Introduction à l'Informatique

Vincent Boyer et Jean Méhat

5 février 2016

Copyright (C) 2009–2016 Jean Méhat

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation ; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Table des matières

1 Introduction **7**

1.1 Définitions 7

1.2 Organisation générale d'un ordinateur 8

1.2.1 La mémoire 8

1.2.2 L'unité centrale 9

1.2.3 Les unités d'entrées sorties 9

1.3 Exécution d'un programme 10

1.3.1 Programme original 10

1.3.2 Programme traduit pour l'ordinateur 10

1.3.3 Instruction de base 11

1.3.4 Codage des instructions 11

1.3.5 Traduction du programme 13

1.3.6 L'unité de controle, le PC 13

1.4 La brique de base : le transistor 15

1.5 Résumé 16

1.6 Exercices 16

2 La représentation des nombres entiers positifs **18**

2.1 Pour les matheux 18

2.2 Pour les programmeurs 18

2.3 Pour les autres 19

2.3.1 Qu'est-ce qu'un nombre ? 19

2.3.2 La notation usuelle : numération en base décimale 20

2.3.3 D'autres représentations usuelles 20

2.3.4 Nommer les chiffres d'un nombre 21

2.3.5 La valeur d'un nombre décimal 21

2.3.6 Représentation des nombres en base constante 22

2.3.7 Conversions vers la base 10 23
2.3.8 Conversion depuis la base dix 25
2.4 Conversions entre binaire, octal et hexadécimal 28
2.4.1 Pour les matheux 28
2.4.2 Pour les autres 29
2.4.3 Récapitulation des conversions entre les bases 10, 2, 8 et 16 29
2.5 Résumé 31
2.6 Exercice 32
2.7 Supplément : le binaire facile 32
2.8 Supplément : la commande `gnome-calculator` 33

3 Opérations en binaire 34
3.1 Addition en base 2 34
3.2 La multiplication en base 2 35
3.2.1 La multiplication en base 10 sans table 36
3.2.2 Le décalage à gauche et à droite 36
3.2.3 Un algorithme de multiplication en binaire 37
3.2.4 Poser la multiplication en base 2 38
3.3 Résumé 38
3.4 Exercices 38

4 Représentation des nombres signés 40
4.1 Le bit de signe, la notation en excédent 40
4.1.1 Le bit de signe 41
4.1.2 La représentation en excédent 41
4.2 Représentation des nombres signés en complément à 2 42
4.2.1 Les bases du complément à deux 42
4.2.2 Calculer l'opposé d'un nombre en complément à 2 43
4.2.3 L'arithmétique en complément à 2 45
4.2.4 La multiplication 46
4.3 Résumé 46
4.4 Exercice 47

5 Calculer des ordres de grandeur entre base 2 et base 10 48
5.1 Exercices 49

6 Représenter les nombres qui ne sont pas entiers 51
6.1 Représentation des nombres fractionnels 51

6.1.1 Conversion de la partie fractionnelle d'un nombre de la base 10 vers la base 2 52
6.1.2 Conversion de la partie fractionnelle d'un nombre de la base 2 à la base 10 53
6.2 Représentation des nombres en virgule fixe 53
6.3 Représentation des nombres en virgule flottante 54
6.3.1 Virgule flottante en base 10 54
6.4 Virgule flottante en base 2 55
6.4.1 Exemples de nombres en virgule flottante 55
6.4.2 Les nombres en virgule flottante dans la mémoire . . . 55
6.4.3 Flottants ou réels 56
6.5 La norme IEEE754 56
6.5.1 Cas particuliers 57
6.6 Résumé 58
6.7 Exercice 58
6.8 Supplément : un outil de calcul 59

7 Représentation des caractères 61
7.1 Les codes de caractères 61
7.2 Le code ASCII 61
7.2.1 Organisation du code ASCII 61
7.2.2 Les limites du code ASCII 62
7.3 La norme iso8859 : une extension du code ASCII 63
7.4 Unicode : un codage sur seize bits 64
7.5 UTF : un codage à longueur variable 64
7.6 la commande `file` 66
7.7 Résumé 67
7.8 Exercices 67
7.9 Supplément : exercice type corrigé sur les conversions . . . 68
7.10 Supplément : saisir des caractères divers au clavier 70
7.10.1 Choisir Compose avec les versions Ubuntu de 2013 . . . 70
7.10.2 Taper un caractère non-standard 70

8 Algèbre de Boole 72
8.1 Les fonctions logiques 72
8.1.1 Fonctions logiques et tables de vérité 74
8.2 Simplification des expressions logiques 78
8.2.1 Simplifications algébriques des expressions logiques . . . 78

8.2.2 Simplification des expressions logiques avec les tables de Karnaugh . . . 79
8.2.3 Simplification des expressions logiques en pratique . . . 85
8.3 Résumé . . . 86
8.4 Exercice . . . 86

9 Des dispositifs pour calculer **89**
9.1 Les relais . . . 89
9.2 Les transistors . . . 91
9.2.1 La structure d'un transistor . . . 91
9.2.2 Les transistors N et P . . . 92
9.2.3 L'inverseur CMOS . . . 92
9.2.4 Le ET et le OU en CMOS . . . 93
9.2.5 Réalisation des circuits . . . 94
9.3 Exercices . . . 94

10 Portes logiques et circuits combinatoires **95**
10.1 Codage des chiffres . . . 95
10.2 Un exemple : conception d'un additionneur . . . 96
10.2.1 Les circuits de base . . . 96
10.2.2 Equivalence entre fonction et circuit . . . 96
10.2.3 Le demi-additionneur . . . 97
10.2.4 L'additionneur 1 bit . . . 99
10.2.5 L'additionneur 4 bits . . . 100
10.3 Le soustracteur 4 bits . . . 101
10.4 L'additionneur–soustracteur 4 bits . . . 103
10.5 Le multiplexeur . . . 105
10.6 L'unité de calcul . . . 106
10.7 Résumé . . . 109
10.8 Exercices . . . 109

11 Circuits séquentiels **112**
11.1 Les boucles d'inverseurs . . . 112
11.2 La bascule RS . . . 113
11.3 La bascule D . . . 114
11.4 Construction d'un mot mémoire avec des bascules D . . . 115
11.5 Lecture dans une mémoire de 4×1 bit . . . 116
11.6 Ecriture dans une mémoire de 4×1 bit . . . 118

11.7 Lire et écrire dans une mémoire $x \times y$ bits . . . 119
11.8 Catégories de mémoire . . . 120
11.8.1 Mémoire RAM . . . 120
11.8.2 Mémoire statique . . . 120
11.8.3 Mémoire dynamique . . . 121
11.8.4 Mémoire ROM et variantes . . . 121
11.9 Exercices . . . 122

12 Automates **123**
12.1 Les automates : un exemple simple . . . 123
12.1.1 Du graphe à la réalisation . . . 124
12.2 Réalisation d'un automate avec une entrée . . . 126
12.3 Un automate général . . . 130
12.4 Exercices . . . 130

13 L'ordinateur en papier **132**
13.1 Anatomie . . . 132
13.2 Fonctionnement . . . 133
13.3 Le langage d'assemblage . . . 134
13.4 Les modes d'addressage . . . 135
13.5 Les microcodes . . . 136
13.6 Le cycle . . . 137
13.7 Exemple de programme : un boot-strap . . . 138
13.8 Exercices . . . 139

14 Projet et évaluation **143**

Chapitre 1

Introduction

Ce support de cours est destiné aux étudiants de la première année de licence. Son but est de permettre aux étudiants d'acquérir les notions de base de l'architecture d'un ordinateur, pour lui permettre de maîtriser la façon dont les programmes qu'il écrit et leurs données sont traités par l'ordinateur pour produire des résultats.

La première partie traite en assez grand détail les codages usuels utilisés pour représenter les données, en particulier les nombres et les caractères. Elle contient également la description de méthodes qu'on emploie sur ces représentations de nombres pour effectuer les opérations arithmétiques usuelles.

La seconde partie montre comment on peut, à aide de la logique et de transistors, réaliser des circuits pour effectuer des calculs et mémoriser des valeurs ; ensuite, en combinant organes de calcul et mémoires, nous montrons la réalisation d'automates.

Finalement, nous présentons un ordinateur très élémentaire mais complet et nous montrons comment écrire des programmes pour cette machine.

Dans cette introduction, nous présentons un exemple de programme qui effectue un calcul, puis nous fabriquons un ordinateur qui peut exécuter ce programme. Le but est de montrer la manière dont les éléments s'imbriquent entre eux avant de rentrer dans les chapitres suivant dans les détails de la représentation des données.

1.1 Définitions

Exercice 1.1 : Trouvez par vous même et écrivez une définition des mots *ordinateur*, *programme* et *information*. (Il est important d'effectuer cet exercice avant le suivant.)

Exercice 1.2 : Chercher dans un dictionnaire et noter les définitions des mots *ordinateur*, *programme* et *information*.

1.2 Organisation générale d'un ordinateur

L'ordinateur est composé principalement de trois parties :
— la mémoire centrale où on stocke les données et les programmes
— l'unité centrale (UC, ou CPU comme *Central Processing Unit*) où se font les calculs
— les unités d'entrées sorties pour échanger des informations avec l'extérieur.

Ces différentes parties sont reliées entre elles par un canal de communication qu'on appelle un *bus*. dans la plupart des bus, les composants ne peuvent pas échanger simultanément de l'information : à un moment donné, il n'y a qu'un seul échange qui a lieu.

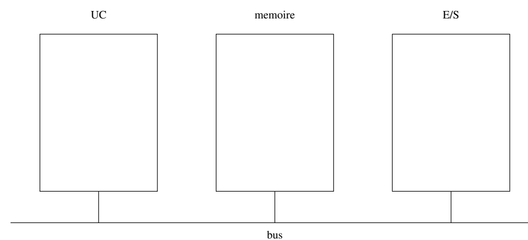


FIGURE 1.1 – Structure d'un ordinateur : première approche

Nous allons examiner chacun de ces composants.

1.2.1 La mémoire

La mémoire peut être considérée comme une série d'endroits numérotés que nous appelons des *mots*, où on peut placer des valeurs. Le numéro de chaque mot est appelé son *adresse*. Chaque mot mémoire peut contenir un nombre, qui peut servir à *représenter n'importe quoi, à condition de se mettre d'accord sur un code*. Par exemple, en utilisant avec le code ASCII, le nombre 65 sert à représenter le caractère `A`, le nombre 33 le caractère *point d'exclamation* `!`, le nombre 32 le caractère *espace*. Les chapitres 2, 4 et 5 et 6 s'étendent sur la manière dont le contenu des mots peut servir à représenter différentes choses (des nombres positifs, négatifs, fractionnels et des caractères).

Il est possible d'effectuer deux opérations sur un mot mémoire : y écrire une valeur et lire ce qu'on y a écrit précédemment.

La mémoire est utilisée pour stocker les programmes exécutés sur l'ordinateur. Ces programmes sont donc au préalable chargés en mémoire. Les données manipulées par le programme sont elles aussi stockées en mémoire. Par conséquent chaque instruction du programme et chaque donnée est codée par un nombre. Le stockage des programmes en mémoire au même titre que les données est une caractéristique essentielle de nos ordinateurs.

1.2.2 L'unité centrale

L'unité centrale est composée de deux parties principales : l'unité de calcul et l'unité de commande (appelée aussi unité de contrôle).

L'unité de calcul permet d'effectuer des opérations simples, comme l'addition de deux nombres. L'ensemble des opérations réalisables par l'unité de calcul est définie par le constructeur de l'unité centrale.

L'unité de contrôle a pour rôle de lire une instruction du programme en mémoire et de la faire réaliser par l'unité de calcul en lui fournissant les opérandes et l'opération à réaliser puis de passer à l'instruction suivante du programme.

1.2.3 Les unités d'entrées sorties

Les unités d'entrées sorties sont les unités d'échanges d'informations entre l'ordinateur et son environnement externe. On abrège souvent leur nom en *E/S* ou *IO* (prononcer *Ailleau*) comme *Input Ouput*

Il existe de nombreux types d'unités d'E/S. Certaines permettent seulement à l'ordinateur de lire des données, comme le clavier ou le CD-Rom. D'autres ne permettent que d'écrire des données, comme l'écran ou les imprimantes. Certaines sont utilisées à la fois en lecture et en écriture, comme l'adaptateur réseau ou le disque dur.

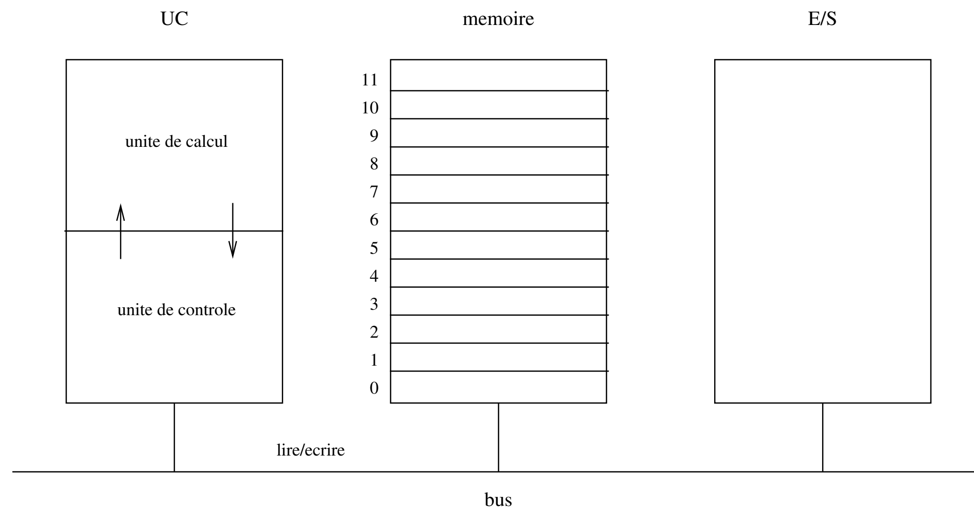


FIGURE 1.2 – Structure d'un ordinateur : seconde approche

1.3 Exécution d'un programme

1.3.1 Programme original

Afin de montrer comment fonctionne un ordinateur, nous allons montrer comment il exécute un programme. Le programme choisi calcule la racine carré r d'un nombre n selon l'algorithme dit *de Newton-Raphson*.

```
début
  r <- 1
  répéter
    r <- (r + n / r) / 2
fin
```

On commence avec une valeur approchée de la racine, ici 1, puis à chaque tour on améliore cette approximation en faisant la moyenne entre r et n/r ; il va de soi que si r est plus petit que la racine exacte, alors n/r est plus grand et inversement si r est trop grand, alors n/r est trop petit. (L'agorithme peut être utilisé pour calculer les valeurs d'autres fonctions que la racine carrée. La raison pour laquelle l'algorithme converge effectivement vers la valeur exacte sort du cadre de ce cours.).

Considérons que $n = 4$. Voici comment évolue la valeur de r à mesure que les calculs se déroulent.

$$n = 4 \qquad r = 1, \tag{1.1}$$
$$r = 2.5, \tag{1.2}$$
$$r = 2.05, \tag{1.3}$$
$$r = 2.0006097560 \tag{1.4}$$
$$r = 2.0000000929 \tag{1.5}$$
$$\ldots \tag{1.6}$$

Nous avons ici le squelette d'un programme ; il lui manque cependant au moins une façon de lire la valeur de n et d'écrire les valeurs successives que prend r, qui sont du ressort des periphériques d'entrée-sortie. Il serait sans doute souhaitable que le programme s'arrête au bout d'un certain temps, par exemple après avoir répété un certain nombre de fois les calculs, ou bien quand la valeur de r est suffisamment proche de la valeur exacte de la racine recherchée. (On peut évaluer facilement cette erreur ; plus $n - r^2$ se rapproche de 0 et plus l'erreur est petite.)

1.3.2 Programme traduit pour l'ordinateur

Ce programme est écrit dans un langage dit *de haut niveau*, destiné à être compris par le programmeur ; il faut le traduire en instructions que l'ordinateur sait exécuter. (Le travail de traduction d'un programme en instructions

élémentaires pour l'ordinateur est en général effectué lui-même par un autre programme appelé *interpréteur* ou *compilateur*.)

L'ordinateur ne possède pas d'instructions complexes, comme *calculer* $(r + n/r)/2$; il ne peut effectuer que des opérations élémentaires avec deux opérandes. Si on traduit le programme en telles instructions, on pourrait avoir :

```
(1) placer 1 dans r
(2) diviser n par r
(3) ajouter r et le resultat du calcul précédent
(4) diviser le résultat du calcul précédent par 2
(5) placer le résultat dans r
(6) recommencer avec l'instruction 2
```

Comme il est difficile de désigner des choses comme *« le résultat du calcul précédent »*, nous prenons des mots de la mémoire, que nous pourrons désigner par leurs adresses, pour contenir les résultats des calculs intermédiaires. Soient t1, t2 et t3 trois mots mémoires. Le programme exprimé en instructions élémentaires devient alors :

```
(a) placer 1 dans r
(b) diviser n par r, placer le résultat dans t1
(c) additionner r et t1, placer le résultat dans t2
(d) placer 2 dans t3
(e) diviser t2 par t3, placer le résultat dans r
(f) recommencer avec l'instruction b
```

1.3.3 Instruction de base

Si on fait la liste des instructions élémentaires que nous utilisons, on voit qu'il n'y en a que quatre :

1. *placer une valeur dans un mot*, utilisé par les instructions a et d.
2. *diviser le contenu d'un mot par celui d'un autre mot, placer le résultat dans un troisième mot*, utilisé par les instructions b et e.
3. *additionner le contenu de deux mots, placer le résultat dans un troisième mot*, utilisé par l'instruction c.
4. *recommencer avec telle instruction*, utilisé par l'instruction f.

Nous avons donc là une liste d'instructions élémentaires que notre ordinateur doit savoir exécuter pour effectuer les calculs de notre programme.

1.3.4 Codage des instructions

Les instructions sont stockées dans la mémoire, comme les données et la mémoire ne contient que des nombres. faut donc choisir un *code* pour représenter les instructions avec des nombres,

Dans les ordinateurs communs, ces codes sont choisis avec un grand soin pour utiliser le moins de mémoire possible mais comme nous cherchons d'abord ici à avoir un ordinateur simple, nous allons choisir un code élémentaire : chaque instruction sera codée sur quatre mots mémoire consécutifs ; le premier mot indique quelle opération doit être effectuée ; les trois mots suivants contiennent la description des valeurs sur lesquelles l'opération doit être effectuée (on appelle cela les *opérandes*).

Détaillons le codage de l'instruction *additionner le contenu du mot mémoire 2805 et du mot mémoire 9591 et placer le résultat dans le mot mémoire 2904* : pour le codage de l'opération, on peut choisir *n'importe quelle valeur*, par exemple 4. L'instruction apparaitra dans la mémoire comme une suite de quatre mots qui contiendront les valeurs 4, 2805, 9591, 2904.

Codons l'instruction *diviser le contenu d'un mot par celui d'un autre mot, placer le résultat dans un troisième mot* de la même manière mais avec un code opération différent, par exmple 2. Ainsi les quatre mots 2, 2805, 9591, 2904 codent l'instruction : *diviser le contenu du mot d'adresse 2805 par celui du mot d'adresse 9591 et placer le résultat dans le mot d'adresse 2904*.

Pour l'instruction *placer une valeur dans un mot*, nous choisissons de nouveau un code opération (*arbitraire*), par exemple 1. Le deuxième mot contiendra la valeur, dans le troisième mot il y aura l'adresse du mot où la placer, le quatrième mot pourra contenir n'importe quoi. Par exemple, la suite de mots 1, 2, 3, 4 placera la valeur 2 dans le mot d'adresse 3.

La dernière instruction dont nous avons besoin est *recommencer avec telle instruction*. Codons l'opération avec la valeur 3 et plaçons dans le mot suivant l'adresse de l'instruction avec laquelle recommencer. Les deux mots qui suivent pourront contenir n'importe quelle valeur sans modifier le sens de l'instruction.

La table suivante résume notre codage des opérations. Les points d'interrogation servent à marquer les mots dont la valeur peut changer sans que la signification de l'instruction soit modifiée. Insistons encore une fois sur le fait que ce codage des opérations et de leurs opérandes est *arbitraire*.

opération	code opération	2ème mot	3ème mot	4ème mot
écrire le nombre x dans le mot adr_1	1	x	adr_1	?
écrire à adr_3 la somme des contenus de adr_1 et adr_2	4	adr_1	adr_2	adr_3
écrire à adr_3 le résultat de la division des contenus de adr_1 et adr_2	2	adr_1	adr_2	adr_3
continuer avec l'instruction qui se trouve à adr_1	3	adr_1	?	?

Exercice 1.3 : Si le mot mémoire d'adresse 4 contient la valeur 251, quel sera l'effet de l'instruction codée par 4, 4, 4, 4 ?

Exercice 1.4 : Quel sera l'effet de l'instruction codée par 2, 4, 4, 4 ?

Exercice 1.5 : Quel sera l'effet de l'instruction 1, 2, 4, 8 ?

1.3.5 Traduction du programme

Maintenant que nous avons choisi un codage pour les instructions élémentaires, nous pouvons encoder notre programme avec des valeurs à placer dans la mémoire. Le tableau suivant suppose que le programme est placé à partir du mot d'adresse (arbitraire) 2997. La première colonne contient l'adresse de chaque mot, la deuxième indique ce qu'il contient et la troisième est un commentaire pour faciliter la lecture.

adresse	contenu	signification
2997 :	1	; ;Placer
2998 :	1	; ;le nombre 1
2999 :	1000	; ;à l'adresse 1000
3000 :	0	
3001 :	1	; ; Placer
3002 :	2	; ;le nombre 2
3003 :	1020	; ;à l'adresse 1020
3004 :	0	
3005 :	2	; ; Diviser
3006 :	1020	; ;le contenu de l'adresse 1020
3007 :	1000	; ;par le contenu de l'adresse 1000
3008 :	1010	; ;et placer le résultat à l'adresse 1010 (n/r)
3009 :	4	; ; Ajouter
3010 :	1020	; ;le contenu de l'adresse 1020
3011 :	1010	; ;au contenu de l'adresse 1010
3012 :	1010	; ;et placer le résultat à l'adresse 1010 (r+n/r)
3013 :	2	; ;Diviser
3014 :	1010	; ;le contenu de l'adresse 1010
3015 :	1020	; ;par le nombre 2
3016 :	1000	; ;et placer le résultat à l'adresse 1000 r=(r+n/r)/2
3017 :	3	; ;continuer avec une autre instruction :
3018 :	3005	; ;la prochaine instruction est à l'adresse 3005
3019 :	0	
3020 :	0	

1.3.6 L'unité de controle, le PC

L'unité de contrôle possède un registre (mémoire d'un mot), qui contient l'adresse de la prochaine instruction à exécuter. (Ce mot mémoire n'est placé dans l'unité de contrôle que parce que cela permet d'y accéder très vite ; on peut parfaitement imaginer d'avoir un mot dans la mémoire ordinaire, à une adresse fixée, qui joue le rôle de PC..) Ce registre est nommé PC pour *Program Counter* (compteur ordinal en français). L'unité de contrôle effectue en boucle un programme cablé (un *automate*), dont les instructions sont les suivantes :

1. charger le contenu du mot dont l'adresse est dans PC,
2. ajouter 1 au contenu de PC.

3. décoder l'instruction, charger les opérandes si nécessaire, faire effectuer l'opération par l'unité de calcul et stocker le résultat à l'adresse spécifiée dans l'instruction,

Le décodage de l'instruction et le chargement des opérandes, le stockage des instructions est une suite d'opérations encore plus élémentaires (on parle de micro-instructions). Elle est réalisée avec un automate. Pour les trois premiers codes opérations, il pourrait s'agir d'effectuer les choses suivantes :

1. Si le code de l'opération vaut 1
2.   lire le mot dont l'adresse est dans PC,
3.   et ajouter 1 à PC
4.   lire le mot dont l'adresse est dans PC,
5.   et ajouter 2 à PC
6.   placer la valeur lue en *2* au mot dont on a lu l'adresse en 4.
7. Si le code de l'opération vaut 2
8.   lire le mot dont l'adresse est dans PC
9.   et ajouter 1 à PC
10.   lire le mot dont l'adresse est dans PC
11.   et ajouter 1 à PC
12.   placer les valeurs lues en 8 et 10 dans le diviseur
13.   lire le mot dont l'adresse est dans PC
14.   et ajouter 1 à PC
15.   placer la sortie du diviseur dans à l'adresse lue en 13.
16. Si le code de l'opération vaut 3
17.   lire le mot dont l'adresse est dans PC
18.   et placer sa valeur dans PC ⋮

Pour commencer l'exécution de notre programme, il suffit maintenant de placer 2997 (l'adresse de la première instruction) dans le registre PC.

Notez que l'instruction « continuer avec telle instruction » (qu'on appelle aussi un *branchement inconditionnel*) consiste simplement à modifier le contenu du PC ; de cette manière, l'adresse de l'instruction suivante sera modifiée.

Exercice 1.6 : La mémoire contient les valeurs suivantes à l'adresse 2000. Que codent ces valeurs si ce sont des instructions ? Quel sera leur effet ?

adresse	contenu	signification
2000 :	3	
2001 :	2000	
2002 :	2001	
2003 :	2002	

1.4 La brique de base : le transistor

Pour construire notre ordinateur, nous disposons uniquement de transistors. Un transistor est un interrupteur électronique qui commute très rapidement. Le transistor présente trois connexions externes : la source, le drain et la grille. Il fonctionne comme un interrupteur entre la source et le drain, controlé par la grille.

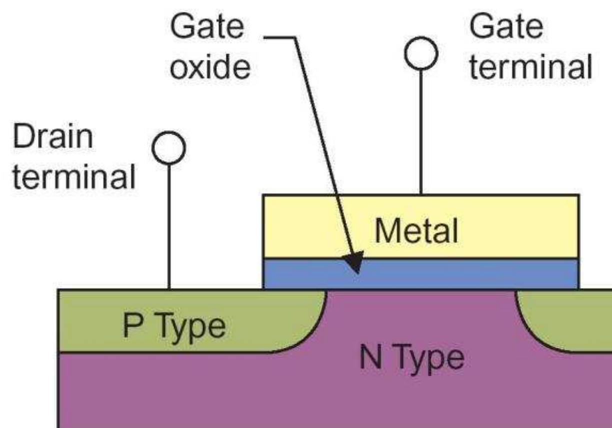


FIGURE 1.3 – La structure d'un des types de transistor

Pour un niveau de tension sur la grille, le transistor est assimilable à un interrupteur ouvert : La tension mesurée sur le drain est celle de la source.

Pour un autre niveau de tension sur la grille, le transistor se comporte comme un interrupteur fermé et la tension mesurée en sortie est celle de la source.

La figure 1.3 montre la structure d'un transistor : le substrat sur lequel est réalisé le transistor est colorié en violet ; la source et le drain sont coloriés en

vert. Suivant la tension présente sur la grille, qui porte l'étiquette *Metal*, le pont en oxyde de silicium, colorié en bleu se comporte soit comme un isolant, soit comme un conducteur.

Le transistor est la brique de base pour construire nos ordinateurs ; nous allons voir dans la suite de ce cours comment ils sont utilisés pour réaliser l'unité de calcul, la mémoire, l'automate de l'unité de contrôle.

1.5 Résumé

Notre ordinateur est composé de trois parties : mémoire, processeur, entrées-sorties. Le processeur possède un jeu d'instructions. Une instruction est écrite sous la forme d'une suite de nombres. Tout programme doit être écrit en utilisant le jeu d'instructions du processeur pour pouvoir être exécuté par ce dernier. Le programme lors de son exécution est stocké en mémoire et le processeur exécute les instructions les unes après les autres.

1.6 Exercices

Exercice 1.7 : Que signifient les instructions suivantes si c'est un programme pour un ordinateur ayant le jeu d'instruction choisi dans ce chapitre ? Quelles seront les valeurs successives que contiendra le mot mémoire 3000 quand le programme suivant sera exécuté (attention, la valeur changera au cours de l'exécution du programme) ?

