboles de contrôle. Le syndrome comportant $n - k$ symboles doit correspondre aux différentes possibilités d'erreur et doit donner la position de l'erreur exprimée en binaire.

Exemple 2.23 (Le code de Hamming (7,4)) *Le code de Hamming de longueur $n = 7$, permet de coder des messages comportant $k = 4$ symboles, en ajoutant $n - k = 3$ symboles de contrôle.*

La construction du syndrome constitué de 3 symboles, repose sur la représentation binaire des entiers :

- *Le 1ąsymbole, c'est à dire le symbole le plus à droite, représente le résultat du premier contrôle de parité, il correspond aux positions 1, 3, 5, 7, 9, ... car les représentations binaires de ces nombres comportent un 1 en 1ąposition.*
- *Le 2ąsymbole représente le résultat du second contrôle de parité, il correspond aux positions 2, 3, 6, 7, 10, ... car les représentations binaires de ces nombres comportent un 1 en 2ąposition.*
- *Le 3ąsymbole représente le résultat du troisième contrôle de parité, il correspond aux positions 4, 5, 6, 7, 12, ... car les représentations binaires de ces nombres comportent un 1 en 3ąposition.*

[7]Hypothèse qui peut être justifiée par une étude probabiliste du domaine de la théorie de l'information.

Les symboles de contrôle ne sont pas placés après les symboles du message à coder, mais le iąsymbole de contrôle est placé en position 2^{i-1} afin de préserver l'indépendance des symboles de contrôle. Le codage et le décodage s'effectuent respectivement de manière suivante :

Codage du code de Hamming (7, 4) *Soit le message* 1010 *à coder, les symboles du message sont placés ailleurs que sur les positions* 1, 2, 4.

1	*2*	*3*	*4*	*5*	*6*	*7*
		1		*0*	*1*	*0*

Le 1ąsymbole de contrôle correspond *à la parité des positions* 3, 5, 7, *donc*

$$1 + 0 + 0 \equiv 1 \pmod 2.$$

Le 2ąsymbole de contrôle correspond *à la parité des positions* 3, 6, 7, *donc*

$$1 + 1 + 0 \equiv 0 \pmod 2.$$

Le 3ąsymbole de contrôle correspond *à la parité des positions* 5, 6, 7, *donc*

$$0 + 1 + 0 \equiv 1 \pmod 2.$$

Le mot du code correspondant est :

1	*2*	*3*	*4*	*5*	*6*	*7*
1	*0*	*1*	*1*	*0*	*1*	*0*

Un mot envoyé ne peut être affecté par plus d'une erreur. Si il n'y a pas d'erreur de transmission le syndrome est égal à 000.

Décodage du code de Hamming (7, 4) *Soit* 1011000 *le mot reçu alors que le mot transmis est* 1011010. *Le syndrome est calculé de la façon suivante : Le* 1ąsymbole de contrôle correspond *à la parité des positions* 1, 3, 5, 7, *donc*

$$1 + 1 + 0 + 0 \equiv 0 \pmod 2.$$

Le 2ąsymbole de contrôle correspond *à la parité des positions* 2, 3, 6, 7, *donc*

$$0 + 1 + 0 + 0 \equiv 1 \pmod 2.$$

Le 3ąsymbole de contrôle correspond *à la parité des positions* 4, 5, 6, 7, *donc*

$$1 + 0 + 0 + 0 \equiv 1 \pmod 2.$$

Le syndrome est 110 *l'erreur se trouve donc en position* 6 *(exprimée en binaire), le mot transmis est bien* 1011010.

Ce code, ayant le meilleur rendement possible pour coder un message de quatre symboles $R = 4/7 = 0,57$, est cité, en 1948, *par C. Shannon dans son article fondamental sur la théorie de l'information.*

Chapitre 3

Introduction à l'algèbre de Boole

La conception des circuits logiques qui constituent la couche physique d'un ordinateur nécessite la construction d'opérateurs logiques dont la description relève d'une théorie mathématique appelée *algèbre de Boole*. Outre ce chapitre qui lui est exclusivement consacrée, on trouvera d'autres éléments introductifs à l'algèbre de Boole dans, par exemple, [DanceaMarchand92] et [Darche02].

On considère l'ensemble $\{0,1\}$ que l'on munit des opérations somme logique, produit logique et inversion logique.

3.1 La somme logique

La somme logique est définie par la table d'addition suivante :

$+$	0	1
0	0	1
1	1	1

Soient x et y deux variables booléennes (c'est à dire que $x \in \{0,1\}$ et $y \in \{0,1\}$), on associe une variable booléenne d'identificateur $x+y$ dont les valeurs sont données par la table de vérité suivante qui est la table de vérité de l'opérateur logique OU :

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

La somme logique possède les propriétés algébriques remarquables suivantes :
- associativité : $\forall (x,y,z) \in \{0,1\}^3 \ (x+y)+z = x+(y+z)$;
- commutativité : $\forall (x,y) \in \{0,1\}^2 \ x+y = y+x$;
- élément neutre : $\forall x \in \{0,1\} \ x+0 = 0+x$.

3.2 Le produit logique

Le produit logique est défini par la table de multiplication :

.	0	1
0	0	0
1	0	1

Soient x et y deux variables booléennes, on associe une variable booléenne d'identificateur $x.y$ dont les valeurs sont données par la table de vérité suivante qui est la table de vérité de l'opérateur logique *ET* :

x	y	$x.y$
0	0	0
0	1	0
1	0	0
1	1	1

Le produit logique possède les propriétés algébriques remarquables suivantes :
- associativité : $\forall(x, y, z) \in \{0, 1\}^3\ (x.y).z = x.(y.z)$;
- commutativité : $\forall(x, y) \in \{0, 1\}^2\ x.y = y.x$;
- élément neutre : $\forall x \in \{0, 1\}\ x.1 = 1.x$.

3.3 L'inversion logique

L'inversion est notée avec une barre et est définie par $\overline{0} = 1$ et $\overline{1} = 0$. Soit x une variable booléenne, on associe une variable booléenne d'identificateur $\overline{x}$ dont les valeurs sont données par la table de vérité suivante qui est la table de vérité de l'opérateur logique *NON* :

x	0	1
$\overline{x}$	1	0

Les propriétés de l'inversion logique sont les suivantes :
- $\forall x \in \{0, 1\}\ x + \overline{x} = 1$;
- $\forall x \in \{0, 1\}\ x.\overline{x} = 0$.

3.4 Propriétés dérivées

Tout ensemble muni de deux éléments particuliers 1 et 0 et des trois opérations somme, produit et inversion logiques vérifiant toutes les propriétés énoncées précédemment est une *algèbre de Boole*.

Les propriétés suivantes sont des propriétés dérivées d'une algèbre de Boole :
- **dualité**
  tout énonçé exprimé en fonction de + et de 0 est vrai lorsque l'on remplace + par . et 0 par 1 ;
- **unicité**

les éléments neutres 0 et 1 pour la somme et le produit logiques, respectivement, sont uniques ;
l'inverse de tout élément est unique ;

- **involution**
  $\forall x \in \{0,1\}\ \overline{\overline{x}} = x$;
- **idempotence**
  $\forall x \in \{0,1\}\ x + x = x$,
  $\forall x \in \{0,1\}\ x.x = x$;
- **élément absorbant**
  $\forall x \in \{0,1\}\ x + 1 = 1$,
  $\forall x \in \{0,1\}\ x.0 = 0$;
- **absorption**
  $\forall (x,y) \in \{0,1\}^2\ x + x.y = x$,
  $\forall (x,y) \in \{0,1\}^2\ x.(x + y) = x$;
- **simplification**
  $\forall (x,y) \in \{0,1\}^2\ x + \overline{x}.y = x + y$,
  $\forall (x,y) \in \{0,1\}^2\ x.(\overline{x} + y) = x.y$;
- **formules de De Morgan**
  $\forall (x,y) \in \{0,1\}^2\ \overline{x + y} = \overline{x}.\overline{y}$,
  $\forall (x,y) \in \{0,1\}^2\ \overline{x.y} = \overline{x} + \overline{y}$;
- **distributivité du produit sur la somme**
  $\forall (x,y,z) \in \{0,1\}^3\ x.(y + z) = (x.y) + (x.z)$;
- **distributivité de la somme sur le produit**
  $\forall (x,y,z) \in \{0,1\}^3\ x + (y.z) = (x + y).(x + z)$.

3.5 La somme logique exclusive

La somme logique exclusive est définie par la table d'addition suivante :

$\oplus$	0	1
0	0	1
1	1	0

Soient x et y deux variables booléennes, c'est à dire que $x \in \{0,1\}$ et $y \in \{0,1\}$, on associe une variable booléenne d'identificateur $x \oplus y$ dont les valeurs sont données par la table de vérité suivante qui est la table de vérité de l'opérateur logique *OU–exclusif* :

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

La somme logique exclusive possède les propriétés algébriques suivantes :

- associativité : $\forall (x,y,z) \in \{0,1\}^3\ (x \oplus y) \oplus z = x \oplus (y \oplus z)$;
- commutativité : $\forall (x,y) \in \{0,1\}^2\ x \oplus y = y \oplus x$;
- élément neutre : $\forall x \in \{0,1\}\ x \oplus 0 = 0 \oplus x = x$;

– élément symétrique : $\forall x \in \{0,1\}\ x \oplus x = 0$.

L'ensemble $\{0,1\}$ muni de la somme logique exclusive a une structure de groupe abélien.

3.6 Propriétés liant somme, inversion, et somme exclusive

– distributivité du produit sur la somme exclusive :
$\forall (x,y,z) \in \{0,1\}^3\ x.(y \oplus z) = (x.y) \oplus (x.z)$;
– relations avec l'inversion :
$\forall (x,y) \in \{0,1\}^2\ \overline{x \oplus y} = \overline{x} \oplus y = x \oplus \overline{y}$;
$\forall (x,y) \in \{0,1\}^2\ \overline{x} \oplus \overline{y} = x \oplus y$.

L'ensemble $\{0,1\}$ muni de la somme logique exclusive et du produit logique a une structure d'anneau[1].

3.7 Expressions booléennes

Définition 3.1 *Soient $x_1, x_2, \ldots, x_n$ un ensemble de variables booléennes, on appelle expression booléenne sur les variables $x_1, x_2, \ldots, x_n$ toute variable exprimée à l'aide des opérations + , . et de l'inversion.*

Exemple 3.1 *Voici un exemple d'expression booléenne : $(x.y + z).(y + z)$.*

Les propriétés des expressions booléennes sont les suivantes :
– La somme de 2 expressions booléennes est une expression booléenne ;
– le produit de 2 expressions booléennes est une expression booléenne ;
– l'inverse d'une expression booléenne est une expression booléenne.

Théorème 3.1 *L'ensemble des expressions booléennes sur un ensemble de variables $x_1, x_2, \ldots, x_n$ est une algèbre de Boole.*

3.8 Fonctions booléennes et formes canoniques

Définition 3.2 *On appelle fonction booléenne, la fonction définie de la façon suivante :*

$$\begin{array}{rcl} f : \{0,1\}^n & \rightarrow & \{0,1\} \\ (x_1, \ldots, x_n) & \mapsto & f(x_1, \ldots, x_n). \end{array}$$

Définition 3.3 *On appelle terme minimal ou* minterm *sur un ensemble de variables booléennes, le produit logique de ces variables ou de leur inverses logiques tel que chaque variable, ou son inverse, n'apparaît qu'une seule fois.*

Exemple 3.2 *Dans le cas où $n = 3$, un minterm est de la forme $x_i\ .\ x_j\ .\ x_k$, avec $(i,j,k) \in \{1,2,3\}$. Dans la suite on omettra d'écrire le "." pour spécifier le produit logique, un minterm s'écrira $x_i\ x_j\ x_k$. Les 2^3 minterms sont les suivants :*

$$\overline{x_1}.\overline{x_2}.\overline{x_3},\ \overline{x_1}.\overline{x_2}.x_3,\ \overline{x_1}.x_2.\overline{x_3},\ \overline{x_1}.x_2.x_3$$

$$x_1.\overline{x_2}.\overline{x_3},\ x_1.\overline{x_2}.x_3,\ x_1.x_2.\overline{x_3},\ x_1.x_2.x_3$$

[1] C'est un anneau unitaire, puisque par convention un anneau est unitaire.

Théorème 3.2 *Soit f une fonction booléenne de n variables $x_1, \ldots, x_n$ alors f s'écrit comme une combinaison linéaire des 2^n minterms :*

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

Les coefficients sont obtenus en remplaçant dans $f(x_1, \ldots, x_n)$, chaque variable par 1 si celle-ci figure dans le minterm correspondant, chaque variable par 0 si son inverse figure dans le minterm correspondant.

Exemple 3.3 *Une fonction booléenne de 3 variables s'écrit :*

$$f(x_1, x_2, x_3) = f(0,0,0).\overline{x_1}.\overline{x_2}.\overline{x_3} + f(0,0,1).\overline{x_1}.\overline{x_2}.x_3 +$$

$$f(0,1,0).\overline{x_1}.x_2.\overline{x_3} + f(0,1,1).\overline{x_1}.x_2.x_3 + f(1,0,0).x_1.\overline{x_2}.\overline{x_3} +$$

$$f(1,0,1).x_1.\overline{x_2}.x_3 + f(1,1,0).x_1.x_2.\overline{x_3} + f(1,1,1).x_1.x_2.x_3.$$

Corollaire 3.1 *Toute fonction booléenne de n variables, $x_1, \ldots, x_n$, s'écrit de façon unique comme une somme de minterms où chacun figure une fois au plus, c'est ce qu'on appelle la* forme canonique disjonctive.

Exemple 3.4

$$\begin{aligned} f(x,y,z) &= (x + \overline{y} + z).(\overline{x} + y).\overline{z} \\ &= x.\overline{x}.\overline{z} + x.y.\overline{z} + \overline{x}.\overline{y}.\overline{z} + y.\overline{y}.z + z.x.\overline{z} + z.y.\overline{z} \\ &= x.y.\overline{z} + \overline{x}.\overline{y}.\overline{z} \end{aligned}$$

On remarque que dans cet exemple, $f(1,1,0) = f(0,0,0) = 1$ et
$f(0,0,1) = f(0,1,0) = f(1,0,0) = f(0,1,1,) = f(1,0,1) = f(1,1,1) = 0.$

Il existe une forme canonique duale, appelée forme canonique conjonctive :

Définition 3.4 *On appelle terme maximal ou* maxterm *sur un ensemble de variables booléennes, la somme logique de ces variables ou de leur inverses logiques tel que chaque variable, ou son inverse n'apparaît qu'une seule fois.*

Exemple 3.5 *Dans le cas où $n = 3$, un maxterm est de la forme $x_i + x_j + x_k$, avec $(i,j,k) \in \{1,2,3\}$. Les 2^3 maxterms sont les suivants :*

$$\overline{x_1} + \overline{x_2} + \overline{x_3},\ \overline{x_1} + \overline{x_2} + x_3,\ \overline{x_1} + x_2 + \overline{x_3},\ \overline{x_1} + x_2 + x_3$$

$$x_1 + \overline{x_2} + \overline{x_3},\ x_1 + \overline{x_2} + x_3,\ x_1 + x_2 + \overline{x_3},\ x_1 + x_2 + x_3$$

Théorème 3.3 *Soit f une fonction booléenne de n variables $x_1, \ldots, x_n$ alors f s'écrit comme un produit des 2^n sommes, où dans chacune figure l'un des 2^n maxterms :*

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

Les coefficients sont obtenus en remplaçant dans $f(x_1, \ldots, x_n)$, chaque variable par 0 si celle-ci figure dans le maxterm correspondant, chaque variable par 1 si son inverse figure dans le maxterm correspondant.

Exemple 3.6 *Une fonction booléenne de* 3 *variables s'écrit :*

$$f(x_1,x_2,x_3) = (f(0,0,0) \ + \ x_1+x_2+x_3).(f(0,0,1) \ + \ x_1+x_2+\overline{x_3}).$$

$$(f(0,1,0) \ + \ x_1+\overline{x_2}+x_3).(f(0,1,1) \ + \ x_1+\overline{x_2}+\overline{x_3}).(f(1,0,0) \ + \ \overline{x_1}+x_2+x_3).$$

$$(f(1,0,1) \ + \ \overline{x_1}+x_2+\overline{x_3}).(f(1,1,0) \ + \ \overline{x_1}+\overline{x_2}+x_3).(f(1,1,1) \ + \ \overline{x_1}+\overline{x_2}+\overline{x_3}).$$

Corollaire 3.2 *Toute fonction booléenne de n variables,* $x_1,\ldots,x_n$*, s'écrit de façon unique comme un produit de maxterms où chacun figure une fois au plus, c'est ce qu'on appelle la* forme canonique conjonctive.

Exemple 3.7

$$\begin{aligned} f(x,y,z) &= x.\overline{y}+z \\ &= (x+z).(\overline{y}+z) \\ &= (x+y.\overline{y}+z).(x.\overline{x}+\overline{y}+z) \\ &= (x+z+y.\overline{y}).(\overline{y}+z+x.\overline{x}) \\ &= (x+z+y).(x+z+\overline{y}).(\overline{y}+z+x).(\overline{y}+z+\overline{x}) \\ &= (x+z+y).(x+\overline{y}+z).(\overline{x}+\overline{y}+z) \end{aligned}$$

On remarque que dans cet exemple, $f(0,0,0) = f(0,1,0) = f(1,1,0) = 0$ *et* $f(0,0,1) = f(1,0,0) = f(0,1,1,) = f(1,0,1) = f(1,1,1) = 1$*.*

Toute fonction booléenne de n variables peut s'écrire comme une combinaison linéaire des 2^n minterms, cela peut se représenter par une table de vérité comportant $n+1$ colonnes et 2^n lignes comme suit :

entier	minterm	x_1	x_2	$\ldots$	x_n	$f(x_1,\ldots,x_n)$
0	$\overline{x_1}\ldots\overline{x_n}$	0	0	$\ldots$	0	$f(0,\ldots,0)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
2^n-1	$x_1\ldots x_n$	1	1	$\ldots$	1	$f(1,\ldots,1)$

Exemple 3.8 *La table de vérité de la fonction booléenne de trois variables est :*

entier	*minterm*	x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	$\overline{x_1}.\overline{x_2}.\overline{x_3}$	0	0	0	$f(0,0,0)$
1	$\overline{x_1}.\overline{x_2}.x_3$	0	0	1	$f(0,0,1)$
2	$\overline{x_1}.x_2.\overline{x_3}$	0	1	0	$f(0,1,0)$
3	$\overline{x_1}.x_2.x_3$	0	1	1	$f(0,1,1)$
4	$x_1.\overline{x_2}.\overline{x_3}$	1	0	0	$f(1,0,0)$
5	$x_1.\overline{x_2}.x_3$	1	0	1	$f(1,0,1)$
6	$x_1.x_2.\overline{x_3}$	1	1	0	$f(1,1,0)$
7	$x_1.x_2.x_3$	1	1	1	$f(1,1,1)$

3.9 Image d'une fonction booléenne

Définition 3.5 *On appelle* image *de la fonction $f(x_1, \ldots, x_n)$, notée : f, le vecteur formé par les 2^n coefficients de la forme canonique disjonctive dans l'ordre croissant de la représentation binaire des entiers de 0 à $2^n - 1$, c'est à dire de $f(0, \ldots, 0)$ à $f(1, \ldots, 1)$.*

Exemple 3.9 *L'image de $f(x, y, z) = x.y.\overline{z} + \overline{x}.\overline{y}.\overline{z}$ est : $f = 10000010$.*

Les propriétés de l'image sont :

$: f \,.g \; = : f. : g$

$: f \; + g \; = : f + : g$

3.10 Simplification des fonctions booléennes

La conception des circuits logiques nécessite l'expression d'une fonction booléenne correspondant à la fonction logique souhaitée, puis la simplification de cette fonction booléenne de façon à réaliser le circuit le plus simple possible et le moins coûteux possible. Il existe de nombreuses méthodes de simplification implémentables, parmi lesquelles l'algorithme de Nelson, l'algorithme de Quine-McCluskey, et l'utilisation des BDDs. Toutefois, dans le cadre de ce cours, nous nous contentons d'étudier les méthodes dites manuelles : simplifications algébriques d'une part et tables de Karnaugh d'autre part.