1997 : 1
1998 : 1
1999 : 2999
2000 : 3000
2001 : 1
2002 : 0
2003 : 3000
2004 : 2004
2005 : 1
2006 : 1
2007 : 3001
2008 : 3001
2009 : 4
2010 : 3000
2011 : 3001
2012 : 3002
2013 : 2
2014 : 3001
2015 : 2999
2016 : 3000
2017 : 2
2018 : 3002
2019 : 2999
2020 : 3001
2021 : 3
2022 : 2009

Chapitre 2

La représentation des nombres entiers positifs

Le chapitre expose la manière de représenter les nombres entiers positifs et de passer d'une représentation à une autre. Il commence par un exposé concis, destiné à celui qui se sent à l'aise avec les mathématiques, puis reprend la matière avec des détails pour les autres.

2.1 Pour les matheux

Un nombre a qu'on représente en base b avec n chiffres $a_{n-1} \cdots a_0$ a pour valeur $\sum_{0 \leq i < n} a_i b^i$. Pour obtenir sa représentation en base 10, il suffit de développer la formule et d'effectuer les calculs. Pour passer de la représentation décimale à la représentation en base b : on obtient le chiffre de droite avec une division euclidienne ; si $a = r + nb$, puisque $\sum_{0 \leq i < n} a_i b^i = a_0 + b \sum_{1 \leq i < n} a_i b^{i-1}$ le chiffre de droite est égal au reste r de la division ; on recommence l'opération sur n pour obtenir les autres chiffres.

2.2 Pour les programmeurs

Un nombre est représenté dans l'ordinateur d'une manière que nous n'avons en général pas besoin de connaître (le plus souvent en binaire).

Pour lire un nombre en base `base` représenté par une suite de chiffres, on peut utiliser le pseudo-code suivant :

```
lirnum(base){
  res = 0;
  pour chaque chiffre (de gauche à droite)
```

```
    res = res * base + val(chiffre)
  return res
}
```

La fonction `val` permet d'obtenir la valeur du nombre représenté par un chiffre.

Pour écrire un nombre `n` en base `base`, on peut utiliser le pseudo-code

```
ecrinum(n, base){
  si (n >= base)
    ecrinum(n / base)       // division entière
  ecrire(code(n % base)) // reste de la division
}
```

La fonction `code` donne le chiffre qui représente un nombre. Noter l'appel récursif qui permet d'écrire les chiffres les plus significatifs en premier.

2.3 Pour les autres

Le reste du chapitre reprend pas à pas ces descriptions concises de la méthode de représentation des nombres, à destination de ceux qui ne sont pas à l'aise avec le formalisme des mathématiques ou la programmation.

2.3.1 Qu'est-ce qu'un nombre ?

On croit souvent qu'un nombre, c'est une série de chiffres. *Ceci est faux.* Une suite de chiffres est *une des façons* de représenter un nombre mais il en existe d'autres. Par exemple, le nombre que nous représentons habituellement avec 12 peut aussi se représenter avec XII (en chiffres romains) ; il y a des nombres qu'on ne peut pas représenter avec une suite (finie) de chiffres, comme $\frac{1}{3}$ (il faut une virgule et un nombre infini de chiffres 3 pour le représenter comme une suite de chiffres). D'autres exemples de nombres communs qui ne se représentent pas bien avec des suites de chiffres sont $\sqrt{2}$ (le nombre qui donne 2 quand on le multiplie par lui-même) ou π. On peut discuter pour décider si $9 + 3$ ou $\sqrt{144}$ sont des représentations du nombre usuellement noté avec 12 ou des indications d'opérations.

Il est important pour nous informaticiens de comprendre que nous ne manipulons pas des nombres mais des *représentations* de nombres. C'est essentiel, particulièrement dans ce chapitre, où nous allons étudier comment passer de la représentation usuelle d'un nombre à d'autres représentations de ce même nombre : il ne faut pas s'imaginer que le nombre *est* sa représentation usuelle.

L'essentiel est donc de garder à l'esprit qu'un nombre est quelque chose de plus vaste que la manière de le représenter et qu'il existe plusieurs représentations pour un même nombre. On trouvera des considérations plus approfondies sur la nature des nombres dans le numéro spécial de la revue *La Recherche*

d'août 1999. En résumé, on peut dire que personne ne sait très bien ce qu'est un nombre (surtout pas les mathématiciens).

2.3.2 La notation usuelle : numération en base décimale

Nous sommes habitués à représenter nos nombres en base dix. Par exemple, nous représentons le nombre d'habitants de la France avec 60 000 000 (environ).

On dit que ces séquences de chiffres représentent le nombre *en base décimale* (on dit aussi *en base 10* ou simplement en *décimal*), parce qu'ils impliquent une série de multiplications et d'additions avec le nombre 10. Ainsi 365 représente le nombre qu'on obtient avec la séquence d'opérations $3 \times 100 + 6 \times 10 + 5$.

Cette manière de représenter les nombres nous est tellement familière qu'il faut faire un petit effort de pensée pour réaliser que nous en connaissons d'autres.

2.3.3 D'autres représentations usuelles

Une autre façon usuelle de représenter les nombres consiste à les représenter en *chiffres romains*. Ici la séquence de chiffres n'implique que des additions et des soustractions élémentaires, pas de multiplication. On utilise couramment :

chiffre romain	équivalent en base 10
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Les valeurs représentées par les chiffres sont à additionner quand il n'y en a pas de plus grandes à droite, à soustraire sinon. Ainsi IV représente le chiffre 4 dans la représentation décimale (on retire le 1 représenté par I au 5 représenté par V, alors que VI représente 6 $(5 + 1)$.

Les différentes représentations des nombres présentent des avantages et des inconvénients divers. Ainsi, la représentation en chiffres romains permet de faire des additions sans avoir besoin de se souvenir des tables d'additions ; en revanche, la méthode de multiplication que nous avons apprise à l'école primaire ne fonctionne pas.

Il y a d'autres représentations usuelles des nombres : par exemple on utilise la base 1 pour représenter (avec des bougies) un nombre d'années sur un gateau d'anniversaire. On utilise aussi la base 1 quand on compte sur ses doigts de la façon usuelle : le nombre de doigts levés (en Europe) représente un nombre.

2.3.4 Nommer les chiffres d'un nombre

J'ai besoin de donner un *nom* aux chiffres d'un nombre pour décrire les calculs à effectuer. Pour cela, je numérote les chiffres d'un nombre en commençant par la droite avec 0. Pour un nombre a qui se représente avec n chiffres, je vais nommer ses chiffres $a_{n-1} \cdots a_1 a_0$.

Par exemple pour le nombre 27483, j'ai $a_4 = 2$, $a_3 = 7$, $a_2 = 4$, $a_1 = 8$ et $a_0 = 3$.

2.3.5 La valeur d'un nombre décimal

Pour un nombre décimal (la représentation usuelle), on nous a appris à l'école primaire à décomposer le nombre en unités, dizaines, centaines, milliers etc. Par exemple le nombre 3141 se compose de trois milliers, une centaine, quatre dizaines et une unité.

On obtient ces blocs dans lesquels on décompose les nombres en multipliant la taille du bloc précédent par 10. Il s'agit des puissances successives de 10. (Y compris pour les unités : $10^0 = 1$; si vous avez besoin d'en être convaincu, notez qu'on a $10^{n-1} = 10^n/10$, par exemple $10^3 = 10^4/10$; donc $10^0 = 10^1/10 = 10/10 = 1$. C'est pareil pour les autres nombres : à peu près n'importe quoi à la puissance 0 vaut 1.)

Donc la décomposition du nombre 3141 peut aussi s'écrire $3 \times 10^3 + 1 \times 10^2 + 4 \times 10^1 + 1 \times 10^0$.

On peut faire la même décomposition pour n'importe quel nombre de n chiffres

$$\begin{aligned} a &= a_{n-1} a_{n-2} \cdots a_1 a_0 \\ a &= a_{n-1} 10^{n-1} + a_{n-2} 10^{n-2} + \cdots + a_1 10^1 + a_0 10^0 \end{aligned}$$

Les points de suspension permettent d'écrire la formule d'une façon générale, quel que soit le nombre de chiffres. On peut aussi utiliser une écriture plus compacte avec l'opérateur $\sum$ qui indique qu'il faut additionner des expressions. Avec cet opérateur, on peut réécrire la formule comme

$$a = a_{n-1} \cdots a_0 = \sum_{0 \leq i < n} a_i b^i$$

C'est pour pouvoir écrire cette formule que j'ai choisi de numéroter les chiffres à partir de 0.

Pour simplifier les calculs,on peut les regrouper :

$$\begin{aligned} a &= a_{n-1} 10^{n-1} + a_{n-2} 10^{n-2} + \cdots + a_1 10^1 + a_0 10^0 \\ a &= (\cdots((a_{n-1} \times 10 + a_{n-2}) \times 10 + \cdots) \times 10 + a_1) \times 10 + a_0 \end{aligned}$$

qu'on peut aussi écrire en échangeant les parties gauche et droite de chaque addition

$$a = a_0 + 10(a_1 + 10(\cdots + 10(a_{n-2} + 10a_{n-1})\cdots))$$

Ceci s'appelle la *forme de Horner*. Son principal avantage pour nous, c'est qu'il n'y a plus que des multiplications par 10 et des additions, au lieu de multiplications par 10, 100, 1000 etc.

2.3.6 Représentation des nombres en base constante

Au lieu du 10 de la notation décimale, on peut utiliser (à peu près) n'importe quel nombre.

Presque tous les ordinateurs utilisent la base 2 (le *binaire*), parce qu'il est bien adapté aux circuits électroniques avec lesquels on construit les ordinateurs. Pour éviter de manipuler les nombreux chiffres utilisés dans la représentation binaire, nous utilisons aussi fréquemment les bases 8 et 16 (on parle alors d'*octal* et d'*hexadécimal*), pour lequelles les conversions avec la représentation binaire sont particulièrement faciles.

Pour le binaire (la base 2), on n'a besoin que des chiffres 0 et 1 pour représenter tous les nombres. Pour l'octal (la base 8), les chiffres de 0 à 7 suffisent. Pour l'hexadécimal, on a besoin de chiffres pour noter toutes les valeurs de 0 à 15; pour les chiffres manquant, on utilise les lettres de l'alphabet : $a = 10$, $b = 11$, $c = 12$, $d = 13$, $e = 14$ et $f = 15$.

A partir de maintenant, je vais m'efforcer d'écrire (dans ce chapitre au moins) tous les nombres sous la forme x_b pour indiquer que le nombre x est représenté en base b. Par exemple le nombre de jours d'une année (non bisextile) est 365_{10} ou $16d_{16}$ ou 555_8 ou 101101101_2.

Les considérations de la section précédente sur les nombres représentés en notation décimale fonctionnent aussi pour tous les nombres dans n'importe quelle base b , simplement en remplaçant les 10 par b. On a donc, pour un nombre a représenté avec n chiffres en base b :

$$\begin{aligned}
a &= a_{n-1}\cdots a_1 a_0 && (2.1)\\
&= a_{n-1}b^{n-1} + \cdots + a_1 b^1 + a_0 b^0 && (2.2)\\
&= a_0 + b(a_1 + b(\cdots + b(a_{n-2} + b a_{n-1})\cdots)) && (2.3)\\
&= \sum_{0 \le i < n} a_i b^i && (2.4)
\end{aligned}$$

Les lignes 2.1 et 2.2 sont importantes, parce qu'elles servent de point d'appui pour trouver les calculs à effectuer lors des conversions.

Pour appliquer les formules générales sur le nombre (en décimal habituel) 314159_{10}, on sait que $b = 10$ et on donne un nom aux chiffres en les numérotant de la droite vers la gauche :

3	1	4	1	5	9
a_5	a_4	a_3	a_2	a_1	a_0

Il n'y a plus qu'a remplacer les valeurs dans la partie droite, en remplissant les points de suspension

$$\begin{aligned}(a_5a_4a_3a_2a_1a_0)_b &= a_5b^5 + a_4b^4 + a_3b^3 + a_2b^2 + a_1b^1 + a_0b^0 \\ &= a_0 + b(a_1 + b(a_2 + b(a_3 + b(a_4 + ba_5)))) \\ 314159_{10} &= 3 \times 10^5 + 1 \times 10^4 + 4 \times 10^3 + 1 \times 10^2 + 5 \times 10^1 + 9 \times 10^0 \\ &= 9 + 10(5 + 10(1 + 10(4 + 10(1 + 10 \times 3))))\end{aligned}$$

La suite du chapitre montre comment effectuer les conversions usuelles entre les nombres.

2.3.7 Conversions vers la base 10

A partir de n'importe quel nombre, on peut facilement calculer sa représentation en décimal en utilisant nos formules. Il suffit de prendre l'équation 2.1 ou 2.2 et d'effectuer les calculs.

Par exemple, pour représenter 4567_8 en décimal en utilisant 2.1, on commence par noter que $b = 8$, $a_3 = 4$, $a_2 = 5$, $a_1 = 6$ et $a_0 = 7$. On prend la formule

$$a = a_{n-1} \cdots a_1a_0 = a_{n-1}b^{n-1} + \cdots + a_1b^1 + a_0b^0$$

et on obtient

$$4567_8 = 4 \times 8^3 + 5 \times 8^2 + 6 \times 8^1 + 7 \times 8^0$$

On commence à effectuer les calculs :

$$4567_8 = 4 \times 512_{10} + 5 \times 64_{10} + 6 \times 8 + 7$$

puis on les termine :

$$4567_8 = 2048_{10} + 320_{10} + 48_{10} + 7 = 2423_{10}$$

Il est souvent plus facile de faire les calculs en utilisant la ligne 2.2 :

$$a = a_{n-1} \cdots a_1a_0 = a_0 + b(a_1 + b(\cdots + b(a_{n-2} + ba_{n-1}) \cdots))$$

Dans notre exemple, on obtient (en montrant tous les calculs intermédiaires) :

$$\begin{aligned}4567_8 &= 7 + 8 \times (6 + 8 \times (5 + 8 \times 4)) \\ &= 7 + 8 \times (6 + 8 \times (5 + 32_{10})) \\ &= 7 + 8 \times (6 + 8 \times 37_{10}) \\ &= 7 + 8 \times (6 + 296_{10}) \\ &= 7 + 8 \times 302_{10}) \\ &= 7 + 2416_{10}) \\ &= 2423_{10}\end{aligned}$$

Depuis la base 16, le processus est le même mais il faut en plus convertir les chiffres vers la base 10. Un exemple pour représenter fac_{16} en base 10, avec 2.2 :

$$\begin{aligned}
FAC_{16} &= C_{16} + 16_{10} \times (A_{16} + 16_{10} \times F_{16}) \\
&= 12_{10} + 16_{10} \times (10_{10} + 16_{10} \times 15_{10}) \\
&= 12_{10} + 16_{10} \times (10_{10} + 240_{10}) \\
&= 12_{10} + 16_{10} \times 250_{10} \\
&= 12_{10} + 4000_{10} \\
&= 4012_{10}
\end{aligned}$$

Le passage de la première à la deuxième ligne a correspondu au passage des chiffres hexadécimaux plus grands que 9 à leur représentation en base 10.

Le même calcul en utilisant 2.1

$$\begin{aligned}
FAC_{16} &= F_{16} \times (16_{10})^2 + A_{16} \times (16_{10})^1 + C_{16} \times (16_{10})^0 \\
&= 15_{10} \times (16_{10})^2 + 10_{10} \times (16_{10})^1 + 12_{16} \times (16_{10})^0 \\
&= 15_{10} \times 256_{10} + 10_{10} \times 16_{10} + 12_{10} \\
&= 3840_{10} + 160_{10} + 12_{10} \\
&= 4012_{10}
\end{aligned}$$

Pour un exemple avec une base non standard, partons de $(31415)_6$ vers le décimal :

$$\begin{aligned}
(31415)_6 &= (3141)_6 \times 6 + 5 \\
&= ((314)_6 \times 6 + 1) \times 6 + 5 \\
&= ((314)_6 \times 6 + 1) \times 6 + 5 \\
&= (((31)_6 \times 6 + 4) \times 6 + 1) \times 6 + 5 \\
&= (((3 \times 6 + 1) \times 6 + 4) \times 6 + 1) \times 6 + 5 \\
&= (((19)_{10} \times 6 + 4) \times 6 + 1) \times 6 + 5 \\
&= ((118)_{10} \times 6 + 1) \times 6 + 5 \\
&= (709)_{10} \times 6 + 5 \\
&= (4259)_{10}
\end{aligned}$$

Ici, j'ai montré la construction de la forme de Horner sur les premières lignes. Tous les calculs peuvent se faire de tête.

Pour un exemple avec une base étonnante, représentons $(172)_{-8}$ en décimal (rien ne nous oblige à utiliser une base positive) :

$$(172)_{-8} = 2 - 8 \times (7 - 8 \times 1))$$

$$
\begin{aligned}
&= 2 - 8 \times (7 - 8)) \\
&= 2 - 8 \times (-1)) \\
&= 2 + 8 \\
&= 10
\end{aligned}
$$

Le dernier exemple, pour représenter $(11011011)_2$ en décimal :

$$
\begin{aligned}
(11011011)_2 &= 1 + 2(1 + 2(0 + 2(1 + 2(1 + 2(0 + 2(1 + 2 \times 1)))))) \\
&= 1 + 2(1 + 2(0 + 2(1 + 2(1 + 2(0 + 2 \times 3))))) \\
&= 1 + 2(1 + 2(0 + 2(1 + 2(1 + 2(0 + 6))))) \\
&= 1 + 2(1 + 2(0 + 2(1 + 2(1 + 2 \times 6)))) \\
&= 1 + 2(1 + 2(0 + 2(1 + 2(1 + 12)))) \\
&= 1 + 2(1 + 2(0 + 2(1 + 2 \times 13))) \\
&= 1 + 2(1 + 2(0 + 2(1 + 26))) \\
&= 1 + 2(1 + 2(0 + 2 \times 27)) \\
&= 1 + 2(1 + 2(0 + 54)) \\
&= 1 + 2(1 + 2 \times 54) \\
&= 1 + 2(1 + 108) \\
&= 1 + 2 \times 109 \\
&= 1 + 218 \\
&= 219
\end{aligned}
$$

Ici, il y a tellement d'additions et de multiplications à effectuer qu'il est délicat de ne pas se tromper dans les calculs, même s'ils sont faciles. Nous verrons un peu plus loin une autre méthode plus facile, en passant par l'octal ou l'hexadécimal.

L'algorithme de conversion peut s'écrire comme dans la figure 2.1. En partant de la représentation de Horner, il revient à effectuer successivement chaque multiplication et chaque addition en partant de l'intérieur de l'expression. Pour être certain de l'avoir bien compris, il peut être intéressant de le traduire en programme.

```
valeur <- a_{n-1}
pour i allant de n-2 à 0 faire
  valeur <- valeur×b + a_i
```

FIGURE 2.1 – Algorithme de conversion de la base b vers la base 10

2.3.8 Conversion depuis la base dix

Pour convertir un nombre de la base dix vers une base quelconque, l'idée est de faire une division avec un résultat et un reste (on dit une division *euclidienne*)

```
 2009 | 8           251 | 8       31 | 8
-16   |-----       -24  |-----   -24  |---
      | 251             | 31          | 3
  40  |            11   |         7   |
 -40  |            -8   |
      |             3   |
   09 |
   -8 |
    1
```

FIGURE 2.2 – Les divisions effectuées pour transformer 2009_{10} en 3731_8.

pour extraire le chiffre le plus à droite, puis de recommencer avec le résultat tant qu'il est plus grand que la base. Cela fait sortir une expression de la forme de la partie droite de l'égalité 2.2.

Sur un exemple : pour représenter 681_{10} en octal, on le divise par 8 : $681_{10} = 1+8\times 85_{10}$. On divise maintenant 85_{10} par 8 : $85_{10} = 5+8\times 10_{10}$. On recommence encore une fois avec $10_{10} = 2 + 8 \times 1$. Quand on met tout ensemble :

$$\begin{aligned} 681_{10} &= 1 + 8 \times 85_{10} \\ &= 1 + 8 \times (5 + 8 \times 10_{10}) \\ &= 1 + 8 \times (5 + 8 \times (2 + 8 \times 1)) \end{aligned}$$

On a maintenant quelque chose qui est dans le même format que la partie droite de 2.2, avec $b = 8$. Donc dans la représentation en octal de 681_{10}, le chiffre a_0 vaut 1, le chiffre a_1 vaut 5, le chiffre a_2 vaut 2 et le chiffre a_3 vaut 1. En conséquence

$681_{10} = 1251_8$

Un autre exemple avec $(2009)_{10}$:

$$\begin{aligned} (2009)_{10} &= 1 + 8 \times 251 \\ &= 1 + 8 \times (3 + 8 \times 31) \\ &= 1 + 8 \times (3 + 8 \times (7 + 8 \times 3)) \\ &= 3731_8 \end{aligned}$$

On peut voir la série de divisions sur la figure 2.2 telle qu'on peut les poser sur sa feuille de brouillon.

Avec une base non standard : $(2003)_{10}$ en base 6 ?

$$\begin{aligned} (2003)_{10} &= 5 + 6 \times 333_{10} \\ &= 5 + 6 \times (3 + 6 \times 55_{10}) \\ &= 5 + 6 \times (3 + 6 \times (1 + 6 \times 9)) \\ &= 5 + 6 \times (3 + 6 \times (1 + 6 \times (3 + 6 \times 1))) \end{aligned}$$

2003 | 6	333 | 6	55 | 6	9 | 6	1 | 6	0 | 6
−18 | 333	−30 | 55	−54 | 9	−6 | 1	−0 | 0	−0 | 0
20	33	(1)	(3)	(1)	(0)
−18	−30				
23	(3)				
−18					
(5)					

FIGURE 2.3 – Conversion de $(2003)_{10}$ vers $(13135)_6$

$$
\begin{aligned}
&= (((1 \times 6 + 3) \times 6 + 1) \times 6 + 3) \times 6 + 5 \\
&= 13135_6
\end{aligned}
$$

Les divisions que nous avons effectuées sont représentées sur la figure 2.3. Les deux dernières divisions de la figure sont inutiles mais elles montrent que 013135 est égal à 13135 (on dit que les zéros à gauche ne *sont* pas *significatifs*). (L'avant dernière ligne permet de remettre les chiffres dans l'ordre dans lequel ils apparaissent dans le nombre.)

Du décimal vers la base 16, la procédure est la même mais il faut une étape supplémentaire pour effectuer la conversion des chiffres supérieurs à 9 (dans cet exemple, $(13)_{10} = (d)_{16}$) :

$$
\begin{aligned}
(2009)_{10} &= 9 + 16_{10} \times 125_{10} \\
&= 9 + 16_{10} \times (13_{10} + 16_{10} \times 7) \\
&= 9 + 16_{10} \times (d + 16_{10} \times 7) \\
&= (7 \times (16)_{10} + d) \times (16)_{10} + 9 \\
&= 7d9_{16}
\end{aligned}
$$

Avec une base négative : 2003_{10} en base -8 ?

$$
\begin{aligned}
2003_{10} &= 3 - 8 \times (-250) \\
&= 3 - 8 \times (6 - 8 \times 32) \\
&= 3 - 8 \times (6 - 8 \times (0 - 8 \times (-4))) \\
&= 3 - 8 \times (6 - 8 \times (0 - 8 \times (4 - 8 \times 1))) \\
&= 3 - 8 \times (6 - 8 \times (0 - 8 \times (4 - 8 \times 1))) \\
&= 14063_{-8}
\end{aligned}
$$

Ici j'ai du réfléchir en faisant les divisions pour obtenir des restes positifs. Par exemple à la deuxième ligne, je n'ai pas utilisé $-250 = 31 \times (-8) - 2$ parce que le reste de la division est négatif mais $-250 = 32 \times -8 + 6$, pour lequel le reste est positif.

Le nombre d'opérations pour convertir des nombres entre les bases 10 et 2 est tel que c'est une très mauvaise idée de faire le travail directement. Il est bien

```
i <- 0
tant que a ≥ b faire
  c_i <- a modulo b
  a <- a / b
  i <- i+1
fin tant que
c_i <- a
```

FIGURE 2.4 – Un algorithme de conversion de la base 10 vers la base b

plus pratique de convertir en base 8 ou en base 16, puis d'utiliser la méthode de conversion rapide entre ces bases et la base 2 que nous présentons dans la prochaine section.

L'algorithme de conversion d'un nombre a de la base 10 représenté sur n chiffres vers un nombre c dans une base b peut ainsi s'écrire comme dans la figure 2.4. Il donne les chiffres du résultat de droite à gauche.

2.4 Conversions entre binaire, octal et hexadécimal

Pour convertir entre les bases 2, 8 et 16, il existe une méthode particulière rapide et facile. Je la présente d'une façon concise puis avec quelques détails.

2.4.1 Pour les matheux

Etant donné deux bases b' et b, si on a $b' = b^p$ avec p entier le passage de la représentation d'un nombre entre les bases b et b' est rapide et facile.

Pour convertir de la base b vers la base b', on ajoute des 0 à gauche du nombre en base b de manière que $n \text{mod} p = 0$) puis on utilise l'équivalence

$$
\begin{aligned}
(a_{n-1} \cdots a_0)_b &= \sum_{0 \leq i < n} a_i b^i \\
&= \sum_{0 \leq i < n/p} (b^i \sum_{0 \leq j < p} a_{pi+j} b'^j)
\end{aligned}
$$

Chaque $\sum_{0 \leq j < p} a_{pi+j} b^j$ est un chiffre de la base b' noté c_i. On obtient ainsi l'égalité $(a_{n-1} \cdots a_0)_b = \sum_{0 \leq i < n/p} c_i b^i$ qui permet d'obtenir avec c_i, la représentation du nombre en base b'.

Ceci est très utilisé avec $b = 2$ et $b' = 8$ ou avec $b = 2$ et $b' = 16_{10}$

2.4.2 Pour les autres

Le principe de la méthode est d'exploiter le fait que chaque chiffre en octal est équivalent à trois chiffres binaires et chaque chiffre en hexadécimal à quatre chiffres binaires.

Pour passer de l'octal au binaire, on remplace chaque chiffre octal par trois chiffres binaires, en utilisant la table (à connaître par cœur)

octal	binaire
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Pour convertir du binaire vers l'octal, on regroupe les chiffres binaires par groupes de trois et on remplace chaque groupe par le chiffre octal équivalent. Pour la base 16, on procède de la même manière mais avec des groupes de quatre chiffres.

Ainsi, pour convertir 7654_8 en hexadécimal :

— on commence par fabriquer le binaire en remplaçant chaque chiffre octal par les trois chiffres binaires équivalents : $111\ 110\ 101\ 100_2$

— on regroupe les chiffres binaires par paquets de quatre en partant de la droite : $1111\ 1010\ 1100_2$

— on remplace chaque groupe de quatre chiffres par son équivalent hexadécimal : fac_{16}

Pour convertir $c0ffee_{16}$ en octal, on utilise la procédure inverse :

$$\begin{aligned} c0ffee_{16} &= 1100\ 0000\ 1111\ 1111\ 1110\ 1110_2 \\ &= 110\ 000\ 001\ 111\ 111\ 111\ 101\ 110_2 \\ &= 60177756_8 \end{aligned}$$

L'opération est facile et peu sujette aux erreurs de calcul. Quand on part du décimal pour obtenir du binaire, c'est bien plus facile de passer par l'octal que d'effectuer sans erreur la longue série de divisions par deux de la méthode de conversion générale.

2.4.3 Récapitulation des conversions entre les bases 10, 2, 8 et 16

Pour s'entraîner aux conversions usuelles, on peut partir d'un nombre écrit en base 10, le convertir en base 8, puis en base 2, puis en base 16, puis à

nouveau en base 10. (Cet exercice présente l'avantage d'être auto-correcteur : si on commet une erreur, on le verra aisément puisqu'on ne retombera pas sur le nombre de départ ; cela permet de chercher l'erreur et de la corriger.) Ainsi, en partant de $(1999)_{10}$, on calculera successivement :

$$
\begin{aligned}
(1999)_{10} &= (249)_{10} \times 8 + 7 \\
&= ((31)_{10} \times 8 + 1) \times 8 + 7 \\
&= ((3 \times 8 + 7) \times 8 + 1) \times 8 + 7 \\
&= (3717)_8 \\
(3717)_8 &= (011\ 111\ 001\ 111)_2 \\
&= (0111\ 1100\ 1111)_2 \\
(0111\ 1100\ 1111)_2 &= (7CF)_{16} \\
(7CF)_{16} &= (7 \times (16)_{10} + (c)_{16}) \times (16)_{10} + (f)_{16} \\
&= (7 \times (16)_{10} + (12)_{10}) \times (16)_{10} + (15)_{10} \\
&= (1999)_{10}
\end{aligned}
$$

On peut faire cet exercice en partant de n'importe laquelle des bases 2, 8, 10 et 16. Ainsi, en partant de la base 16 :

$$
\begin{aligned}
(cafe)_{16} &= (1100\ 1010\ 1111\ 1110)_2 \\
(1100101011111110)_2 &= (1\ 100\ 101\ 011\ 111\ 110)_2 \\
&= (145376)_8 \\
(145376)_8 &= ((((1 \times 8 + 4) \times 8 + 5) \times 8 + 3) \times 8 + 7) \times 8 + 6 \\
&= (((12)_{10} \times 8 + 5) \times 8 + 3) \times 8 + 7) \times 8 + 6 \\
&= (((101)_{10} \times 8 + 3) \times 8 + 7) \times 8 + 6 \\
&= ((811)_{10} \times 8 + 7) \times 8 + 6 \\
&= (6489)_{10} \times 8 + 6 \\
&= (51966)_{10} \\
(51966)_{10} &= (((12)_{10} \times (16)_{10} + (10)_{10}) \times (16)_{10} + (15)_{10}) \times (16)_{10} + (14)_{10} \\
&= ((c \times (16)_{10} + a) \times (16)_{10} + f) \times (16)_{10} + e \\
&= (cafe)_{16}
\end{aligned}
$$

Pour effectuer facilement les opérations de conversion entre les bases 2, 8, 10 et 16, il est utile de connaître par cœur la représentation des nombres entre 0 et 15_{10} dans ces bases, présentée dans la table 2.1

Il est aussi utile de connaître les premières puissances de 2 :

décimal	octal	hexadécimal	binaire
0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	10	8	1000
9	11	9	1001
10	12	A	1010
11	13	B	1011
12	14	C	1100
13	15	D	1101
14	16	E	1110
15	17	F	1111
16	20	10	10000

TABLE 2.1 – La représentation des nombres entre 0 et 16_{10} dans les bases 10, 8, 16 et 2. Il est utile de connaitre cette table par coeur.

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024

2.5 Résumé

Pour convertir d'une base quelconque vers la base 10, écrire la formule développée puis effectuer les opérations.

Pour convertir de la base 10 vers une base quelconque, effectuer la série de divisions par la base : les restes des divisions donnent les chiffres du nombre de la droite vers la gauche.

Entre le binaire et l'octal ou l'hexadécimal, utiliser la méthode rapide.

2.6 Exercice

Exercice 2.1 : Convertir 31 et 96 en chiffres romains et effectuer l'addition dans cette représentation. Convertir le résultat dans la base 10 usuelle et vérifier qu'il est égal à 127.

Exercice 2.2 : Décrire dans le détail, avec des phrases, la suite d'opérations qu'il faut effectuer pour additionner deux nombres écrits en chiffres romains (faute d'avoir tous les éléments pour écrire un programme).

Exercice 2.3 : (difficile pour le moment) Réaliser un programme qui imprime un nombre en chiffres romains. A défaut de programme, décrire avec des phrases les opérations à effectuer.

Exercice 2.4 : Écrire les nombres 5342_{10} et 4009_{10} sous leur forme de Horner.

Exercice 2.5 : Convertir 452_8 de la base 8 vers la base 10.

Exercice 2.6 : Convertir faf_{16} en décimal et en binaire en indiquant les étapes intermédiaires.

Exercice 2.7 : Convertir 0234_8 en hexadécimal.

Exercice 2.8 : Convertir 5926_{10} en octal, hexadécimal et binaire en indiquant les étapes intermédiaires.

Exercice 2.9 : Convertir le nombre $(1254)_7$ de la base 7 vers le décimal.

Exercice 2.10 : Convertir le nombre $(1254)_{-8}$ de la base -8 vers la base 10.

Exercice 2.11 : Convertir le nombre $(1254)_{-6}$ de la base -6 vers la base 10.

Exercice 2.12 : Convertir le nombre 3391_{10} en base -5. (Rappel : les restes des divisions par -5 doivent être positifs ; on ne doit pas utiliser des choses comme $-18 = 3 \times -5 - 3$ mais plutôt $-18 = 4 \times -5 + 2$).

Exercice 2.13 : (long et répétitif) Reprenez le programme écrit au chapitre précédent et écrivez les codes des opérations en binaire.

2.7 Supplément : le binaire facile

Une façon facile et amusante de compter en binaire consiste à compter sur ses doigts mais d'une manière inhabituelle. Chaque doigt joue le rôle d'un chiffre binaire, qui vaut 0 s'il est replié et 1 s'il est tendu.

Pour commencer, on peut écrire sur le bout de ses doigts la valeur du nombre en base 10 qu'il représente : 1 sur le pouce de la main gauche, 2 sur l'index, 4 sur le majeur, 8 sur l'annulaire et 16 sur l'auriculaire. Sur la main droite, on place 32 sur l'auriculaire, 64 sur l'annulaire, 128 sur le majeur, 256 sur l'index et 512 sur le pouce. Avec un peu d'exercice, on se souvient facilement de ces valeurs et il n'est plus nécessaire de se barbouiller les doigts.

Un exercice facile consiste à énumérer les nombres : tous les doigts repliés

(= 00000 00000_2) vaut 0. Le pouce gauche levé seul (= 00000 00001_2) vaut 1, l'index gauche seul (= 00000 00010_2) vaut 2, ce qui est facile à voir puisque c'est écrit dessus. Le pouce et l'index (= 00000 00011_2) vaut 3, puisqu'on additione le "1" marqué sur le pouce et le "2" inscrit sur l'index. Avec le majeur gauche seul (= 00000 00100), on a 4, puis 5 en levant aussi le pouce, 6 en levant l'index et 7 si on les lève tous les deux. Avec cette méthode, on peut compter jusqu'à 1023 sur ces dix doigts.

Une autre forme d'exercice consiste à choisir un nombre et à chercher les doigts à lever pour le représenter. Prenons le nombre 27_{10} par exemple : si on lève n'importe quel doigt de la main droite, on obtient une valeur trop grande : la main droite est donc complètement fermée. Si on lève l'auriculaire gauche, on a la valeur 16 et il reste 11 à représenter. En levant aussi l'annulaire, on ajoute 8 et il reste 3 à représenter. On ne lève donc pas le majeur puisqu'il vaut 4 mais on lève l'index et le pouce qui valent 2 et 1 : il ne reste plus rien à compter. Donc le nombre 27_{10} se représente sur les doigts avec la main droite complètement fermée et sur la main gauche, seul le majeur replié : cela se note 11011 si on utilise des chiffres.

2.8 Supplément : la commande `gnome-calculator`

Quand on utilise le système Linux avec la gestion de fenêtre installée, il y a un programme `gnome-calculator` qui permet d'explorer d'une manière interactive la représentation des nombres dans des bases différentes de 10.

On lance le programme en passant par les menus (si on utilise Gnome) ou bien entrant dans un terminal la commande `gnome-calculator` (si on utilise un autre système de fenêtrage, par exemple Kde). Cela fait apparaître une fenêtre qui ressemble à une calculette, qu'on peut utiliser pour effectuer les calculs usuels.

Dans la barre du haut de la fenêtre, cliquer sur `View` pour faire sortir un menu ; choisir `Programming` : cela modifie l'affichage des nombres et les fonctions disponibles.

Dans ce mode, la représentation des nombres en binaire est toujours disponible pour les nombres entiers positifs ; leurs valeurs sont toujours affichées sous la forme de 64 bits ; on peut modifier ces valeurs en cliquant dessus.

On peut faire des conversions entre binaire, octal, décimal et hexadécimal en sélectionnant `Bin`, `Oct`, `Dec` ou `Hex`.

De nombreuses autres opérations sont possibles.

Chapitre 3

Opérations en binaire

Nous montrons ici comment on peut effectuer des opérations directement en binaire. Ceci nous sera nécessaire dans la suite, quand nous aborderons la représentation des nombres signés.

Chaque chiffre d'un nombre écrit en base 2 est appelé bit. (Le mot bit vient de *binary digit*. Le terme consacré en français est celui de e.b., comme dans *éléments binaires* mais il est peu employé.) Le bit de droite d'un nombre est appelé *bit de poids faible*, celui de gauche le *bit de poids fort*.

3.1 Addition en base 2

Quelle que soit la base dans laquelle on représente des nombres, on peut continuer à poser les opérations comme nous l'avons appris à l'école primaire. Celà est vrai aussi en base deux. Cependant, il faut changer les tables d'addition et de multiplication.

a_i	$+$	b_i	$=$	c_i
0	+	0	=	0
0	+	1	=	1
1	+	0	=	1
1	+	1	=	10

TABLE 3.1 – Table d'addition pour la base 2.

La table d'addition de la base 2 est présentée dans la table 3.1. Comme on n'a que deux chiffres, elle ne comporte que quatre entrées ! De plus, les trois premières lignes sont identiques à celles de la base dix. La seule petite difficulté provient de la dernière ligne : puisqu'on compte en base deux, le résultat de $1 + 1$ est égal à 10_2, qui est la manière de représenter la valeur 2_{10} en binaire.

Avec cette table d'addition, on peut opérer une addition avec les règles usuelles que nous employons pour la base 10. Par exemple, additionnons 38_{10} et 20_{10}.

On commence par les convertir en base 2, en passant par la base 8.

$$38_{10} = 4 \times 8 + 6 = 46_8 = 100\,110$$
$$20_{10} = 2 \times 8 + 4 = 24_8 = 010\,100$$

On additionne ensuite, colonne par colonne. La retenue, quand elle existe, est toujours égale à 1 : nous la notons dans l'opération avec un r :

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

Finalement, il ne reste plus qu'à convertir le résultat en base dix pour vérifier qu'on n'a pas fait d'erreur de calcul : $111\,010_2 = 72_8 = 7 \times 8 + 2 = 58_{10}$.

Sur certaines colonnes, on peut avoir à additionner 1 et 1 et une retenue en plus. Il est donc pratique d'ajouter dans la table d'addition une ligne pour tenir compte de ce cas où il faut additionner trois fois 1 ; c'est ce que nous faisons dans la table 3.2.

r_i	+	a_i	+	b_i	=	c_i
		0	+	0	=	0
		0	+	1	=	1
		1	+	0	=	1
		1	+	1	=	10
1	+	1	+	1	=	11

TABLE 3.2 – Table d'addition pour la base 2 complétée.

Ceci nous permet d'effectuer des additions plus difficiles, par exemple $85_{10} + 119_{10} = 125_8 + 167_8$:

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

On obtient bien : $11\,001\,100 = 314_8 = 204_{10}$.

3.2 La multiplication en base 2

Nous commençons par examiner une méthode qui permet, en base 10, d'effectuer des multiplications sans avoir besoin des tables de multiplication. Nous montrons ensuite comment elle conduit directement à une méthode simple et facile pour effectuer les multiplications en base 2.

3.2.1 La multiplication en base 10 sans table

Supposons que nous souhaitions calculer 27×43 mais que nous ne nous connaissons pas les tables de multiplication. Une méthode consiste à partir d'une ligne qui contient 1 et 43, puis d'additionner le contenu de chaque ligne à lui-même, jusqu'à obtenir sur la colonne de gauche quelque chose de plus grand que le second multiplicande :

1	43
2	86
4	172
8	344
16	688
32	

On s'arrête là puisque 32 est plus grand que 27. On remonte ensuite en marquant les valeurs à additionner dans la colonne de gauche pour obtenir 27 ; on commence avec $27 = 16 +$ quelque chose, puis $27 = 16 + 8 +$ quelque chose mais $16+8+4$ est trop grand, puis $27 = 16+8+2+$quelque chose et finalement $27 = 16 + 8 + 2 + 1$.

Dans le tableau de départ, on ne conserve que les lignes qui nous servent à obtenir 27 (sur le papier, on barre les autres) ; on fait la somme de la colonne de droite et on obtient le résultat recherché :

1	43
2	86
8	344
16	688
	1161

L'explication est simple : on s'est contenté dans cette procédure d'utiliser la suite d'égalités :

$$\begin{aligned} 27 \times 43 &= (1 + 2 + 8 + 16) \times 43 \\ &= 1 \times 43 + 2 \times 43 + 8 \times 43 + 16 \times 43 \\ &= 43 + 86 + 344 + 688 \\ &= 1\,161 \end{aligned}$$

L'intérêt de la chose, c'est qu'on a effectué tout le travail seulement avec des additions.

Il est à noter pour la suite que le choix des lignes à additionner est d'une certaine façon une conversion vers la base 2 : on a

$$\begin{aligned} 27_{10} &= (1 + 2 + 8 + 16_{10}) \\ &= (((1 \times 2 + 1) + 0) \times 2 + 1) \times 2 + 1 \\ &= 11011_2 \end{aligned}$$

3.2.2 Le décalage à gauche et à droite

Quand on rajoute un zéro à droite d'un nombre, on le multiplie par sa base. Cette opération est familière en base 10 : ajouter un 0 à droite d'un nombre représenté en base 10 revient à le multiplier par 10.

On appelle cette opération un *décalage à gauche*. Ceci est vrai dans n'importe quelle base constante. Puisque le nombre $a_n \cdots a_0 = \sum_{0 \leq i \leq n} a_i b^i$, on a

$$\begin{aligned} b \times (a_n \cdots a_0) &= b \times \sum_{0 \leq i \leq n} a_i b^i \\ &= \sum_{0 \leq i \leq n} a_i b^{i+1} \\ &= a_n \cdots a_0 0 \end{aligned}$$

Ce décalage à gauche permet en binaire d'effectuer facilement des multiplications par 2.

Inversement et pour la même raison un décalage à droite consiste à supprimer le chiffre de droite d'un nombre ; on peut ajouter un 0 à gauche, dans les chiffres de poids forts, si on souhaite conserver le même nombre de chiffres. Le résultat de cette opération est une division par la base.

3.2.3 Un algorithme de multiplication en binaire

On peut appliquer en base 2 la même méthode que celle qu'on a employée en base 10 mais les choses sont plus faciles puisqu'on a déja décomposé le nombre de gauche en base 2 et que les multiplications par deux sont en fait simplement des décalages. Pour multiplier deux nombres a et b, l'algorithme est simplement

```
r <- 0
tant que a est différent de 0
   si le bit de droite de a vaut 1
     r ← r + b
   décaler a à droite
   décaler b à gauche
```

On peut faire tourner ce programme à la main, en écrivant les valeurs successives des variables ; par exemple, si on reprend les valeurs précédentes et qu'on multiplie entre eux 27 et 43 :

a	b	r	commentaire
11011	101011	0	départ
		101011	r ← r + b
01101	1010110		décaler a et b
		10000001	r ← r + b
00110	10101100		décaler a et b
			on n'ajoute pas b à r
00011	101011000		décaler a et b
		111011001	r ← r + b
00001	1010110000		décaler a et b
		10010001001	r ← r + b
00000	1010110000		décaler a et b

On justifiera cet algorithme dans le paragraphe suivant.

a_i	$\times$	b_i	$=$	c_i
0	$\times$	0	$=$	0
0	$\times$	1	$=$	0
1	$\times$	0	$=$	0
1	$\times$	1	$=$	1

TABLE 3.3 – Table de multiplication pour la base 2

3.2.4 Poser la multiplication en base 2

La table de multiplication en base 2 est particulièrement facile à retenir. Elle est présentée dans la table 3.3. Puisqu'on est en binaire, elle ne contient que les multiplications par 0 et par 1 !

On peut poser les multiplications en base 2 comme on le fait en base 10, à condition d'utiliser cette table. Par exemple, toujours avec notre exemple :

```
     101011
    x 11011
 ----------
     101011
    101011.
   000000..
  101011...
 101011....
 ----------
10010001001
```

Il est amusant de constater que cette méthode effectue les mêmes opérations que l'algorithme de la section précédente mais dans un ordre différent : pour chaque ligne du résultat intermédiaire, l'ajout du point correspond au décalage à gauche, et ici nous faisons toutes les additions à la fin au lieu de les effectuer pour chaque résultat intermédiaire.

3.3 Résumé

Les tables d'addition et de multiplication en binaire sont faciles à retenir. On peut les utiliser pour poser des additions en binaire comme on le fait en base 10.

3.4 Exercices

Exercice 3.1 : (exercice type) Convertir 183_{10} et 73_{10} en binaire, les additionner dans cette représentation. Convertir le résultat en décimal pour vérifier

qu'on n'a pas fait d'erreur. **Exercice** 3.2 : Multiplier 528_{10} et 321_{10} avec l'algorithme de multiplication sans table.

Exercice 3.3 : (comparer avec le précédent) Convertir 528_{10} et 321_{10} en binaire et faire tourner à la main l'algorithme de multiplication.

Exercice 3.4 : (comparer avec les deux précédents) Convertir 528_{10} et 321_{10} en binaire, poser et effectuer la multiplication en binaire.

Exercice 3.5 : Établir les tables d'addition et de multiplication en base 5, refaire l'exercice 1 en base 5 au lieu de la base 2.

Exercice 3.6 : La base -2 n'utilise que les chiffres 0 et 1. (facile) Traduire en décimal les seize valeurs entre 0000_{-2} et 1111_{-2}. (plus difficile) Quelle est la table d'addition de la base -2 ?

Supplément

Dans le programme de gnome présenté au chapitre précédent, on peut effectuer des décalages avec les boutons étiquetés < (à gauche) et > (à droite).

Chapitre 4

Représentation des nombres signés

Dans la partie précédente, nous avons examiné comment passer de la représentation d'un nombre d'une base dans une autre. Nous nous sommes particulièrement intéressés à la représentation utilisée dans les ordinateurs, la base 2. Ici, nous allons examiner comment représenter les nombres négatifs dans cette base.

Dans la plupart des cas, nous allons devoir spécifier quel est le nombre de bits utilisé pour représenter un nombre. Nous commençons par présenter brièvement des systèmes de représentation des nombres avec un signe que nous n'utiliserons que plus loin, puis nous entrerons dans le détail de la notation *en complément à 2* qui est la plus employée pour les nombres entiers.

4.1 Le bit de signe, la notation en excédent

Nous savons que toute information est codée dans un ordinateur avec un 1 ou un 0. Aucun autre symbole ne peut être utilisé. Jusqu'alors nous avons représenté les nombres en supposant qu'ils étaient positifs. Pour les nombres avec un signe, deux systèmes possibles sont la définition d'un bit de signe et la notation en excédent, qui sont toutes les deux employées pour la représentation des nombres en virgule flottante, que nous verrons plus loin.

Attention, la représentation la plus courante pour les nombres entiers utilise une autre méthode, le complément à deux, qui est présentée dans la section suivante.

4.1.1 Le bit de signe

Le principe est d'utiliser un bit pour représenter le signe du nombre. Par exemple, dans un nombre sur 8 bits $b_7 \cdots b_0$, on a la valeur absolue du nombre sur les bits $b_6 \cdots b_0$ et le bit b_7 vaut 0 pour un nombre positif et 1 pour un bit négatif.

Ainsi sur 8 bits, on représentera $+53$ avec 0011 0101 et -53 avec 1011 0101. Dans les deux cas, les sept bits de poids faibles codent, avec 011 0101, la valeur absolue 53 et le bit de signe vaut 0 pour le nombre positif et 1 pour le nombre négatif.

L'utilisation du bit de signe présente des inconvénients : d'une part, il y a deux représentations distinctes pour une même valeur ; sur huit bits, 0000 0000 et 1000 0000 valent tous les deux 0. D'autre part, l'addition et la soustraction sont un peu compliquées : pour additionner deux nombres, il faut additionner les deux valeurs absolues si les bits de signe sont identiques. S'ils sont différents il faut soustraire la plus petite valeur absolue de la plus grande. La comparaison de deux nombres est aussi un peu compliquée, puisqu'il faut là aussi traiter le bit de signe comme un cas particulier.

4.1.2 La représentation en excédent

Le principe de la représentation en excédent E est de représenter le nombre $-E + n$ en stockant la valeur n. Ainsi, quand on stocke un 0, on représente en fait $-E$. Sur 8 bits en excédent 127 on aura les valeurs suivantes :

valeur binaire	valeur décimale	valeur représentée
0000 0000	0	$-127 + 0 = -127$
0000 0001	1	$-127 + 1 = -126$
0111 1101	125	$-127 + 125 = -2$
0111 1110	126	$-127 + 126 = -1$
0111 1111	127	$-127 + 127 = 0$
1000 0000	128	$-127 + 128 = 1$
1000 0001	129	$-127 + 129 = 2$
1111 1111	255	$-127 + 255 = 128$

Nous pouvons remarquer qu'en choisissant correctement l'excédent, le bit de poids fort permet de connaître immédiatement le signe du nombre.

L'addition des nombres en excédent est elle aussi plus compliquée que celle de deux nombres positifs ordinaires (voir exercice).

Le bit de signe et la représentation en excédent sont utilisées pour les nombres représentés avec une virgule flottante, que nous examinerons au prochain chapître.

4.2 Représentation des nombres signés en complément à 2

La représentation la plus courante pour les entiers dans l'ordinateur est la représentation dite en *complément à 2*, qui présente deux avantages : d'une part on peut additionner des nombres en complément à deux avec les règles usuelles de l'addition, sans se soucier de leur signe : le résultat de l'addition aura le bon signe ; d'autre part, chaque nombre qu'on peut représenter ne possède qu'une représentation unique.

Il y a deux manières de considérer les nombres en complément à 2 : on peut comprendre sur quelles bases arithmétiques ils sont construits et on peut maitriser les techniques qui permettent de calculer l'opposé d'un nombre.

Attention, les opérations en complément à deux imposent d'effectuer les calculs sur un nombre de bits bien défini.

4.2.1 Les bases du complément à deux

L'idée de la notation en complément à deux est d'utiliser le bit de poids fort pour indiquer la partie négative du nombre représenté. Dans un nombre représenté par les bits $a_{n-1}a_{n-2}\cdots a_0$, tous les bits de poids faibles, de a_{n-2} jusqu'à a_0 comptent pour la partie positive : ils se traduisent comme d'usage par la somme des puissances de 2 qui correspondent à leurs indices ($\sum_{n-1>i\geq 0} a_i 2^i$) ; le bit de poids le plus fort, a_{n-1}, indique la partie négative ; il participe à la valeur du nombre pour $-a_{n-1}2^{n-1}$.

Le bit de poids le plus fort (le bit a_{n-1}) joue le rôle d'un bit de signe. Il va de soi qu'un nombre dans lequel il est égal à 0 n'a pas de partie négative : c'est donc un nombre positif ou nul. Quand ce bit est égal à 1, la partie négative vaut 2^{n-1} et sera toujours plus grande que la partie positive qui vaut au plus $\sum_{0\leq i<n-1} 2^i = 2^{n-1} - 1$ si tous les bits sont à 1. On n'a donc pas besoin de calculer la valeur d'un nombre pour savoir s'il est positif ou négatif : le bit de gauche joue le rôle d'un bit de signe.

Quel est le plus grand nombre positif représentable sur n bits ? Ce sera un nombre dont la partie négative sera nulle et dont la partie positive sera la plus grande possible. Il aura donc un bit à 0 à gauche et tous les autres bits à 1 ; par exemple sur 8 bits, ce sera 0111 1111. Sa valeur sera $2^{n-1} - 1$, ici 127.

Quel est le plus petit nombre négatif représentable sur n bits ? Ce sera un nombre avec une partie négative différente de 0 et une partie positive nulle. On le représentera avec un bit à 1 à gauche suivi de $n-1$ bits à 0 ; par exemple, sur 8 bits, ce sera 1000 0000. Sa valeur sera -2^{n-1}, ici -128.

Notez qu'on peut représenter un nombre négatif de plus que de nombres strictement positifs (c'est à dire supérieurs à 0) ; c'est un (léger) inconvénient

de la représentation en complément à 2.

Comment représenter 0 ? Bien sur, avec seulement des bits à 0. Notez qu'en complément à 2, le bit de signe de 0 est égal à 0, comme pour un nombre strictement positif.

Comment représenter -1 sur n bits ? On va représenter le nombre décimal -1 avec une partie négative différente de 0 (et donc égale à -2^{n-1} ; la partie positive devra donc valoir $2^{n-1} - 1$, qui est égal à la somme de toutes les puissances de 2 de 0 à $n - 2$; on l'obtient avec rien que des bits à 1. Donc, le nombre -1 se représente avec un mot dont tous les bits sont à 1.

On peut partir des propriétés de la représentation pour convertir entre la représentation en complément à 2 et la représentation usuelle en décimal avec signe mais ce n'est pas pratique du tout. La section suivante contient une méthode bien plus aisée. Voici néammoins deux exemples d'une manière de procéder.

Représenter -84 en complément à 2 sur 8 bits. Le nombre est négatif, il faudra donc un bit de poids fort à 1 qui codera pour $-(2^7) = -128$. Il faut lui ajouter $128 - 84 = 44$ pour obtenir -84 ; $44_{10} = 5 \times 8 + 4$ se représente en base 2 avec 101 100. Le résultat est donc 1010 1100

Calculer la valeur en décimal du nombre représenté en complément à 2 avec 1100 0011. Le nombre se compose d'une partie négative qui vaut -128_{10} et d'une partie positive $1\,000\,011_2 = 103_8 = 67_{10}$. Sa valeur en décimal avec signe est donc $-128_{10} + 67_{10} = -61_{10}$.

Encore une fois, les méthode de conversion de la section suivante sont plus faciles à mettre en oeuvre.

4.2.2 Calculer l'opposé d'un nombre en complément à 2

Il est facile de calculer l'opposé d'un nombre en complément à 2 en effectuant deux opérations successives : on bascule tous les bits et on ajoute 1.

Le basculement de tous les bits consiste à remplacer chaque bit à 0 de la chaîne de bits d'origine par un bit à 1 dans le résultat et chaque bit à 1 de la chaîne de bits d'origine par un bit à 0. Cela s'appelle le *complément à 1* d'un nombre. Nous le noterons $C_1(a)$. (On peut utiliser le complément à 1 pour représenter les nombres négatifs mais nous n'en parlerons pas ici.)

Ajouter 1 au complément à 1 de la représentation d'un nombre permet d'obtenir son *complément à 2*. Nous le noterons $C_2(a)$.

Pour convertir un nombre négatif du décimal vers le complément à 2 On calcule la représentation binaire de sa valeur absolue (le nombre sans le

signe), puis on calcule le complément à 2 de ce nombre.

Avec un exemple sur 8 bits : comment représenter -84_{10} en complément à 2 ? On calcule sa représentation en binaire en passant par l'octal

$$84_{10} = (1 \times 8 + 2) \times 8 + 4 = 124_8 = 1\,010\,100_2$$

Le nombre se représente donc sur 8 bits avec 01 010 100. (*Attention à ne pas oublier d'ajouter autant de bits à 0 à gauche que nécessaire pour avoir une représentation sur 8 bits.*) On calcule son complément à 1 :

$$C_1(01\,010\,100) = 10\,101\,011$$

Finalement on ajoute 1 pour obtenir son complément à 2

$$C_2(01\,010\,100) = C_1(01\,010\,100) + 1 = 10\,101\,011 + 1 = 10\,101\,100$$

Cette séquence d'opérations se pose aisément dans la table suivante :

$84_{10} =$	$122_8 =$	`01 010 100`	
		`10 101 011`	Complément à 1
		`+1`	
$-84_{10} =$		`10 101 100`	Complément à 2

Pour convertir un nombre négatif du complément à 2 vers le décimal On calcule son complément à 2 (on obtient sa valeur absolue), on la convertit en décimal et on rajoute un signe $-$ devant le nombre obtenu.