Simplifications algébriques

La simplification algébrique d'une fonction booléenne s'effectue directement à partir des propriétés de l'algèbre de Boole.

Exemple 3.10

$$\begin{aligned} f(x,y,z) &= \overline{x}.y.z + x.\overline{y}.z + x.y.\overline{z} + x.y.z \\ &= \overline{x}.y.z + x.\overline{y}.z + x.y.\overline{z} + x.y.z + x.y.z + x.y.z \\ &= (\overline{x}.y.z + x.y.z) + (x.\overline{y}.z + x.y.z) + (x.y.\overline{z} + x.y.z) \\ &= (\overline{x} + x).y.z + (\overline{y} + y).x.z + (\overline{z} + z).x.y \\ &= y.z + x.z + x.y \end{aligned}$$

Simplification par les tables de Karnaugh

Une table de Karnaugh est une table associée à la valeur de la fonction booléenne dont les cases correspondent aux différentes lignes de la table de vérité.

Table de Karnaugh à 2 variables

x_2 \ x_1	0	1
0	$f(0,0)$	$f(1,0)$
1	$f(0,1)$	$f(1,1)$

Table de Karnaugh à 3 variables

x_3 \ $x_1\ x_2$	00	01	11	10
0	$f(0,0,0)$	$f(0,1,0)$	$f(1,1,0)$	$f(1,0,0)$
1	$f(0,0,1)$	$f(0,1,1)$	$f(1,1,1)$	$f(1,0,1)$

Table de Karnaugh à 4 variables

$x_3\ x_4$ \ $x_1\ x_2$	00	01	11	10
00	$f(0,0,0,0)$	$f(0,1,0,0)$	$f(1,1,0,0)$	$f(1,0,0,0)$
01	$f(0,0,0,1)$	$f(0,1,0,1)$	$f(1,1,0,1)$	$f(1,0,0,1)$
11	$f(0,0,1,1)$	$f(0,1,1,1)$	$f(1,1,1,1)$	$f(1,0,1,1)$
10	$f(0,0,1,0)$	$f(0,1,1,0)$	$f(1,1,1,0)$	$f(1,0,1,0)$

Les lignes et les colonnes d'une table de Karnaugh sont écrites dans l'ordre binaire réfléchi, ce qui met en évidence la propriété d'adjacence, c'est-à-dire, deux cases adjacentes de la table contenant la valeur booléenne "1" correspondent à 2 minterms qui ne diffèrent que par l'état d'une seule variable. Soit x_i cette variable,

$$\begin{aligned} f(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_n) &= \ldots + x_1\ldots\overline{x_i}\ldots x_n + x_1\ldots x_i\ldots x_n + \ldots \\ &= \ldots + x_1\ldots x_{i-1}.(\overline{x_i}+x_i).x_{i+1}\ldots x_n + \ldots \\ &= \ldots + x_1\ldots x_{i-1}.x_{i+1}\ldots x_n + \ldots \end{aligned}$$

En utilisant la propriété $\overline{x_i} + x_i = 1$, on obtient, à partir de 2 minterms, un produit qui ne contient plus la variable x_i. Les tables de Karnaugh sont construites pour mettre en évidence graphiquement la propriété d'adjacence. La simplification s'opère en regroupant, par un nombre égal à une puissance de 2, les cases adjacentes contenant la valeur 1. De plus, *les regroupements se font de telle sorte qu'un maximum de valeurs* 1 *soient englobées dans un minimum de regroupements.* Autrement dit, on cherche à réaliser les plus gros regroupements possibles, et en plus petit nombre possible. Enfin, il est important de noter que la propriété d'adjacence permet de regrouper des valeurs 1 aux extrémités du tableaux. On obtient autant de minterms dans l'écriture simplifiée de la fonction qu'il y a de regroupements dans la table.

Exemple 3.11 *Soit la fonction*

$$f(x_1,x_2,x_3) = x_1.\overline{x_2}.\overline{x_3} + x_1.\overline{x_2}.x_3 + x_1.x_2.x_3 + \overline{x_1}.\overline{x_2}.x_3$$

La table de Karnaugh correspondante est :

x_3 \ $x_1\ x_2$	00	01	11	10
0				1
1	1		1	1

Il y a trois regroupements, dont un qui réunit les 1 *des deux extrémités de la deuxième ligne. Une forme simplifiée de la fonction est :* $f(x_1, x_2, x_3) = x_1.\overline{x_2} + x_1.x_3 + \overline{x_2}.x_3$

Exemple 3.12 *Soit la fonction*

$$f(x_1, x_2, x_3, x_4) = \quad \overline{x_1}.\overline{x_2}.\overline{x_3}.x_4 + \overline{x_1}.x_2.\overline{x_3}.x_4 + \overline{x_1}.x_2.x_3.\overline{x_4} + \overline{x_1}.x_2.x_3.x_4 + x_1.\overline{x_2}.x_3.x_4 + x_1.x_2.\overline{x_3}.\overline{x_4} + x_1.x_2.\overline{x_3}.x_4 + x_1.x_2.x_3.x_4$$

La table de Karnaugh correspondante est :

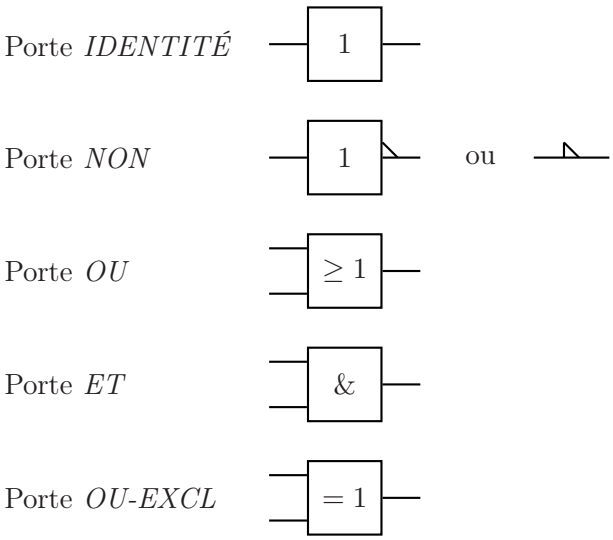
$x_3\ x_4$ \ $x_1\ x_2$	00	01	11	10
00			1	
01	1	1	1	
11		1	1	1
10		1		

On peut remarquer que les regroupements sont bien les plus gros possibles et que chacun des 1 *composant le regroupement central est contenu dans un autre regroupement ; par conséquent, et compte-tenu du critère de minimisation du nombre de regroupements, le regroupement central de quatre* 1 *n'est pas retenu dans l'énoncé de la forme simplifiée. Une forme simplifiée de la fonction est donc :* $f(x_1, x_2, x_3, x_4) = x_1.x_2.\overline{x_3} + \overline{x_1}.\overline{x_3}.x_4 + \overline{x_1}.x_2.x_3 + x_1.x_3.x_4$

3.11 Réalisation d'un circuit logique

Il est commode de donner une représentation graphique d'une fonction booléenne sous la forme d'un schéma normalisé, réalisé à l'aide d'un ensemble de *portes logiques* associées aux opérateurs booléens de base. Chacune de ces portes comporte une ou plusieurs entrées (les variables booléennes sur lesquelles porte l'opérateur en question) et une sortie. Le branchement en cascade des sorties de certaines portes sur les entrées d'autres portes permet de représenter n'importe quelle fonction booléenne sous la forme d'un *circuit logique* associé à cette fonction. Autrement dit, l'ensemble de toutes les entrées du circuit sont les variables booléennes sur lesquelles porte la fonction, et la sortie du circuit correspond à la réalisation de la fonction. De même, plusieurs circuits logiques peuvent être combinés comme les composants élémentaires d'un ensemble plus important, par exemple une unité arithmétique et logique.

La représentation suivante des portes logiques de base est issue de la norme ANSI/IEEE Std 91-1984, qui a pour objectif de faciliter la représentation de circuits intégrés complexes :



La démarche suivie dans la conception d'un circuit passe globalement par les étapes suivantes :

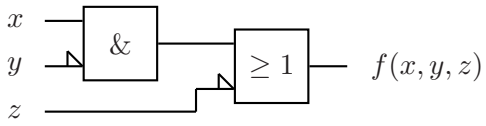
1. Spécification de la fonction logique à réaliser (en français ou tout autre dialecte familier...).
2. Tables de vérité, si nécessaire, ou lorsque c'est possible.
3. Simplifications booléennes.
4. Tracé du circuit (éviter l'esthétique « spaghettis »).

Exemple 3.13 *On considère la fonction logique suivante :*

1. $f(x,y,z) = \overline{x}.\overline{y}.\overline{z} + \overline{x}.y.\overline{z} + x.\overline{y}.\overline{z} + x.\overline{y}.z + x.y.\overline{z}$
2. *La table de vérité correspondante est :*

x	y	z	$f(x,y,z)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

3. *Après simplification,* $f(x,y,z) = x.\overline{y} + \overline{z}$
4. *Le circuit correspondant est :*



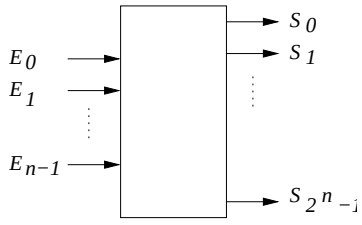
Chapitre 4

Quelques circuits combinatoires

Un circuit *combinatoire* est un circuit logique dont l'état des sorties ne dépend que des valeurs assignées aux variables d'entrée au moment considéré (par opposition aux circuits *séquentiels*, étudiés au chapitre suivant, et dont les sorties dépendent non seulement des entrées, mais aussi de l'*histoire* du circuit). Le présent chapitre décrit succintement le développement théorique de quelques structures combinatoires de base qui jouent un rôle fondamental dans la conception des différents blocs d'un ordinateur : décodeurs, multiplexeurs, démultiplexeurs, réseaux logiques programmables, et additionneurs. L'accent est mis sur les étapes de conception d'un circuit combinatoire : spécification de la fonction logique réalisée, traduction en tables de vérité quand c'est possible, optimisation par rapport à certains critères (comme la minimisation du nombre de portes, ou l'emploi d'un certain type de portes seulement), tracé du circuit. Enfin, l'analyse de certains circuits montre que d'autres outils, plus faciles à comprendre ou utiliser, peuvent être employés (par exemple les *tables de fonctionnement* en lieu et place des tables de vérité dans le cas d'un multiplexeur).

4.1 Décodeur

Un décodeur est un circuit comportant :
- n entrées ;
- 2^n sorties, dont une toujours à 1 ;
- telle qu'elle est celle dont le numéro est l'entier codé par la configuration binaire des entrées.



Autrement dit, un décodeur permet de sélectionner une sortie $S_i, 0 \leq i \leq 2^n - 1$, avec $i = \text{valeur}_{10}(E_n, \ldots, E_0)$.

Exemple 4.1 *Soit un décodeur 3–8 (3 entrées, 8 sorties) et tel que la configuration binaire des entrées soit* $E_2 = 1, E_1 = 1, E_0 = 0$*, soit* 110*. L'entier codé par* $E_2E_1E_0$ *est* 6*, par conséquent la sortie* S_6 *est sélectionnée, elle vaut* 1*.*

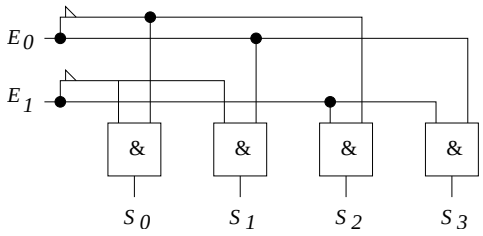
Spécification du circuit logique réalisant une fonction de décodage. Pour fixer les idées, on décide de réaliser un décodeur 2–4. La table de vérité comporte deux entrées E_1 et E_0, quatre sorties S_3, S_2, S_1, S_0 telles que la sortie dont le numéro d'indice de la sortie est codé par la configuration binaire des entrées E_1E_0 est mise à 1.

E_1	E_0	S_0	S_1	S_2	S_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

On déduit immédiatement les équations booléennes des sorties en fonction des entrées :

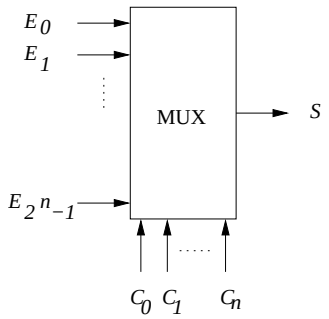
$$\begin{aligned} S_0 &= \overline{E_1}.\overline{E_0} \\ S_1 &= \overline{E_1}.E_0 \\ S_2 &= E_1.\overline{E_0} \\ S_3 &= E_1.E_0 \end{aligned}$$

d'où le circuit :



4.2 Multiplexeur

Un multiplexeur est un circuit comprenant :
- n fils de contrôle $C_n, C_{n-1}, \ldots, C_0$;
- 2^n entrées ;
- Une sortie S qui prend la valeur de l'une des entrées, de telle sorte que la configuration binaire des n fils de contrôle code le numéro d'indice de cette entrée.



Autrement dit, un multiplexeur permet de connecter une entrée $E_i, 0 \leq i \leq 2^n - 1$, à la sortie S de telle sorte que $i = \text{valeur}_{10}(C_n, \ldots, C_0)$.

Exemple 4.2 *Soit un multiplexeur 8–1 (8 entrées, 1 sortie) et tel que la configuration binaire des trois variables de contrôle soit* $C_2 = 1, C_1 = 1, C_0 = 0$*, soit* 110*. L'entier codé par* $C_2C_1C_0$ *est* 6*, par conséquent la sortie* S *prend la valeur (*1 *ou* 0*) qui se trouve sur l'entrée sélectionnée* E_6*.*

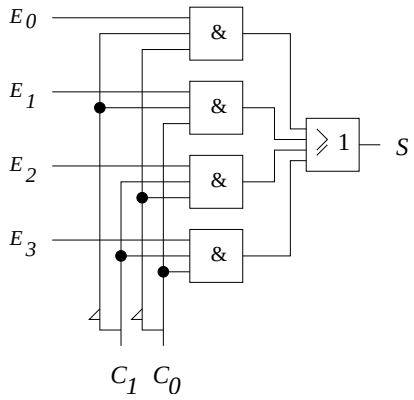
Spécification du circuit logique réalisant une fonction de multiplexage. Pour fixer les idées, on décide de réaliser un multiplexeur 4–1. Parce que les entrées sont mutuellement exclusives, il est possible d'envisager la construction d'une table de vérité simplifiée, appelée *table de fonctionnement*, et dont la colonne de sortie, plutôt que de contenir des 1 et des 0, contient les variables d'entrées. La table en question contient donc deux variables de contrôle C_1 et C_0, et une sortie S qui prend la valeur de l'une des quatre entrées E_3, E_2, E_1, E_0 quand la configuration binaire C_1C_0 code le numéro d'indice de la dite entrée.

C_1	C_0	S
0	0	E_0
0	1	E_1
1	0	E_2
1	1	E_3

Bien sûr, il existe, équivalent à cette table de vérité un peu inhabituelle, une table "classique" comportant une colonne asscié à E_0, une à E_1, une à E_2, et une à E_3, ainsi que les colonnes associées à C_1 et C_0 et la colonne de sortie S... Mais cette table est moins aisée à manipuler. En effet, elle "masque" la propriété qui nous intéresse directement (connexion de la sortie à l'une des entrées en fonction des variables de contrôle), et il est plus laborieux d'en extraire la fonction booléenne. L'équation boléenne de sortie se déduit sans difficulté de la table de fonctionnement réalisée ici :

$$S = \overline{C_1}.\overline{C_0}.E_0 + \overline{C_1}.C_0.E_1 + C_1.\overline{C_0}.E_2 + C_1.C_0.E_3$$

d'où le circuit :



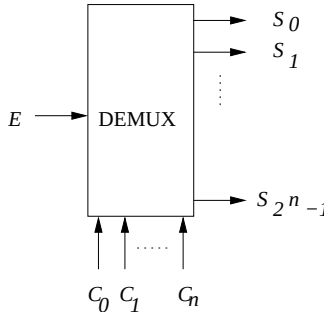
Remarque : Un multiplexeur permet de réaliser n'importe quelle fonction booléenne de ses variables de contrôle (pas toujours de manière économique). Il suffit pour cela de fixer les valeurs des variables d'entrées au regard de la fonction qu'on désire réaliser. Par exemple, sur un multiplexeur 4–1, en fixant $E_0 = 0, E_1 = 1, E_2 = 1, E_3 = 0$, on obtient la fonction :

$$f(C_1, C_0) = \overline{C_1}.C_0 + C_1.\overline{C_0}$$

qui représente le *OU–exclusif* des deux variables C_1, C_0.

4.3 Demultiplexeur

Un démultiplexeur est un circuit comprenant :
- n fils de contrôle $C_n, C_{n-1}, \ldots, C_0$;
- 2^n sorties $S_0, \ldots S_{2^n-1}$;
- Une entrée E qui donne sa valeur à l'une des sorties, de telle sorte que la configuration binaire des n fils de contrôle code le numéro d'indice de cette sortie.



Autrement dit, un démultiplexeur permet d'aiguiller l'entrée E vers la sortie $S_i, 0 \leq i \leq 2^n - 1$, de telle sorte que $i = \text{valeur}_{10}(C_n, \ldots, C_0)$.

Exemple 4.3 *Soit un démultiplexeur 1–8 (1 entrée, 8 sorties) et tel que la configuration binaire des trois variables de contrôle soit* $C_2 = 1, C_1 = 1, C_0 = 0$*, soit* 110*. L'entier codé par* $C_2C_1C_0$ *est* 6*, par conséquent la sortie* S_6 *est sélectionnée. Elle prend la valeur (*1 *ou* 0*) qui se trouve sur l'entrée* E*.*

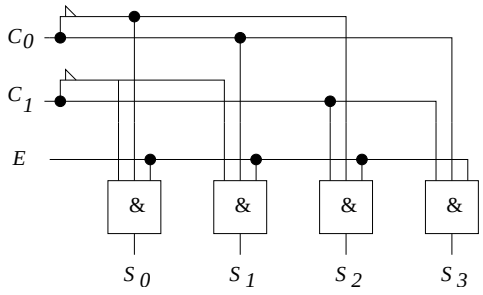
Spécification du circuit logique réalisant une fonction de démultiplexage. Comme d'habitude (!), on décide de réaliser (par exemple...) un démultiplexeur 4–1. La table de vérité comporte deux variables de contrôle C_1 et C_0, quatre sorties S_0, S_1, S_2, S_3 dont une prend la valeur de l'entrée E quand la configuration binaire C_1C_0 code le numéro d'indice de cette sortie. En suivant cette contrainte, on obtient une table de fonctionnement dont la colonne de sortie, plutôt que de contenir des 1 et des 0, contient les variables d'entrées.

C_1	C_0	S_0	S_1	S_2	S_3
0	0	E	0	0	0
0	1	0	E	0	0
1	0	0	0	E	0
1	1	0	0	0	E

Remarque : D'une certaine manière, le démultiplexage peut être vu comme un raffinement de la fonction de décodage : la sortie n'est plus systématiquement mise à 1 (cas du décodage), mais prend la valeur de l'entrée E.
Les équations booléennes de sortie se déduisent sans difficulté de la table de vérité :

$$\begin{aligned} S_0 &= \overline{C_1}.\overline{C_0}.E \\ S_1 &= \overline{C_1}.C_0.E \\ S_2 &= C_1.\overline{C_0}.E \\ S_3 &= C_1.C_0.E \end{aligned}$$

d'où le circuit :



4.4 Réseaux logiques programmables

Les réseaux logiques programmables sont des matrices de type ET-OU-NON qui peuvent être programmés pour réaliser n'importe quelle fonction logique. Il existe deux grandes familles de réseaux logiques programmables : les PLA (Programmable Logic Arrays) et les PAL (Programmable Array Logic).