Avec un exemple sur 8 bits : quelle est la représentation en décimal d'un nombre a représenté en complément à 2 par 11 000 011 ? On calcule le complément à 2.

a	`11 000 011`
$C_1(a)$	`00 111 100`
	`+ 1`
$C_2(a)$	`00 111 101`

On convertit le résultat : $111\,101 = 75_8 = 61_{10}$

On ajoute le signe et on obtient le résultat -61_{10}.

Calcul rapide du complément à 2 (facultatif)

Avec un peu d'expérience, on se passe de poser l'opération pour calculer le complément à 2. Remarquons une propriété de l'addition de 1 à un nombre binaire : tous les bits de poids faibles qui valent 1 passent à 0 et le bit à 0 de poids le plus faible passe à 1 ; le reste est inchangé. On peut combiner cette observation avec le calcul du complément à 1, en parcourant les bits de droite à gauche. Tant qu'on a des bits à 0, on ne les modifie pas ; on laisse le premier bit à 1 intact puis on inverse tous les bits suivants.

```
i ← 0
Première partie : copier jusqu'au premier 1
tant que a_i ≠ 1 faire
  b_i ← a_i
  i ← i+1
fin tant que
  b_i ← a_i
  i ← i+1
Deuxième partie : inverser jusqu'à la fin
tant que i ≤ n faire
  b_i ← 1−a_i
  i ← i+1
fin tant que
```

4.2.3 L'arithmétique en complément à 2

Comme mentionné au début de la section, le principal attrait de la notation en complément à 2 est que les additions peuvent se faire sans s'inquiéter du signe des opérandes.

Addition

Voyons l'addition sur un exemple. On a vu plus haut que sur 8 bits $-84_{10} = 10\,101\,100$ et que $-61_{10} = 11\,000\,011$. Essayons donc d'ajouter $23_{10} = 27_8 = 10\,111_2$ à -84

	10 101 100	-84
+	10 111	$+23$
=	11 000 011	$= -61$

La valeur obtenue correspond bien à ce qu'on attendait.

Il faut prendre garde à ne conserver du résultat que le nombre de bits utilisés pour la représentation. L'addition peut produire un bit à 1 supplémentaire qu'il est important d'ignorer. Pour un exemple, additionnons les représentations en complément à 2 de -61 et de -23. Pour cela, il faut commencer par calculer la représentation à 2 de -23 :

$$C_2(00\,010\,111) = C_1(00\,010\,111) + 1 = 11\,101\,000 + 1 = 11\,101\,001$$

On peut maintenant poser l'addition :

		11 000 011	-61
+		11 101 001	-23
=	1	10 101 100	= -84

On ignore le neuvième bit du résultat de l'addition et le résultat obtenu est bien égal à -84

Débordement de capacité

Quand on additionne deux nombres de même signe, on peut rencontrer un débordement de capacité : le résultat est trop grand (s'il est positif) ou trop petit (s'il est négatif) pour être représenté avec le nombre de bits dont on dispose.

Les débordements de capacité sur les additions sont faciles à détecter, puisqu'il s'agit toujours d'ajouter deux nombres de même signe et d'avoir un résultat d'un signe différent. Pourtant, la plupart des environnements actuels ne les traitent pas. Ceci conduit à des situations qui semblent ridicules aux mathématiciens. Par exemple, si on ajoute 1 au plus grand nombre positif, on obtient le plus petit nombre négatif.

Le programme C suivant énumére tous les entiers qu'on peut représenter : il démarrre une boucle avec une variable à 1 et lui ajoute 1 jusqu'à arriver à 0 ! Contrairement à ce qu'on pourrait penser, ce programme ne boucle pas indéfiniment.

```
int
main(){
  int t = 1;

  while(t != 0)
    t = t + 1;

  return 0;
}
```

4.2.4 La multiplication

La multiplication de nombres en complément à 2 peut se faire directement avec un algorithme dit de *Booth*, que nous n'examinerons pas ici. Si on a besoin de multiplier des nombres signés entre eux, le plus simple à la main est de calculer le signe qu'aura le résultat (négatif si les signes sont différents, positif s'ils sont identiques), de multiplier leurs valeurs absolues et de prendre le complément à 2 du résultat s'il doit être négatif.

4.3 Résumé

La façon la plus usuelle de représenter les nombres négatifs est la représentation en complément à deux. Pour trouver la représentation en complément à deux d'un nombre, on part de sa représentation en binaire, on change la valeur de chaque bit et on ajoute 1. On peut additionner entre eux ces nombres de la façon usuelle mais pas les multiplier.

4.4 Exercice

Exercice 4.1 : (simple) A partir de deux nombres représentés en excédent E, comment obtenir avec le moins d'opérations possible la représentation de leur somme en excédent E ?

Exercice 4.2 : Avec quatre bits, on peut représenter 16 valeurs différentes ; enumérer ces 16 valeurs et indiquer pour chacune d'entre elle le nombre qu'elle code en complément à 2.

Exercice 4.3 : (simple) Un mot de 8 bits contient 10100101. Que représente-t-il si c'est un nombre positif sur huit bits ? si c'est un nombre avec un bit de signe ? en exécdent 127 ? En complément à 2 (sur 8 bits) ?

Exercice 4.4 : (simple) Calculer le complément à deux de 0101 1010.

Exercice 4.5 : (simple) Représenter -39 en complément à deux sur 8 bits.

Exercice 4.6 : (exercice type) Convertir 72 et 29 en binaire (en passant par l'octal) sur 8 bits. Calculer leurs compléments à 2 pour obtenir la représentation de -72 et -29, puis calculer, avec les règles d'addition usuelle de la base 2 : $72+29$, $72-29$, $-72+29$, $-72-29$. Convertir les résultats en décimal et vérifier que vous n'avez pas fait d'erreur de calcul.

Exercice 4.7 : (amusant) Quel est l'équivalent, pour les nombres en base 10, du complément à 2 ?

Exercice 4.8 : (à faire) Étant donné deux nombres représentés en complément à 2, comment peut-on savoir lequel est le plus grand en n'utilisant que des comparaisons de bits entre eux ? Écrire un programme pour effectuer cette comparaison. (Pour esquiver les questions de représentation des nombres dans le langage de programmation utilisé, on pourra représenter le nombre comme une liste de chiffres (en Lisp) ou une chaîne de caractères (en C) ou un tableau (en Python) ; si votre maitrise d'un langage de programmation est insuffisante, décrivez l'algorithme avec des phrases.)

Exercice 4.9 : (à faire) Identifier les deux valeurs pour lesquelles le complément à deux ne change pas le bit de signe de la représentation d'un nombre. (Facultatif) Prouver qu'il le change pour tous les autres.

Exercice 4.10 : (difficile) Prouver que le complément à deux du complément à deux d'un nombre est égal à ce nombre.

Supplément

Le programme `gnome-calculator` présenté dans les suppléments du chapitre 2 calcule les compléments à 1 et à 2 des nombres, avec les boutons `1's` et `2's`. Le nombre de bits sur lequel le calcul sera effectué peut être fixé à 16, 32 ou 64 bits.

Chapitre 5

Calculer des ordres de grandeur entre base 2 et base 10

En informatique, on est souvent amené à manipuler simultanément de très grandes et de très petites valeurs. Par exemple, estimer la durée, en heure ou en jours, que prendra un calcul en comptant le nombre d'instructions, sachant qu'une instruction utilisera environ 10^{-9} seconde.

Quand on s'intéresse aux ordres de grandeur, on peut faire des calculs approchés en se fondant sur le fait que $2^{10} = 1024$ est très proche de $10^3 = 1000$.

Ainsi, pour répondre à la question suivante : combien de bits faut-il pour représenter une valeur entière de l'ordre de 10^{18}, le plus simple est de calculer la puissance de 2 à laquelle correspond cette valeur, il suffira d'ajouter 1 pour obtenir la réponse recherchée. Une séquence de transformations simples nous dit que :

$$\begin{aligned} 10^{18} &= (10^3)^6 \\ &= 1000^6 \\ &\approx 1024^6 \\ &\approx (2^{10})^6 \\ &\approx 2^{60} \end{aligned}$$

Il faudra donc environ 61 bits pour représenter cette valeur.

Dans l'autre sens, quel est, en base 10, l'ordre de grandeur du plus grand nombre qu'on peut représenter sur 32 bits ?

Le plus grand nombre qu'on peut représenter sur 32 bits est $2^{32} - 1$. Pour approximer 2^{32} en puissance de 10 :

$$\begin{aligned}
2^{32} &= 2^2 \times 2^{30} \\
&= 4 \times (2^{10})^3 \\
&= 4 \times 1024^3 \\
&\approx 4 \times 1000^3 \\
&\approx 4 \times (10^3)^3 \\
&\approx 4 \times 10^9
\end{aligned}$$

La réponse est donc : de l'ordre de 4 milliards.

Pour les puissances de 2, on utilise des préfixes suivants :

1 octet	=	8 bits	2^3 bits
1 Kilo octet(Ko)	=	1024 octets	2^{13} bits
1 Mega octet(Mo)	=	1024 Ko	$2^{23} bits$
1 Giga octet(Go)	=	1024 Mo	$2^{33} bits$
1 Tera octet(To)	=	1024 Go	$2^{43} bits$
1 Peta octet(Po)	=	1024 To	$2^{53} bits$

5.1 Exercices

Exercice 5.1 : Estimer combien de caractères contient un livre de 250 pages (y compris les espaces). En utilisant un octet par caractère, combien d'octets sont nécessaires pour encoder un tel livre ?

Exercice 5.2 : Au moment de la rédaction de ce support, sur le site de la BNF (Bibliothèque Nationale de France), à la rubrique BNF en chiffres, on peut lire : “ Plus de treize millions de livres et d'imprimés, deux cent cinquante mille volumes de manuscrits, trois cent cinquante mille collections de périodiques, environ douze millions d'estampes, photographies et affiches, plus de huit cent mille cartes et plans, deux millions de pièces musicales, un million de documents sonores, plusieurs dizaines de milliers de vidéos et de documents multimédias, cinq cent trente mille monnaies et médailles..., telle est l'évaluation actuelle des fonds de la Bibliothèque.”
Supposons que ces 13 000 000 de livres et d'imprimés fassent en moyenne 250 pages. En utilisant le code ASCII étendu, combien d'octets sont nécessaires pour encoder ces documents ?

Exercice 5.3 : A chaque fois qu'on remonte d'une génération, le nombre d'ancêtres est multiplié par 2 : on a 2 parents, 4 grand-parents, 8 arrières grand-parents etc. En supposant une génération tous les 25 ans, il y a 40 générations qui nous séparent de nos ancêtres de l'an 1000. Approximer en base 10 le nombre d'ancêtres que nous avons en l'an 1000.

Exercice 5.4 : (Hors sujet) Comparer la réponse à la question précédente avec le nombre d'êtres humains vivant en l'an 1000. Trouver une explication.

Exercice 5.5 : Une légende dit qu'en guise de récompense, l'inventeur (indien) du jeu d'échec demande un grain de blé sur la première case de l'échiquier, deux grains sur la seconde, quatre sur la troisième et ainsi de suite jusqu'à la soixante quatrième. Approximer en base 10 le nombre de grains à poser sur la dernière case. En supposant qu'il faut dix grains de blés pour faire un gramme,

approximer le poids que cela représente. Convertir le résultat en tonne.

Exercice 5.6 : On estime le nombre de parties possibles au Échecs à quelque chose comme 10^{123}.

Combien faut-il de bits pour représenter ce nombre ?

Exercice 5.7 : Quel est l'ordre de grandeur du plus grand nombre qu'on peut représenter en utilisant 2 Kilo octets de mémoire (16384 bits).

Exercice 5.8 : Même question pour un giga octets au lieu d'un kilo octets.

Exercice 5.9 : (instructif) : Considérons le programme de 4.2.3, (a) évaluer l'ordre de grandeur du nombre de tours de boucles qu'il effectue. Considérons que chaque tour de boucle fait trois opérations (*comparer avec 0*, *ajouter 1*, *recommencer la boucle*) et que chaque opération prend un tic d'horloge (la fréquence du processeur indique le nombre de tics d'horloge par seconde) ; (b) évaluer l'ordre de grandeur du temps nécessaire pour exécuter le programme. (c) Placer le code du programme dans un fichier nommé `foo.c`, le compiler avec la ligne de commande `gcc foo.c`, mesurer son temps d'exécution avec la ligne de commande `time a.out` ; comparer le temps mesuré avec l'estimation de (b).

Chapitre 6

Représenter les nombres qui ne sont pas entiers

Dans les chapitres précédents, nous avons examiné comment représenter les nombres entiers, positifs ou négatifs. Nous examinons ici comment représenter les nombres avec des chiffres à droite de la virgule.

6.1 Représentation des nombres fractionnels

Comment représenter un nombre comme $(0.3)_{10}$ en base 2 ? Ce qui ne fonctionne absolument pas, c'est de convertir les chiffres à droite de la virgule en base 2 (ici $3_{10} = (11)_2$) et de placer le résulat à droite de la virgule, pour obtenir $(0.11)_2$. En réalité, on a $(0.11)_2 = (0.75)_{10}$.

Dans la numération en base 10, les chiffres à droite de la virgule indiquent des fractions de la base : le premier chiffre à droite de la virgule indique le nombre de dixièmes, le second le nombre de centièmes, le troisième le nombre de millièmes, etc. Autrement dit, le $i$ème chiffre à droite de la virgule indique le nombre de fois où le nombre contient $1/10^i$. Comme $1/10^n$ peut aussi s'écrire 10^{-n}, il suffit d'indexer les chiffres à droite de la virgule dans la représentation d'un nombre avec des index négatifs à partir de -1. On a alors pour une partie franctionnelle quelconque

$$(0.a_{-1}a_{-2}\cdots a_{-m})_{10} = \sum_{-m \leq i < 0} a_i 10^i$$

Ceci est vrai dans une base b quelconque. On aura donc

$$(0.a_{-1}a_{-2}\cdots a_{-m})_b = \sum_{-m \leq i < 0} a_i b^i$$

On peut combiner cette relation pour exprimer à la fois la partie entière et la partie fractionnelle, en numérotant les chiffres à gauche de la virgule à partir de 0 en montant et ceux à droite de la virgule à partir de −1 en descendant. On a alors

$$a_{n-1} \cdots a_0.a_{-1} \cdots a_{-m} = \sum_{-m \le i < n} a_i b^i$$

Pour la représentation en base 2, le premier chiffre à droite de la virgule vaut $1/2 = (0.5)_{10}$, le second $1/4 = (0.25)_{10}$, le troisième $1/8 = (0.125)_{10}$ et ainsi de suite.

6.1.1 Conversion de la partie fractionnelle d'un nombre de la base 10 vers la base 2

Pour convertir la partie fractionnelle d'un nombre décimal en binaire, il existe une procédure simple : on multiplie la partie fractionnelle du nombre par 2 et on utilise le chiffre à gauche de la virgule comme prochain chiffre de la représentation en base 2. Par exemple, pour représenter $(0.3)_{10}$ en base 2, les étapes sont les suivantes :

travail	représentation partielle	reste à faire
0.3	0. . . .	0.3
$2 \times 0.3 = 0.6$	0.0 . . .	
$2 \times 0.6 = 1.2$	0.01 . . .	$0.3 - 1/4$
$2 \times 0.2 = 0.4$	0.010 . . .	
$2 \times 0.4 = 0.8$	0.0100 . . .	
$2 \times 0.8 = 1.6$	0.01001 . . .	$0.3 - 1/4 - 1/16$

A partir de la dernière ligne du tableau, on repart avec la valeur 0.6 qu'on a déja rencontrée à la seconde ligne ; il est évident que la procédure va produire des séquences de chiffres 1001 sans jamais s'arrêter : il n'y a pas moyen de représenter exactement 0.3 en base 2 avec un nombre fini de chiffres. On a donc $(0.3)_{10} = (0.0\,1001\,1001\,1001\ldots)_2$.

Un autre exemple avec $(0.7)_{10}$

travail	représentation partielle	reste à faire
0.7	0. . . .	0.7
$2 \times 0.7 = 1.4$	0.1 . . .	$0.7 - 1./2$
$2 \times 0.4 = 0.8$	0.10 . . .	
$2 \times 0.8 = 1.6$	0.101 . . .	$0.7 - 1/2 - 1/8$
$2 \times 0.6 = 1.2$	0.1011 . . .	$0.7 - 1.2 - 1/8 - 1/16$
$2 \times 0.2 = 0.4$	0.10110 . . .	

Donc $(0.7)_{10} = (0.1\,0110\,0110\ldots)_2$.

Cette procédure fonctionne parce qu'en partant d'un nombre n, après la première multiplication $2n$ aura un 1 à gauche de la virgule si n est supérieur à

1/2 : si ce n'est pas le cas, il faut placer un zéro à droite de la virgule, sinon il faut placer un 1 et il ne reste à représenter que $n - \frac{1}{2}$. On peut prendre ce qui reste à représenter et le multiplier par 4 : si on obtient une valeur inférieure à 1, alors le second chiffre à droite de la virgule est égal à 0, sinon il est égal à 1 et ainsi de suite. La multiplication des parties fractionnelles à chaque ligne fait qu'on multiplie ce qui reste à représenter par 2 sur la première ligne, par 4 sur la ligne suivante, par 8 sur la troisième ligne et ainsi de suite.

6.1.2 Conversion de la partie fractionnelle d'un nombre de la base 2 à la base 10

Pour convertir un nombre fractionnel de la base 2 vers la base 10, il y a deux méthodes pratiques.

Dans la première méthode, on calcule la valeur en décimal de chaque bit à 1 et on additionne ces valeurs. Par exemple, pour convertir $n = (0.101001)_2$ on voit que

$$\begin{aligned} n &= (0.101001)_2 \\ &= (0.1)_2 + (0.001)_2 + (0.000001)_2 \\ &= 2^{-1} + 2^{-3} + 2^{-6} \\ &= 1/2 + 1/8 + 1/64 \\ &= (0.5)_{10} + (0.125)_{10} + (0.015625)_{10} \\ &= (0.640625)_{10} \end{aligned}$$

Cette méthode présente l'inconvénient qu'on est obligé de faire des divisions pour chaque bit à 1, puis une addition. Elle n'est à recommander que quand le nombre contient peu de bits à 1.

Dans la seconde méthode, on fait la conversion d'un nombre de la base 2 vers la base 10, puis une seule division.

$$\begin{aligned} n &= (0.101001)_2 \\ &= (101001)_2/64_{10} \\ &= (51)_8/64_{10} \\ &= 41_{10}/64_{10} \\ &= (0.640625)_{10} \end{aligned}$$

L'unique division est plus compliquée que dans la première méthode mais elle est unique.

6.2 Représentation des nombres en virgule fixe

Une façon simple et pratique de représenter les nombres avec des parties fractionnelles consiste à choisir, en fonction des nombres que l'on souhaite représenter, une position fixe pour la virgule.

Par exemple, si on utilise 8 bits pour représenter un nombre, on peut choisir de placer la virgule entre le quatrième et le cinquième bit. Les huit bits $a_7a_6a_5a_4a_3a_2a_1a_0$ représenteront alors un nombre dont la partie entière est égale à $a_7a_6a_5a_4$ et la partie fractionnelle à $a_3a_2a_1a_0$.

L'avantage d'une telle représentation est que les additions et les multiplications sont simples. On additionne deux nombres en virgule fixe de la même manière que deux nombres entiers. Pour la multiplication, on ne conserve que les bits de gauche de la partie fractionnelle (en utilisant les bits de droite pour l'arrondi) et les bits de droite de la partie entière (si les bits gauche de la partie entière — ceux qu'on élimine — contiennent autre chose que 0, c'est qu'on a un *débordement de capacité* : on ne peut pas représenter exactement le résultat).

L'inconvénnient de la virgule fixe, c'est qu'on ne peut pas représenter à la fois de très grands et de très petits nombres (voir exercice). Pour cela, on utilise la représentation en virgule flottante, présentée dans la section suivante.

6.3 Représentation des nombres en virgule flottante

Le principe de la représentation en virgule flottante est de séparer les chiffres significatifs et la position de la virgule. Nous commençons par rappeler la méthode que nous utilisons couramment en base 10.

6.3.1 Virgule flottante en base 10

On utilise couramment la représentation en virgule flottante en base 10 pour les très grands et très petits nombres, particulièrement en physique. Sur les calculettes, cette représentation est souvent appelée la *notation scientifique*.

Ainsi, les chiffres significatifs d'une grandeur (la *mantisse*) peuvent être 1245. L'usage est de positionner la virgule dans la mantisse à droite du chiffre le plus significatif : la mantisse devient 1.2345. On ajoute ensuite un déplacement de la virgule, noté comme une multiplication par une puissance de 10.

1.2345×10^{0}	1.2345
1.2345×10^{1}	12.345
1.2345×10^{2}	123.45
1.2345×10^{10}	12345000000.
1.2345×10^{-1}	0.12345
1.2345×10^{-2}	0.012345
1.2345×10^{-10}	0.00000000012345

Comme illustré dans la table précédente, l'exposant de la puissance de 10 indique en fait où se place la virgule, ce qui permet de représenter des nombres très grands ou très petits avec seulement quelques chiffres. Il est plus facile d'écrire la masse (supposée) d'une particule comme le boson Z avec 9.1187×10^{31} eV plutôt que 91187000000000000000000000000000 eV.

Pour multiplier ces nombres en virgule flottante, il suffit de multiplier les mantisses et d'ajouter les exposants. En revanche, l'addition est plus délicate : il faut représenter les deux nombres avec le même exposant avant de pouvoir additionner les mantisses. Ces caractéristiques sont indépendantes de la base (entière) choisie et s'appliquent donc aussi dans la base 2.

6.4 Virgule flottante en base 2

Pour représenter les nombres en virgule flottante d'une manière avec laquelle les calculs sont aisés, les ordinateurs utilisent une représentation équivalente à la notation scientifique des calculettes mais fondée la base 2.

Un nombre se représente avec un signe s, une mantisse m et un exposant e. La valeur du nombre est $(-1)^s \times m \times 2^e$.

6.4.1 Exemples de nombres en virgule flottante

Un nombre représenté en virgule flottante contiendra donc un signe, une mantisse qui indique la valeur et un exposant qui indique où se situe la virgule dans la mantisse. Le tableau suivant donne des exemples de nombres positifs.

Dans la colonne de gauche, la mantisse et l'exposant sont tous les deux représentés en binaire ; dans la suivante, on trouve la représentation en décimal de l'exposant binaire de départ qui indique donc comment il faut déplacer la virgule. Cela permet dans la troisième colonne d'écrire le nombre comme un nombre à virgule ordinaire en binaire. La dernière colonne contient la représentation du nombre en décimal.

flottant binaire	exposant décimal	valeur binaire	valeur décimale
$1.101_2 \times 2^0$	0	1.101_2	1.625_{10}
$1.101_2 \times 2^1$	1	11.01_2	3.25_{10}
$1.101_2 \times 2^{10_2}$	2	110.1_2	6.5_{10}
$1.101_2 \times 2^{-1}$	-1	0.1101_2	0.8125_{10}
$1.101_2 \times 2^{-10_2}$	-2	0.01101_2	0.40625_{10}
$1.101_2 \times 2^{1000_2}$	8	110100000_2	416_{10}
$1.101_2 \times 2^{-1000_2}$	-8	0.00000001101_2	0.00634765625_{10}

6.4.2 Les nombres en virgule flottante dans la mémoire

Les nombres en virgule flottante sont stockés dans les mots de la mémoire. Dans les bits d'un mot, certains contiennent la mantisse, d'autres l'exposant. Le rôle des bits du mot est en général fixé d'une manière spécifique au processeur. On trouvera plus loin la présentation du format préconisé par la norme IEEE 754, qui est le format plus employé à l'heure actuelle.

Je rappelle qu'il n'y a pas moyen, en examinant le contenu d'un mot de

mémoire de déterminer s'il contient un entier ou bien un nombre flottant (ou une suite de caractères, ou le code d'une instruction, ou l'adresse d'un autre mot mémoire ou n'importe quoi d'autre) : c'est le contexte seul qui permet de déterminer la manière dont le contenu de la mémoire doit être interprété.

6.4.3 Flottants ou réels

Dans certains langage de programmation, les nombres représentés ainsi sont appelés des nombres *réels* mais il s'agit d'une approximation ; avec le nombre fixe de bits attribués à l'exposant, il existe des limites au delà desquelles les nombres sont trop petits ou trop grands pour être représentés ; avec le nombre fixe de bits attribués à la mantisse, il existe une infinité de nombres réels voisins qui auront la même représentation en virgule flottante. (On le voit aisément en fixant un nombre de chiffres pour la mantisse, disons m et en choisissant une valeur $1.a_{-1} \cdots a_{-m}$ pour cette mantisse. Les nombres différents $1.a_{-1} \cdots a_{-m}1$, $1.a_{-1} \cdots a_{-m}01$, $1.a_{-1} \cdots a_{-m}001$ etc. ont tous la même approximation dans notre représentation. Il est intéressant que ce raisonnement puisse se tenir sans préciser si la mantisse est représentée en base 10 ou en base 2.)

6.5 La norme IEEE754

La norme IEEE 754 est la plus utilisée à l'heure actuelle pour représenter les nombres en virgule flottante dans les ordinateurs. Elle fixe un nombre de bits pour la représentation des nombres sur 32 et 64 bits, elle traite de quelques cas particuliers et spécifie des manières d'arrondir le résultat des opérations quand on ne peut pas le représenter de manière exacte.

Les caractéristiques les plus importantes de la norme IEEE 754 sont

- un nombre est codé sur 32 bits (simple précision), ou 64 bits (double précision).
- la mantisse appartient à l'intervalle [1,0 ; 10,0[(en binaire).
- le seul chiffre à gauche de la virgule étant toujours 1, n'est pas représenté (cela signifie que 0 sera représenté comme un cas particulier).
- la partie fractionnelle de la mantisse sera représentée sur 23 bits (simple précision) ou 52 bits (double précision).
- l'exposant est codé sur 8 bits en excédent 127 (simple précision) ou sur 11 bits en excédent 1023 (double précision).

En simple précision, le bit de signe est codé par 1 bit, l'exposant sur 8 bits et la mantisse sur 23 bits. Ainsi par exemple, soit un mot de 32 bits

(1 10000000 01001000000000000000000)

Il se découpe en un bit de signe qui vaut 1, les bits de l'exposant qui valent 10000000 et les bits de la mantisse 01001000000000000000000).

Le bit de signe est égal à 1, donc le nombre est négatif.

Les bits de l'exposant valent $(10000000)_2 = 128_{10}$, donc la valeur de l'exposant est $128 - 127 = 1$.

Les bits de la mantisse sont 01001000000000000000000), donc la valeur de la mantisse est $(1.01001000000000000000000)_2 = (1.01001)_2 = 1+9/32 = 1.28125$.

La valeur du nombre est donc

$$-1.28125 \times 2^1 = -2.5625$$

On peut aussi résumer cette séquence d'opérations avec :

$$\begin{aligned} x &= -1 \times (1.01001)_2 \times 2^{(10000000)_2-(127)_{10}} \\ &= -(1 + 1/2^2 + 1/2^5) \times 2^{128-127} \\ &= -(2 + 1/2 + 1/2^4) \\ &= -2.5625 \end{aligned}$$

Exemple : représentation de 21.78125 en simple précision On commence par convertir le nombre en binaire. Pour la partie entière, $21_{10} = 25_8 = 10101_2$. Pour la partie fractionnelle :

$$\begin{array}{r|l} 0.78125 & 0.\ldots \\ 2 \times 0.78125 = 1.5625 & 0.1\ldots \\ 2 \times 0.56250 = 1.125 & 0.11\ldots \\ 2 \times 0.125 = 0.25 & 0.110\ldots \\ 2 \times 0.25 = 0.5 & 0.1100\ldots \\ 2 \times 0.5 = 1.0 & 0.11001\ldots \end{array}$$

On a donc $(21.7812)_{10} = (10100.11001)_2$.

Il faut ensuite calculer la valeur de l'exposant :

$$\begin{aligned} (10101.11001)_2 &= (10101.11001)_2 \times 2^0 \\ &= (1.010111001)_2 \times 2^4 \end{aligned}$$

- Puisque le nombre est positif, le bit de signe vaut 0.
- Puisque la mantisse vaut $(1.010011001)_2$, il faut placer 01011100100000000000000 dans les 23 bits de la mantisse.
- Puisque l'exposant vaut 4, il faut placer $127 + 4 = 131_{10} = (10000011)_2$ dans les huit bits de l'exposant.

Les 32 bits seront donc :

0 10000011 01011100100000000000000

6.5.1 Cas particuliers

La norme IEEE 754 prévoit un certain nombre de cas particuliers, qui sont représentés avec les bits de l'exposant qui codent 0 et 255 en simple précision (0

et 2047 en double précision). Si ce n'étaient pas des cas particulier, l'exposant vaudrait alors $0 - 127 = -127$ et $255 - 127 = 128$.

Si les bits de l'exposant valent 0 et que la mantisse vaut 0, alors c'est le nombre 0 qui est représenté (cela signifie qu'il existe deux représentations du nombre 0, suivant que le bit de signe vaut 0 ou 1, avec tous les autres bits à 0).

Si les bits de l'exposant valent 0 et que la mantisse est différente de 0, alors on a la représentation *dénormalisée* d'un nombre : le chiffre à gauche de la virgule est égal à 0 et l'exposant vaut -126 ; cela permet de représenter de tous petits nombres.

Si les bits de l'exposant valent 255 et que la mantisse vaut 0, alors le nombre représente l'infini ($+\infty$ ou $-\infty$ suivant la valeur du bit de signe). Cela est utilisé pour indiquer un débordement de capacité.

Si les bits de l'exposant valent 255 et que la mantisse est différente de 0, alors ce n'est pas un nombre qui est représenté. C'est ce qu'on obtient si on extrait, par exemple, la racine carrée d'un nombre négatif. Une opération dont l'un des deux opérandes n'est pas un nombre produit un résultat qui n'est pas un nombre non plus : le cas particulier est *propagé*.

Les fonctions qui impriment des valeurs de nombres en virgule flottante impriment souvent `Inf` et `NaN` pour ces deux derniers cas particuliers.

6.6 Résumé

Une représentation des nombres réels existe. Elle ne permet pas de représenter de façon exacte l'ensemble des réels. Elle donne au pire une représentation du nombre avec une précision dépendant de la mantisse. Cette approximation, pouvant être considéré comme négligeable, peut devenir problématique lors d'opérations arithmétiques ou lors de conversion de type. (Un exemple de conversion ratée : l'échec du premier tir d Ariane 5 a été causé par une erreur du système informatique de guidage. Cette erreur est survenue lors d'une conversion de type qui a causé un dépassement de capacité d'une variable. Parmi les recommandations émises suite à cet accident on notera : Identifier toutes les hypothèses implicites faites par le code et ses documents de justification sur les paramètres fournis par l'équipement. Vérifier ces hypothèses au regard des restrictions d'utilisation de l'équipement. Vérifier la plage des valeurs prises dans les logiciels par l'une quelconque des variables internes ou de communication.)

6.7 Exercice

Exercice 6.1 : Écrire l'algorithme qui convertit la partie fractionnelle d'un nombre de la base 10 vers la base 2. Vérifier que l'algorithme ne rentre pas dans une boucle infinie, même avec les nombres comme $(0.3)_{10}$ qui ont besoin d'un nombre infini de chiffres en base 2 pour être représentés exactement.

Exercice 6.2 : Convertir $(1000100100.0010010001)_2$ en base 10 (sans arrondi).

Exercice 6.3 : Convertir $(1234.4321)_{10}$ en base 2 avec 16 bits à gauche et à droite de la virgule.

Exercice 6.4 : Les méthodes de conversion entre les base 2, 8 et 16 sont-elles utilisables pour la partie fractionnelle ? Donner une opinion, puis un exemple (ou un contre-exemple), puis une démonstration.

Exercice 6.5 : Si on représente un nombre sur 32 bits avec 16 bits à gauche et 16 bits à droite de la virgule, quel est le plus petit et le plus grand nombres positifs non nuls qu'on peut représenter ?

Exercice 6.6 : Même question que la précédente, sur $2n$ bits de représentation avec n bits à gauche et n bits à droite de la virgule.

Exercice 6.7 : Écrire un programme pour multiplier deux nombres représentés en base 10 et en virgule flottante entre eux. Penser à *normaliser* le résultat de façon à n'avoir qu'un seul chiffre à gauche de la virgule dans le résultat.

Exercice 6.8 : Déterminer à la main quelle est, en base 10, l'ordre de grandeur de la plus grande valeur qu'on peut représenter en simple précision avec la norme IEEE 754 ? La plus petite strictement positive sans utiliser les nombres dénormalisés ? Avec les nombres dénormalisés ?

Variante : quelle l'ordre de grandeur des nombres suivants : (a) $(2-2^{-23}) \times 2^{127}$ (b) 2^{-126} (c) $2^{-23} \times 2^{-126}$?

Exercice 6.9 : Il existe une plus grande valeur v représentée en virgule flottante telle que $x+v=x$, à cause des erreurs d'arrondi. Écrire un programme pour déterminer cette valeur pour $x=1$, $x=2$, $x=4$. Généraliser pour $x=2^n$. Vérifier avec le programme.

Exercice 6.10 : Quelle manipulation faut-il effectuer sur les bits de la représentation d'un nombre flottant pour le multiplier par 4 ? (c'est facile si on ne tient pas compte du cas particulier des nombres dénormalisés).

Exercice 6.11 : Écrire un programme qui calcule la fonction exponentielle avec la formule de Taylor :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots + \frac{x^n}{n!}$$

Constater l'erreur d'approximation produite.

6.8 Supplément : un outil de calcul

Sous Unix/Linux, il existe une commande `bc` qui permet de faire des calculs sur des nombres de taille quelconque. On lance la commande puis on entre au clavier des expressions sur une ligne et `bc` affiche leur valeur ; on sort avec Controle+D (la touche CTRL et la touche D pressées en même temps).

La taille des nombres manipulés n'est limitée que par celle de la mémoire

disponible, ce qui permet de faire des calculs sur de très grands nombres. On peut calculer en virgule fixe, en affectant à `scale` le nombre de bits à conserver à droite de la virgule. On peut contrôler les bases dans lesquelles les nombres seront lues ou écrites avec les variables `ibase` et `obase`.

Voici une courte session avec bc, avec des commentaires en italique.

```
$ bc -q            lancement de la commande
2 * 3              entrée d'une expression
6                  affichage du résultat
2^100              entrée d'une autre expression
                   affichage du résultat
1267650600228229401496703205376
2^1000             entrée d'une troisième expression
                   résultat sur cinq lignes
10715086071862673209484250490600018105614048117055336074437503883703\
51051124936122493198378815695858127594672917553146825187145285692314\
04359845775746985748039345677748242309854210746050623711418779541821\
53046474983581941267398767559165543946077062914571196477686542167660\
429831652624386837205668069376
1/3                entrée d'une expression
0                  résultat : les calculs se font sur des entiers
scale=10           pour avoir 10 chiffres à droite de la virgule
1/3                entrée de la même expression
.3333333333        le résultat avec la partie fractionnelle
obase=2            modification de la base de sortie
4321               entrée d'une expression
1000011100001      affichage du résultat
123.321            entrée d'une expression
1111011.0101001000 affichage du résultat
obase=10           retour à la base de sortie originale
ibase=2            modification de la base d'entrée
1000011100001      entrée d'une expression
4321               affichage du résultat
^D                 Contrôle D
$                  Fin de la session
```

Chapitre 7

Représentation des caractères

Nous présentons dans ce chapitre les systèmes de codage de caractères les plus utilisés, en insistant tout particulièrement sur les codes ASCII, ISO8859, Unicode et UTF.

7.1 Les codes de caractères

Les ordinateurs ne stockent et ne traitent que des nombres mais nous souhaitons aussi stocker des textes, qui se composent de caractères ; pour cette raison, on utilise une convention dans laquelle chaque caractère est codé par un nombre.

On peut choisir n'importe quel codage pour les caractères mais il est bien sur indispensable, quand on échange de l'information, que l'émetteur et le récepteur soient d'accord sur le codage employé.

De nombreux codes ont été utilisés ; on n'en rencontre certains que très rarement, comme le code BAUDOT qui était utilisé sur les Telex ou le code EBCDIC, spécifique des gros ordinateurs d'IBM. A l'heure actuelle, la plupart des systèmes utilisent le code ASCII ou l'une de ses nombreuses variantes.

7.2 Le code ASCII

A l'heure actuelle, le codage de caractère le plus utilisé est le code ASCII (*American Standard Code for Information Interchange*). Comme il est très couramment utilisé, il joue le rôle de *lingua franca* entre ordinateurs.

7.2.1 Organisation du code ASCII

Le code ASCII utilise sept bits pour représenter un caractère ; cela signifie donc qu'il permet de représenter $2^7 = 128$ caractères différents. En pratique, on

utilise le plus souvent un octet pour contenir un caractère codé en ASCII, avec le huitième bit à zéro. (Certains systèmes de transmission utilisent le huitième bit pour autre chose, par exemple pour transmettre une somme de contrôle qui permet de détecter les erreurs de transmission les plus communes. On parle dans ce cas d'un bit de *parité*.)

NUL	0	DLE	10		20	0	30	@	40	P	50	’	60	p	70
SOH	1	DC1	11	!	21	1	31	A	41	Q	51	a	61	q	71
STX	2	DC2	12	”	22	2	32	B	42	R	52	b	62	r	72
ETX	3	DC3	13	#	23	3	33	C	43	S	53	c	63	s	73
EOT	4	DC4	14	$	24	4	34	D	44	T	54	d	64	t	74
ENQ	5	NAK	15	%	25	5	35	E	45	U	55	e	65	u	75
ACK	6	SYN	16	&	26	6	36	F	46	V	56	f	66	v	76
BEL	7	ETB	17	’	27	7	37	G	47	W	57	g	67	w	77
BS	8	CAN	18	(	28	8	38	H	48	X	58	h	68	x	78
HT	9	EM	19	)	29	9	39	I	49	Y	59	i	69	y	79
LF	A	SUB	1A	*	2A	:	3A	J	4A	Z	5A	j	6A	z	7A
VT	B	ESC	1B	+	2B	;	3B	K	4B	[	5B	k	6B	{	7B
FF	C	FS	1C	,	2C	<	3C	L	4C	\	5C	l	6C	|	7C
CR	D	GS	1D	-	2D	=	3D	M	4D	]	5D	m	6D	}	7D
SO	E	RS	1E	.	2E	>	3E	N	4E	ˆ	5E	n	6E	~	7E
SI	F	US	1F	/	2F	?	3F	O	4F	_	5F	o	6F	DEL	7F

TABLE 7.1 – Le jeu de caractères ASCII et leur correspondance en hexadécimal

Les 33 premiers codes (de 0 à 32) et le dernier (code 127) codent des choses qui ne sont pas imprimables mais qui peuvent être utilisées pour contrôler les comportements des périphériques. Par exemple le code 7 (BEL) provoque un bip sur l'ordinateur qui l'affiche. Certains de ces codes controlent la position du prochain caractère : le code 10 (A_{16}, LF comme *Line Feed*) provoque un passage à la ligne suivante et le code 13 (D_{16}, CR comme *Cariage Return*) un retour sur la première colonne de la ligne. Le code numéro 32 (20_{16}) est utilisé pour le caractère espace.

Les autres codes sont utilisés pour représenter les caractères alphabétiques usuels (en majuscule et en minuscule), les chiffres et les symboles de ponctuation les plus courants.

Le code 27 ($1B_{16}$, ESC pour *escape*, échappement) joue un rôle particulier : il sert à indiquer que les codes qui suivent ne sont pas du code ASCII mais doivent être interprétés d'une façon spéciale qui dépend du contexte.

7.2.2 Les limites du code ASCII

Le code ASCII présente quelques limites qui sont dues au petit nombre de caractères qu'il peut coder ; il lui manque des caractères importants, comme ×

le symbole de la multiplication. (Pour cette raison, les informaticiens ont pris l'habitude de noter la multiplication en utilisant le caractère *.)

Surtout, le code ASCII, comme son nom l'indique, est un code américain et ne permet donc de coder aucun des caractères spéciaux qui apparaissent dans les autres langues européennes comme les caractères accentués ou le double ss de l'allemand. Pour les langues qui utilisent d'autres alphabets comme le grec, les langues slaves, l'arabe et le persan, l'hébreu ou de nombreuses langues indiennes, ou pour les langues orientales (chinois, japonais, coréen), qui s'écrivent avec des idéogrammes, le code ASCII est complètement inadéquat.

Il existe de nombreux codes pour pallier aux insuffisances du code ASCII, si nombreux qu'il est parfois difficile de s'y retrouver ; les plus importants sont probablement les codes iso8859 et le couple formé par Unicode et UTF.

7.3 La norme iso8859 : une extension du code ASCII

La norme iso8859 exploite le fait que le code ASCII n'utilise que sept bits pour un caractère qui est presque toujours stocké sur huit bits, dans un octet ; ceci permet de coder 128 nouveaux caractères, avec les codes entre 128 et 255, alors que les codes entre 0 et 127 s'interprètent de la même façon que dans le code ASCII.

Ceci présente l'avantage d'une compatibilité ascendante et descendante avec le code ASCII. La compatibilité est *ascendante* parce qu'un programme qui s'attend à lire un texte codé en iso8859 pourra interpréter correctement ce texte s'il est codé en ASCII. Elle est aussi *descendante* parce qu'un programme qui attend quelque chose codé en ASCII reconnaitra dans le texte tous les caractères qui peuvent être codés en ASCII. (Naturellement, un programme qui attend du code ASCII et qui reçoit un texte codé en iso8859 ne pourra pas interpréter comme il convient les codes compris entre 128 et 255 qui n'existent pas dans le code ASCII.)

Comme il existe bien plus de 128 caractères à coder en dehors de ceux qui sont déja présents dans le code ASCII, la norme iso8859 est construite autour de variantes, qui indiquent de quelle façon les codes entre 128 et 255 doivent être interprétés ; la variante la plus importante pour nous est nommée iso8859-1 (parfois aussi nommée *iso-latin-1*) qui contient les caractères accentués utilisés en Europe occidentale. (Pour mémoire, d'autres variantes communes sont l'iso8859-2 pour l'Europe de l'est, l'iso8859-3 pour l'Europe du sud, l'iso8859-4 pour l'Europe du nord, l'iso8859-5 pour l'alphabet cyrillique, l'iso8859-6 pour l'arabe, etc.)

La norme iso8859 présente quelques inconvénients majeurs ; d'une part il manque dans la variante iso8859-1 un caractère essentiel, le « e dans l'o » (œ) qu'on trouve en français dans des mots courants comme *cœur* ou *œuvre*. D'autre part, la division du code en page oblige au minimum à accompagner chaque

texte d'une information supplémentaire qui indique quelle variante du code est utilisée ; ceci complique singulièrement les choses pour les textes qui doivent mélanger des caractères de plusieurs variantes, comme un texte en français qui contient une citation du grec par exemple. Finalement, les 128 nouveaux codes sont largement insuffisants pour les langues à idéogrammes qui ont besoin de plusieurs milliers de codes différents.

7.4 Unicode : un codage sur seize bits

Une façon de résoudre les problèmes des normes comme l'iso8859 consiste à modifier le nombre de bits utilisés pour coder un caractère ; c'est ce que fait la norme Unicode, dans laquelle chaque caractère est représenté sur 16 bits, ce qui permet de coder 65536 caractères différents ; c'est suffisant pour coder à peu près toutes les écritures connues, à condition d'utiliser des codes identiques pour les idéogrammes chinois, coréens et japonais qui se dessinent de manière identique, même quand leurs significations sont différentes dans ces trois langues.

En utilisant Unicode, on peut mélanger plusieurs alphabets dans un même document, sans avoir besoin de changer de système de codage, ce qui est précieux. D'ici quelques années, Unicode (avec UTF présenté dans la section suivante) va probablement s'imposer comme le standard de codage de caractères le plus répandu.

Le codage Unicode présente deux inconvénients : les documents sont deux fois plus gros qu'avec le code ASCII et il est incompatible avec l'*ascii*.

Le fait que les documents soient deux fois plus volumineux n'est pas un gros problème ; la capacité des disques et les vitesses de transport de l'information augmentent suffisamment vite pour que ce problème puisse être ignoré. En revanche, l'incompatibilité avec le code ASCII pose un problème difficile ; il est bien sûr hors de question de modifier tous les systèmes pour qu'ils utilisent Unicode et il est hors de question de ne pas laisser les nouveaux systèmes qui utilisent Unicode dialoguer avec les anciens systèmes qui utilisent l'ASCII ou une de ses extensions. C'est la raison d'être de la norme UTF que nous présentons dans la section suivante.

7.5 UTF : un codage à longueur variable

Unicode est incompatible avec le code ASCII, ce qui est un réel problème. La solution consiste à lui associer un autre code, UTF comme *Unicode Translation Format*, qui permet de maintenir la compatibilité avec le code ASCII.

En UTF, chaque caractère est codé sur un, deux ou trois octets. Tous les caractères du code ASCII sont codés en UTF sur un octet *avec la même valeur* qu'en ASCII. En revanche, un octet qui contient une valeur supérieure à 127 annonce un caractère sur deux ou trois octets et UTF est construit de telle

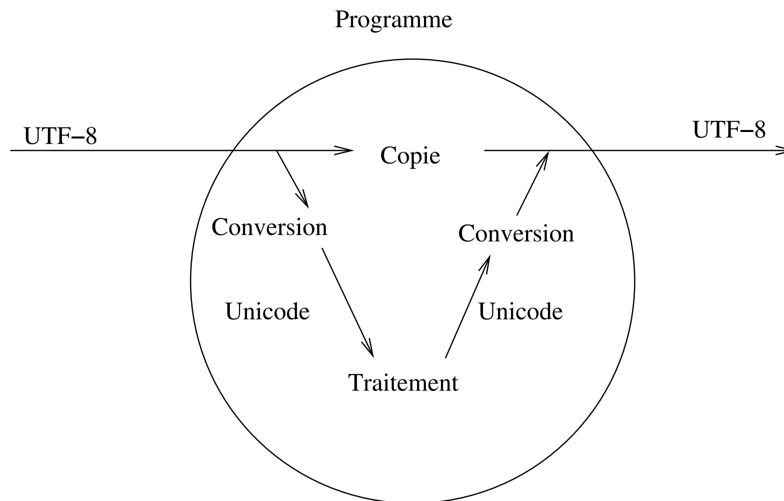


FIGURE 7.1 – Un programme lit et écrit du texte codé en UTF mais le convertit en Unicode quand le texte doit être modifié.

manière que tous ces octets contiennent un code supérieur à 127. Ce mécanisme garantit une compatibilité ascendante et descendante avec l'ASCII.

La compatibilité est ascendante, puisqu'un programme qui lit des caractères codés en UTF interprétera correctement un fichier ASCII (les codes UTF entre 0 et 128 codent les mêmes caractères que l'ASCII). Elle est aussi descendante, puisqu'un programme qui traite de l'ASCII et qui reçoit un fichier UTF interprétera correctement tous les caractères de ce fichier qui apparaissent dans le code ASCII (puisque quand un caractère est codé sur plusieurs octets, tous les octets contiennent des valeurs supérieures à 127, qui n'appartiennent pas au code ASCII).

Un algorithme assez simple permet de traduire l'Unicode en UTF et inversement.

Les programmes, en général, ne peuvent pas utiliser simplement l'UTF pour manipuler du texte, parce que c'est un code de longueur variable. Or un codage de longueur variable complique sérieusement l'écriture des programmes : par exemple, pour trouver le 2843ème caractère d'un texte, il est nécessaire d'examiner les 2842 caractères précédents, qui peuvent occuper un nombre variable d'octets, afin de savoir dans quels octets précisément le caractère est stocké. Avec un codage de longueur fixe, on sait sans examiner les caractères précédents que le 2843ème caractère est stocké à partir de l'octet numéro 2842 $\times$ *la taille d'un caractère*.

De ce fait, dans les systèmes qui supportent Unicode + UTF, la structure des programmes est celle décrite dans la figure 7.1. Tous les programmes lisent des données textuelles encodées en UTF ; les programmes qui ne manipulent

pas le texte peuvent transmettre directement ces données ; ceux qui ont des traitements à effectuer sur le texte commencent par le convertir en Unicode, incompatible avec l'ASCII mais de longueur fixe, ils traitent les données sous leur forme Unicode, puis ils reconvertissent les données en UTF au moment de produire les résultats.

Dans la bibliothèque du langage C, les caractères Unicode sont appelés des *wide char* et définis avec le type `wchar_t` ; les caractères en UTF sont appelés des *multi-byte string*. La plupart des fonctions qui effectuent des opérations usuelles ont un équivalent pour les *wide chars* ; par exemple la fonction `strcat` définie pour les chaînes ASCII a un équivalent `wcscat` pour les chaînes Unicode.

7.6 la commande `file`

Il existe une commande `file` qui s'efforce de deviner le contenu d'un fichier en examinant son contenu. On peut l'utiliser pour distinguer entre les fichiers qui contiennent de l'UTF et ceux qui contiennent de l'iso8859. Voici un exemple commenté d'utilisation

```
$ ls                    # Le répertoire est vide
$ cat > f1.txt          # Fabrication d'un fichier.
Ce texte ne contient pas d'accents
$                       # Pour terminer la commande cat,
$                       #   j'ai tapé sur les touches Controle et D en même temps.
$ ls                    # Le fichier a bien été créé.
f1.txt
$ file f1.txt           # Que contient le fichier ?
f1.txt: ASCII text
$ cat > f2.txt          # Un deuxième fichier.
Avec un caractère accentué.
$                       # Ici aussi, contrôle-D pour finir
$ file f2.txt           # Que contient le fichier ?
f2.txt: UTF-8 Unicode text
$                       # Conversion du contenu du fichier
$ tcs -f utf -t 8859-1 < f2.txt > f3.txt
$ file f3.txt           # Que contient le résultat ?
f3.txt: ISO-8859 text
$                       # Conversion dans l'autre sens.
$ tcs -f 8859-1 -t utf < f3.txt > f4.txt
$ file f4.txt           # Résultat.
f4.txt: UTF-8 Unicode text
$ cmp f2.txt f4.txt     # Comparaison avec le fichier de départ :
                        # Pas de message = ils sont identiques.
# On peut utiliser file avec n'importe quel fichier
$ file .
.: directory
```

```
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses share
```

7.7 Résumé

Pour représenter des caractères dans un ordinateur, il suffit de choisir un codage qui utilise des nombres pour représenter des caractères. Le système de codage le plus utilisé est le codage ASCII, sur 7 bits, qui ne peut coder que les caractères américains. Le codage ISO8859 étend l'ASCII sur 8 bits, pour coder de nouveaux caractères; ainsi l'ISO8859-1, appelé aussi ISO-LATIN-1, code les caractères utilisés en europe occidentale.

Un autre système de codage utilise en combinaison, Unicode, un code sur 16 bits pour représenter les informations dans les programmes et UTF (comme *Unicode Translation Format*), un code de longueur variable, pour transmettre les informations.

Entre iso8859 et UTF8, préférez UTF8.

7.8 Exercices

Exercice 7.1 : Installer sur votre ordinateur un programme capable de convertir entre différentes normes de codage de caractères. Par exemple, un programme nommé `iconv` ou `tcs`, comme *Translate Character Set* font l'affaire. Quels sont les codages que supporte ce programme ? En citer deux qui ont une compatibilité ascendante avec l'ASCII (en dehors de l'UTF et de l'ISO8859) et un qui est incompatible. Ne pas oublier de mentionner la manière dont vous avez déterminé cette compatiblité.

Exercice 7.2 : Sauver un fichier, avec des caractères accentués avec les deux systèmes de codage iso8859-1 et utf-8. Les lire avec les commandes `cat`, `more` et `less`. Décrire ce que vous observez. (Facultatif) S'il y a un problème, provient-il de la commande, du terminal ou du système ?

(Sous emacs, il y a un caractère en bas à gauche qui indique avec quel système de codage le contenu du fichier sera écrit. Il y a un 'u' pour Unicode-UTF, un 1 pour l'ISO8859-1 etc. Avec la commande
`M-X set-buffer-file-coding-system` on peut choisir le codage employé pour le fichier.

Le but de l'exercice est d'avoir des fichiers qui contiennent les mêmes caractères accentués en iso8859-1 et en utf-8 ; vouz pouvez utiliser un autre outil qu'emacs si vous le préférez ; notez cependant qu'Emacs, bien qu'un peu compliqué à maîtriser au départ, est l'éditeur de texte préféré de beaucoup de programmeurs.)

Exercice 7.3 : Dans les fichiers html, on peut indiquer le codage de caractère utilisé dans le fichier avec une ligne dans l'en-tête de la forme :

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

ou

```
<meta http-equiv="Content-Type" content="text/html; charset=iso8859-1" />
```

Que voit-on quand on essaye d'afficher comme de l'`utf-8` un fichier qui contient de `iso8859-1` ? Et quand on essaye d'afficher comme de l'`iso8859-1` un fichier qui contient de l'`utf-8` ?

Un fichier html élémentaire peut contenir les lignes

```
<html>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
<body>
  Caractères accentués
</body>
</html>
```

Si le fichier s'appelle `toto.html`, on peut voir son contenu avec `firefox toto.html`.

Exercice 7.4 : (assez difficile, pas de corrigé) Trouver une séquence de caractères accentués du français qui, encodée en iso-8859, donne autre chose qu'une erreur en utf8.

7.9 Supplément : exercice type corrigé sur les conversions

Ce chapitre conclue la partie sur différentes manières de représenter de l'information avec des bits dans la mémoire de l'ordinateur. Voici un exercice corrigé sur le sujet.

Enoncé : Étant donné quatre octets qui contiennent les valeurs binaires 0010 1000, 1001 1100, 0100 0011 et 1000 0000, que représentent-ils si

1. ce sont quatre nombres non signés (sur 8 bits) ?
2. ce sont deux nombres nombres non signés (sur 16 bits) ? Que vaut leur somme en binaire ? en décimal ?
3. ce sont quatre nombres en complément à deux (sur 8 bits) ?
4. ce sont deux nombres en complement à deux (sur 16 bits) ? Que vaut leur somme en binaire ? en décimal ?
5. S'il s'agit d'un nombre entier sur 32 bits, quel est son ordre de grandeur ?
6. S'il s'agit d'un nombre flottant sur 32 bits, quel est-il avec la mantisse exprimée en binaire ? En décimal ?
7. S'ils sont sensés coder quatre caractères ASCII ? ISO-8859-1 ? UTF ?

1 $0010\,1000_2 = 00\,101\,000_2 = 050_8 = 40_{10}$, $1001\,1100_2 = 10\,011\,100_2 = 234_8 = (2\times 8+3)\times 8+4 = 156_{10}$, $0100\,0011_2 = 01\,000\,011_2 = 103_8 = 67_{10}$, $1000\,0000 = 2^7 = 128_{10}$.

2 En supposant que le premier octet contient les poids forts : [Ce n'est pas un problème si on suppose que c'est le second octet qui contient les poids forts.] $0\,010\,100\,010\,011\,100_2 = 024234_8 = (((2\times 8+4)\times 8+2)\times 8+3)\times 8+4 = 10\,396_{10}$, $0\,100\,001\,110\,000\,000_2 = 041600_8 = ((4\times 8+1)\times 8+6)\times 8\times 8 = 17\,280_{10}$

Leur somme en binaire :

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

Sa valeur en décimal : $0\,110\,,110\ 000\ 011\,100_2 = 066034_8 = 27676_{10}$. [On peut vérifier facilement que le résultat est correct avec $10\,396_{10} + 17\,280_{10}$.]