- Un PLA est constitué de la réunion de portes ET, OU, NON, liées ensemble pour réaliser une structure logique combinatoire sous la forme ET-OU, ou sous la forme NON-ET-OU. Deux connexions, l'une directe, l'autre complémentée, partent de chaque entrée vers toutes les portes ET. Un autre niveau de connexion fait les liaisons entre chaque sortie d'une porte ET et toutes les entrées des portes OU. Les sorties des portes OU, qui représentent les sorties du circuit, sont elles aussi doublées, données à la fois sous forme directe et complémentée. Les trois niveaux de connexions (entrées vers les portes ET, portes ET vers les portes OU, portes OU vers les sorties) sont réalisés à l'aide de microfusibles. Durant la programmation, certains des microfusibles sont détruits tandis que d'autres restent intacts, de façn à réaliser la forme canonique disjonctive des fonctions combinatoires imposées.
- Un PAL est un PLA ne disposant que d'un seul niveau de microfusibles (au lieu de trois) entre les entrées et physiques et les portes ET. Les sorties des portes ET sont liées durant le processus de fabrication directement aux portes finales OU, sans l'utilisation de microfusibles. Autrement dit, les portes ET ne peuvent être partagées, et chaque porte OU doit avoir ses propres portes ET.

Les réseaux logiques programmables sont utilisés surtout pour la matérialisation de structures combinatoires complexes.

4.5 Additionneur

La réalisation d'un additionneur complet (1 bit avec 1 bit avec retenue d'entrée) passe par la réalisation d'un demi-additionneur (1 bit avec 1 bit sans retenue d'entrée). Le *demi-additionneur* est un circuit qui réalise l'addition de deux bits. La somme s'obtient sur deux bits, S pour le poids faible, R pour le poids fort (la retenue). A partir de la table de vérité suivante :

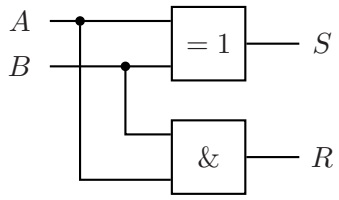
A	B	R	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

on obtient :

$$\begin{aligned} S &= A.\overline{B} + \overline{A}.B \\ &= A \oplus B \end{aligned}$$

$$R = A.B$$

Le demi-additionneur est réalisé par le circuit suivant :



Le demi-additionneur correspond à l'addition de deux bits isolés. Pour réaliser l'addition de deux nombres binaires, il faut tenir compte lors de l'addition de deux bits de rang n de la retenue de rang $n-1$. L'opérateur qui réalise l'addition de deux bits, en tenant compte d'une éventuelle retenue en entrée, s'appelle *étage d'additionneur*. Il possède trois entrées A, B, et R' et deux sorties S et R :

A	B	R'	R	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

On déduit les équations :

$$\begin{aligned} S &= \overline{A}.\overline{B}.R' + \overline{A}.B.\overline{R'} + A.\overline{B}.\overline{R'} + A.B.R' \\ &= \overline{R'}.(\overline{A}.B + A.\overline{B}) + R'.(\overline{A}.\overline{B} + A.B) \\ &= \overline{R'}.(A \oplus B) + R'.(\overline{A \oplus B}) \\ &= R' \oplus A \oplus B \end{aligned}$$

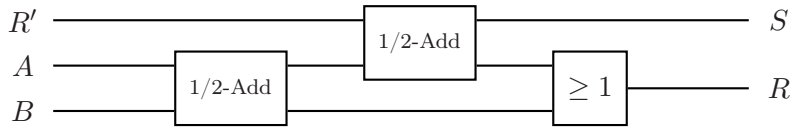
$$\begin{aligned} R &= \overline{A}.B.R' + A.\overline{B}.R' + A.B.\overline{R'} + A.B.R' \\ &= \underbrace{(A \oplus B).R'}_{R_1} + \underbrace{A.B}_{R_2} \end{aligned}$$

La retenue comporte donc deux termes :

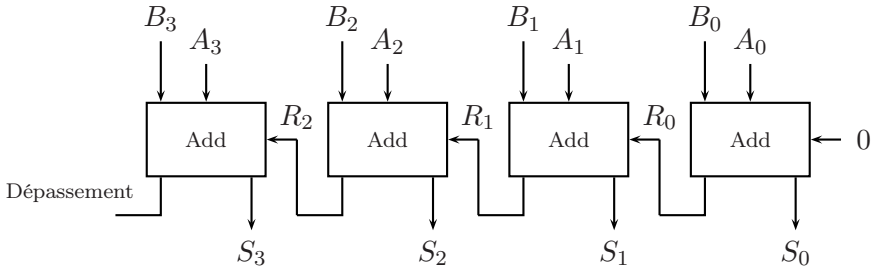
- $R_2 = A.B$ qui représente la retenue générée dans l'étage d'additionneur ;
- $R_1 = (A \oplus B).R'$ qui représente la retenue propagée par l'étage d'additionneur.

On peut alors construire l'étage d'additionneur en réalisant la somme de A et B dans un premier demi-additionneur, puis en additionnant au résultat la retenue R' en provenance de l'étage

précédent. On obtient la somme S et deux retenues : R_2 générée dans l'étage, R_1 propagée par l'étage, réunies dans un OU logique pour obtenir la retenue R :



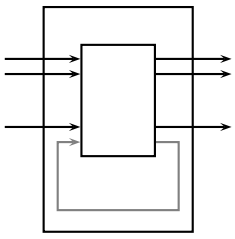
Le schéma d'un additionneur pour deux nombres sur quatre bits est :



Chapitre 5

Circuits séquentiels, registres, compteurs

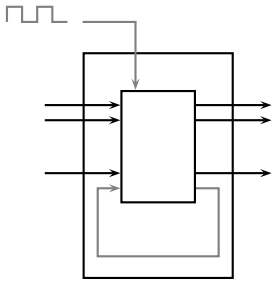
Un circuit *séquentiel* est un circuit combinatoire dont les valeurs des variables de sortie dépendent non seulement des valeurs de variables d'entrée, mais aussi de *variables internes*, autrement dit de variables réalisant un rebouclage du circuit combinatoire sur lui-même, comme cela est représenté dans la figure suivante :



Δt : temps de propagation rétroactif

L'introduction d'une boucle de rétroaction a pour effet de rendre nécessaire la prise en compte d'un *temps de propagation* des valeurs des variables internes qui permette une stabilisation du circuit dans un état donné. On obtient ainsi la définition suivante : un circuit *séquentiel* est un circuit dont l'état des sorties à un instant t donné dépend à la fois des valeurs assignées aux variables d'entrée et de l'état qu'avaient les sorties à l'instant $t - 1$. La stabilisation du circuit séquentiel dans un état donné, lorsqu'elle est possible, induit un effet de *mémorisation*, c'est pourquoi on utilise ce type de circuit pour représenter le fonctionnement des registres d'un ordinateur. L'*élément de mémoire* est défini comme un circuit séquentiel à deux états, 0 et 1, utilisé pour stocker le contenu d'un bit : l'état du circuit est modifié par des signaux de commutation aux entrées ; il possède deux sorties dont l'une est le complément de l'autre.

Les circuits séquentiels se divisent en deux catégories : *asynchrone* et *synchrone*. Dans un circuit asynchrone, les variables du système évoluent librement au cours du temps, autrement dit l'action des entrées est prise en compte dès leur changement d'état. Au contraire, un circuit synchrone est tel que l'évolution des variables dépend d'une impulsion d'horloge comme un des signaux d'entrée :



Δt : temps de propagation rétroactif

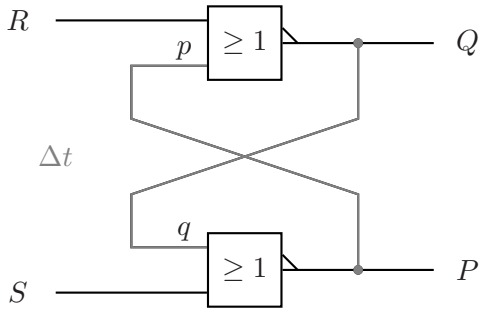
Par conséquent, les entrées d'un circuit synchrone ne sont sensibles aux signaux qui leur sont appliqués que pendant un court intervalle de temps déterminé par le signal d'horloge.

De fait, les changements synchrones rendent plus fiables le fonctionnement des circuits séquentiels, c'est pourquoi les circuits séquentiels synchrones sont utilisés pour la réalisation de la plupart des blocs fonctionnels d'un ordinateur. Inversement, les circuits asynchrones sont peu nombreux et les exemples étudiés dans la suite (bascules RS, D, JK, et T) servent essentiellement à introduire leurs pendants synchrones.

5.1 Circuits asynchrones

Bascule RS asynchrone

La bascule RS est un circuit séquentiel à deux entrées R et S (pour *Reset* et *Set* respectivement), défini par le schéma suivant :



Les équations des sorties sont :

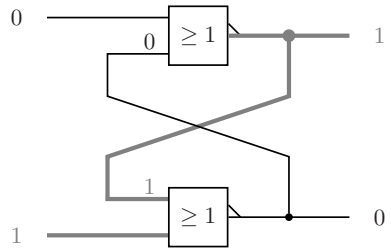
$$\begin{array}{rcccl} Q & = & \overline{R+p} & = & \overline{R}.\overline{p} \\ P & = & \overline{S+q} & = & \overline{S}.\overline{q} \end{array}$$

Dans ces équations, q et p caractérisent l'état actuel du circuit, et Q et P l'état suivant du circuit, état qui se produit après l'application des signaux aux entrées. Notons qu'il existe une version duale comportant des portes *NON-ET* à la place des portes *NON-OU*. Il est intéressant d'illustrer le fonctionnement de la bascule RS par une étude de cas.

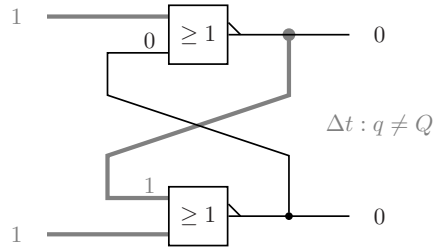
Exemples de fonctionnement de la bascule RS

Exemple 5.1 $\boxed{R = 0, S = 1 \rightarrow R = 1, S = 1}$

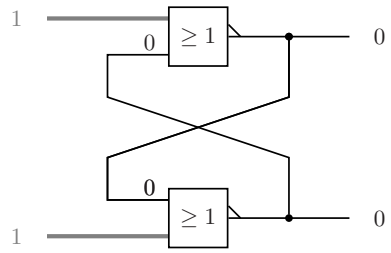
On suppose la bascule dans un état stable, avec $R = 0$, $S = 1$, et on commute l'entrée R à 1. La bascule passe successivement par les états suivants :



$R = 0, S = 1$



$R = 1, S = 1$

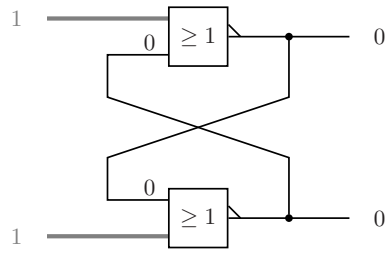


Propagation du signal

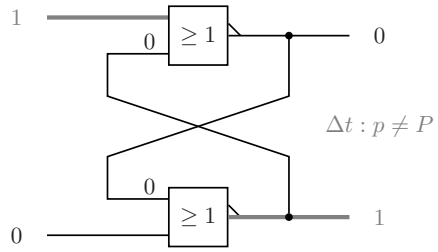
On constate sur cet exemple que la mise à 1 de l'entrée R correspond effectivement à l'émission d'un signal Reset, la bascule mémorisant l'état 0 ; les sorties Q et P n'étant plus complémentées, la bascule ne fonctionne clairement pas comme un élément de mémoire.

Exemple 5.2 $\boxed{R = 1, S = 1 \rightarrow R = 1, S = 0}$

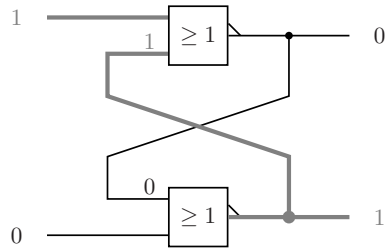
On repart de la configuration obtenue dans l'exemple précédent. L'entrée R étant positionnée à 1, on commute l'entrée S de 1 à 0. La bascule passe successivement par les états suivants :



$R = 1, S = 1$



$R = 1, S = 0$



Propagation du signal

Avec la mise à 0 de l'entrée S ($Set = 0$), la bascule mémorise bien l'état 0 ; cette fois les sorties Q et P sont complémentées.

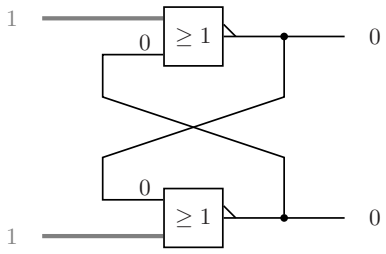
En résumé, et à partir des deux exemples précédents, on constate que :

- Avec $S = 1$ (Set), la bascule mémorise la valeur 1.
- Avec $R = 1$ ($Reset$), la bascule mémorise la valeur 0.
- En raison du temps de propagation rétroactif Δt, la bascule passe par une phase d'instabilité ($q \neq Q$ ou $p \neq P$) avant de se stabiliser dans un état déterminé.
- La bascule ne vérifie pas la dernière contrainte caractéristique d'un élément de mémoire (les sorties Q et P ne sont pas complémentaires l'une de l'autre dans tous les cas).

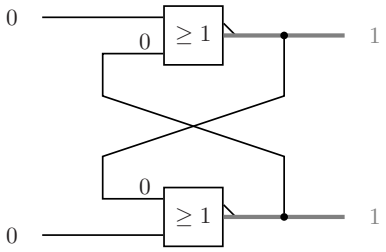
En fait les problèmes entrevus jusqu'ici ne sont pas les seuls, comme le montre l'exemple suivant.

Exemple 5.3 $\boxed{R = 1, S = 1 \rightarrow R = 0, S = 0}$

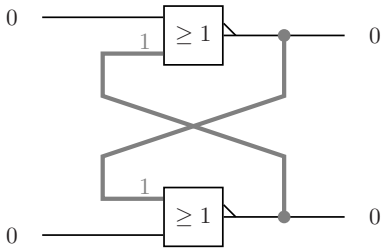
On fait passer simultanément les entrées R et S de 1 à 0.



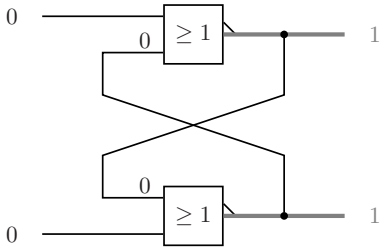
$R = 1, S = 1$



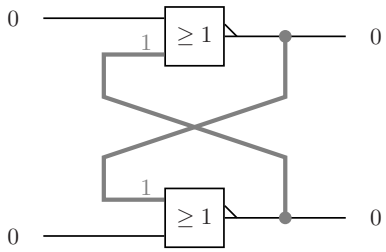
$R = 0, S = 0$



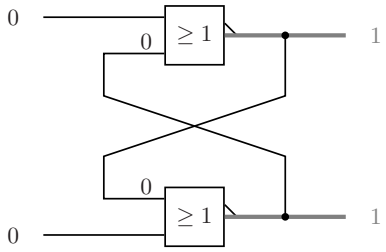
oscillation...



oscillation...



oscillation...



oscillation...

⋮ ⋮

On constate une instabilité chronique de la bascule.

En basculant simultanément les entrées de $R = S = 1$ à $R = S = 0$:

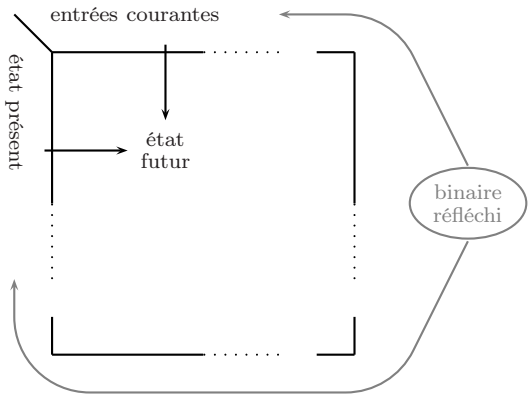
- La bascule entre dans une phase d'oscillation permanente due au fait qu'à tout instant $q \neq Q$ et $p \neq P$.
- A aucun moment les sorties Q et P ne sont complémentaires l'une de l'autre, ce qui contredit totalement la dernière propriété caractéristique d'un élément de mémoire.

Etude systématique

Une étude systématique du comportement du circuit passe par la détermination de ses *états stables* : un circuit est dans un état stable s'il peut rester dans le même état pour un temps indéfini, ce qui s'exprime ici par les conditions suivantes :

$$Q = q \quad \text{et} \quad P = p$$

La détermination des états stables est réalisée à partir d'une forme particulière de diagramme de Karnaugh, appelée *matrice d'états assignés*, réalisée en fonction des entrées présentes et de l'ensemble des valeurs courantes des sorties du circuit. Dans chaque cellule du diagramme, on positionne l'état futur de la bascule (les valeurs futures des sorties) à partir des entrées considérées et de l'état courant de la bascule (les valeurs courantes des sorties), comme illustré par la figure suivante :



Ainsi, la matrice d'états assignés réalisée pour la bascule RS étudiée ici est telle que dans chaque cellule de la matrice la valeur de la sortie Q est inscrite dans la partie gauche et la valeur de la sortie P dans la partie droite.

qp \ RS	00	01	11	10
00	...	...	...	...
01	01	...	...	...
11	...	...	...	...
10	...	...	...	...

Par exemple, dans le cas pour lequel les valeurs des entrées R et S sont respectivement 0 et 0 et l'état courant est décrit par $q = 0$, $p = 1$, la cellule correspondante se trouve à l'intersection de la colonne 00 et de la ligne 01 ; les équations de sorties donnent alors :

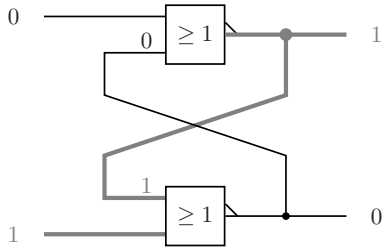
$$\begin{aligned} Q &= \bar{R}.\bar{p} = \bar{0}.\bar{1} = 1.0 = 0 \\ P &= \bar{S}.\bar{q} = \bar{0}.\bar{0} = 1.1 = 1 \end{aligned}$$

On place donc 01 dans la cellule en question. On remarque que pour les valeurs de cette cellule $Q = q$ et $P = p$, par conséquent elle désigne un état stable. En procédant ainsi pour chaque cellule du tableau, et en marquant l'ensembles des états stables, on obtient la matrice d'états assignés suivante :

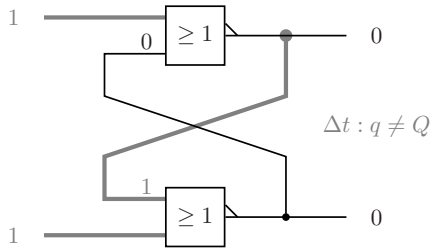
qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00

Les transitions d'un état de la bascule à un autre représentent alors un ensemble de chemins dans la matrice. En reprenant les exemples précédents, on a :

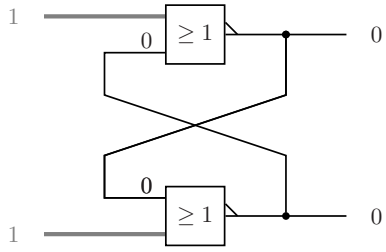
Exemple 5.1 (suite) $\boxed{R = 0, S = 1 \rightarrow R = 1, S = 1}$



qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00



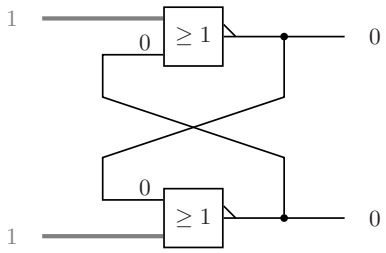
qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00



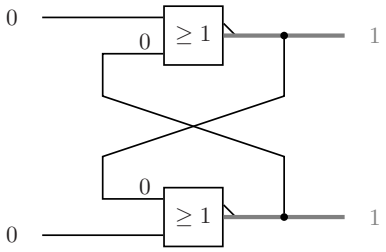
qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00

En partant de l'état stable $QP = 10$ et en faisant passer les entrées de $R = 0$, $S = 1$ à $R = 1$, $S = 1$, on passe par l'état instable $QP = 00$ (premier arc). Un état instable ne peut exister qu'un très court laps de temps, et q passe alors de 1 à 0, ce qui implique un déplacement vertical dans la matrice jusqu'à l'état stable $qp = QP = 00$ (deuxième arc). Il est à noter que, durant ce régime transitoire, les signaux de sortie restent inchangés.

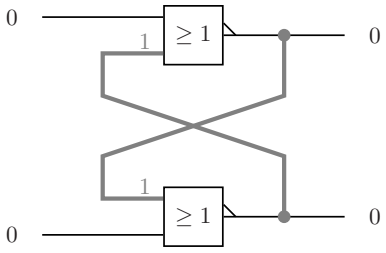
Exemple 5.3 (suite) $\boxed{R = 1, S = 1 \rightarrow R = 0, S = 0}$



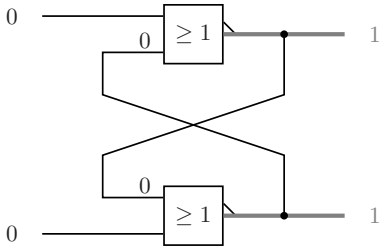
qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00



qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00



qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00

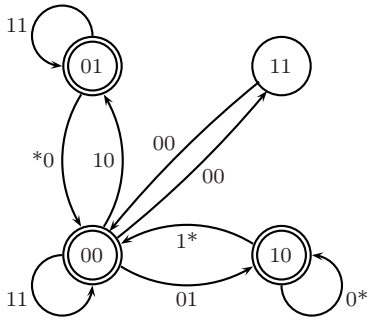


qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00

La bascule entre dans une phase d'oscillation entre les deux états instables $QP = 11$ *et* $QP = 00$.

L'ensemble des chemins dans la matrice d'états assignés permet la construction d'un *graphe de transitions* sous forme de diagramme d'états finis : à chaque état de la bascule correspond un état du graphe, les états stables pouvant être vus comme des états finaux. Le graphe de

transitions obtenu à partir de la matrice ci-dessus est le suivant :



En résumé :

- Pour la combinaison $R = 0$ et $S = 0$ aux entrées, il existe deux états stables. Le changement $R = 1, S = 0 \rightarrow R = 0, S = 0$ conduit à l'état stable $QP = 01$; le changement $R = 0, S = 1 \rightarrow R = 0, S = 0$ conduit à l'état stable $QP = 10$. Cette particularité exprime l'essence de la fonction de mémorisation élémentaire du circuit : si l'on passe d'une situation dans laquelle le signal actif est transmis à une seule entrée à une situation dans laquelle des signaux inactifs sont transmis aux deux entrées, le circuit mémorise l'état imposé par la dernière situation active.
- Pour la combinaison $R = 1$ et $S = 1$ aux entrées, le circuit se trouve dans l'état stable $QP = 00$, ce qui met en évidence que $P \neq \bar{Q}$.
- Pour le changement $R = 1, S = 1 \rightarrow R = 0, S = 0$, le circuit entre dans une phase d'oscillation permanente entre les états 00 et 11 sans jamais atteindre d'état stable. On peut remarquer à ce propos que dans la pratique, si on peut considérer qu'à une échelle donnée la commutation des deux variables est simultanée, il est en principe toujours possible de trouver une échelle assez fine pour que l'une varie avant l'autre. Toutefois, même en posant l'hypothèse que les commutations ne peuvent être réellement simultanées, on constate qu'il existe à partir de l'état 11 deux chemins possibles menant à deux états stables différents :

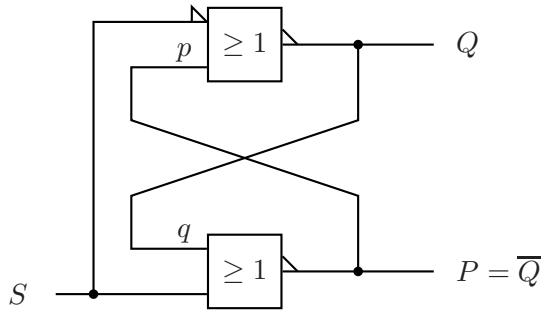
qp \ RS	00	01	11	10
00	11	10	00	01
01	01	00	00	01
11	00	00	00	00
10	10	10	00	00

- Si on suppose que la durée minimale d'un état est supérieure ou égale au temps de réponse du système, c'est-à-dire au temps nécessaire pour qu'il atteigne un état stable, l'état instable 00 ne peut plus être atteint.

En conclusion il apparaît nécessaire d'éviter toute combinaison des entrées qui mène à l'état 11. Plusieurs solutions sont envisageables, dont la plus immédiate est la bascule D, et une solution plus élaborée, la bascule JK...

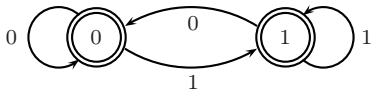
Bascule D asynchrone

La bascule D est une bascule RS dont une des entrées est connectée à l'autre via une inversion logique.



Il s'agit d'une bascule asynchrone qui fonctionne bien comme un élément de mémoire car :
- Les sorties sont complémentées.
- La bascule mémorise la dernière valeur de l'entrée.

Le diagramme de transitions de la bascule D est très simple :



La bascule fonctionne de façon *transparente* c'est-à-dire que la sortie Q recopie l'état de l'entrée. La bascule étant asynchrone, elle présente peu d'intérêt car elle n'opère que comme un retardateur de signal.

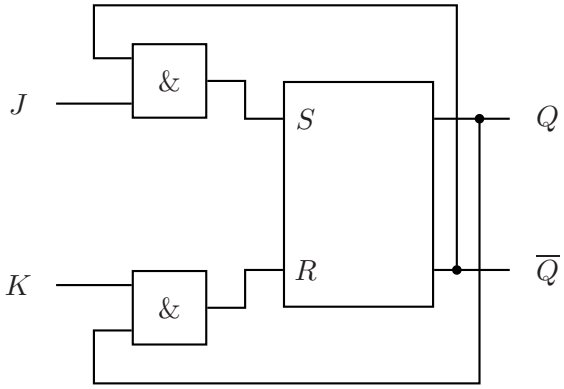
Bascule JK asynchrone

La bascule JK est une bascule dont on attend le comportement suivant :
- si $J = 0$ et $K = 0$ alors la sortie Q ne change pas d'état ;
- si $J = 0$ et $K = 1$ alors la sortie Q passe à 0 ;
- si $J = 1$ et $K = 0$ alors la sortie Q passe à 1 ;
- si $J = 1$ et $K = 1$ alors la sortie Q passe à $\overline{Q}$.

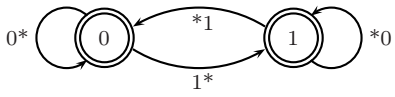
La *table d'excitation* suivante résume ce comportement, Q_t et Q_{t+1} représentant les valeurs de la sortie Q aux instant t et $t+1$ respectivement, en fonction des entrées J et K :

J	K	Q_{t+1}	Q_t
0	0	Q_t	$*$
0	1	0	$*$
1	0	1	$*$
1	1	$\overline{Q_t}$	$*$

Une bascule JK peut être réalisée à partir d'une bascule RS en rebouclant les sorties sur chacune des entrées :



Le diagramme d'états finis de la bascule est le suivant :

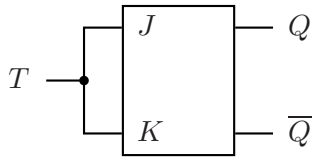


Bascule T asynchrone

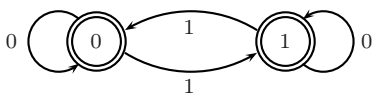
A partir de la table d'excitation d'une bascule JK asynchrone, on constate que l'état de la sortie est inversé ($Q_{t+1} = Q_t$) lorsque $J = K = 1$. Une bascule T (Trigger) est obtenue à partir d'une bascule JK, en injectant le même signal sur les entrées J et K. On obtient la table d'excitation suivante :

T	Q_{t+1}	Q_t
0	Q_t	$*$
1	$\overline{Q_t}$	$*$

correspondant au circuit suivant :



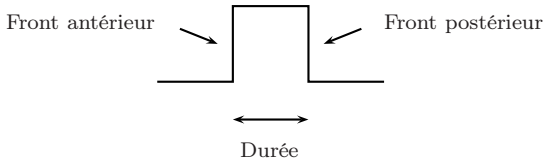
Le diagramme d'états finis de la bascule est le suivant :



Les bascules T sont utilisées en particulier dans la conception des compteurs.

5.2 Circuits synchrones

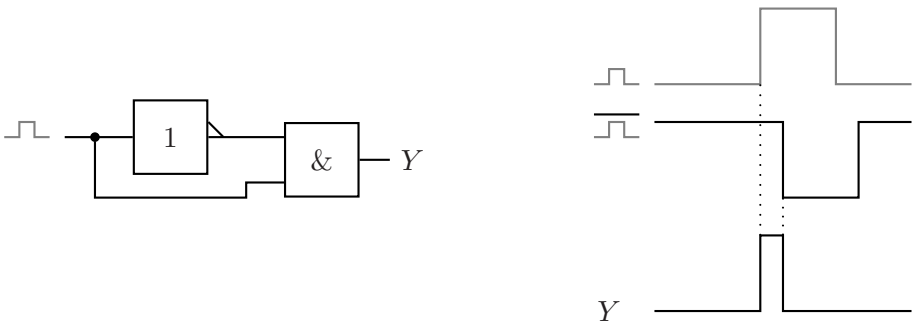
Une bascule synchrone est contrôlée par un signal d'horloge de type impulsion : le signal se trouve à la valeur logique 0, passe à la valeur logique 1, puis repasse à la valeur logique 0, de façon périodique et régulière. Le temps de commutation entre valeurs basse et haute étant négligeable, on peut sans aucun problème considérer que la transition est instantanée. L'impulsion d'horloge prend donc idéalement l'allure d'un signal carré périodique :



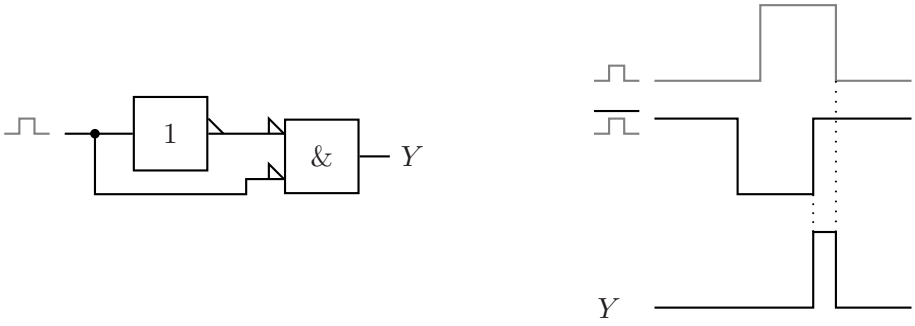
L'observation du signal d'horloge permet immédiatement d'envisager deux techniques de commutation des bascules synchrones.

- Les bascules *verrou* : ces bascules sont considérées comme actives pendant toute la durée du signal haut (en logique positive). Autrement dit, une bascule verrou bloque la dernière information qui se trouve aux entrées, avant que l'impulsion d'horloge effectue la transition de 1 à 0. Pour obtenir une seule transition par impulsion d'horloge, ce qui est fréquemment demandé en pratique, il est nécessaire que le ou les signaux d'entrée demeurent constants pendant toute la durée active de l'impulsion. Cette condition est, en général, assez difficile à remplir.
- Les bascules à *commutation sur front* : ces bascules sont actives au seul moment de la transition sur front (antérieur ou postérieur) du signal d'horloge. La commutation sur front pose le problème du choix du front actif ; en fait, la plupart des bascules commutent sur front postérieur, et sont généralement réalisées par la connexion de plusieurs bascules de type verrou et l'ajout de rétroactions supplémentaires (cf. les exemples donnés plus bas). Une autre technique consiste à utiliser un circuit détecteur de front [Tisserant03]. Ce type de circuit produit en coïncidence avec le front antérieur ou le front postérieur une impulsion de largeur juste suffisante pour permettre un basculement d'état. Les figures suivantes illustrent respectivement le principe d'un détecteur de front antérieur et d'un détecteur de front postérieur : l'idée est de prendre en compte le faible retard induit par la traversée d'un inverseur.

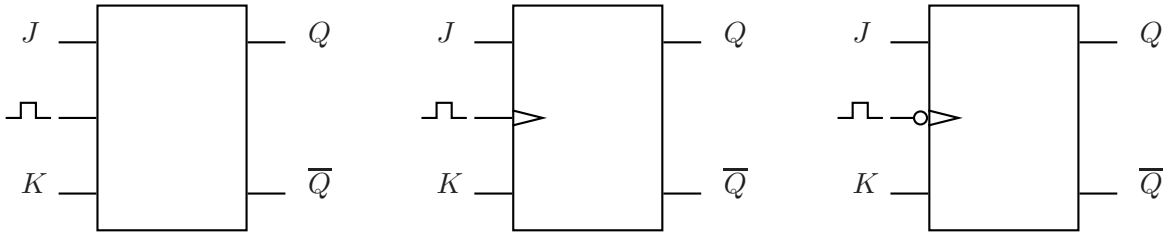
a) Détecteur de front antérieur



b) Détecteur de front postérieur



Graphiquement, on distingue une bascule verrou d'une bascule à commutation sur front par l'introduction d'un symbole supplémentaire au niveau de l'entrée d'horloge. Par exemple, pour une bascule JK :



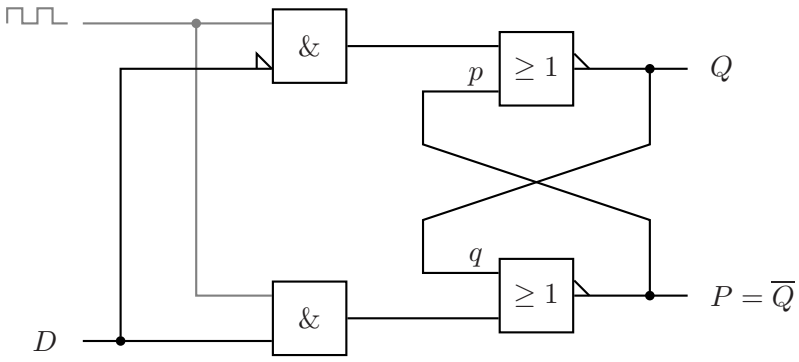
a) bascule verrou

b) bascule à commutation sur front antérieur

c) bascule à commutation sur front postérieur

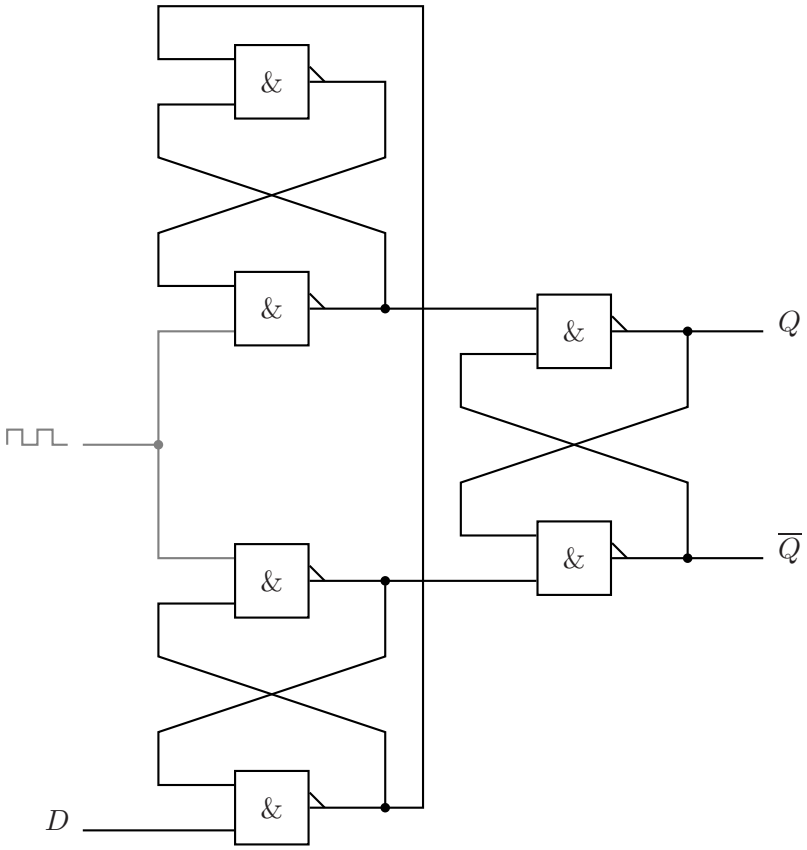
Les exemples qui suivent illustrent quelques-unes des plus courantes parmi les bascules synchrones :

Bascule D verrou



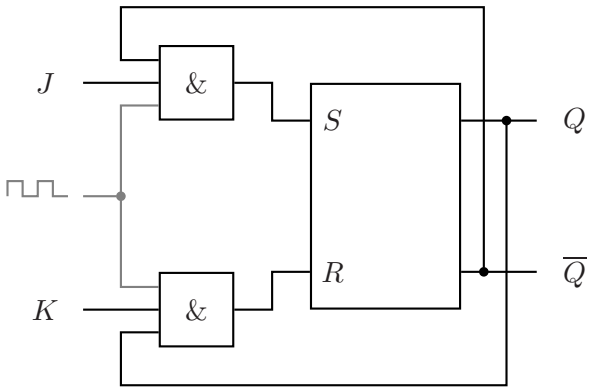
La propagation du signal d'entrée se fait sur la valeur haute du signal d'horloge (logique positive) qui fait alors office d'autorisation en écriture. La durée du cycle doit être supérieure ou égale à la durée de propagation dans la bascule : Cycle d'horloge $\geq \Delta t$.

Bascule D à commutation sur front antérieur

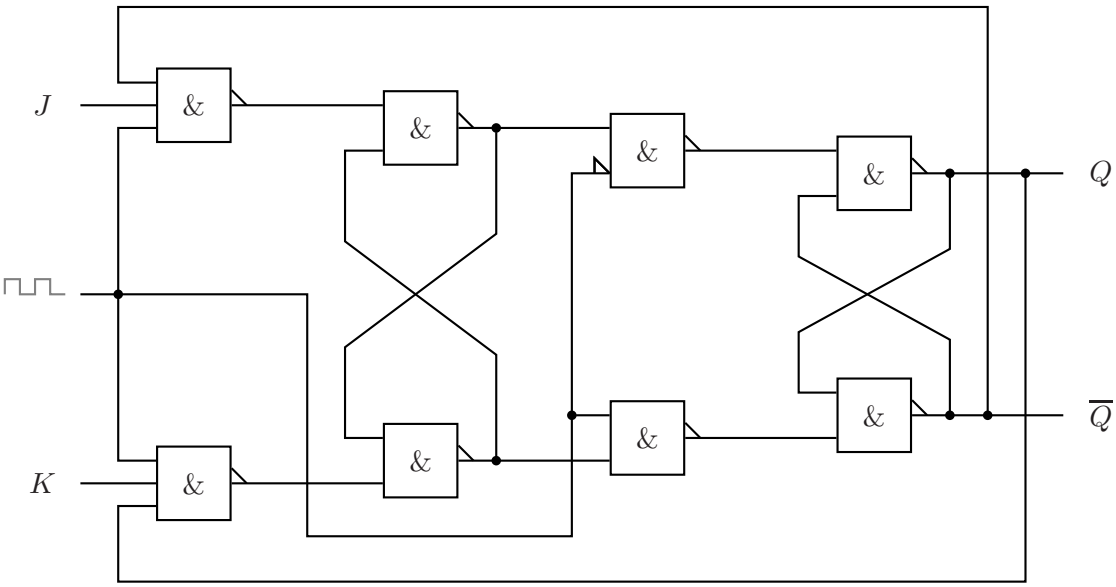


La bascule D à commutation sur front antérieur est ici constituée de trois bascules *RS NON-ET* sous forme d'un étage d'entrée et d'un étage de sortie. La rétroaction effectuée au niveau de l'étage d'entrée permet la commutation de la bascule sur le front antérieur.