3 Si ce sont des nombres en compléments à 2, le premier et le troisième sont des nombres positifs qui ont la même valeur qu'au (1).

Le deuxième et le quatrième sont des nombres négatifs. Commme $C_2(10\,011\,100) = C_1(10\,011\,100) + 1 = 01\,100\,001 + 1 = 01\,100\,010_2 = 142_8 = 98_{10}$ le deuxième représente -98 ; le quatrième vaut -128.

4 Si ce sont des nombres en complément à 2 sur 16 bits ils sont positifs tous les deux puisque leurs bits de signe sont à 0 et ils valent (comme au 2) $10\,396_{10}$ et $17\,280_{10}$. Le résultat de leur addition (effectuée au 2) est aussi un nombre positif et vaut donc aussi 27676_{10}.

5 Toujours en supposant que les bits de poids forts sont dans le premier octet, le nombre vaut $00\,101\,000\,100\,111\,000\,100\,001\,110\,000\,000_2$. Le bit à 1 le plus à gauche est en position numéro 29, il code pour $2^{29} = 2^9 \times 2^{20}$ soit environ $512 \times 10^6 = 5.12 \times 10^8$. L'ordre de grandeur du nombre est donc de 10^8,

6 En supposant qu'on a d'abord le bit de signe, puis l'exposant, puis la partie fractionnelle de la mantisse :

Le signe contient 0 : c'est un nombre positif.

L'exposant contient $01\,010\,001_2 = 121_8 = 73_{10}$. L'exposant vaut donc $-127 + 73 = -54$

La partie fractionnelle de la mantisse contient $100\,111\,000\,100\,001\,110\,000\,000$, la mantisse vaut donc $1.100\,111\,000\,100\,001\,110_2$

La valeur du nombre est donc $1.100\,111\,000\,100\,001\,110_2 \times 10_2{}^{-110110_2}$ en binaire.

La partie fractionnelle de la mantisse est $0.100\,111\,000\,100\,001\,110_2 = 0.470416_8 = 470416x2^{-18}$, donc le nombre vaut $(1 + 470416x2^{-18}) \times 2^{-54}$ en décimal.

7 Le deuxième et le quatrième sont supérieurs à 127 : il ne peuvent pas reprénter des caractères ASCII. La commande `man ascii` permet de constater que le premier code un parentèse ouvrante et le troisième le caractère C. [Si les conditions d'examen ne prévoient pas une table des caractères ASCII, cette dernière partie est facultative.]

En ISO-8859-1 le premier et le troisième codent toujours la parenthèse ouvrante et le C majuscule comme dans le code ASCII ; le deuxième et le quatrième codent des caractères qui n'appartiennent pas au code ASCII. Ce sont tous les deux des *« caractères de contrôle étendus »* ce qui signifie qu'ils ne sont-il pas imprimables. [Si les conditions d'examen ne prévoient pas une table des caractères ISO-8859-1, cette dernière partie est facultative.]

En UTF, le premier octet est inférieur à 128 et code donc le même caractère que l'ASCII ; le second est supérieur ou égal à 128 et devrait donc être le premier octet d'un caractère sur 2 ou 3 octets mais comme le caractère suivant est inférieur à 128, cela signifie qu'il y a une erreur de codage. C'est pareil pour le troisième et le quatrième (ASCII et erreur).

7.10 Supplément : saisir des caractères divers au clavier

Pour taper les caractères qui n'apparaissent pas sur le clavier, il existe une méthode simple pour les obtenir avec une suite de frappe de touches sous X11, avec la touche `Compose`. Cette touche n'est en général pas présente sur le clavier mais les systèmes de gestion de fenêtre Gnome et KDE (ou la commande `xmodmap`) permettent d'utiliser à la place une touche inutilisée du clavier, par exemple une touche `Windows`.

7.10.1 Choisir Compose avec les versions Ubuntu de 2013

Avec les distributions de Linux qui utilisent des systèmes de gestion de fenêtres, on peut choisir sa touche Compose directement avec des clics. Dans le menu de gestion du clavier, choisir la disposition du clavier puis les options ; parmi les options, il y a une entrée pour choisir la position de la touche Compose.

Sur la ligne de commande, on peut utiliser la commande `xmodmap` par exemple avec quelque chose comme `xmodmap -e "keycode 115 = Multi_key"`, la difficulté étant d'identifier le *keycode*, le code interne qui correspond à la touche ; la commande `xev` permet de résoudre ce problème assez facilement.

7.10.2 Taper un caractère non-standard

Les caractères composés s'obtiennent en tapant sur le clavier la touche *Compose* puis deux autres touches. Par exemple, `<Compose> 1 2` donne le caractère

pour '$\frac{1}{2}$'. `<Compose> ' e` donne le caractère 'é'. `<Compose> s s` donne le caratère 'ß'. On devine facilement les caractères à taper : il s'agit en général de l'accent puis de la lettre.

Vous pouvez maintenant utiliser les guillemets français « et », qu'on obtient bien sur avec `Compose < <` et `Compose > >`.

Chapitre 8

Algèbre de Boole

Comment représenter avec deux valeurs des connaissances (propositions) et effectuer des opérations sur ces connaissances ? Par exemple comment représenter et traiter : *si ((un panneau ralentir est présent) et (automobiliste discipliné)) ou (voyant allumé) alors ralentir.*

La solution consiste à définir *ralentir* comme étant une fonction logique qui dépend des variables binaires *panneau ralentir présent*, *automobiliste discipliné* et *voyant allumé*. Pour calculer le résultat de la fonction *ralentir*, on utilise une algèbre, nommée algèbre de Boole.

8.1 Les fonctions logiques

Dans cette algèbre une variable peut prendre deux valeurs, vrai ou faux (codées respectivement par V et F). (On rencontre souvent des auteurs qui codent les valeurs *faux* et *vrai* avec les chiffres 0 et 1. J'ai fait ici le choix d'utiliser F et V pour éviter la confusion avec les chiffres utilisés pour représenter les nombres.) Il y a trois fonctions de base, qui permettent de construire d'autres fonctions : *et*, *ou*, *non*.

Pour chacune d'elles, nous donnons son écriture algébrique ainsi que sa *table de vérité*. La table de vérité d'une fonction logique donne la valeur de la fonction pour toutes les combinaisons des valeurs des variables. Pour une fonction qui contient n variables, la table de vérité comporte 2^n lignes. Il existe plusieurs manières courantes de représenter les fonctions logiques, que nous indiquons devant chaque table de vérité.

La fonction *non* : $f(x) = \overline{x} = \neg x = \tilde{}x$

x	$\overline{x}$
F	V
V	F

La fonction *ou* : $f(x,y) = x + y = x \vee y = x|y$

x	y	$x+y$
F	F	F
F	V	V
V	F	V
V	V	V

La fonction *et* : $f(x,y) = x.y = xy = x \wedge y = x\&y$

x	y	xy
F	F	F
F	V	F
V	F	F
V	V	V

Sur la base de ces trois fonctions, nous pouvons réécrire la fonction du premier exemple et en donner la table de vérité :

$$f(x,y,z) = x.y + z$$

x	y	z	$f(x,y,z)$
F	F	F	F
F	F	V	V
F	V	F	F
F	V	V	V
V	F	F	F
V	F	V	V
V	V	F	V
V	V	V	V

Une fonction logique avec n variables se décrit avec une table de vérité de 2^n lignes. Chaque ligne peut prendre l'une des deux valeurs V et F. Il existe donc 2^{2^n} fonctions logiques différentes avec n variables. Par exemple, il existe quatre fonctions à une seule variable : $f(x) = V$ et $f(x) = F$ qui donnent toujours le même résultat quel que soit la valeur de la variable, $f(x) = x$, l'identité et $f(x) = \overline{x}$, la fonction NON déja vue. D'autres fonctions booléennes existent qu'il est utile de connaître :

La fonction *ni* ou *non-ou* ou *nor* : $f(x,y) = \overline{x+y} = \overline{x}.\overline{y}$

x	y	$\overline{x+y}$
F	F	V
F	V	F
V	F	F
V	V	F

La fonction *non-et* ou *nand* : $f(x,y) = \overline{xy} = \overline{x} + \overline{y}$

x	y	$\overline{x.y}$
F	F	V
F	V	V
V	F	V
V	V	F

La fonction *ou exclusif* ou *xor* : $f(x, y) = x \oplus y = x\overline{y} + \overline{x}y$

x	y	$x \oplus y$
F	F	F
F	V	V
V	F	V
V	V	F

8.1.1 Fonctions logiques et tables de vérité

Pour acquérir un peu de familiarité avec la manipulation des fonctions logiques, il est utile de savoir déterminer la table de vérité de n'importe quelle fonction à partir de son équation et inversement de déterminer l'équation d'une fonction à partir d'une table de vérité.

De l'équation de la fonction à la table de vérité

Pour déterminer la table de vérité d'une façon systématique, le plus simple est de commencer par déterminer n le nombre de variables, qui fixe le nombre de lignes (2^n) et de placer et remplir dans la table les colonnes qui correspondent à ces variables, pour énumérer tous les cas possibles sans en manquer. Par exemple, traitons la fonction :

$$f(x, y, z) = \overline{x}.\overline{y}.\overline{z} + x.\overline{y}.z + x.y.z$$

La fonction a trois variables ; la table de vérité a donc 8 lignes. On commence donc par remplir les trois premières colonnes avec toutes les combinaisons possibles de valeurs pour ces variables :

x	y	z	à faire
F	F	F	
F	F	V	
F	V	F	
F	V	V	
V	F	F	
V	F	V	
V	V	F	
V	V	V	

Un truc pour ne pas oublier de ligne dans la table de vérité : si on remplace dans les trois colonnes les F par des 0 et les V par des 1, on voit qu'on a sur chaque ligne, dans l'ordre, la représentation binaire des nombres entre 0 et 7.

On identifie ensuite les composants de l'équation et on rajoute des colonnes intermédiaires, où on calcule, ligne par ligne, la valeur de ces composants. (Quand on a une certaine familiarité avec les fonctions logiques, on peut se dispenser de certaines colonnes intermédiaires et les calculer de tête.) Avec la fonction f, les colonnes intermédiaires peuvent être (1) $\overline{x}$, (2) $\overline{y}$, (3) $\overline{z}$, (4) $\overline{x}.\overline{y}.\overline{z}$, (5) $x.\overline{y}.z$ et (6) $x.y.z$. La façon qui conduit à se tromper le moins consiste à remplir le tableau colonne par colonne. Ainsi pour remplir la colonne $x.\overline{y}.z$, on repère sur la gauche les colonnes x, $\overline{y}$ et z et sur chaque ligne, on place un V si les trois colonnes d'origine contiennent V, on place un F sinon. La table devient :

x	y	z	$\overline{x}$	$\overline{y}$	$\overline{z}$	$\overline{x}.\overline{y}.\overline{z}$	$x.\overline{y}.z$	$x.y.z$	à faire
F	F	F	V	V	V	V	F	F	
F	F	V	V	V	F	F	F	F	
F	V	F	V	F	V	F	F	F	
F	V	V	V	F	F	F	F	F	
V	F	F	F	V	V	F	F	F	
V	F	V	F	V	F	F	V	F	
V	V	F	F	F	V	F	F	F	
V	V	V	F	F	F	F	F	V	

On continue jusqu'a ajouter la colonne qui correspond à la fonction recherchée.

x	y	z	$\overline{x}$	$\overline{y}$	$\overline{z}$	$\overline{x}.\overline{y}.\overline{z}$	$x.\overline{y}.z$	$x.y.z$	f
F	F	F	V	V	V	V	F	F	V
F	F	V	V	V	F	F	F	F	F
F	V	F	V	F	V	F	F	F	F
F	V	V	V	F	F	F	F	F	F
V	F	F	F	V	V	F	F	F	F
V	F	V	F	V	F	F	V	F	V
V	V	F	F	F	V	F	F	F	F
V	V	V	F	F	F	F	F	V	V

Pour prendre un autre exemple, calculons la table de vérité de la fonction $f(a,b,c) = \overline{\overline{\overline{a}+b}+c}$. On va partir des valeurs des trois variables a, b et c (la table aura donc $2^3 = 8$ lignes), puis ajouter et remplir l'une apres l'autre des colonnes pour les expressions intermédiaires $\overline{a}$, puis $\overline{\overline{a}+b}$ et finalement $\overline{\overline{\overline{a}+b}+c}$. La table complètement remplie sera donc la suivante :

a	b	c	$\overline{a}$	$\overline{a}+b$	$\overline{\overline{a}+b}$	$\overline{\overline{a}+b}+c$	$\overline{\overline{\overline{a}+b}+c}$
F	F	F	V	V	F	F	V
F	F	V	V	V	F	V	F
F	V	F	V	V	F	F	V
F	V	V	V	V	F	V	F
V	F	F	F	F	V	V	F
V	F	V	F	F	V	V	F
V	V	F	F	V	F	F	V
V	V	V	F	V	F	V	F

De la table de vérité à l'équation

Pour déterminer une équation à partir d'une table de vérité, une méthode simple consiste, pour chaque ligne où la fonction vaut V, à rajouter un *terme* qui exprime la valeur des variables sur cette ligne. La fonction sera un OU entre ces *termes*, puisque l'un d'entre eux sera vrai dans chacun des cas où la fonction est vraie. Repartons de la table de vérité de la fonction précédente, en omettant les colonnes que nous avons utilisées pour les calculs intermédiaires :

a	b	c	f
F	F	F	V
F	F	V	F
F	V	F	V
F	V	V	F
V	F	F	F
V	F	V	F
V	V	F	V
V	V	V	F

Il y a trois lignes où la fonction a la valeur vraie : l'expression logique qui traduira la fonction sera donc un *ou* entre trois termes. La première ligne où la fonction est vraie est

a	b	c	f
F	F	F	V

La fonction sera donc vraie quand le terme $\overline{a}.\overline{b}.\overline{c}$ sera vrai. De même la seconde ligne où la fonction est vraie :

a	b	c	f
F	V	F	V

correspond au cas où le terme $\overline{a}.b.\overline{c}$ sera vrai. La troisième ligne :

a	b	c	f
V	V	F	V

correspond à $a.b.\overline{c}$. La fonction sera donc vraie quand l'un de ces trois cas se produira et on peut la traduire par l'équation $f(a, b, c) = \overline{a}.\overline{b}.\overline{c} + \overline{a}.b.\overline{c} + a.b.\overline{c}$.

La méthode démontre au passage que n'importe quelle fonction logique peut s'exprimer avec une combinaison de *et*, *ou* et *non*, puisque n'importe quelle fonction logique peut s'exprimer par une table de vérité et que nous pouvons traduire de cette façon n'importe quelle table de vérité en équation à base de *Et*, de *Ou* et de *Non*. On dit que ces opérateurs logiques sont *universels*. Il y a d'autres combinaisons d'opérateurs universels (voir exercice).

Cette méthode pour obtenir une équation à partir d'une table de vérité est simple mais elle ne conduit pas nécessairement à l'équation la plus simple. Par exemple, la fonction $f(a, b) = V$ (qui est toujours vraie, quelles que soient les valeurs de a et b) a la table de vérité suivante :

a	b	f
F	F	V
F	V	V
V	F	V
V	V	V

L'application de notre méthode nous donne, pour la fonction l'équation

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

qui n'est vraiment pas plus simple que l'équation de départ !

Simplification algébrique des expressions logiques

Des propriétés peuvent être démontrées qui sont utiles pour la manipulation des expressions, résumées dans la table suivante : (Une méthode de démonstration longue mais simple pour démontrer ces théorèmes consiste à écrire la table de vérité des parties gauche et droite de l'égalité et de constater leur égalité. Elle manque cependant d'élégance ; de plus, on ne peut pas l'employer quand le nombre de variables est indéterminé, comme dans le théorème de Morgan.)

1 variable	$\overline{\overline{x}} = x$ $x + F = x$ $x.F = F$ $x + V = V$ $x.V = x$	
2 variables	$x + y = y + x$ $x.y = y.x$	commutativité
3 variables	$x + y + z = (x + y) + z = x + (y + z)$ $x.y.z = (x.y).z = x.(y.x)$ $x.(y + z) = x.y + x.z$ $x + (y.z) = (x + y).(x + z)$	associativité distributivité
nb quelconque	$\overline{x + y + ... + z} = \overline{x}.\overline{y}.....\overline{z}$ $\overline{x.y.....z} = \overline{x} + \overline{y} + ... + \overline{z}$	Théorème de Morgan

Chacune de ces expressions peut se traduire en français et correspond à une propriété évidente (sauf peut-être pour le théorème de Morgan) des expressions logiques.

Attention à ne pas confondre les opérateurs logiques *Et* et *Ou* avec l'addition et la multiplication (qu'on représente avec les mêmes caractères mais dans un autre contexte) : la différence la plus déroutante est que le *Ou* est distributif sur le *Et* ($x + (y.z) = (x + y).(x + z)$) alors que l'addition ne l'est pas sur la multiplication.

Une propriété magnifique de l'algèbre de Boole, qui apparait bien dans cette table, est que si on échange tous les V et les F, tous les ET et les OU, le système continue à fonctionner de la même manière.

8.2 Simplification des expressions logiques

Comme nous l'avons vu, il y a plusieurs expressions logiques qui sont toujours égales entre elles. Quand on réalise un circuit électronique, il est souhaitable d'utiliser l'expression la plus simple, puisque c'est celle qui utilisera le moins de transistors.

Nous montrons brièvement dans cette section deux méthodes de simplification. L'une des méthodes est *algébrique* : elle n'utilise que des manipulations de symboles. L'autre méthode utilise un support graphique, dit des *tables de Karnaugh* (prononcer *Karnaugh* comme *carno*) et permet de trouver la forme simplifiée des expressions à trois ou quatre variables.

8.2.1 Simplifications algébriques des expressions logiques

En utilisant les axiomes et les théorèmes de l'algèbre de Boole, on peut simplifier les fonctions logiques. L'opération ressemble à la factorisation des polynomes, en utilisant la ressemblance entre le *et* de la logique et la multiplication de l'arithmétique d'une part et entre le *ou* de la logique et l'addition d'autre part.

On peut aussi s'appuyer à l'occasion sur le fait que $x.\overline{x}$ vaut *Faux* alors que $x + \overline{x}$ vaut *Vrai*.

Par exemple, à partir de la fonction
$f(x, y, z) = x.y.z + x.(y.\overline{z} + \overline{y}.z)$
on développe en :
$f(x, y, z) = x.y.z + x.y.\overline{z} + x.\overline{y}.z$
Il y a deux factorisations tentantes, par $x.y$ et par $x.z$. Utilisons le fait $a = a + a$ pour dupliquer le terme qui entre dans les deux factorisations :
$f(x, y, z) = x.y.z + x.y.z + x.y.\overline{z} + x.\overline{y}.z$
Ré-ordonnons les termes :
$f(x, y, z) = x.y.z + x.y.\overline{z} + +x.y.z + x.\overline{y}.z$
Factorisons par $(x.y)$ et par $x.z$:

$f(x,y,z) = x.y.(z+\overline{z}) + +x.z.(y+\overline{y})$
Puisque $a + \overline{a} = V$:
$f(x,y,z) = x.y.V + +x.z.V$
Puisque $a.V = a$:
$f(x,y,z) = x.y + x.z$
Factorisons par x :
$f(x,y,z) = x.(y+z)$

En partant d'une expression qui contenait 9 opérateurs logiques, nous avons réussi à obtenir une expression équivalente qui n'utilise plus que deux opérateurs logiques.

Un deuxième exemple :

$$\begin{aligned}
a.(a+b) &= a.a + a.b && \text{développement}\\
&= a + a.b && \text{puisque } x.x = x\\
&= a.V + a.b && \text{puisque } x = x.V\\
&= a.(V+b) && \text{factorisation par } a\\
&= a.V && \text{puisque } V + x = V\\
&= a && \text{puisque } x.V = x
\end{aligned}$$

La mise en œuvre des simplifications algébriques demande une certaine pratique et un peu de flair. Il convient de répéter les exercices proposés jusqu'à être en mesure de les faire sans recourir aux corrigés. La prochaine partie montre une façon mécanique de simplifier certaines expressions logiques.

8.2.2 Simplification des expressions logiques avec les tables de Karnaugh

La méthode des tables de Karnaugh permet de trouver mécaniquement la forme en minterme la plus simple de n'importe quelle expression logique qui ne contient pas plus de quatre variables.

La méthode de Karnaugh est basée sur l'identité $ax + a\overline{x} = a$. C'est une méthode graphique qui consiste à placer des termes les uns à coté des autres quand ils n'ont qu'une variable qui change de valeur, de façon à mettre en évidence cette identité.

Deux termes sont dits adjacents si l'état des variables qui les composent sont identiques à l'état d'une variable près. Par exemple $\overline{x}.y.\overline{z}.t$ est adjacent à $\overline{x}.y.\overline{z}.\overline{t}$, puisque seul l'état de la variable t est différent. En revanche $x.\overline{y}.z.t$ et $x.y.\overline{z}.t$ ne le sont pas puisque les deux variables y et z diffèrent entre les deux termes.

La méthode consiste à :

1. Construire la table de Karnaugh,
2. Réaliser des groupements de 2^n termes en
   — minimisant le nombre de groupements,
   — maximisant le nombre de termes par groupement.

3. Pour chaque regroupement de 2^n termes, éliminer les n variables qui changent d'état et déduire le terme résultant composé du produit des variables directes ou inverses qui n'ont pas changé d'état.
4. Écrire l'expression logique finale qui est la réunion des termes trouvés à l'étape précédente.

Construction de la table de Karnaugh

Nous donnons ici les tables de Karnaugh pour 2, 3 et 4 variables.

$b \backslash a$	F	V
V		
F		

$c \backslash ab$	FF	FV	VV	VF
F				
V				

$cd \backslash ab$	FF	FV	VV	VF
FF				
FV				
VV				
VF				

En haut à gauche on rappelle quelles sont les variables considérées : dans la dernière table, chaque colonne rendra compte des valeurs de variables a et b, chaque ligne de celles des variables c et d.

Chaque case joue le même rôle qu'une ligne dans une table de vérité : elle indique une combinaison de la valeur des variables. Par exemple, la case en bas à droite de la dernière table est dans la colonne VF, donc pour laquelle a est *vrai* et b est *faux* et sur la ligne VF, donc pour laquelle c est *vrai* et d est faux. Elle correspond donc au terme $a.\overline{b}.c.\overline{d}$.

Les lignes et les colonnes sont placées dans un ordre bien précis, pour que les termes adjacents soient placés dans des cases connexes : entre deux cases voisines, il n'y a qu'une seule variable qui change de valeur. *Attention, cet ordre est différent de celui utilisé pour les tables de vérité.*

Enfin, il faut considérer que la table est située sur un cylindre : la colonne la plus à droite et la colonne la plus à gauche sont voisines. La colonne la plus à gauche représente $\overline{a}.\overline{b}$ et la colonne la plus à droite $a.\overline{b}$: entre les deux, seule la variable b change de valeur. C'est la même chose entre la première et la dernière ligne.

Remplissage de la table de Karnaugh

Dans chacune des cases on écrit la valeur de la fonction, obtenue par la table de vérité. On fait une transposition de la table de vérité vers la table de Karnaugh. Prenons comme exemple la fonction $f(a,b,c,d)$ décrite par la table de vérité suivante :

a	b	c	d	f(a, b, c, d)
F	F	F	F	V
F	F	F	V	V
F	F	V	F	F
F	F	V	V	V
F	V	F	F	F
F	V	F	V	F
F	V	V	F	V
F	V	V	V	V
V	F	F	F	V
V	F	F	V	F
V	F	V	F	F
V	F	V	V	V
V	V	F	F	F
V	V	F	V	F
V	V	V	F	V
V	V	V	V	V

De la table de vérité, nous pouvons extraire une expression algébrique de la fonction

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

L'expression algébrique avant simplification utilise huit *Ou*, seize *Non* et vingt sept *Et*.

On peut transcrire la table de vérité dans la table de Karnaugh, ce qui donne la table remplie suivante :

$cd\backslash ab$	FF	FV	VV	VF
FF	V			
FV	V			V
VV	V	V	V	V
VF		V	V	

Nous pourrions bien sur, à partir de cette table traduire chaque case par un terme qui rende compte de la valeur des quatre variables pour cette case : nous retomberions alors sur l'équation non simplifiée déja vue. On peut aussi utiliser le fait qu'entre des cases voisines, il n'y a qu'une seule variable qui change de valeur.

Groupement

Quand deux cases voisines contiennent toutes les deux la valeur V, cela signifie qu'elle est vraie, quelle que soit la valeur de la variable qui change de valeur de vérité entre les deux cases.

Les deux premières cases de la colonne de gauche contiennent toutes les deux V. Elles correspondent à l'expression $\overline{a}.\overline{b}.\overline{c}.\overline{d}$ pour l'une et à l'expression $\overline{a}.\overline{b}.\overline{c}.d$. pour l'autre. On peut les prendre en compte toutes les deux avec l'expression $\overline{a}.\overline{b}.\overline{c}$, la valeur de d n'est pas utile. Sous une forme algébrique, on peut voir ceci comme la factorisation :

$$\begin{aligned}\overline{a}.\overline{b}.\overline{c}.\overline{d} + \overline{a}.\overline{b}.\overline{c}.d \quad &= \overline{a}.\overline{b}.\overline{c}.(\overline{d} + d)\\ &= \overline{a}.\overline{b}.\overline{c}.V\\ &= \overline{a}.\overline{b}.\overline{c}\end{aligned}$$

mais nous n'avons pas eu besoin de deviner le terme par lequel il convenait de factoriser (ce n'est pas toujours aussi facile que dans cet exemple) : c'est la disposition des cases dans la table qui nous l'a indiqué.

Regardons maintenant la troisième ligne, dont toutes les cases contiennent V : cela signifie qu'on peut couvrir ces quatre cas avec l'expression $c.d$, sans tenir compte des valeurs de a ni de b.

Il nous reste trois cas où la fonction est vraie : les deux cas de la ligne du bas pourraient se traduire par $b.c.\overline{d}$ mais on peut faire mieux : en regroupant ces deux cas avec ceux de la ligne précédente, on voit qu'en réalité la valeur de d est sans importance et qu'on peut couvrir ces quatre cas avec l'expression plus simple $c.b$. Le fait que les deux cases avec lesquelles nous les avons regroupé étaient déja couvertes n'est pas important, puisque $V + V = V$. D'un point de vue algébrique, nous avons utilisé le fait que $c.d + b.c.\overline{d} = c.d + b.c$.

De la même manière, le dernier cas non couvert, sur la droite de la deuxième ligne peut se regrouper avec celui du dessous *et ceux de la première colonne* : il correspond à $\overline{b}.d$.

L'expression finale simplifiée de la fonction est donc :

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

Si nous n'avons pas fait d'erreurs lors du calcul des regroupements, nous avons la certitude qu'il s'agit de l'équation la plus simple de la fonction sous forme de mintermes.

Extensions des tables de Karnaugh

Il existe des variantes de la méthode simple d'utilisation des tables de Karnaugh que nous avons présentée ici. La plus importante est est celle des situations *indifférentes* : certains cas de la fonction logique peuvent être sans intérêt pour nous : le dispositif que nous construisons fonctionnera de la même manière que la fonction soit vraie ou fausse dans ce cas.

Quand on a des cas indifférents, on les note avec des points d'interrogation dans la table de Karnaugh. Quand on procède aux groupements, on peut utiliser ces cases si cela permet de faire de plus grands groupes avec les les cases qui contiennent vrai mais on n'est pas obligé de les recouvrir.

Il existe une variante des tables de Karnaugh pour les fonctions à 5 ou 6 variables et on peut traduire certaines dispositions de cases par des *Ou exclusif* mais nous ne l'utiliserons pas dans le cours.