Bascule JK verrou



Bascule JK à commutation sur front postérieur



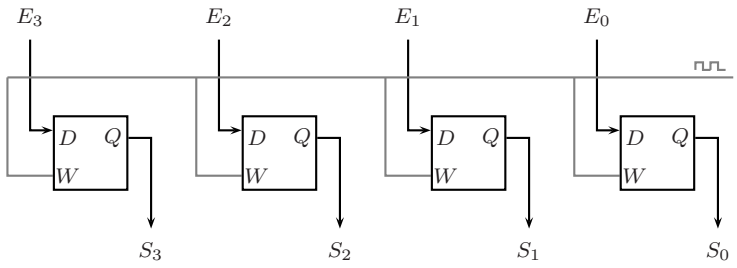
La bascule JK à commutation sur front postérieur est ici constituée de deux bascules *RS NON-ET* sous forme d'un étage d'entrée et d'un étage de sortie. La première bascule lit l'information qui se trouve aux entrées, immédiatement après le front antérieur de l'impulsion d'horloge : quand l'impulsion d'horloge se trouve au niveau bas (valeur logique 0), les sorties des premières portes *NON-ET* de l'étage d'entrée se trouvent à la valeur logique 1 ; après la transition de 0 à 1 de l'impulsion d'horloge ces portes sont débloquées et les signaux d'entrée J et K entrent sur la première bascule qui effectue la lecture de l'information. Simultanément, en raison de la présence d'un inverseur, l'étage de sortie est isolé de l'étage d'entrée. Quand l'impulsion d'horloge effectue la transition de 1 à 0, l'étage d'entrée se trouve à nouveau isolé des entrées d'information tandis que l'information sauvegardée dans la première bascule est transmis à la deuxième bascule. L'information stockée dans l'étage d'entrée est donc transférée vers l'étage de sortie par

l'intermédiaire du front postérieur de l'impulsion d'horloge. Du point de vue externe, le circuit constitue une seule bascule qui commute sur le front postérieur.

5.3 Registres, transfert d'information, bus de données

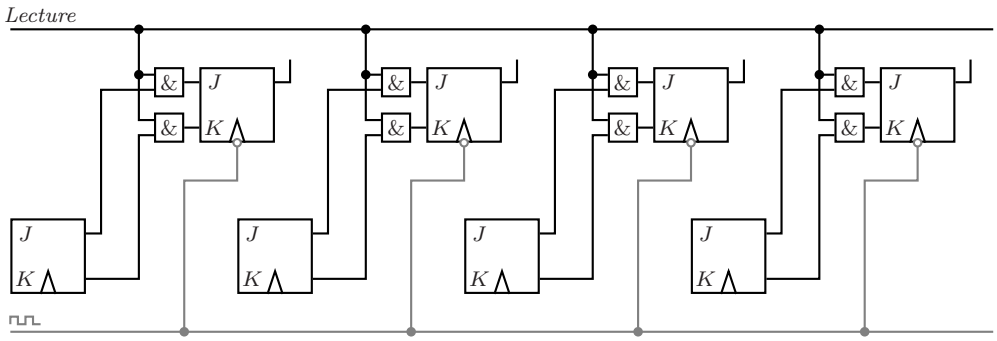
Le circuit séquentiel synchrone affecté au stockage temporaire d'une information de plusieurs bits est appelé *registre*. Un registre est constitué d'une collection de bascules de même type, actionnées par la même impulsion d'horloge. Le nombre de bascules constituant un registre détermine sa capacité.

Registre verrou D



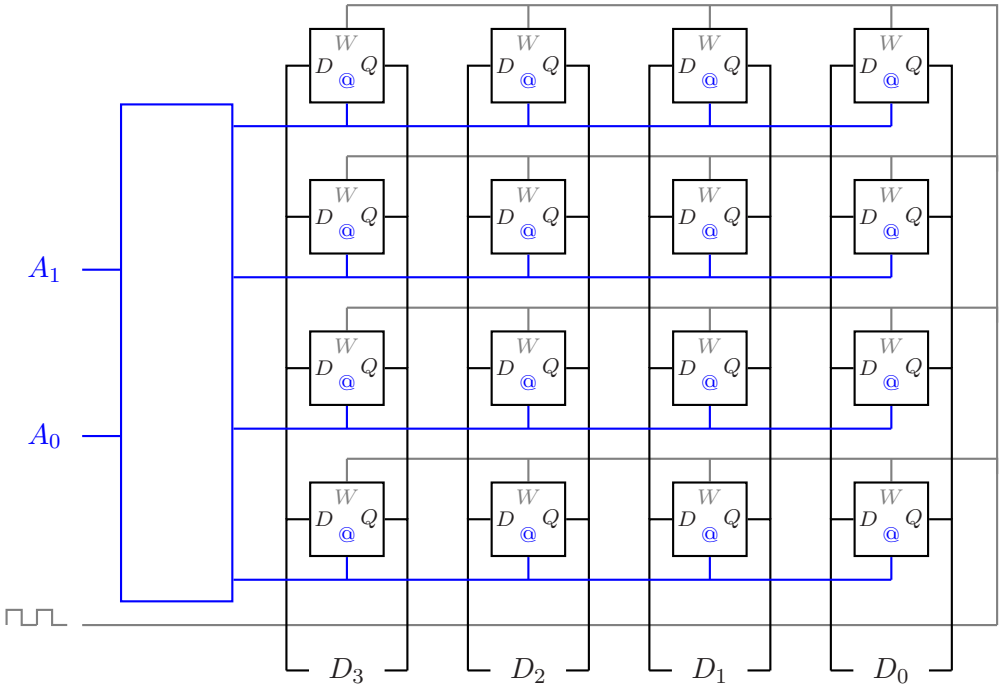
Un registre de type verrou est utilisé comme mémoire tampon, c'est-à-dire pour stocker l'information pendant un temps assez court, avant de l'acheminer vers une autre destination.

Transfert entre registres



Si la condition *Lecture* est vraie et que l'impulsion d'horloge arrive, l'information stockée par le registre source est copiée dans le registre destination. Cette information est saisie dans les sorties du registre destination, après le front postérieur de l'impulsion d'horloge (cf. [DanceaMarchand92]).

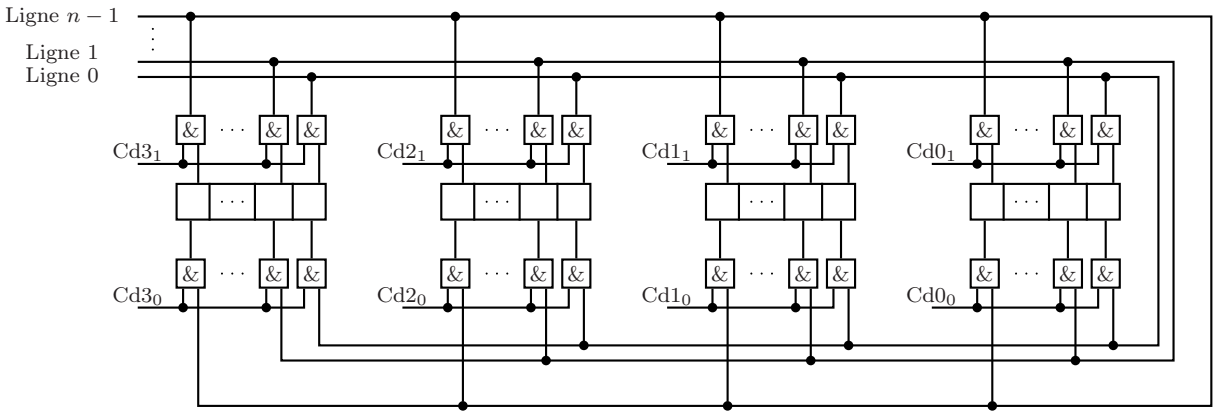
Banc de registres



Le décodeur $2-4$ permet de sélectionner un registre parmi les quatre composant le banc de registres. Le signal d'horloge joue le rôle d'autorisation en écriture.

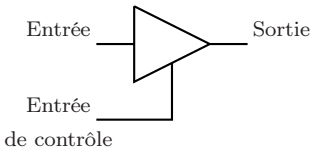
Bus de données

Un *bus* (abbréviation de liaison omnibus) est utilisé pour connecter entre eux plusieurs registres : il s'agit d'un ensemble de fils sur lesquels les informations binaires sont maintenues sous forme de tension par l'un des registres source. Un bus comporte au choix un, ou deux fils par bit, portant respectivement la sortie, ou les sorties complémentées de chaque élément mémoire auquel est relié le bus. L'utilisation d'un bus reliant quatre registres est présentée dans la figure ci-dessous (cf. [DanceaMarchand92]) :

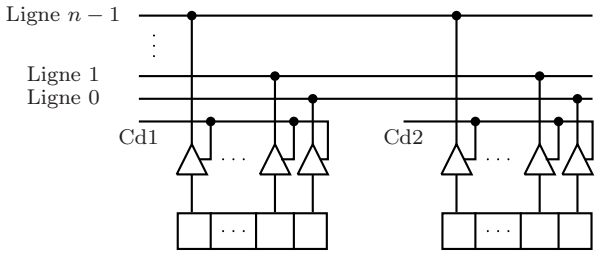


Chacune des sorties, tout comme chacune des entrées, de chaque bascule composant chaque registre est connectée à une ligne du bus à l'aide d'une porte *ET* contrôlée par un signal de commande. Par exemple, le transfert d'information du registre 2 vers le registre 0 nécessite que les signaux de commande $Cd2_1$ et $Cd0_0$ soient à 1 et que les autres signaux de commande soient à 0 : les sorties des bascules du registre 2 ainsi que les entrées des bascules du registre 0 se trouvent connectées simultanément au bus, ce qui permet le transfert parallèle de l'information au moment de l'arrivée de l'impulsion d'horloge, non représentée sur ce schéma.

Les portes *ET* qui controlent la connexion entre un registre et le bus sont généralement remplacées par des *barrières à trois états*, qui permettent au registre de poser un 0, un 1, ou de se déconnecter électriquement du bus. Une entrée de contrôle séparée est utilisée soit pour ouvrir la barrière de sortie, soit pour la mettre dans un état de haute impédance (électriquement déconnectée) :



La communication à travers un bus à l'aide de barrières à trois états placées par exemple en sorties de registres, est alors représentée de la façon suivante :



5.4 Compteurs et séquenceurs

Un *compteur* est un registre dont le contenu est incrémenté à chaque impulsion d'une horloge. Il s'agit donc d'un ensemble de bascules interconnectées avec un circuit combinatoire affecté au mode de changement d'état durant le processus de comptage, un compteur à n bascules pouvant passer successivement par 2^n états distincts. Outre leur fonction initiale de comptage, les compteurs permettent aussi de construire des *séquenceurs* dont le rôle est la distribution du signal d'horloge dans un circuit complexe. On classe les compteurs en fonction :

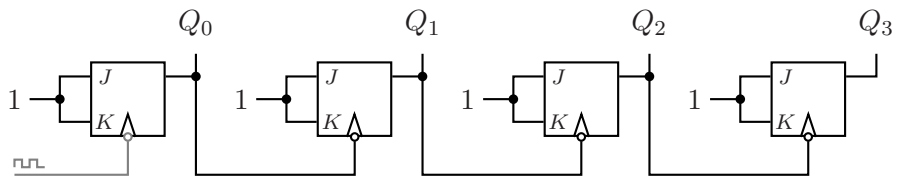
- du type de codage de l'information stockée (binaire, modulo n, binaire-décimal) ;
- du mode de commutation des bascules durant le processus de comptage (asynchrone ou synchrone) ;
- du mode d'avancement du compteur durant le processus de comptage (direct, inverse, réversible).

Le compteur le plus simple utilise directement le système de numération binaire, c'est-à-dire qu'il compte en binaire les impulsions d'horloge qui lui sont transmises à l'entrée de comptage, modulo n. Les deux exemples suivants illustrent le fonctionnement d'un tel compteur sur quatre bits,

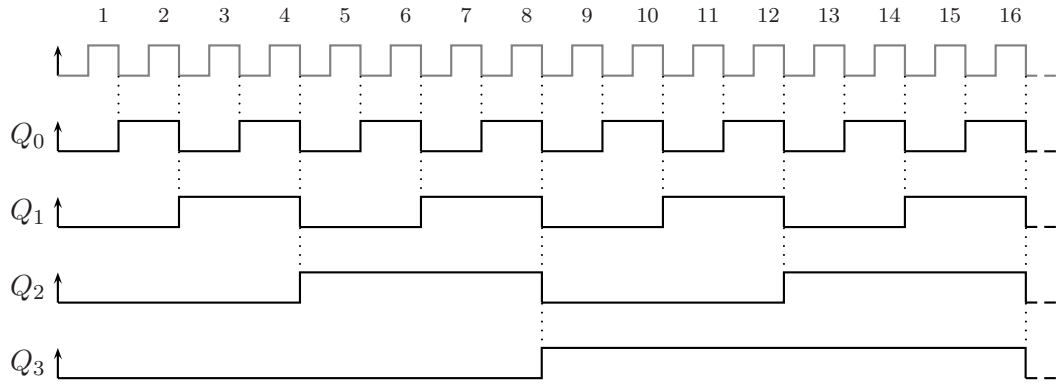
d'abord en mode asynchrone, puis en mode synchrone. Ces compteurs sont construits à partir de bascules JK à commutation sur front postérieur.

Compteur binaire modulo 16 asynchrone

On sait qu'une bascule JK à commutation sur front postérieur, utilisée en mode T ($J = K$), passe à l'état complémentaire à chaque front postérieur du signal d'horloge. En transmettant l'impulsion d'horloge à la bascule servant au codage du bit de poids le plus faible, et en reliant successivement chaque sortie d'une bascule codant un bit de poids inférieur à l'entrée de la bascule servant au codage du bit de poids immédiatement supérieur, l'impulsion se propage de proche en proche à l'ensemble des bascules, et on obtient le compteur recherché :



L'évolution des sorties $Q_0Q_1Q_2Q_3$ en fonction du signal d'horloge est représentée par le *chronogramme* [1] suivant :



On voit que chacune des sorties divise la fréquence du signal d'entrée correspondant par deux : c'est sur cette propriété du circuit que repose la capacité de comptage en binaire. Afin de mieux comprendre le comportement du circuit, considérons par exemple son évolution après comptage de la troisième impulsion d'horloge : $Q_0 = 1$, $Q_1 = 1$, $Q_2 = 0$, $Q_3 = 0$. Au front postérieur de la nouvelle impulsion d'horloge (la quatrième), la bascule Q_0 passe à l'état complémentaire, soit $Q_0 = 0$. Cette commutation produit donc un front postérieur à la sortie de la bascule, front qui actionne l'entrée de commutation de la bascule Q_1 : du coup, la bascule Q_1 passe à son tour à l'état complémentaire, soit $Q_1 = 0$. La commutation de la bascule Q_1 de l'état 1 à l'état 0 produit, elle aussi, un front postérieur qui oblige la bascule Q_2 à passer à l'état 1. Cette commutation ne produit pas de front postérieur, et par conséquent, la bascule Q_3 conserve son état initial.

[1] schéma donnant l'état des sorties d'un circuit en fonction du temps

L'obtention de la liste complète des états successifs des sorties à partir du chronogramme est immédiate :

Numéro d'impulsion	Q_3	Q_2	Q_1	Q_0
état initial	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
16	0	0	0	0

Bien que chaque bascule fonctionne individuellement de manière synchrone, le fonctionnement du compteur est, lui, asynchrone, puisque l'état de chacune des bascules ne dépend que des entrées correspondantes, sans référence globale à une impulsion d'horloge.

Compteur binaire modulo 16 synchrone

L'exemple proposé est tiré de [DanceaMarchand92]. A la différence du compteur asynchrone vu précédemment où l'horloge n'agit que sur une bascule, ici chaque impulsion d'horloge synchronise l'ensemble du circuit. L'idée est donc de construire une table d'excitation qui, outre les sorties du circuit, tienne compte de l'état de *l'ensemble des bascules* à chaque impulsion d'horloge. A partir de la table d'excitation d'une bascule JK individuelle, on introduit dans la table d'excitation du circuit les signaux qui doivent être nécessairement assignés aux entrées des bascules 0, 1, 2, et 3, pour que la succession des états soit respectée. Par exemple, après comptage de la troisième impulsion, les bascules doivent être respectivement dans l'état $Q_0 = 1$, $Q_1 = 1$, $Q_2 = 0$, $Q_3 = 0$. L'arrivée d'une nouvelle impulsion d'horloge (la quatrième) doit provoquer la transition vers un nouvel état dans lequel $Q_0 = 0$, $Q_1 = 0$, $Q_2 = 1$, $Q_3 = 0$. Au cours de cette transition, la bascule Q_3 ne change pas d'état, alors que les autres bascules doivent changer d'état. En consultant la table d'excitation d'une bascule JK, on observe que les transitions suivantes :

Q_3 : 0 à 0
Q_2 : 0 à 1
Q_2 : 1 à 0
Q_0 : 1 à 0

ont lieu lorsque les conditions suivantes sont imposées :

$$\begin{array}{ll} J_3 = 0, & K_3 = * \\ J_2 = 0, & K_2 = * \\ J_1 = *, & K_1 = 1 \\ J_0 = *, & K_0 = 1 \end{array}$$

En considérant de la même manière et de façon exhaustive toutes les transitions imposées par le processus de comptage, on obtient la table d'excitation suivante :

Numéro d'impulsion	Q_3	Q_2	Q_1	Q_0	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
état initial	0	0	0	0	0	*	0	*	0	*	1	*
1	0	0	0	1	0	*	0	*	1	*	*	1
2	0	0	1	0	0	*	0	*	*	0	1	*
3	0	0	1	1	0	*	0	*	*	1	*	1
4	0	1	0	0	0	*	*	0	0	*	1	*
5	0	1	0	1	0	*	*	0	1	*	*	1
6	0	1	1	0	0	*	*	0	*	0	1	*
7	0	1	1	1	1	*	*	1	*	1	*	1
8	1	0	0	0	*	0	0	*	0	*	1	*
9	1	0	0	1	*	0	0	*	1	*	*	1
10	1	0	1	0	*	0	0	*	*	0	1	*
11	1	0	1	1	*	0	1	*	*	1	*	1
12	1	1	0	0	*	0	*	0	0	*	1	*
13	1	1	0	1	*	0	*	0	1	*	*	1
14	1	1	1	0	*	0	*	0	*	0	1	*
15	1	1	1	1	*	1	*	1	*	1	*	1
16	0	0	0	0	0	*	0	*	0	*	1	*

Il est alors possible d'établir les huit diagrammes de Karnaugh qui correspondent aux entrées :

$\boxed{J_3}$

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00				
01	*	*	*	*
11	*	*	*	*
10			1	

$\boxed{K_3}$

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00	*	*	*	*
01				
11			1	
10	*	*	*	*

$\boxed{J_2}$

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00			1	
01			1	
11	*	*	*	*
10	*	*	*	*

$\boxed{K_2}$

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00	*	*	*	*
01	*	*	*	*
11			1	
10			1	

J_1

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00		*	*	1
01		*	*	1
11		*	*	1
10		*	*	1

K_1

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00	*		1	*
01	*		1	*
11	*		1	*
10	*		1	*

J_0

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00	*	*	1	1
01	*	*	1	1
11	*	*	1	1
10	*	*	1	1

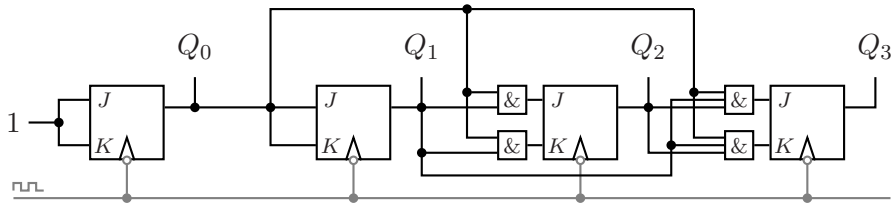
K_0

Q_2Q_3 \ Q_0Q_1	00	01	11	10
00	1	1	*	*
01	1	1	*	*
11	1	1	*	*
10	1	1	*	*

Après simplification, on obtient les équations suivantes :

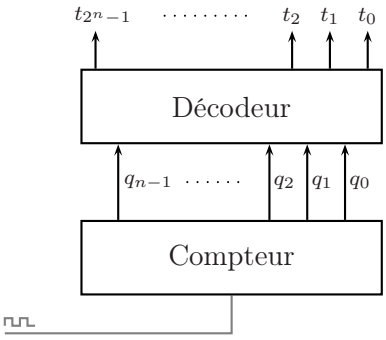
$$\begin{array}{ll} J_3 = Q_0.Q_1.Q_2, & K_3 = Q_0.Q_1.Q_2 \\ J_2 = Q_0.Q_1, & K_2 = Q_0.Q_1 \\ J_1 = Q_0, & K_1 = Q_0 \\ J_0 = 1, & K_0 = 1 \end{array}$$

D'où le circuit :



Générateur de séquences

On appelle *générateur de séquences* l'association d'un circuit combinatoire et d'un circuit séquentiel. Le dispositif résultant permet de créer des signaux de commande décalés dans le temps, et constitue ainsi le coeur de l'unité de contrôle d'un ordinateur : les mots binaires générés sont en effet à la base du fonctionnement des architectures cablées et microprogrammées. Le séquenceur ayant la structure la plus simple résulte de l'association d'un compteur et d'un décodeur :



Un compteur comportant n bascules pourra passer par 2^n états distincts successifs, le décodeur relié aux sorties du compteurs comportant alors n entrées et 2^n sorties permettant de distribuer successivement 2^n signaux de commandes $t_0, t_1, \ldots t_{2^n-1}$ dans diverses parties d'une architecture.

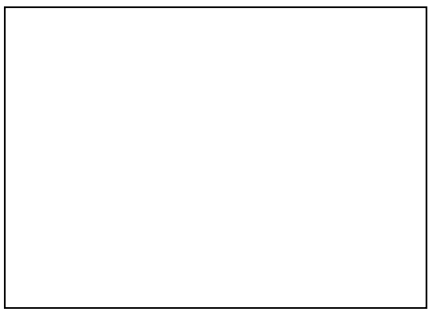
Chapitre 6

Architecture du processeur

6.1 Introduction

Un ordinateur est un dipositif servant à l'exécution d'un programme, programme qui se présente sous la forme d'un ensemble d'instructions rangées dans la mémoire principale : chaque instruction est successivement lue à partir de la mémoire, c'est-à-dire transférée à l'UC (pour *Unité Centrale*) pour y être décodée puis exécutée. Ce processus se fait par l'intermédiaire de cycles rythmés par une horloge. On distingue :

- un *cycle de recherche*, au cours duquel une instruction est extraites de la mémoire pour être copiée vers un registre du processeur, appelé *registre instruction*, et noté IR ;
- un *cycle d'exécution*, au cours duquel l'instruction en cours est interprétée par le dispositif de commande et des signaux sont générés et envoyés au sein de l'UC pour permettre le traitement de cette instruction.



Si le cycle de recherche nécessite un nombre constant d'impulsions d'horloge, le cycle d'exécution peut requérir une ou plusieurs impulsions d'horloge suivant la complexité de l'instruction. Les opérations élémentaires effectuées durant une seule impulsion d'horloge sont appelées *micro-opérations*, et décrites à l'aide d'un formalisme simple, le *langage transfert*.

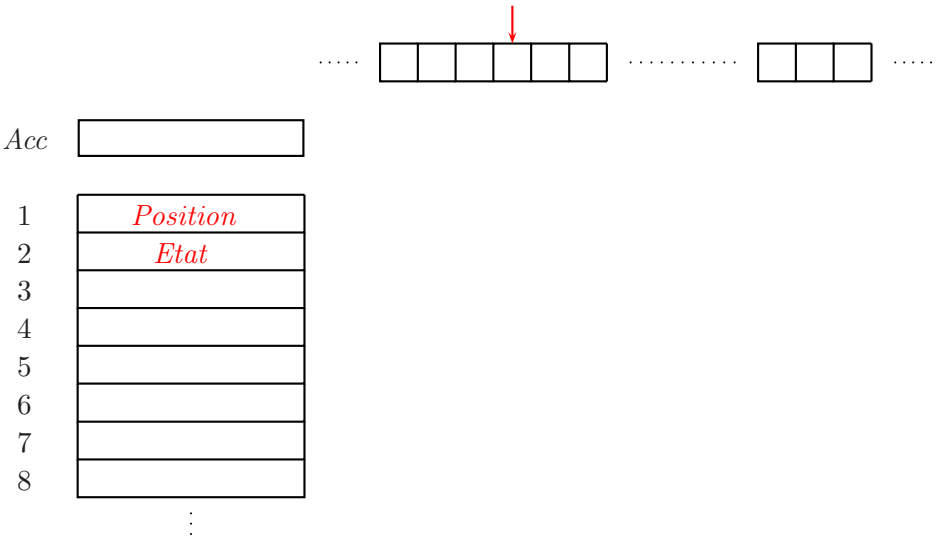
6.2 Jeux d'instructions et modèle d'exécution

6.3 Equivalence machine à accumulateur – machine de Turing

Il est relativement facile de montrer que les différents modèles d'exécution considérés au paragraphe précédent peuvent être simulés sur une machine à accumulateur. Nous montrons ici qu'une machine à accumulateur possédant une mémoire virtuellement infinie est par ailleurs équivalente à une machine de Turing, en montrant d'abord qu'une telle machine à accumulateur peut simuler toute machine de Turing, et qu'inversement une machine de Turing est en mesure de simuler toute machine à accumulateur.

Simulation d'une machine de Turing par une machine à accumulateur

Etant données une machine de Turing et une machine à accumulateur, la mémoire de la première est utilisée pour simuler le ruban de la seconde, comme dans la figure suivante [Dewdney93] :



Partant de la bijection ainsi obtenue entre mot-mémoires de la machine à accumulateur et cellules du ruban de la machine de Turing, il ne reste plus qu'à écrire pour la machine à accumulateur un programme qui simule les opérations de la machine de Turing : à chacune de ces opérations correspond un ensemble d'instructions de la machine à accumulateur qui doivent produire sur la mémoire de cette machine l'effet que produisent les opérations en question sur le ruban de la machine de Turing. A cet effet, il est nécessaire que le programme de la machine à accumulateur soit en mesure de se souvenir à la fois de la position de la tête de lecture-écriture, et de l'état courant de la machine de Turing, comme cela est représenté dans la figure.

Simulation d'une machine à accumulateur par une machine de Turing

6.4 Modes d'adressage, alignement

6.5 Séquencement des micro-opérations

Les instructions composant le programme que l'ordinateur doit exécuter sont chargées dans des emplacements consécutifs de la mémoire principale. Au cours du *cycle de recherche*, l'UC va chercher les instructions une à une et exécute la fonction demandée. L'UC mémorise l'adresse mémoire de l'instruction suivante à exécuter par l'intermédiaire d'un registre spécialisé, appelé PC (pour *compteur de programme*). Après avoir obtenu une instruction, le contenu du PC est mis à jour pour désigner l'instruction suivante de la séquence d'instruction à exécuter. Si l'on suppose, pour simplifier, que chaque instruction occupe un mot mémoire, alors l'exécution d'une instruction se décompose en trois étapes, correspondant à l'exécution d'un ensemble de micro-opérations :

1. Obtention du contenu de la case mémoire désignée par PC : IR $\leftarrow$ [[PC]]
2. Incrémentation du *compteur de programme* : PC $\leftarrow$ [PC] + 1
3. Effectuer les actions spécifiées par l'instruction rangée dans IR.

Les deux premières étapes correspondent au *cycle de recherche*, la troisième étape au *cycle d'exécution*. A ce stade, il est important de rappeler que les différents blocs composant l'UC peuvent

être organisés et interconnectés de façons différentes, en fonction du nombre de registres à disposition au sein de l'UC, et selon que l'on considère une architecture à un bus unique, ou plusieurs bus. Cependant, à quelques exceptions près, la plupart des opérations composant le *cycle de recherche* et le *cycle d'exécution* peuvent réalisées en exécutant une ou plusieurs des fonctions suivantes dans un ordre prédéfini :

- Obtention d'une case mémoire et chargement dans un registre de l'UC.
- Rangement d'un mot de données contenu dans un registre de l'UC dans une case mémoire déterminée.
- Transfert d'un mot de données d'un registre à un autre ou vers l'Unité Arithmétique et Logique.
- Exécution d'une opération arithmétique ou logique, et rangement du résultat dans un registre de l'UC.

6.6 Contrôle cablé

6.7 Contrôle microprogrammé

Chapitre 7

Evaluation des performances

La *mesure* et la possibilité de *comparaison* des performances obtenues par une machine déterminent les choix de conception d'une architecture. Si toute mesure de performance repose *in fine* sur la mesure du temps que passe l'unité centrale à exécuter un ensemble de tâches choisies, différentes unités de mesures sont envisageables en fonction des caractéristiques que l'on cherche à évaluer. D'autre part, la comparaison des performances obtenues à partir d'un dispositif d'amélioration permet de mesurer l'accélération effectivement obtenue par l'implémentation de ce dispositif, et représente donc un outil théorique absolument nécessaire. Les concepts présentés dans ce chapitre sont développés notamment par John Hennessy et David Patterson dans [HennessyPatterson96].

7.1 Equation de performance de l'Unité Centrale

L'équation de performance de l'UC dépend de la période d'horloge, exprimée sous forme :
- de durée (par exemple : 2ns) ;
- de fréquence (par exemple 700MHz).

Le temps UC d'un programme s'exprime donc de deux manières :

(1) $\textit{Temps d'exécution UC} = \textit{Nb de cycles UC} \times \textit{Temps cycle d'horloge}$

(2) $\textit{Temps d'exécution UC} = \frac{\textit{Nb de cycles UC}}{\textit{Fréquence d'horloge}}$

Or, en plus du nombre de cycles utilisés pour l'exécution d'un programme, on peut compter le nombre d'instructions exécutées, symbolisé par la constante *NI*. D'autre part, à partir du nombre de cycles et du nombres d'instructions exécutées, on peut calculer le nombre moyen de cycles par instruction, représenté par la variable *CPI*. On a :

(3) $\textit{CPI} = \frac{\textit{Nb de cycles UC}}{\textit{NI}}$

d'où : $\textit{Nb de cycles UC} = \textit{NI} \times \textit{CPI}$, et en remplaçant dans l'équation (1) :

(1') $\textit{Temps d'exécution UC} = \textit{NI} \times \textit{CPI} \times \textit{Temps cycle d'horloge}$

(2') $\textit{Temps d'exécution UC} = \frac{\textit{NI} \times \textit{CPI}}{\textit{Fréquence d'horloge}}$

Remarque :

- L'équation de performance de l'UC montre quels sont les facteurs sur lesquels un concepteur de circuit peut agir pour obtenir un gain de performance ; Par exemple, une amélioration de 10% dans la fréquence d'horloge, le *CPI* ou le *NI* conduit à une amélioration de 10% du temps UC.
- Malheureusement, les technologies de base impliquées dans une modification sont interdépendantes ; on peut relever le constat suivant :
  - la *Fréquence d'horloge* dépend de la technologie matérielle ;
  - le *CPI* dépend de l'architecture et de la structure du jeu d'instructions ;
  - le *NI* dépend de la structure du jeu d'instructions et du compilateur.

  Heureusement, beaucoup des techniques accessibles pour accroître les performances améliorent essentiellement une composante avec un impact petit ou prévisible sur les deux autres.

Il est à noter que l'équation de performance de l'UC peut être encore raffinée de la manière suivante :

On considère

$$\begin{aligned} NI_i &= \textit{le nombre de fois où l'instruction } i \textit{ est exécutée dans un programme} \\ CPI_i &= \textit{le nombre moyen de cycles nécessaires à l'exécution de l'instruction } i \end{aligned}$$

On a alors :

(4) $\textit{Nb de cycles} = \sum_{i=1}^{n}(CPI_i \times NI_i)$

d'où, en remplaçant dans les équations (1) et (2) :

(1") $\textit{Temps d'exécution UC} = \sum_{i=1}^{n}(CPI_i \times NI_i) \times \textit{Temps cycle d'horloge}$

(2") $\textit{Temps d'exécution UC} = \frac{\sum_{i=1}^{n}(CPI_i \times NI_i)}{\textit{Fréquence d'horloge}}$

et dans l'équation (3) :

(3')

$$\begin{aligned} CPI &= \frac{\sum_{i=1}^{n}(CPI_i \times NI_i)}{NI} \\ &= \sum_{i=1}^{n}(CPI_i \times \frac{NI_i}{NI}) \end{aligned}$$

7.2 Unités de mesure de la performance : MIPS et MFLOPS

On désigne par *MIPS*, le million d'instructions exécutées par seconde. On a :

$$\begin{aligned} \textit{Nb de MIPS} &= \frac{NI}{\textit{Temps d'exécution} \times 10^6} \\ &= \frac{\textit{Fréquence d'horloge}}{CPI \times 10^6} \end{aligned}$$

A priori les machines rapides ont un débit MIPS plus élevé. Toutefois :

– Le nombre de MIPS dépend du jeu d'instructions ce qui rend difficile la comparaison de machines ayant des jeux d'instructions différents.
– Il dépend des programmes sur un même ordinateur.
– Il peut varier en sens inverse de la performance si on ne prend pas garde au contexte de la machine sur laquelle sont faites les mesures (cas typique : une machine avec du matériel flottant optionnel).

Dans le cas de programmes pour lesquels le traitement des flottants revêt une importance particulière, il est en général plus pertinent d'utiliser comme unité de mesure le MFLOPS, qui désigne le million d'opérations flottantes par seconde. On a

$$Nb\ de\ MFLOPS = \frac{Nb\ d'opérations\ flottantes\ réalisées}{Temps\ d'exécution\ UC \times 10^6}$$

Le MFLOPS dépend de la machine et du programme (bien qu'on considère ici non pas des instructions mais des opérations).

7.3 Mesure de l'accélération : la loi d'Amdahl

L'accélération indique quel est le gain en vitesse pour une tâche utilisant une machine munie d'un dispositif d'amélioration par rapport à la même machine sans ce dispositif. Appelons *Temps d'exécution ancien*, le temps d'exécution de la tâche sans utiliser le dispositif d'amélioration. De façon similaire, appelons *Temps d'exécution nouveau*, le temps d'exécution de la tâche en appliquant le dipositif d'amélioration. On a :

$$Accélération = \frac{Temps\ d'exécution\ ancien}{Temps\ d'exécution\ nouveau}$$

En fait, et de manière plus précise, l'accélération dépend de :

1. La *fraction* du temps total de calcul pendant laquelle l'amélioration peut être utilisée (appelée *fraction améliorée*, ≤ 1).
2. Le *gain* apporté par le mode d'exécution amélioré, c'est à dire le gain en vitesse si le programme entier s'exécutait en mode amélioré (appelé *accélération améliorée*, ≥ 1).

On a :

$$\begin{aligned} Fraction\ améliorée &= \frac{fraction\ de\ temps\ d'utilisation\ du\ dispositif}{Temps\ d'exécution\ nouveau} \\ Accélération\ améliorée &= \frac{Temps\ d'exécution\ ancien}{temps\ d'exécution\ total\ avec\ dispositif} \end{aligned}$$

En détaillant, on voit que :

$$\begin{aligned} &Temps\ d'exécution\ nouveau = \\ &\quad fraction\ de\ temps\ d'utilisation\ du\ dispositif + \\ &\quad fraction\ du\ temps\ restant\ sans\ dispositif \end{aligned} \tag{7.1}$$

Le temps d'utilisation avec dispositif si le dispositif était applicable tout au long de la tâche est donné par *Temps d'exécution ancien*/*Accélération améliorée* ; le temps réel d'utilisation sur la fraction améliorée est alors donné par

$$(Temps\ d'exécution\ ancien/Accélération\ améliorée) \times Fraction\ améliorée$$

autrement dit :

$$\textit{fraction de temps d'utilisation du dispositif} = \textit{Temps d'exécution ancien} \times \frac{\textit{Fraction améliorée}}{\textit{Accélération améliorée}}$$

D'autre part, la fraction de temps pendant laquelle le dispositif n'est pas appliqué s'exprime en retranchant du temps d'origine le temps pendant lequel le dispositif s'applique, donné par *Temps d'exécution ancien* × *Fraction améliorée*, d'où :

$$\begin{aligned} &\textit{fraction du temps restant sans dispositif} \\ &= \textit{Temps d'exécution ancien} - \\ &\quad \textit{Temps d'exécution ancien} \times \textit{Fraction améliorée} \\ &= \textit{Temps d'exécution ancien} \times (1 - \textit{Fraction améliorée}) \end{aligned}$$

En remplaçant alors dans l'expression du *Temps d'exécution nouveau* donnée par l'équation 7.1 :

$$\begin{aligned} &\textit{Temps d'exécution nouveau} \\ &= \textit{Temps d'exécution ancien} \times (1 - \textit{Fraction améliorée}) + \\ &\quad \textit{Temps d'exécution ancien} \times \frac{\textit{Fraction améliorée}}{\textit{Accélération améliorée}} \\ &= \textit{Temps d'exécution ancien} \times \\ &\quad ((1 - \textit{Fraction améliorée}) + \frac{\textit{Fraction améliorée}}{\textit{Accélération améliorée}}) \end{aligned}$$

On obtient alors (Loi d'Amdahl)

$$\textit{Accélération} = \frac{1}{1 - \textit{Fraction améliorée} + \frac{\textit{Fraction améliorée}}{\textit{Accélération améliorée}}}$$

Exemple 7.1 *Un dispositif d'amélioration est* 10 *fois plus rapide que la machine de base, mais on ne peut l'appliquer que* 40% *du temps. Quelle est l'accélération obtenue en intégrant ce dispositif ? On a :*

$$\begin{aligned} \textit{Fraction améliorée} &= 0,4 \\ \textit{Accélération améliorée} &= 10 \end{aligned}$$

L'accélération globale obtenue est donnée par la loi d'Amdhal :

$$\begin{aligned} \textit{Accélération} &= \frac{1}{1 - 0,4 + \frac{0,4}{0,6}} \\ &= \frac{1}{0,64} \\ &= 1,56 \end{aligned}$$

Annexe A

Le code ASCII

Le codage d'un caractère donné dans la table ci–après est obtenu par juxtaposition de la combinaison des bits de la colonne à celle de la ligne où est situé le caractère concerné. Par exemple, le code du caractère D est 1000100, celui du caractère %, 0100101. Les chiffres décimaux sont exprimés par le groupe de trois bits 011, suivi de leur équivalent en DCB. Enfin le code ASCII contient une série de caractères de commande, comme par exemple le retour chariot (CR), le retour arrière (BS), ou encore la tabulation horizontale (HT), indispensables dans tout processus d'édition de texte ou de communication.