Résumé de la façon de simplifier une expression logique à quatre variables avec une table de Karnaugh

On commence par remplir la table Karnaugh avec les valeurs de vérité, puis on recherche les plus grands rectangles ou carrés qui ne contiennent que des V : il vont se traduire par des termes. Plus le rectangle est grand et plus le terme sera simple. Quant on aura traduit tous les V de la table dans des termes (on dit qu'on les a *couverts*), on peut s'arrêter ; sinon il faut continuer.

La table peut être remplie de V comme dans l'exemple suivant :

$cd\backslash ab$	FF	FV	VV	VF
FF	V	V	V	V
FV	V	V	V	V
VV	V	V	V	V
VF	V	V	V	V

Dans ce cas, quelle que soit la valeur des variables, la fonction vaut vrai. On peut l'écrire $f(a, b, c, d) = V$. On a traduit tous les V de la table par le terme, on peut donc s'arrêter.

On cherche ensuite les rectangles 4×2 complètement remplis de V. Il se traduisent par des termes composés d'une seule variable. Par exemple

$cd\backslash ab$	FF	FV	VV	VF
FF		V	V	
FV		V	V	
VV		V	V	
VF		V	V	

correspond au terme b

$cd\backslash ab$	FF	FV	VV	VF
FF	V	V	V	V
FV				
VV				
VF	V	V	V	V

correspond au terme $\overline{d}$ On a intérêt à utiliser des V de la table plusieurs fois quand cela permet d'agrandir des rectangles. Ainsi dans la table

$cd\backslash ab$	FF	FV	VV	VF
FF	V	V	V	V
FV		V	V	
VV		V	V	
VF	V	V	V	V

on a les *deux* rectangles des deux exemples précédents ; elle correspond donc à la fonction $f(a, b, c, d) = b + \overline{d}$

Pour les termes qu'on a pas encore couverts, on essaye de les placer dans des rectangles 4×1 ou des carrés 2×2 ; ils vont se traduire par des termes avec deux variables. Ainsi

$cd\backslash ab$	FF	FV	VV	VF
FF				
FV	V	V	V	V
VV				
VF				

correspond à $\overline{c}.d$

$cd\backslash ab$	FF	FV	VV	VF
FF	V	V		
FV	V	V		
VV				
VF				

correspond à $\overline{a}.\overline{c}$.

$cd\backslash ab$	FF	FV	VV	VF
FF				
FV		V	V	
VV		V	V	
VF				

correspond à $b.d$. De nouveau, il faut utiliser tous les V qui permettent d'agrandir le rectangle, même s'ils sont déja couverts. Ainsi

$cd\backslash ab$	FF	FV	VV	VF
FF	V	V		
FV	V	V	V	
VV		V	V	
VF				

contient les deux carrés 2×2 des exemples précédents et correspond donc à $\overline{a}.\overline{c} + b.d$

Attention, on ne peut pas traduire directement un rectangle 2×3 par une seule expression ; il faut le considérer comme deux carrés 2×2 qui se recouvrent partiellement. Ainsi la table

$cd \backslash ab$	FF	FV	VV	VF
FF	V	V	V	
FV	V	V	V	
VV				
VF				

contient deux carrés 2×2 qui se recouvrent ; elle va se traduire par $f(a, b, c, d) = \overline{a}.\overline{c} + b.\overline{c}$.

Pour les V qui restent, on essaye de les placer dans des rectangles 2×1, toujours en utilisant plusieurs fois des V déjà couverts si cela permet de construire un rectangle. Ils vont être traduits par des termes à 3 variables. Ainsi

$cd \backslash ab$	FF	FV	VV	VF
FF				
FV	V	V		
VV				
VF				

correspond à $f(a, b, c, d) = \overline{a}.\overline{c}.d$

$cd \backslash ab$	FF	FV	VV	VF
FF				
FV		V	V	
VV				
VF				

correspond à $f(a, b, c, d) = b.\overline{c}.d$

$cd \backslash ab$	FF	FV	VV	VF
FF				
FV	V	V	V	
VV				
VF				

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

Finalement les V complètement isolés sont traduits par un terme à quatre variables.

8.2.3 Simplification des expressions logiques en pratique

En pratique, la simplification des expressions logiques est un problème difficile à traiter mais dans lequel les ordinateurs sont en général bien plus efficaces que nous. Des applications logicielles existent pour faire ce travail.

Il faut cependant acquérir une certaine familiarité avec les simplifications algébriques ordinaires. Les tables de Karnaugh, présentent de plus l'avantage de faire de bonnes questions d'examen.

8.3 Résumé

A partir des valeurs *Vrai* et *Faux*, on peut construire des fonctions logiques de variables booléennes. Ces expressions peuvent se manipuler sous forme algébrique ou sous forme de tables de vérité. Pour chaque fonction logique, on peut construire un circuit électronique équivalent.

On peut simplifier les expressions logiques avec des manipulations algébriques ou des tables de Karnaugh.

8.4 Exercice

Exercice 8.1 : Soit $f(x,y,z) = x + \overline{y}.z$. Calculer $\overline{f}$ avec des manipulations algébriques.

Exercice 8.2 : Vérifiez le résultat de l'exercice précédent en construisant les tables de vérité de f et de $\overline{f}$ (avec toutes les colonnes intermédiaires).

Exercice 8.3 : (Exercice type ; long) Construire la table de vérité de la fonction $f(a,b,c,d) = a.b + \overline{a}.\overline{b} + c.d + \overline{c}.\overline{d}$

Exercice 8.4 : Déterminer une équation de la fonction décrite par la table de vérité suivante.

a	b	c	f
F	F	F	V
F	F	V	V
F	V	F	V
F	V	V	V
V	F	F	F
V	F	V	F
V	V	F	F
V	V	V	V

Simplifier algébriquement cette fonction.

Exercice 8.5 : (à faire) A l'aide d'une table de vérité, regarder si *Non-Et* est associatif (c'est à dire si $\overline{a.\overline{\overline{b.c}}} = \overline{\overline{\overline{a.b}}.c}$

Exercice 8.6 : (à faire) On montre que toutes les fonctions logiques peuvent s'écrire seulement avec des *Non-Et* en notant que les trois fonctions de base peuvent se réaliser des *Non-Et* :

$$
\begin{aligned}
\overline{x} &= \overline{x.x} \\
a.b &= \overline{\overline{a.b}.\overline{a.b}} \\
a+b &= \overline{\overline{a.a}.\overline{b.b}}
\end{aligned}
$$

Démontrer ces trois égalités avec des tables de verités.

Exercice 8.7 : Trouver les trois équations qui permettent de prouver que toutes les fonctions logiques peuvent s'écrire en utilisant seulement des *Non-Ou*. (On peut s'inspirer des équations de l'exercice précédent.)

Exercice 8.8 : En fait, une seule équation aurait suffi dans l'exercice précédent, pour montrer qu'on pouvait calculer un *Non-Et* en utilisant seulement des *Non-Ou*. Laquelle ?

Exercice 8.9 : Imaginons que pour notre langage, nous utilisions l'algèbre de Boole. Que repondrait alors la sage femme lorsque le nouveau père lui demande : "Alors, c'est un garçon *ou* une fille ?".

Exercice 8.10 : Dans quel cas la réponse de l'infirmière serait elle différente si le père lui demandait :"C'est un garçon *ou exclusif* une fille ?".

Exercice 8.11 : Simplifier algébriquement les expressions algébriques de gauche, jusqu'à obtenir l'expression de droite.

A simplifier	Forme simple
(a) $(p+r).(p+s).(q+r).(q+s)$	$p.q+r.s$
(b) $(a.c+b.\overline{c}).(\overline{a}+\overline{c}).b$	$b.\overline{c}$
(c) $a.\overline{b}.\overline{c}+a.b.\overline{c}+a.b.c$	$a.(b+\overline{c})$
(d) $b.d+c.d+\overline{c}.d+\overline{a}.b.\overline{c}.\overline{d}+a.b.\overline{c}$	$b.\overline{c}+d$
(e) $(a+b).(\overline{a}+b)$	b
(f) $(x.\overline{y}+z).(x+\overline{y}).z$	$(x+\overline{y}).z$
(g) $a+a.b.c+\overline{a}.b.c+\overline{a}.b+a.d+a.\overline{d}$	$a+b$
(h) $\overline{(p.q+\overline{p}.r)}$	$p.\overline{q}+\overline{p}.\overline{r}+\overline{q}.\overline{r}$
(i) $\overline{(\overline{p}+q.(\overline{(r+\overline{s})}))}$	$p.(\overline{q}+r+\overline{s})$

Exercice 8.12 : Utiliser les tables de Karnaugh suivantes pour obtenir la forme algébrique simplifiée de la fonction qu'elles décrivent :

(a)

$cd\backslash ab$	FF	FV	VV	VF
FF	V	V	V	V
FV	V			V
VV	V			V
VF	V	V	V	V

(b)

$cd\backslash ab$	FF	FV	VV	VF
FF		V	V	
FV	V			V
VV	V			V
VF		V	V	

(c)

$cd\backslash ab$	FF	FV	VV	VF
FF		V	V	
FV	V	V	V	
VV	V	V	V	V
VF		V	V	

(d)

$cd\backslash ab$	FF	FV	VV	VF
FF	V			V
FV		V	V	
VV	V	V	V	
VF	V			

Chapitre 9

Des dispositifs pour calculer

Dans ce chapitre, nous examinons comment combiner les signaux logiques pour représenter des informations et effectuer des calculs. Ce dont on a besoin, c'est de quelque chose susceptible de *controler* un signal.

Nous présentons d'abord les relais qui présentent l'avantage de la simplicité, puis nous nous étendons un peu plus longuement sur les transistors, qu'on utilise dans les ordinateurs actuels.

9.1 Les relais

Les relais sont contruits à base d'*électro-aimants*. Un électro-aimant est simplement un fil qui entoure un coeur métallique ; quand il n'y a pas de courant qui passe dans le fil, le coeur métallique n'est pas magnétique ; en revanche, quand il y a du courant qui traverse le fil, le coeur métallique se comporte comme un aimant. L'électro-aimant est utilisé pour déplacer une lame métallique mobile de manière à controler le passage du courant dans un autre circuit, comme suggéré dans les schémas de la figure 9.1.

Avec des relais, on peut construire des circuits qui effectuent les calculs de la logique booléenne, si on utilise la circulation du courant pour encoder les valeurs *vrai* et *faux*; ainsi le circuit de la figure 9.1 se comporte comme un *Non* : s'il y a du courant entre a et b, alors il n'y en a pas entre X et Y ; inversement, s'il n'y en a pas entre a et b, alors il circule entre X et Y. Un autre exemple de circuit est celui de la figure 9.2 qui utilise deux relais.

Considérons la figure 9.2 et décidons de coder *Vrai* avec du courant qui circule et *Faux* avec du courant qui ne circule pas. Pour que le courant circule entre x_1 et x_2, (et donc que $x =$*Vrai*), il faut qu'il n'en circule ni entre a_1 et a_2, ni entre b_1 et b_2 (et donc que $a =$*Faux* et $b =$*Faux*. On a donc alors un circuit qui calcule $x = \overline{a}\overline{b}$.

Les relais ont été utilisés pour construire des calculateurs entre 1920 et 1940

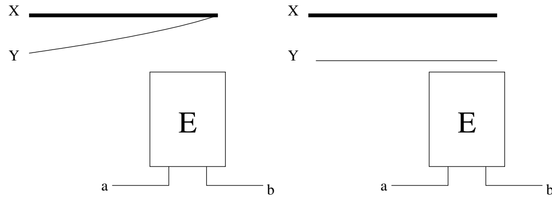


FIGURE 9.1 – Quand il n'y a pas de courant entre a et b, la lame mobile (reliée à Y) est en contact avec la lame fixe (reliée à X) et le courant peut circuler (dans le relai de gauche). En revanche, quand le courant circule entre a et b, l'électroaimant attire la lame mobile et le courant ne peut pas passer entre X et Y (dans le relai de droite).

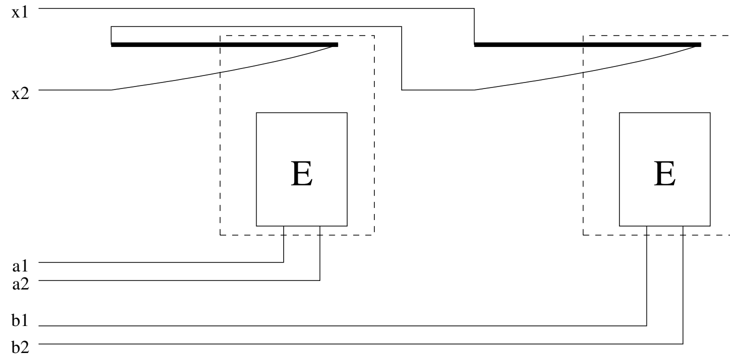


FIGURE 9.2 – S'il y a du courant qui circule entre a_1 et a_2 ou entre b_1 et b_2 (ou les deux à la fois), alors il y a un relai en position ouverte et le courant ne peut pas circuler entre x_1 et x_2.

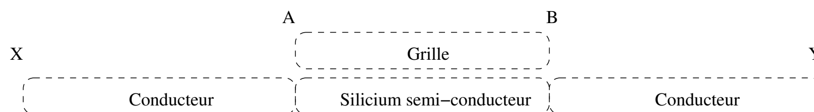


FIGURE 9.3 – Une vision schématique d'un transistor

mais des considérations électriques et mécaniques limitent la vitesse de commutation entre les deux positions du relais (il faut un temps de l'ordre du dixième ou du centième de seconde pour déplacer le bras mobile) ce qui ne permet pas de calculer très vite. On les utilise encore, notamment pour controler, à partir d'un circuit qui utilise des tensions basses (comme une unité de calcul), des circuits qui utilisent des tensions plus fortes (comme un moteur).

A partir des années 1940, on s'est mis à utiliser pour la même fonction des lampes électroniques qui jouent le même rôle que les relais mais n'ont pas de partie mécanique et commutent beaucoup plus rapidement. Elles se composent d'un filament qui émet des électrons (c'est la *source*) en direction d'un récepteur (le *drain*) ; entre les deux, on intercale une *grille* ; quand on met la grille sous tension, elle crée un champs électrique qui bloque le passage des électrons entre la source et le drain ; la grille joue ici le même rôle que l'électro-aimant du relai. Les inconvénients principaux des lampes, c'est qu'elles sont grosses, qu'elles consomment beaucoup de courant et surtout qu'elles tombent en panne, comme les lampes que nous utilisons pour nous éclairer ; à partir de la découverte du transistor au milieu des années 1950, on les a utilisé à la place des lampes.

9.2 Les transistors

Nous ne présentons ici que les transistors qu'on trouve dans les circuits VLSI et nous nous limitons à la variante CMOS.

9.2.1 La structure d'un transistor

Le transistor utilise un matériau dit *semi-conducteur* : ce matériau est isolant ou conducteur suivant qu'on lui applique ou pas une tension. Le principal transistor utilisé dans les ordinateurs est le transistor *MOSFET*, fondé sur du silicium.

On voit sur la figure 9.3 une vision schématique de la structure d'un transistor : le courant pourra circuler ou pas entre X et Y si le silicium semi-conducteur est dans l'état conducteur ou pas ; ce sera le cas selon qu'il y aura une tension ou pas entre A et B. Il se comporte donc comme un interrupteur reliant X à Y sous le contrôle de AB, comme le relai de la figure 9.1. Quand il se comporte comme un interrupteur ouvert, on dit que le transistor est *bloquant*; quand c'est comme un interrupteur fermé, on dit qu'il est *passant*.

FIGURE 9.4 – Une représentation schématique des transistors N (à gauche) et P (à droite).

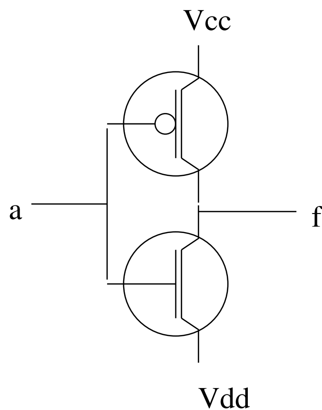


FIGURE 9.5 – Un inverseur CMOS

9.2.2 Les transistors N et P

Il y a deux types de transistors : N et P (comme *Négatif* et *Positif*) qui se comportent de façon inverse : un transistor N sera passant quand il n'y aura pas de tension sur la grille, alors qu'un transistor P le sera quand il y aura une tension. On les représente couramment par les deux symboles de la figure 9.4.

Avec ces transistors N et P, on peut construire des fonctions de calcul en technologie CMOS. On utilise deux niveaux de tensions, V_{CC} et V_{DD} pour coder les deux valeurs de la logique booléenne. Les autres niveaux de tensions sont indéterminés ; ils ne codent rien et on conçoit les circuits de manière qu'ils ne puissent pas apparaître. On parle alors d'électronique *numérique* ou *digitale*.

9.2.3 L'inverseur CMOS

On se sert, par exemple de la tension V_{CC} pour représenter *Vrai* et V_{DD} pour *Faux*. Nous pouvons réaliser un circuit qui calcule la fonction *non* avec le dispositif de la figure 9.5. Quand il n'y a pas de tension en a, le transistor du haut est passant et celui du bas est bloquant : f est donc connecté à V_{CC}. Inversement, quand a est à V_{CC}, le transistor du haut est bloquant, celui du bas est passant, si bien que f est connecté à V_{DD}.

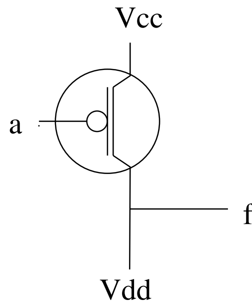


FIGURE 9.6 – Un montage à éviter en CMOS.

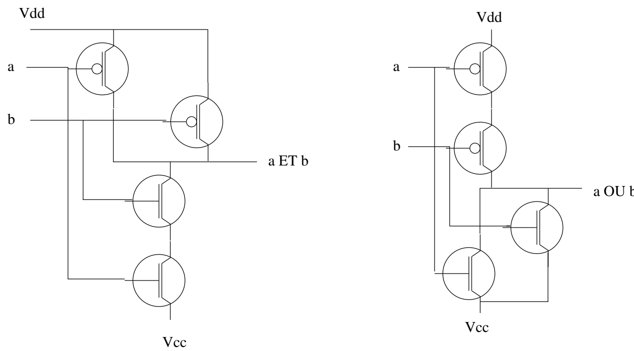


FIGURE 9.7 – Un *et* et un *ou* réalisés en CMOS.

En pratique, V_{CC} sera relié à l'alimentation et V_{DD} à la masse. Pour cette raison, on évite de monter des circuits comme celui de la figure 9.6 : quand il y a une tension en a, le transistor est bloquant, la sortie f est reliée à V_{dd} et le signal est inversé. En revanche, quand il n'y a pas de tension en a, le transistor est passant, le courant peut circuler librement entre V_{CC} et V_{DD} et le circuit fond (comme, dans une installation électrique classique, quand on fait se toucher les deux fils d'une prise électrique).

9.2.4 Le ET et le OU en CMOS

Les fonctions *et* et *ou* peuvent se réaliser de la même manière en combinant des transistors N et P comme dans la figure 9.7. Vous pouvez vérifier que les circuits ont été conçus avec soin pour que la sortie soit connectée soit à V_{DD}, soit à V_{CC} suivant les tensions présentes en entrées. En revanche, quelle que soient les valeurs de a et b, on n'a jamais de connexion entre V_{CC} et V_{DD}. Si c'était le cas, là aussi courant circulerait librement et le circuit fondrait.

Pour des fonctions plus complexes, on peut construire directement des circuits CMOS qui les calculent, ou bien on peut combiner les trois circuits que

nous venons de voir.

9.2.5 Réalisation des circuits

La plupart des transistors sont à l'heure actuelle réalisés dans des circuits intégrés fondés sur le silicium. Le processus de fabrication commence par la réalisation d'un cylindre de silicium cristallin très pur qui est ensuite découpé en fines tranches (des *wafers*) de quelques dizaines de centimètres de diamètre.

La tranche de silicium passe ensuite par une série d'étapes de fabrication de chaque couche : elle est recouverte de résine photo-sensible qui durcit à la lumière ; la tranche est ensuite illuminée dans certaines zones seulement à l'aide d'un masque. La résine qui est restée à l'ombre est lessivée avec un solvant doux. Le silicium qui n'est plus protégé par la résine est alors modifié : soit il est oxydé (l'oxyde de silicium est isolant), soit il est dopé en le bombardant d'atomes qui vont en modifier les propriétés et le rendre semi-conducteur, soit une couche de métal (conducteur) y est déposée. La résine qui reste est alors retirée avec un solvant plus énergique et on recommence pour la couche suivante.

Finalement, la tranche est découpée en dés (des *dies* en anglais), sur lequel sont fixés des pattes et le tout est coulé dans un boitier en plastique.

Les techniques de fabrication actuelles permettent de placer quelques millions de transistors dans un circuit intégré : on parle de circuits VLSI comme *Very Large Scale Integrated Circuit.*

9.3 Exercices

Exercice 9.1 : Si on adopte le codage inverse de celui du texte, c'est à dire qu'on code *Vrai* avec le courant qui ne circule pas et *Faux* avec le courant qui circule, quelle est la fonction calculée par le circuit de la figure 9.2

Exercice 9.2 : Comment construire avec des relais un circuit qui calcule un *OU*?

Exercice 9.3 : Dessiner un circuit CMOS qui calcule la fonction *ou exclusif.*

Exercice 9.4 : Combiner les circuits CMOS qui calculent un *ou* et un *non* pour obtenir un circuit qui calcule un *Non-Ou* en utilisant 6 transistors. (Voir aussi l'exercice suivant.)

Exercice 9.5 : Concevoir un circuit qui n'utilise que 4 transistors pour calculer la fonction *Non-Ou.* (Voir aussi l'exercice précédent.)

Exercice 9.6 : Que calcule le circuit CMOS de l'exercice précédent si on adopte la convention que V_{CC} code la valeur logique *Faux* et V_{DD} la valeur *Vrai* ?

Chapitre 10

Portes logiques et circuits combinatoires

Ce chapitre présente les circuits *combinatoires*, dont la sortie à un moment donné ne dépend que des entrées. On utilise ces circuits notamment pour faire des calculs mais on ne peut pas les utiliser pour fabriquer des mémoires.

10.1 Codage des chiffres

On utilise les fonctions logiques pour coder les chiffres 0 et 1 que manipulent nos ordinateurs. Ceci va nous permettre de concevoir des circuits pour effectuer des calculs.

Notons qu'il existe plusieurs niveaux de codages différents : à la base, il y a des transistors qui sont passant ou bloquant suivant des niveaux de tensions V_{CC} et V_{DD} ; avec ces niveaux de tensions, on peut coder les valeurs *Vrai* et *Faux* : deux codages distincts sont possibles. De même, dans les circuits on peut utiliser les valeurs logiques *Vrai* et *Faux* pour coder les chiffres 0 et 1. Dans les langages de programmation de plus haut niveau, on a une autre représentation des valeurs logiques *Vrai* et *Faux*, à partir de valeurs représentées dans la mémoire.

On suppose souvent, comme nous le ferons ici que l'absence de tension est utilisée pour représenter le *Faux* de la logique, qui sert à représenter le chiffre 0. De même nous utiliserons V_{CC} pour représenter le *Vrai* de la logique et pour le chiffre 1. Les autres codages sont aussi possibles et pratiques dans certaines occasions.

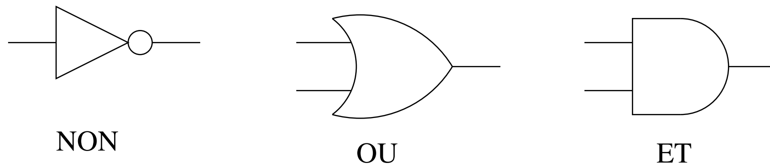


FIGURE 10.1 – Les circuits de base

10.2 Un exemple : conception d'un additionneur

Nous allons montrer les étapes de la conception d'un circuit destiné à effectuer des additions.

10.2.1 Les circuits de base

Dans la figure 10.1, on voit les trois circuits que nous utilisons comme briques de base : à gauche, les traits indiquent les *entrées* du circuit ; à droite, le circuit produit une sortie.

Nous utilisons la convention anglo-saxonne qui fait dessiner les circuits qui calculent des fonctions différentes avec des formes spécifiques. Le *Non* se dessine avec un petit rond blanc ; le triangle qui le précède rappelle que l'inverseur amplifie également le signal. Le *Et* et le *Ou* que nous montrons ont deux entrées mais nous pouvons utiliser les variantes avec un plus grand nombre d'entrées.

10.2.2 Equivalence entre fonction et circuit

Étant donné une fonction booléenne quelconque, on peut facilement concevoir un circuit équivalent qui calcule cette fonction. Inversement, étant donné un circuit combinatoire, on peut trouver la fonction équivalente.

Pour concevoir le circuit qui calcule une fonction, on place les variables de la fonction comme entrées du circuit, puis on construit étape par étape les circuits qui traduisent les sous-expressions de la fonction. Ainsi, la fonction de trois variables

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

se traduit par le circuit de la figure 10.2 Les points noirs indiquent où les fils ne se croisent pas mais sont en contact.

Inversement, on peut extraire d'un circuit combinatoire la fonction booléenne équivalente. Il suffit de noter, à la sortie de chaque circuit de base, l'expression qui est calculée en fonction de ses entrées, en continuant jusqu'aux sorties du circuit.

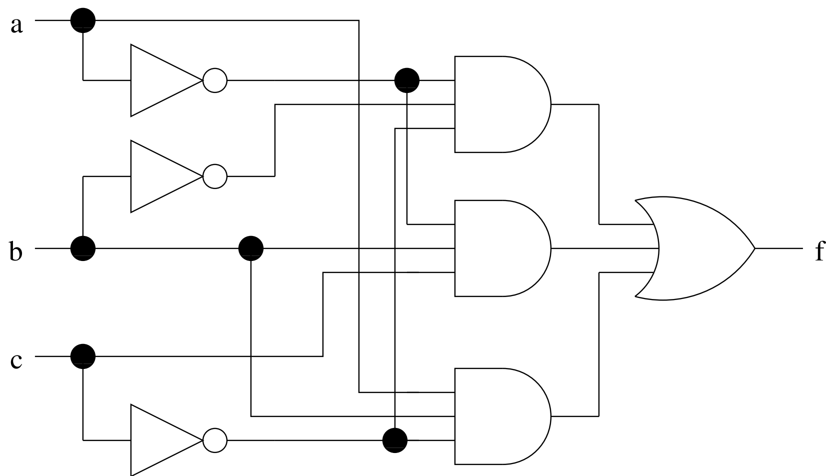


FIGURE 10.2 – Le circuit équivalent à la fonction $f(a, b, c) = \overline{a}.\overline{b}.\overline{c} + \overline{a}.b.c + a.b.\overline{c}$.

10.2.3 Le demi-additionneur

Le circuit de la figure 10.3, représente ce qu'on appelle un *demi-additionneur* : il possède deux entrées, a et b et calcule le résultat de l'addition des deux nombres représentés par ces deux chiffres. Rappelons la table de l'addition dans la base 2 :

a	b	a + b
0	0	0
0	1	1
1	0	1
1	1	10

Un problème, simple à résoudre, vient de la dernière ligne dans cette table d'addition : il faut deux chiffres pour représenter la sortie de notre demi-additionneur : l'un sera le chiffre de droite, que nous noterons s et l'autre la retenue que nous noterons r. Nous pouvons compléter notre table avec ces deux valeurs :

a	b	$a + b$	r	s
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	10	1	0

En utilisant les méthodes vues au chapitre précédent, on constate aisément que $r = a.b$ et que $s = a.\overline{b} + \overline{a}.b$. C'est ce que calcule notre circuit.