On utilise conventionnellement l'ensemble d'acronymes suivants pour les différents caractères de commande disponibles :

NUL	nul	DC1	contrôle dispositif 1
SOH	début d'en tête	DC2	contrôle dispositif 2
STX	début de texte	DC3	contrôle dispositif 3
ETX	fin de texte	DC4	contrôle dispositif 4
EOT	fin de bande	NAK	accusé de reception négatif
ENQ	demande	SYN	synchronisation
ACK	accusé de reception	ETB	fin de bloc de transmission
BEL	sonnerie	CAN	annulation
BS	retour arrière	EM	fin de support
HT	tabulation horizontale	SUB	substitution
LF	saut de ligne	ESC	échappement
VT	tabulation verticale	FS	séparateur de fichier
FF	saut de page	GS	séparateur de groupe
CR	retour chariot	RS	séparateur d'article
SO	hors–code	US	séparateur de sous–article
SI	en–code	SP	espacement
DLE	échappement transmission	DEL	effacement

Le code ASCII est alors :

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Annexe B

Bases de numération sur les entiers

B.1 Division euclidienne dans $\mathbb{N}$

Lemme B.1

$$(\forall a \in \mathbb{N})(\forall b \in \mathbb{N}^*)(\exists k \in \mathbb{N})(a < k.b)$$

Preuve : De $1 \leq b$, on déduit $a \leq a.b$ d'où $a + b \leq a.b + b$. De plus, $0 < b$ d'où $a < a + b$. On a donc

$$\left.\begin{array}{c} a < a + b \\ a + b \leq a.b + b \end{array}\right\} \text{ d'où } a < a.b + b$$

soit $a < (a + 1).b$. Par conséquent il existe au moins un entier k tel que $a < k.b$. *Q.E.D.*

Théorème B.1 *Quels que soient $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$, il existe un couple unique $(q, r) \in \mathbb{N} \times \mathbb{N}$ tels que $a = b.q + r$ et $0 \leq r < b$.*

Preuve : Soit E, l'ensemble des entiers k tels que $a < k.b$. D'après le lemme précédent $E \neq \emptyset$; comme sous-ensemble de $\mathbb{N}$ il admet donc un plus petit élément. Cet élément n'est pas nul, on le note $q + 1$ d'où $a < (q + 1).b$. L'élément $q + 1$ étant le plus petit élément de E, q n'est pas dans E, d'où $a \geq q.b$. Donc il existe un entier q unique tel que

$$b.q \leq a \leq b.(q + 1)$$

Si on pose $a - b.q = r$ (q étant unique, r l'est aussi), on a

$$\left\{\begin{array}{c} a - b.q = r \\ b.q \leq a \leq b.(q + 1) \end{array}\right.$$

qui équivaut à

$$\left\{\begin{array}{c} a = b.q + r \\ 0 \leq r \leq b \end{array}\right.$$

On a effectué la *division euclidienne* de a par b, q est le *quotient*, r le *reste*. *Q.E.D.*

B.2 Développement polynômial d'un entier naturel

La représentation d'un entier naturel dans un système de numération donné se fonde sur le choix de b symboles ($b > 1$), appelés *chiffres* du système de numération. Le système en question est dit de *base* b, et la représentation d'un entier dans ce système repose sur le résultat suivant :

Théorème B.2 *Soit $b \in \mathbb{N}, b > 1$. Pour tout $a \in \mathbb{N}$, supérieur ou égal à b, il existe une séquence unique de n entiers naturels $q_0 \dots q_{n-1}$ tels que :*

$$a = q_{n-1}.b^{n-1} + \ldots + q_2.b^2 + q_1.b + q_0$$

avec $0 < q_{n-1} < b$ et $\forall i \in \{0, 1, \ldots, (n-2)\}, 0 \leq q_i < b$.

Preuve : Soit $a \in \mathbb{N}, a \geq b$ ($b > 1$). En effectuant la division euclidienne de a par b, il existe un couple unique $(q_1, r_0) \in \mathbb{N}^* \times \mathbb{N}$ tel que $a = q_1.b + r_0$

$$\text{avec } \begin{cases} 0 \leq r_0 \leq b \\ 0 < q_1 < a \end{cases}$$

1. si $q_1 < b$ le théorème est démontré
2. si $q_1 \geq b$, il existe un couple unique $(q_2, r_1) \in \mathbb{N}^* \times \mathbb{N}$ tel que $q_1 = q_2.b + 1$

$$\text{avec } \begin{cases} 0 \leq r_0 \leq b \\ 0 < q_1 < a \end{cases}$$

   (a) si $q_2 < b$ le théorème est démontré

   (b) si $q_2 \geq b$ on effectue la division euclidienne de q_2 par b et on procède ainsi de suite jusqu'à ce que le quotient $q_{n-1} < b$ (ceci se produira nécessairement car la suite des quotients $q_1, q_2, \ldots, q_{n-1}$ est décroissante).

On obtient successivement

$$\begin{array}{lllllll}
(0) & a & = & q_1.b + r_0 & \text{avec} & 0 \leq r_0 \leq b, 0 < q_1 < a \\
(1) & q_1 & = & q_2.b + r_1 & \text{avec} & 0 \leq r_1 \leq b, 0 < q_2 < q_1 \\
(2) & q_2 & = & q_3.b + r_2 & \text{avec} & 0 \leq r_2 \leq b, 0 < q_3 < q_2 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
(n-1) & q_{n-2} & = & q_{n-1}.b + r_{n-2} & \text{avec} & 0 \leq r_{n-2} \leq b, 0 < q_{n-1} < q_{n-2}
\end{array}$$

En multipliant les égalités $(0), (1), (2), \ldots, (n-1)$ respectivement par 1, b, b^2, …, b^{n-2}, et en les additionnant membre à membre, on obtient

$$a = q_{n-1}.b^{n-1} + r_{n-2}.b^{n-2} + \ldots + r_2.b^2 + r_1.b + r_0$$

avec $0 < q_{n-1} < b$ et $\forall i \in \{0, 1, \ldots, (n-2)\}, 0 \leq r_i < b$. La décomposition est unique car les couples q_i, r_{i-1} successifs sont uniques. *Q.E.D.*

Soit b la base du système de numération choisi, et soit a un entier naturel dans le système de base b. La *représentation de a en base b* s'effectue conventionnellement de la façon suivante :

– Si $a < b$, l'entier naturel a est représenté par l'un des b symboles définissant le système de numération de base b.

– Si $a \geq b$, a s'écrit de manière unique sous la forme

$$a = q_{n-1}.b^{n-1} + \ldots + q_2.b^2 + q_1.b + q_0$$

avec $0 < q_{n-1} < b$ et $\forall i \in \{0, 1, \ldots, (n-2)\}, 0 \leq q_i < b$. Par convention, l'entier naturel a, dans le système de numération de base b est alors écrit :

$$a = q_{n-1} \ldots q_2 q_1 q_0$$

Exemple B.1 *Dans le système de numération de base* 5*, les symboles utilisés sont* $0, 1, 2, 3, 4$. *L'entier naturel a noté* 4302 *correspond au développement*

$$a = 4.5^3 + 3.5^2 + 0.5^1 + 2.5^0$$

Exemple B.2 *Soit le nombre* 9456*, écrit en base* 10 *(*$b = 10$*) et* $q_i \in \{0, \ldots, 9\}$

$$9456 = 9.10^3 + 4.10^2 + 5.10^1 + 6.10^0$$

Notation

Par convention on indice la représentation d'un nombre dans une base par la représentation de cette base dans le système décimal. Par exemple, on notera la représentation de l'entier 377 par : 377_{10} en base 10, ou 4302_5 en base 5.

Le théorème B.2 donne lieu au corollaire suivant :

Corollaire B.1 *Soit* $b \in \mathbb{N}, b > 1$. *Pour tout* $a \in \mathbb{N}$*, il existe un entier* n *tel que*

$$b^{n-1} \leq a < b^n$$

Preuve : Soit a, un entier naturel quelconque, a s'écrit de manière unique sous la forme

$$a = q_{n-1}.b^{n-1} + \ldots + q_2.b^2 + q_1.b + q_0$$

avec $0 < q_{n-1} < b$ et $\forall i \in \{0, 1, \ldots, (n-2)\}, 0 \leq q_i < b$. On obtient immédiatement l'encadrement

$$b^{n-1} \leq a < b^n$$

Q.E.D.

B.3 Division euclidienne dans $\mathbb{Z}$

Théorème B.3 *Quels que soient* $a \in \mathbb{Z}$ *et* $b \in \mathbb{Z}^*$*, il existe un couple unique* $(q, r) \in \mathbb{Z} \times \mathbb{Z}$ *tels que* $a = b.q + r$ *et* $0 \leq r < b$.

Preuve :

Cas 1 $a \geq 0$

On est ramené à la division euclidienne dans $\mathbb{N}$ et l'on sait (théorème B.1) qu'il existe un couple unique (q, r) d'entiers naturels tels que

$$a = b.q + r \quad \text{et} \quad 0 \leq r < b$$

Cas 2 $a < 0$

Considérons la division euclidienne dans $\mathbb{N}$ de $|\,a\,|$ par b. Il existe un entier naturel q' unique tel que

$$b.q' \leq |\,a\,| < b.(q'+1)$$

(a) si $b.q' = |\,a\,|$, $a = b.(-q')$ et en posant $q = -q'$, $a = b.q$

(b) si $b.q' \neq |\,a\,|$, on a $b.q' < |\,a\,| < b.(q'+1)$ soit $b.q' < -a < b.(q'+1)$, ou encore $b.(-q'-1) < a < b.(-q')$. En posant $q = -q' - 1$, on obtient $b.q < a < b.(q+1)$. Autrement dit :

$$(\forall a \in \mathbb{Z})(\forall b \in \mathbb{N}^*)(\exists! q \in \mathbb{Z})(b.q < a < b.(q+1))$$

En posant $a - b.q = r$ (q étant unique, r l'est aussi), on a :

$$\left\{ \begin{array}{c} a - b.q = r \\ b.q < a < b.(q+1) \end{array} \right.$$

équivalent à

$$\left\{ \begin{array}{c} a = b.q + r \\ 0 \leq r < b \end{array} \right.$$

Q.E.D.

B.4 Congruences modulo n

Définition B.1 *Etant donnés $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, et $n \in \mathbb{N}^*$, x est dit* congru à y modulo n *(noté $x \equiv y\ (mod\, n)$) si et seulement s'il existe un entier rationnel k tel que $x = y + k.n$.*

Remarque : La relation de congruence modulo n est une relation d'équivalence. L'ensemble des classes d'équivalence est noté $\mathbb{Z}/n\mathbb{Z}$.

Théorème B.4 *Soient $x \in \mathbb{Z}$, $y \in \mathbb{Z}$, $n \in \mathbb{N}^*$.*

$$x \equiv y\ (mod\, n) \quad ssi \quad x \text{ et } y \text{ ont le même reste dans la division euclidienne par } n$$

Preuve :

($\Leftarrow$) Supposons que x et y ont le même reste dans la division euclidienne par n. On a :

$$\left\{ \begin{array}{rcl} x & = & n.q + r \\ y & = & n.q' + r \end{array} \right.$$

d'où $x - y = n.(q - q')$, autrement dit $x = y + n.(q - q')$, i.e. $x \equiv y\ (\mathrm{mod}\ n)$.

($\Rightarrow$) On procède par contraposée : montrons que si x et et y ont des restes différents dans la division euclidienne par n, alors x n'est pas congru à y. En raisonnant par l'absurde, on suppose que x et y ont des restes identiques dans la division euclidienne par n, i.e. $x = n.q + r$ et $y = n.q' + r$. Alors $x - y = n.(q - q')$, i.e. $x \equiv y\ (\mathrm{mod}\ n)$: contradiction !

Q.E.D.

Un entier rationnel x est dit donné *modulo* n pour dire qu'il est congru au reste de sa division par n, autrement dit qu'il appartient à la même classe d'équivalence que ce reste (par exemple si 9 est donné *modulo* 4, il est congru à 1 *modulo* 4). Il est en fait commode de se munir de l'opération *modulo*, définie de manière suivante :

Définition B.2 *Etant donnés* $x \in \mathbb{Z}$, *et* $n \in \mathbb{N}^*$,

$$x \text{ modulo } n = y \quad ssi \quad x = n.k + y, \text{avec } r \in \mathbb{Z}, k \in \mathbb{Z}$$

Remarque : La relation de congruence est compatible avec l'addition et la multiplication dans $\mathbb{Z}$. En effet : $\forall (x, x') \in \mathbb{Z} \times \mathbb{Z}$, $\forall (y, y') \in \mathbb{Z} \times \mathbb{Z}$,

$$\text{si } \begin{cases} x \equiv x' \pmod{n} \\ y \equiv y' \pmod{n} \end{cases}$$

$$\text{alors} \quad x + y \equiv x' + y' \pmod{n} \quad \text{et} \quad x.y \equiv x'.y' \pmod{n}$$

Les classes d'équivalence de $\mathbb{Z}/n\mathbb{Z}$ sont caractérisées par les restes de la division euclidienne par n : les restes de la division euclidienne par n ne pouvant être que $0, 1, \ldots n-1$, $\mathbb{Z}/n\mathbb{Z}$ contient n éléments qu'on note $\hat{0}, \hat{1}, \ldots \hat{n-1}$, autrement dit $\mathbb{Z}/n\mathbb{Z} = \{\hat{0}, \hat{1}, \ldots \hat{n-1}\}$.

Exemple B.3 *Tout entier rationnel a pour reste dans la division euclidienne par* 4, *soit* 0, *soit* 1, *soit* 2, *soit* 3. *Les éléments de* $\mathbb{Z}/4\mathbb{Z}$ *sont donc :*

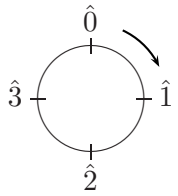
$$\begin{aligned} \hat{0} &= \{\ldots, -8, -4, 0, 4, 8, 12, \ldots\} \\ \hat{1} &= \{\ldots, -7, -3, 1, 5, 9, 13, \ldots\} \\ \hat{2} &= \{\ldots, -6, -2, 2, 6, 10, 14, \ldots\} \\ \hat{3} &= \{\ldots, -5, -1, 3, 7, 11, 15, \ldots\} \end{aligned}$$

Remarque : L'élément noté $\hat{0}$ dans $\mathbb{Z}/4\mathbb{Z}$ est différent de l'élément noté $\hat{0}$ dans $\mathbb{Z}/5\mathbb{Z}$. En effet :

$$\begin{aligned} \hat{0} &= \{\ldots, -8, -4, 0, 4, 8, \ldots\} \quad \text{dans } \mathbb{Z}/4\mathbb{Z} \\ \hat{0} &= \{\ldots, -10, -5, 0, 5, 10, \ldots\} \quad \text{dans } \mathbb{Z}/5\mathbb{Z} \end{aligned}$$

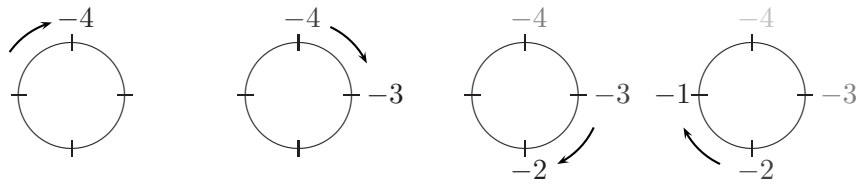
Représentation sur un cercle

Il est parlant de visualiser les éléments de chaque classe d'équivalence sur un cercle ; en tournant sur ce cercle dans le sens conventionnel des aiguilles d'une montre, et en ajoutant un à chaque fois, on passe d'une classe d'équivalence à une autre. Par exemple pour $\mathbb{Z}/4\mathbb{Z}$:

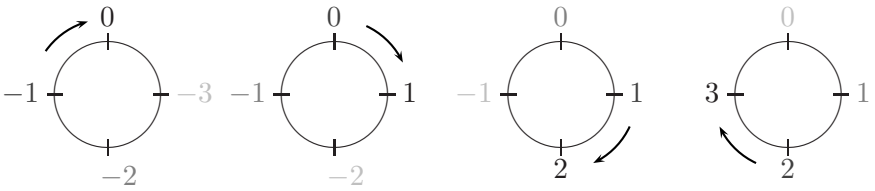


La position des différents éléments de chaque classe d'équivalence sur le cercle est matérialisée par le nombre de tours successifs réalisés sur le cercle. Par exemple, en comptant à partir de -4 :

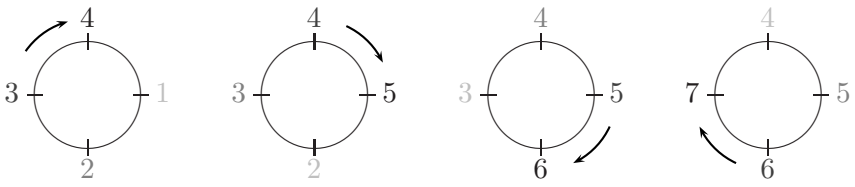
Tour 1



Tour 2



Tour 3



On voit par exemple que -4 (tour 1), 0 (tour 2), et $+4$ (tour 3) appartiennent à la même classe d'équivalence.

Annexe C

La norme IEEE 754

C.1 Représentation IEEE 754

[GermainEtiemble] La norme IEEE 754 comporte deux aspects : la définition d'une représentation commune des nombres fractionnaires (ou «réels»), et des contraintes sur la précision des calculs. Il existe deux formats : *simple* et *double* précision, chacun de ces deux formats pouvant être en outre *étendus*.

Simple précision (32 bits)

1	8	23
s	E (exposant)	M (mantisse)

Double Précision (64 bits)

1	11	52
s	E (exposant)	M (mantisse)

- En simple précision, e est l'interprétation de E en excès à 128 ($e = E_{10} - 127$). On a donc $e_{min} = -127$, $e_{max} = +128$.
- En double précision, e est l'interprétation de E en excès à 1024 ($e = E_{10} - 1023$). On a donc $e_{min} = -1023$, $e_{max} = +1024$.

On distingue cas normalisé ($e_{min} < e < e_{max}$) et cas exceptionnel ($e = e_{min}$ ou $e = e_{max}$).

Cas normalisé Pour $e_{min} < e < e_{max}$, le codage s'interprète de la façon suivante :

- Le bit de poids le plus fort correspond au bit de signe.
- La mantisse utilise un 1 implicite ; donc pour une partie fractionnaire $f_1\ f_2 \dots f_n$, la mantisse m est définie par

$$m = 1, f_1\ f_2 \dots f_n = 1 + \sum_{i=1}^{n} f_i.2^{-i} = (1 f_1\ f_2 \dots f_n)_2.2^{-n}$$

- Au total le nombre est évalué par :

$$x = (-1)^s \times 1, f_1\ f_2 \dots f_n \times 2^e$$

Par exemple, $C8900000_{16}$ code $-2^{18} \times (1+2^{-3})$. En effet : $C8900000_{16} = 1100\,1000\,1001\,0000\ \dots\ 0000_2$. On a donc

1. bit de signe = 1
2. $E = 10010001_2 = 145$, donc $e = 17$
3. $f = 0010\ldots0$, donc $m = 1,001_2 = 1 + 2^{-3}$

Cas exceptionnel La table suivante montre que les valeurs extrêmes sont réservées à la représentation des nombres non représentables en mode normalisé :

Cas	e	f	valeur
Normalisé	$e_{min} < e < e_{max}$	quelconque	$(-1)^s \times 1, f \times 2^e$
Dénormalisé	$e = e_{min}$	$\neq 0$	$(-1)^s \times 0, f \times 2^{e_{min}}$
Zéro	$e = e_{min}$	0	$(-1)^s \times 0$
Infini	$e = e_{max}$	0	$(-1)^s \times \infty$
NaN	$e = e_{max}$	$\neq 0$	*NaN*

– Il existe un plus grand nombre exactement représentable en normalisé :

$$x_m = 1,1\ldots1 \times 2^{emax-1}$$

Les nombres plus grands sont représentés par les mots où le signe est positif, $e = e_{max}$ et $f = 0$. L'interprétation de tous ces mots est identique : $+\infty$. Pour les nombres négatifs, on a une représentation analogue de $-\infty$.

– Il existe un plus petit nombre strictement positif représentable en normalisé. Cependant, la représentation dénormalisée permet de représenter des nombres plus petits. On notera que le bit implicite n'est plus 1, mais 0, d'où l'appellation dénormalisé.
– Il existe deux représentations de 0, suivant le bit de signe.
– Enfin, un code est réservé pour représenter le résultat d'une opération aberrante, par exemple 0/0. Ce code est *NaN*.

C.2 Opérations flottantes

Les opération flottantes impliquent un traitement simultané des mantisses (c'est à dire des parties fractionnaires) et des exposants. Les principales opérations sont les comparaisons et les opérations arithmétiques : addition, soustraction, multiplication, et division sont réalisées par des opérateurs matériels spécifiques. Le standard IEEE 754 ne prescrit rien sur l'implémentation. Il spécifie simplement les bits d'une représentation, et du résultat des opérations arithmétiques flottantes. Ainsi le langage java offre un calcul flottant conforme IEEE 754 indépendamment de toute plate-forme matérielle.

Comparaisons L'utilisation du bit de signe permet le test rapide du signe. La notation en excès pour l'exposant permet de comparer les flottants en utilisant la comparaison des entiers naturels.

Addition et soustraction L'addition implique une dénormalisation du nombre le plus petit pour que les exposants deviennent égaux, suivie d'une addition des mantisses, qui est suivie éventuellement d'une renormalisation. La dénormalisation et la renormalisation peuvent entraîner une perte d'information. L'addition peut entraîner la sortie du champs des nombres représentables en normalisé : par exemple $x_m + x_m$.

Multiplication Soit deux nombres flottants $x_1 = s_1\, m_1\, 2^{e_1}$ et $x_2 = s_2\, m_2\, 2^{e_2}$. Le produit $x_1.x_2$ est donné par $s_1\, s_2\, m_1\, m_2\, 2^{e_1+e_2}$. Il y a multiplication des mantisses, ce qui correspond à une multiplication entière, où l'on arrondit pour obtenir un résultat correspondant au nombre de bits de la partie fractionnaire. Compte-tenu du codage en excès, l'addition des exposants correspond à l'opération $E_1 + E_2 - c$ où les E_i sont interprétés comme des entiers naturels, $c = 127$ en simple précision et 1023 en double précision. Là encore, il peut y avoir perte d'information, lors de l'arrondi, ou sortie du champs des nombres représentables, lorsque l'exposant est trop grand ou trop petit.

Traitement des situations anormales Comme on vient de le voir, les calculs peuvent entraîner soit un dépassement de capacité, le résultat n'étant pas représentable au format normalisé, soit une erreur, le résultat étant arrondi. La norme IEEE 754 vise à permettre à un programme d'adopter un comportement adéquat dans ce type de situation anormale. Les résultat non-représentables sont pris en compte par l'arithmétique étendue et la représentation dénormalisée. Les erreurs sont prises en compte par une contrainte de précision.

Arithméthique étendue L'idée de l'arithmétique étendue est qu'il peut être profitable de laisser survivre un programme qui a, par exemple, effectué une division par 0. L'exemple plus simple est un solveur qui cherche les zéros d'une fonction dont l'ensemble de définition n'est pas commodément calculable. Le solveur travaille en évaluant la fonction en divers points. S'il tombe sur une valeur en dehors de l'ensemble de définition de la fonction, il peut fort bien calculer $0/0$ ou $\sqrt{-1}$. Cette erreur n'est pas nécessairement gênante, si le solveur peut recommencer en un autre point, indépendamment. La norme définit une arithmétique étendue à trois valeurs supplémentaires, $-\infty$, $+\infty$, et *NaN*. Les règles utiliées sont les règles usuelles, par exemple :
- $(+\infty) + (+\infty) = +\infty$
- $(+\infty) + (-\infty) = NaN$
- $(\pm\infty) \times (\pm\infty) = (\pm\infty)$ avec règle des signes.
- Toute opération dont un des opérateurs est *NaN* a pour résultat *NaN*.

Du point de vue de l'application, si une des opération produit un résultat trop grand en valeur absolue, ou bien mathématiquement incorrect $(0/0)$, le résultat tombe dans l'ensemble $\{-\infty, +\infty, NaN\}$, et les calculs peuvent être poursuivis en utilisant l'arithmétique étendue. Tout FPU conforme IEEE 754 doit implémenter cette arithmétique étendue.

Représentation dénormalisée La différence de deux nombres normalisés peut ne pas être représentable au format normalisé. Par exemple, en simple précision ($e_{min} = 127$), $x = 1,1\ldots11 \times 2^{-126}$ et $y = 1,\ldots10 \times 2^{-126}$ sont représentables en normalisé (respectivement par $00FFFFFF_{16}$ et $00FFFFFE_{16}$). Mais $x - y = 0,0\ldots01 \times 2^{-126}$ n'est pas représentable en normalisé. On pourrait tout simplement arrondir le résultat au plus proche représentable, soit 0. Mais supposons que le FPU réalise correctement la comparaison de deux flottants en testant l'égalité de tous leurs bits. Le test $x = y$ donne *faux*, et le test $x - y = 0$ donne *vrai*. Deux codes identiques du point de vue mathématique auront des résultats différents : par exemple, le code `if not (x=y) then z=1/(x-y)` pourrait aboutir à une division par 0, et le code `if not (x-y=0) then z=1/(x-y)` ne produit pas d'erreur. La représentation dénormalisée permet précisément la représentation de ces nombres trop petits : $0,0\ldots01 \times 2^{-126} = 0,0\ldots10 \times 2^{-127}$, qui se représente comme 00000002_{16}. Dans ce cas les deux test ont

le même résultat. La différence de deux nombres dénormalisés peut elle-même être trop petite pour être représentée, même en dénormalisé, et donc peut être arrondie à 0 (voir le paragraphe consacré à la précision ci-après).

C.3 Drapeaux et gestionnaires d'exceptions

Le standard IEEE 754 fournit un fonctionnement par défaut, qui est de continuer le calcul dans l'arithmétique étendue. Continuer l'exécution est souvent la solution appropriée, mais pas toujours. Sur erreur arithmétique flottante, une application peut donc souhaiter trois comportements : arrêt immédiat, ou contrôle par elle-même, ou non-traitement.

Un exemple typique d'application d'application qui demande un arrêt immédiat est celui du calcul de $\frac{x}{1+x^2}$. En simple précision, quand $x = 2^{65}$ (représentable par 60000000_{16}), x^2 produit ∞ et le résultat est 0, alors qu'il est de l'ordre de $1/x$, qui est parfaitement représentable ; le résultat est donc complètement erroné et continuer l'exécution en général sans intérêt (et éventuellment coûteux en temps machine). Un exemple typique d'application qui demande un traitement nuancé est une application qui génère des nombres dénormalisés : elle ne souhaite pas s'interrompre prématurément à chaque résultat dénormalisé, mais veut s'interrompre si elle obtient un 0 comme résultat de la soustraction de nombres dénormalisés.

Le standard prescrit que les évènements anormaux soient enregistrés dans des drapeaux (flags), et recommande fortement que des gestionnaires d'exceptions soient installés. Les drapeaux permettent un traitement personnel à l'utilisateur (éventuellement rien) ; les gestionnaires d'exception fournissent un traitement par le logiciel de base («système »), qui permet en particulier l'arrêt immédiat. La figure suivante présente ces drapeaux :

Flag	Condition	Résultat
Underflow	Nombre trop petit	$0, \pm e_{min}$, ou nombre dénormalisé
Overflow	Nombre trop grand	$\pm\infty$ ou x_{max}
Division par 0	Division par 0	$\pm\infty$
Invalid Operation	Résultat = NaN et Opérateurs $\neq NaN$	NaN
Inexact	Résulat arrondi	Résultat arrondi

Des instructions du processeur permettent de les tester. Finalement, des routines des bibliothèques numériques offrent une interface utilisateur vers ces instructions, par exemple la *libm* qui est la librairie numérique standard associée au langage C. L'utilisateur peut donc choisir de ne pas tester les drapeaux, ou de les tester et d'appliquer un algorithme de son choix. La norme prescrit que les drapeaux soient persistants (sticky) : à la différence des drapeaux entiers, un drapeau positionné ne sera remis à 0 que par une instruction explicite. Les drapeaux et les librairies numériques permettent donc le contrôle par l'application.

Un traitement générique, en général l'arrêt immédiat, est rendu possible si chaque dépassement de capacité flottant peut déclencher une exception. Le drapeau n'est alors pas positionné.

Le choix entre positionnement des drapeaux (avec poursuite de l'exécution du programme utilisateur) et exception est programmable. Les micro-processeurs offrent des bits de contrôle qui font généralement partie d'un mot de contrôle du processeur, et des instructions pour les positionner, qui permettent ce choix. Il y a un bit de contrôle par exception. Lorsque l'erreur ne produit pas l'appel d'un gestionnaire d'exception, on dit que l'exception est *masquée*. Par

exemple, l'exception *Résultat Inexact* est presque toujours masquée : la plupart des calculs effectuent des arrondis et le programme doit continuer. L'interface d'un langage de haut niveau vers les instructions de masquage est réalisée par des options du compilateur.

C.4 Précision

Les opérations d'alignement et d'arrondi perdent de l'information, ce qui peut être sans effet, ou catastrophique. Par exemple sur trois digits décimaux de mantisse, soient $x = 2,15 \times 10^2$ et $y = 1,25 \times 10^{-5}$. En alignant y sur x et en arrondissant, $y = 0$ et $x \ominus y = 2,15 \times 2^{12}$, ce qui est le résultat arrondi correct ($\ominus$ correspondant à l'opérateur de soustraction sur trois digits) . Mais pour $x = 1,01 \times 10^1$ et $y = 9,93$ en alignant y sur x et en arrondissant, $y = 0,99 \times 10^1$ et $x \ominus y = 0,02$, alors que le résultat sur trois digits $10,1 - 9,93 = 0,17$. Deux digits sont donc faux : l'erreur sur le dernier digit s'est propagée.

Une mesure de précision est le *ulp* (units in last position). Dans le calcul précédent, l'erreur est de 15 *ulp*.

Le standard IEEE impose que les opérations d'addition, soustraction, multiplication et division soient *arrondies exactement* : tout se passe comme si le résultat était calculé exactement, puis arrondi au plus près. Ceci requiert que l'unité de calcul dispose de plus de bits que le format, pour stocker temporairement des informations. Il a été montré que trois bits suffisent (garde, garde supplémentaire et sticky bit).

Le contrat assuré par le standard IEEE ne garantit évidemment rien sur le cumul des erreurs dans une séquence de calculs. L'étude de la qualité numérique des résultats d'un algorithme est une branche de l'analyse numérique.

L'exemple le plus simple est l'associativité, que le calcul flottant ne respecte pas. Considérons l'exemple suivant, toujours avec une mantisse de trois digits décimaux : d'une part

$$-2,15 \times 10^{12} \oplus (2,15 \times 10^{12} \oplus 1,25 \times 10^5) = -2,15 \times 10^1 2 \oplus 2,15 \times 10^1 2 = 0,$$

d'autre part

$$(-2,15 \times 10^{12} \oplus 2,15 \times 10^{12}) \oplus 1,25 \times 10^5 = 1,25 \times 10^5.$$

Annexe D

Relations fondamentales de l'algèbre de Boole

Relation	*Intitulé*
$\forall x \in \{0,1\}\ x + \bar{x} = 1$	tautologie
$\forall x \in \{0,1\}\ x.\bar{x} = 0$	contradiction
$\forall x \in \{0,1\}\ \bar{\bar{x}} = x$	involution
$\forall x \in \{0,1\}\ x + x = x$ $\forall x \in \{0,1\}\ x.x = x$	idempotence
$\forall x \in \{0,1\}\ x + 1 = 1$ $\forall x \in \{0,1\}\ x.0 = 0$	élément absorbant
$\forall (x,y) \in \{0,1\}^2\ x + x.y = x$ $\forall (x,y) \in \{0,1\}^2\ x.(x + y) = x$	absorption
$\forall (x,y) \in \{0,1\}^2\ x + \bar{x}.y = x + y$ $\forall (x,y) \in \{0,1\}^2\ x.(\bar{x} + y) = x.y$	simplification
$\forall (x,y) \in \{0,1\}^2\ \overline{x + y} = \bar{x}.\bar{y}$ $\forall (x,y) \in \{0,1\}^2\ \overline{x.y} = \bar{x} + \bar{y}$	formules de De Morgan
$\forall (x,y,z) \in \{0,1\}^3\ x.(y + z) = (x.y) + (x.z)$	distributivité de . sur +
$\forall (x,y,z) \in \{0,1\}^3\ x + (y.z) = (x + y).(x + z)$	distributivité de + sur .
$\forall (x,y) \in \{0,1\}^2\ x \oplus y = \overline{x}.y + x.\overline{y}$	expression de $\oplus$ en fonction de . et +
$\forall x \in \{0,1\}\ x \oplus 0 = 0 \oplus x = x$	élément neutre pour $\oplus$
$\forall x \in \{0,1\}\ x \oplus x = 0$	élément symétrique pour $\oplus$
$\forall (x,y,z) \in \{0,1\}^3\ x.(y \oplus z) = (x.y) \oplus (x.z)$	distributivité de . sur $\oplus$
$\forall (x,y) \in \{0,1\}^2\ \overline{x \oplus y} = \overline{x} \oplus y = x \oplus \overline{y}$ $\forall (x,y) \in \{0,1\}^2\ \overline{x} \oplus \overline{y} = x \oplus y$	relations avec l'inversion

Annexe E

Aide-mémoire MIPS

E.1 Registres

Nom	Numéro	Utilisation	A sauvegarder
`$zero`	0	constante 0	
`$at`	1	*réservé* à l'assembleur	
`$v0`–`$v1`	2–3	retours de fonctions	non
`$a0`–`$a3`	4–7	passage de paramètres	oui
`$t0`–`$t7`	8–15	registres généraux	non
`$s0`–`$s7`	16–23	registres généraux	oui
`$t8`–`$t9`	24–25	registres généraux	non
`$k0`–`$k1`	26–27	*réservé* au noyau système	
`$gp`	28	pointeur global	oui
`$sp`	29	pointeur de pile	oui
`$fp`	30	pointeur de frame	oui
`$ra`	31	adresse de retour de fonction	oui

E.2 Formats d'instruction

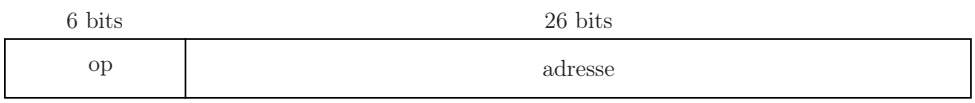
R

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	décal	fonct

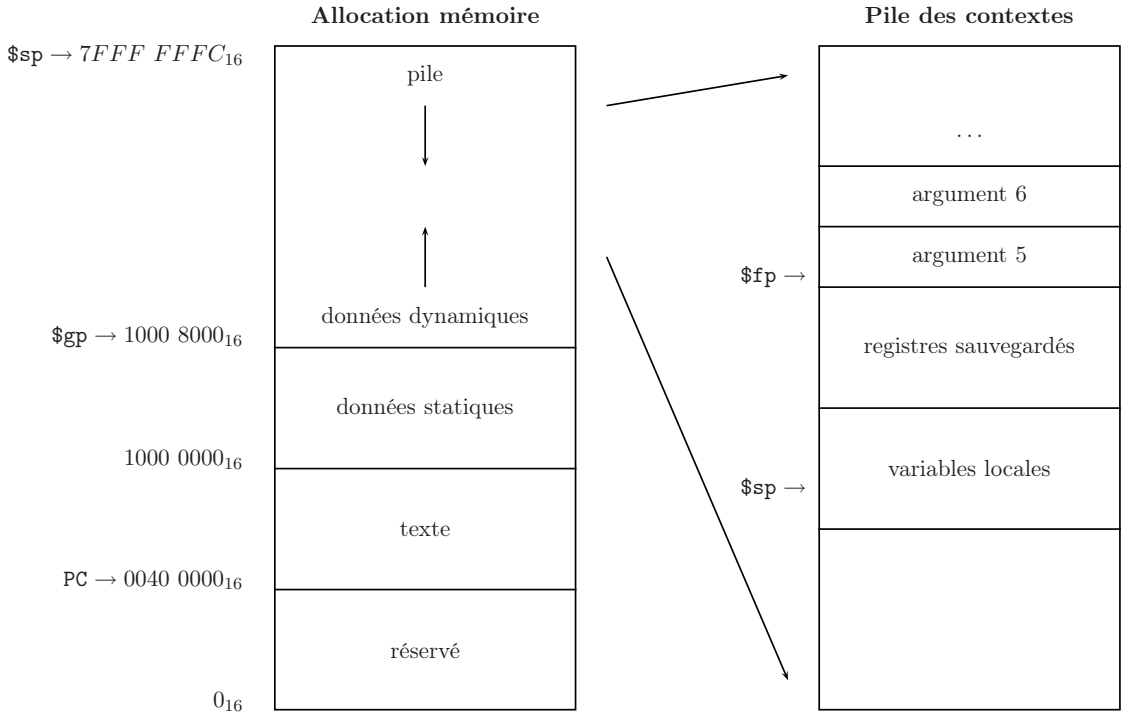
I

6 bits	5 bits	5 bits	16 bits
op	rs	rt	immédiat

J



E.3 Gestion de la mémoire



E.4 Instructions

Nom	Exemple	Format	Description
add	`add $s3, $s2, $s1`	R	`s3=s2+s1 ;`
addi	`add $s3, $s2, 100`	I	`s3=s2+100 ;`
and	`and $s3, $s2, $s1`	R	`s3=s2&&s1 ;`
beq	`beq $t0, $t1, Etiquette`	I	`if (t0=t1) goto Etiquette ;`
bne	`bne $t0, $t1, Etiquette`	I	`if (t0 !=t1) goto Etiquette ;`
j	`j Etiquette`	J	`goto Etiquette ;`
jal	`jal Etiquette`	J	`ra=PC+4 ; goto Etiquette ;`
jr	`jr $ra`	J	Saut à l'adresse contenue dans le registre `ra`, cette adresse étant alignée sur une frontière de mot
la	`la $t0, Etiquette`	pseudo-instr.	`lui $t0, adresse Etiquette»16` `ori $t0, $t0, adresse Etiquette & 0xFFFF`
lui	`lui $t0, 0xFF14`	I	`t0=0xFF14«16 ;`
lw	`lw $s0, 32($sp)`	I	`s0=Mem[sp+32] ;`
mul	`mul $s3, $s2, $s1`	R	`s3=s2*s1 ;`
nor	`nor $s3, $s2, $s1`	R	`s3=~(s2|s1) ;`
or	`or $s3, $s2, $s1`	R	`s3=s2|s1 ;`
ori	`ori $s3, $s2, 0xFFFF`	I	`s3=s2|0xFFFF ;`
sll	`sll $s2, $s1, 4`	I	`s2=s1«4 ;`
slr	`slr $s2, $s1, 4`	I	`s2=s1»4 ;`
slt	`slt $t0, $s1, $s0`	R	`if (s1<s0) t0=1 ; else t0=0 ;`
slti	`slt $t0, $s1, 2`	I	`if (s1<2) t0=1 ; else t0=0 ;`
sub	`sub $s3, $s2, $s1`	R	`s3=s2-s1 ;`
sw	`sw $s0, 32($sp)`	I	`Mem[sp+32]=s0 ;`
xor	`xor $s3, $s2, $s1`	R	`s3=s2||s1 ;`

Bibliographie

[Arnold98] Douglas N. Arnold. Some disasters attributable to bad numerical computing, 1998. URL : http ://www.ima.umn.edu/ arnold/disasters/.

[Breton90] Philippe Breton. *Une histoire de l'informatique*. Collection Points Sciences. Seuil, 1990. ISBN 2-02-012348-7.

[DanceaMarchand92] Ioan Dancea and Pierre Marchand. *Architecture des ordinateurs*. Gaëtan Morin, 1992. ISBN 2-89105-438-5.

[Darche02] Philippe Darche. *Architecture des ordinateurs–tome 2 : Fonctions booléennes, logique combinatoire et séquentielle*. Passeport pour l'Informatique. Vuibert, 2002. ISBN 2-7117-8688-9.

[Dewdney93] A. K. Dewdney. *The (New) Turing Omnibus*. W. H. Freeman and Company, 1993. ISBN 0-7167-8271-5.

[GermainEtiemble] Cécile Germain and Daniel Etiemble. Architecture des ordinateurs. Licence d'Informatique–IUP Miage–FIIFO, Université Paris XI, URL : http ://www.lri.fr/ENSEIGNANTS/LM/archi/L4.html.

[Goldberg91] David Goldberg. What every computer scientist should know about floating–point arithmetic. Copyright, Association for Computing Machinery, Inc, 1991. Disponible (.html) à l'URL : http ://docs.sun.com/source/806-3568/ncg_goldberg.html, ou téléchargeable (.pdf) à l'URL : http// :www.validlab.com/goldberg/paper.pdf.

[HennessyPatterson96] John Hennessy and David Patterson. *Computer Architecture–A Quantitative Approach*. Morgan Kaufmann Publishers, 2e edition, 1996. ISBN 2-84180-022-9.

[LewisPapadimitrou81] H. R. Lewis and C. H. Papadimitrou. *Elements of the Theory of Computation*. Prentice–Hall, Englewood Cliffs, N. J., 1981.

[MeyerBaudoin80] Bertrand Meyer and Claude Baudoin. *Méthodes de Programmation*. Collection de la Direction des Etudes et Recherches d'Electricité de France. Eyrolles, 2e edition, 1980. ISSN 0399-4198.

[Minsky67] Marvin Minsky. *Computation : Finite and Infinite Machines*. Prentice–Hall, Englewood Cliffs, N. J., 1967.

[Tisserant03] Sylvain Tisserant. Architecture et technologie des ordinateurs. Département d'Informatique – Ecole Supérieure d'In-

génieurs de Luminy, Université de la Méditerranée, URL : http ://tisserant.developpez.com/, 2003.

[Wikipedia] Wikipedia. URL : http ://fr.wikipedia.org/.