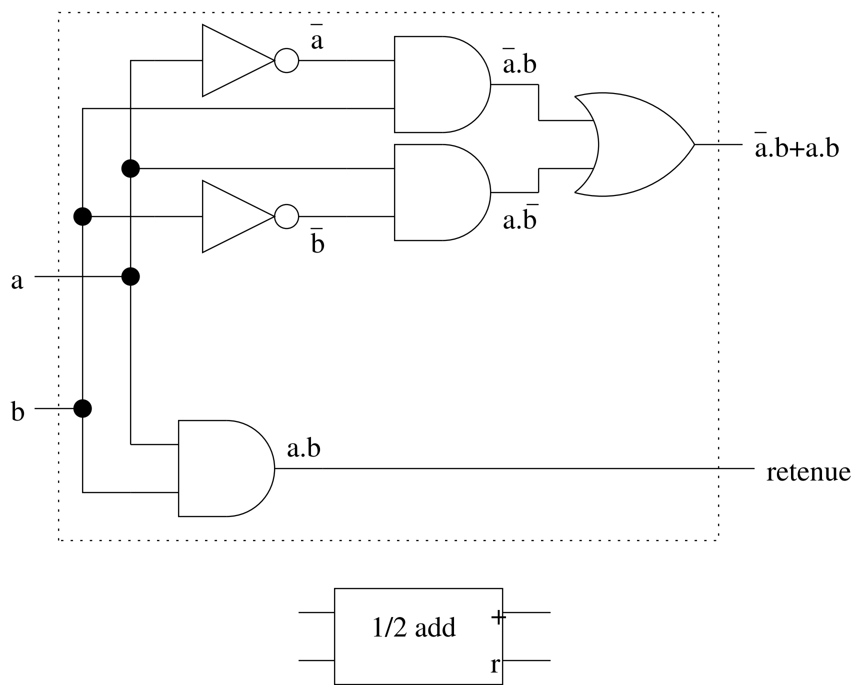


FIGURE 10.3 – Le circuit *demi-additionneur*, brique de base pour effectuer des additions.

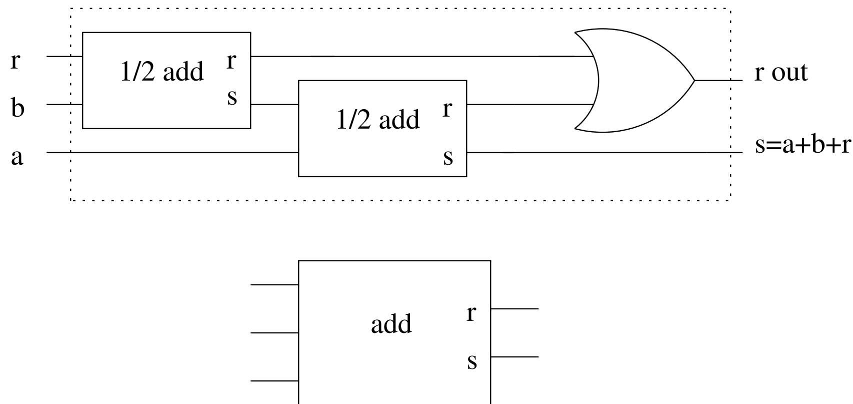


FIGURE 10.4 – Le circuit *additionneur complet*, qui additionne trois nombres binaires à un chiffre.

Une fois le circuit construit, nous pouvons le considérer comme une brique supplémentaire qu'on peut ré-utiliser dans des constructions plus complexes. Désormais, nous représenterons le demi-additionneur comme indiqué en bas de la figure : un rectangle qui porte l'étiquette `1/2 add`, avec ses deux entrées et ses deux sorties.

10.2.4 L'additionneur 1 bit

Avec un demi additionneur, on peut avoir le sentiment que nous n'avons pas beaucoup avancé : à partir de deux entrées, nous avons calculé deux sorties. Or pour faire l'addition de deux nombres nous avons besoin d'additionner trois bits dans chaque position : le bit de chacun des nombres mais aussi la retenue de la colonne précédente. Pour cette raison, considérons une table d'addition de trois valeurs :

a	b	c	$a+b+c$	s	r
0	0	0	0	0	0
0	1	0	1	0	1
1	0	0	1	0	1
1	1	0	10	1	0
0	0	1	1	0	1
0	1	1	10	1	0
1	0	1	10	1	0
1	1	1	11	1	1

Une façon simple de construire un tel circuit consiste à utiliser deux demi-additionneurs (d'où leur nom). Si le premier demi-additionneur produit une

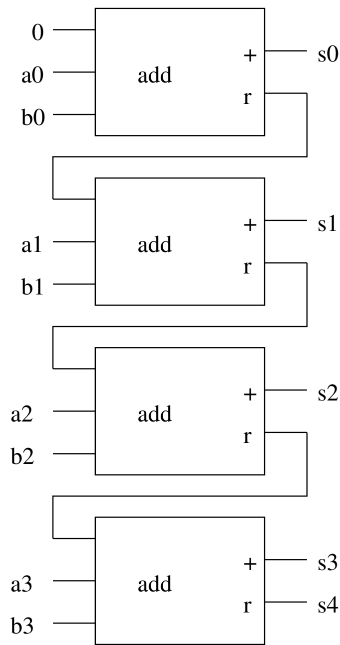


FIGURE 10.5 – Un circuit pour additionner deux nombres de quatre chiffres, en utilisant quatre additionneurs complets.

retenue, alors l'entrée du second qui correspond à la somme sera à 0 et ce second demi-additionneur ne produira pas de retenue. On peut donc faire un simple *Ou* entre les deux retenues qui sortent des deux demi-additionneurs pour obtenir la retenue de l'addition de trois valeurs. Nous présentons un tel circuit dans la figure 10.4 ; il calcule la somme de ses entrées. C'est un additionneur *complet* que nous pouvons désormais utiliser comme brique dans la construction de circuits plus complexes.

10.2.5 L'additionneur 4 bits

Pour additionner des nombres représentés sur plusieurs chiffres, on peut construire simplement un circuit avec autant d'additionneurs complets qu'il y a de chiffres. C'est ce que fait le circuit de la figure 10.5. Si on présente sur les entrées $a_3a_2a_1a_0$ et $b_3b_2b_1b_0$ deux nombres binaires sur quatre chiffres, il produit une somme (sur 5 bits, à cause de la retenue) $s_4s_3s_2s_1s_0$.

Pour construire ce circuit, nous nous sommes contentés de connecter la retenue de l'additionneur de chaque paire de chiffres à l'entrée de l'additionneur suivant. Nous allons maintenant utiliser ce circuit comme une nouvelle brique

dans nos constructions, en le représentant comme sur la droite de la figure, avec un rectangle étiquetté avec *add*.

Sur le même modèle, on peut faire des additionneurs pour des nombres sur n'importe quel nombre de chiffres binaires : il suffit de multiplier le nombre d'additionneurs 1 bit.

Il existe d'autres circuits plus rapides pour faire des additions mais celui-ci présente l'avantage de la simplicité.

Vous vous demandez pourquoi nous avons utilisé, pour additionner les bits a_0 et b_0 un additionneur complet, alors qu'un demi-additionneur aurait suffi ? La réponse se trouve dans la section suivante.

10.3 Le soustracteur 4 bits

Comment faire un circuit soustracteur ? Puisque nos nombres sont représentés en complément à deux, il suffit de calculer le complément à deux du nombre à soustraire puis d'utiliser un additionneur pour effectuer le calcul.

Pour calculer le complément à 2, rappelons qu'on remplace chaque 0 par 1 et réciproquement, puis qu'on ajoute 1. Cet ajout final est génant parce que la retenue peut se propager dans tout le nombre et il faudrait donc une seconde ligne (qui pourrait se faire avec des demi-additionneurs seulement) pour propager cette retenue.

Cependant, il existe un raccourci. Si on note $c_1(x)$ le complément à 1 d'un nombre x, alors son complément à 2 vaut $c_1(x) + 1$. Donc, on a :

$$a - b = a + (c_1(b) + 1)$$

Or l'addition est associative, donc on peut aussi écrire

$$a - b = a + c_1(b) + 1$$

Dans l'additionneur 4 bits que nous venons de construire, nous avons injecté 0 comme une des trois entrées de l'additionneur complet pour les bits de poids faible. Nous pouvons aussi y injecter 1 quand nous souhaitons faire une soustraction.

Cela nous permet de réaliser un circuit soustracteur sur le même modèle que l'additionneur : il suffit d'inverser chacun des bits du nombre à soustraire et de placer un 1 dans l'entrée de l'additionneur pour les bits de poids faible. C'est ce que nous faisons dans la figure 10.6. Cette figure est à comparer avec la figure 10.5.

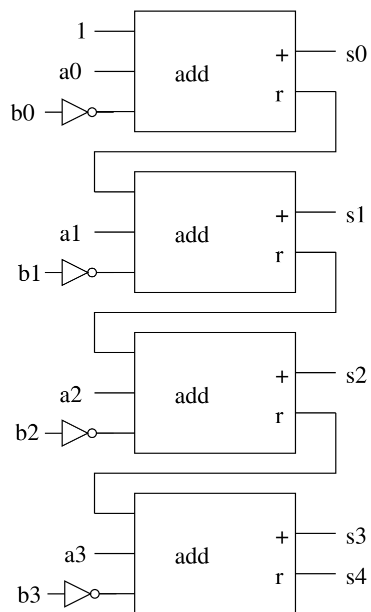


FIGURE 10.6 – Un circuit *soustracteur* : en inversant tous les bits de b et en ajoutant 1 au bits de poids faible, on additionne en fait le complément à 2 de b, ce qui revient à soustraire b.

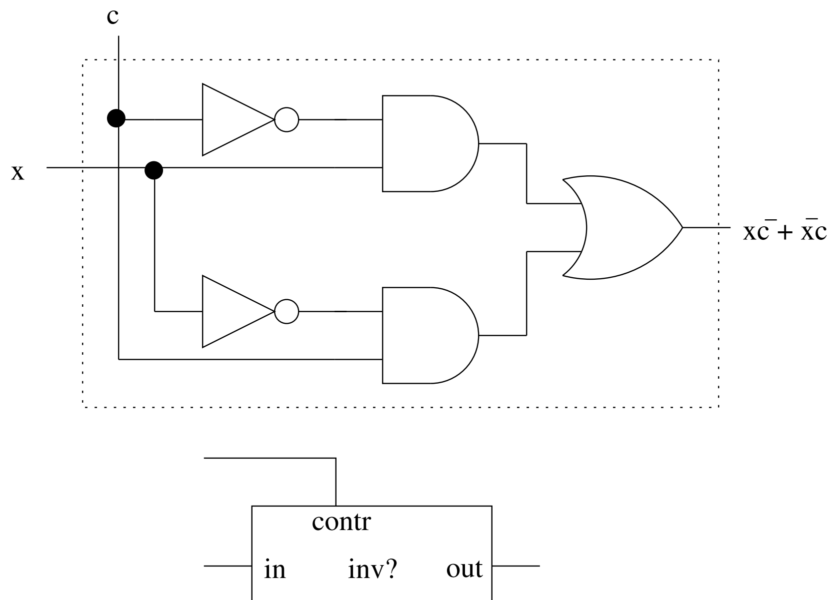


FIGURE 10.7 – Un circuit *inverseur ?* qui inverse ou pas son entrée x suivant la valeur de c.

10.4 L'additionneur–soustracteur 4 bits

Nous considérons maintenant la réalisation d'un circuit qui permette soit d'additionner soit de soustraire, en fonction d'un signal d'entrée supplémentaire qui indique l'opération à effectuer. Pour cela, nous commençons par construire un *inverseur conditionnel*. Le circuit aura une entrée de contrôle c : quand c vaudra 0, la sortie sera égale à x ; en revanche, quand le controle c vaudra 1, la sortie sera égale à $\overline{x}$. La table de vérité de notre inverseur sera donc :

c	x	f
0	0	0
0	1	1
1	0	1
1	1	0

Nous présentons ce circuit sur la figure 10.7. Comme d'habitude, nous avons placé en bas un schéma qui va nous permettre de réutliser ce circuit dans avoir besoin d'en préciser les détails. Soulignons que nous avons déja rencontré ce circuit comme une partie du demi-additionneur.

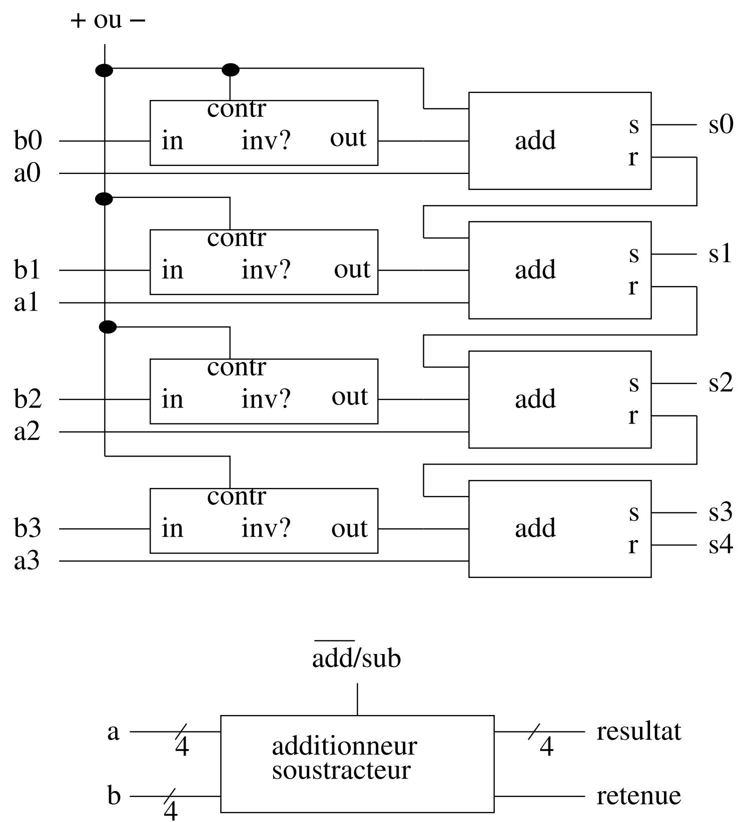


FIGURE 10.8 – Un circuit aditionneur/soustracteur pour mots de quatre bits.

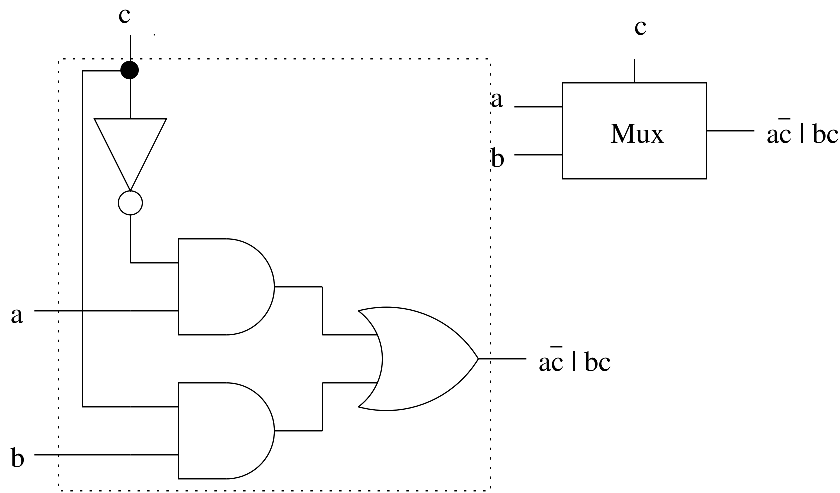


FIGURE 10.9 – Un circuit multiplexeur élémentaire

Pour obtenir un circuit qui permet de faire à la fois les additions ou les soustractions, nous remplaçons les inverseurs du soustracteur de la figure 10.6 par le circuit *inverseur conditionnel* que nous venons de concevoir, comme dans la figure 10.8 : si le signal de contrôle est à zéro, l'inverseur conditionnel ne modifie pas le signal et nous avons un additionneur comme dans la figure 10.5 ; si le signal de contrôle est à 1, les inverseurs conditionnels inversent les signaux qu'ils reçoivent en entrée et le circuit fonctionne comme le soustracteur de la figure 10.6.

Nous avons montré en dessous une version sans détails de notre circuit qu'on peut utiliser comme élément de construction de circuits plus complexes. Pour simplifier le schéma, plutôt que de dessiner tous les fils pour a, b et s, on ne dessine qu'un trait et on indique le nombre de fils représentés par ce trait. Nous avons nommé le signal de contrôle $\overline{\text{add}}/\text{sub}$ pour indiquer que la valeur 0 sur ce fil provoque une addition et la valeur 1 une soustraction : c'est un usage courant pour nommer les signaux d'une façon qui permette de se souvenir de leur fonction.

10.5 Le multiplexeur

Nous allons maintenant construire un circuit qui va nous permettre de choisir, entre plusieurs signaux en entrée, lequel nous souhaitons obtenir en sortie. On appelle ces circuits des *multiplexeurs*.

La figure 10.9 montre un multiplexeur élémentaire qui permet de choisir lequel des signaux a et b sera présenté en sortie, suivant la valeur du signal de contrôle c. Le circuit calcule s tel que $s = a.\bar{c} + b.c$; donc si $c = 0$, on a $s = a.1 + b.0 = a + 0 = a$, alors que si $c = 1$ on a $s = a.0 + b.1 = 0 + b = b$. On peut donc considérer ce circuit comme un aiguillage qui connecte la sortie soit à a, soit à b.

A l'aide du circuit multiplexeur élémentaire, on peut construire facilement un circuit multiplexeur qui va permettre de choisir le signal à présenter en sortie parmi un nombre quelconque de signaux. La figure 10.10 présente un tel multiplexeur pour huit entrées : il faut trois signaux de contrôle dans ce cas.

Comme on le voit sur la figure, on utilise le premier signal de contrôle c_0 pour éliminer la moitié des signaux entrants : il ne reste que quatre signaux parmi les huit de départ ; le deuxième signal de contrôle c_1 en élimine encore 2 et le dernier contrôle c_2 permet de choisir le signal qui sera présenté en sortie.

Une autre façon de considérer la chose, c'est que $c_2c_1c_0$ représentent un nombre binaire sur trois chiffres, compris entre 0 et 7, qui contient le numéro du signal présenté en sortie. Cette vision synthétique correspond au schéma de droite, dans lequel le multiplexeur est représenté avec deux traits qui entrent et un trait qui sort : sur chaque trait est indiqué le nombre de fils représentés sur ce trait.

En bas à droite de la figure, on a une version plus générale : avec n signaux de contrôle, on peut choisir un signal parmi 2^n.

Il existe un circuit du même genre que le multiplexeur, qu'on appelle un *décodeur*. Voir les exercices.

10.6 L'unité de calcul

Nous avons maintenant tous les éléments pour construire une unité arithmétique et logique : la figure 10.11 ne contient que des éléments que nous connaissons déja.

Cette UAL est susceptible d'effectuer 5 opérations : trois opérations logiques (*Et, Ou, Non*) et deux opérations arithmétiques : addition et soustraction. Elle possède deux entrées, a et b et une retenue entrante, notée r_{in}. Le signal est présenté en entrée de circuits qui calculent le résultat possible de toutes les opérations logiques et celui de l'addition ou de la soustraction suivant l'état du bit de poids fort du contrôle. Ensuite un multiplexeur permet de choisir, parmi les quatre valeurs calculées, laquelle sera présentée en sortie comme résultat.

Les codes opérations seront donc :

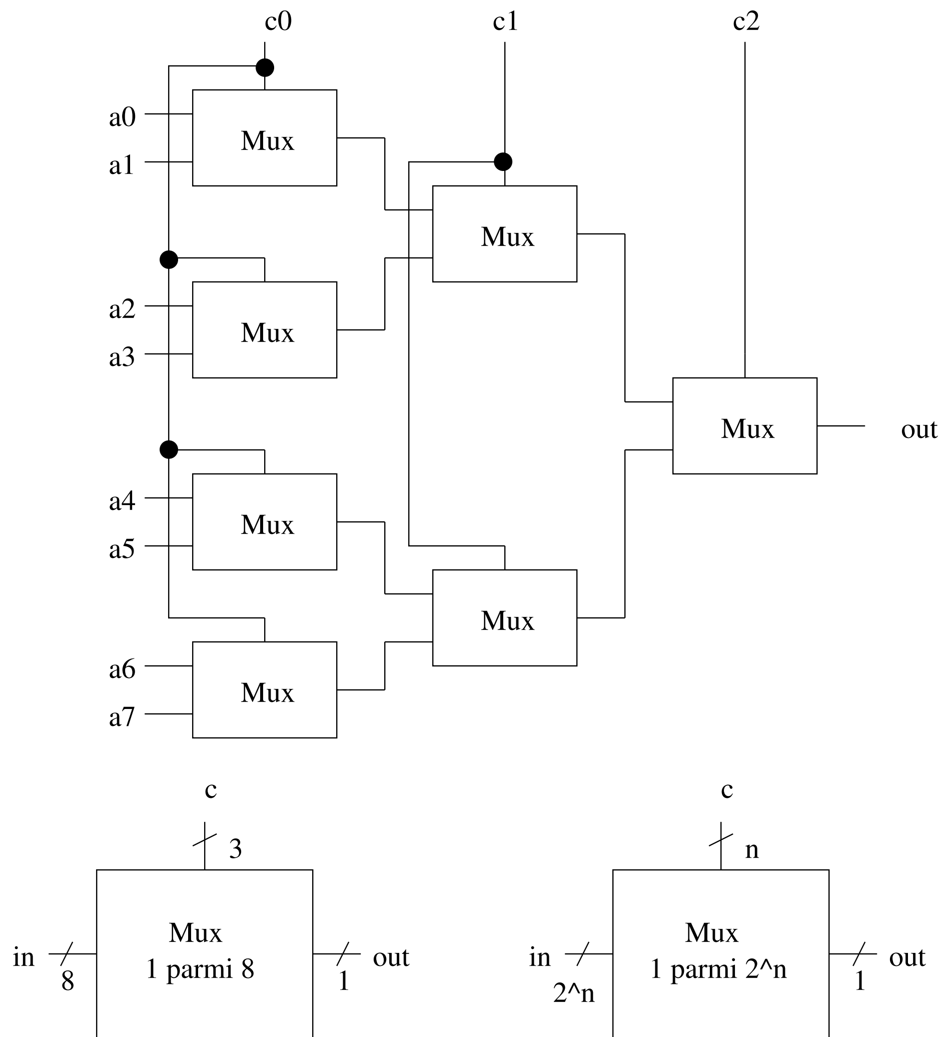


FIGURE 10.10 – Un circuit multiplexeur 1 parmi 8.

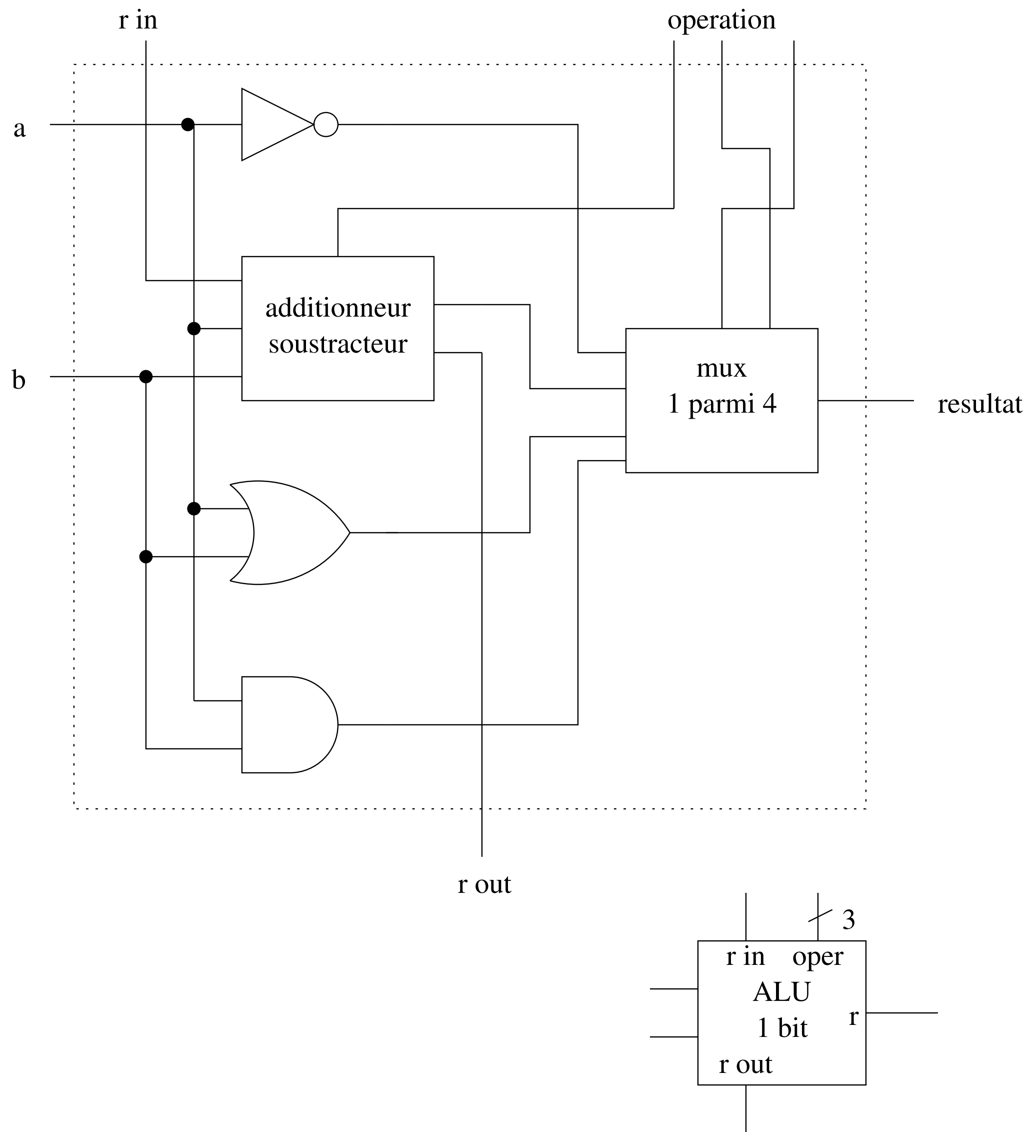


FIGURE 10.11 – Une unité de calcul 1 bit

code	opération
000	`Non a`
001	`a + b`
010	`a OU b`
011	`a ET b`
101	`a - b`

Une UAL 1 bit n'est pas très utile mais on peut combiner des unités de traitement d'un bit pour traiter le nombre de bits que l'on veut. La figure 10.12 montre une unité de calcul pour des mots de quatre bits, contruite avec quatre unités de calcul d'un bit. La partie de droite de la figure contient le symbole qu'on emploie usuellement pour représenter une UAL qui traite des nombres de n bits.

10.7 Résumé

Aux expressions logiques, on peut faire correspondre des circuits qu'on peut réaliser avec des transistors. Quand les circuits ne contiennent pas de boucles, on peut aussi faire correspondre une expression logique à chaque circuit. Des circuits intéressants sont l'additionneur, le multiplexeur, le décodeur. En les utilisant, on peut contruire une unité arithmétique et logique.

10.8 Exercices

Exercice 10.1 : Concevoir un circuit à trois entrées et une sortie, dont la sortie vaudra 1 si une et une seule de ses entrées est à 1.

Exercice 10.2 : Concevoir un circuit qui reçoit un nombre codé sur quatre bits et renvoie 1 si ce nombre est supérieur ou égal à dix.

Exercice 10.3 : Concevoir un circuit qui reçoit un nombre codé sur quatre bits et renvoie 1 si ce nombre est divisible par 3.

Exercice 10.4 : (Amusant) Concevoir un circuit, réalisé seulement avec des *Ou*, *Et* et *Non*, qui réalise un *Ou exclusif* sans qu'aucun fil ne se croise (et bien sur avec les entrées et les sorties qui arrivent de l'extérieur et ne sont pas parachutées au milieu du circuit).

Exercice 10.5 : (Amusant et difficile) Concevoir un circuit à trois entrées a, b et c, qui utilise des *Ou*, des *Et* et seulement deux circuits *Non*, dont les trois sorties valent $\overline{a}$, $\overline{b}$ et $\overline{c}$.

Exercice 10.6 : Chaque fois qu'un signal traverse une porte *Non*, *Et*, *Ou*, les transistors qui le composent ont besoin d'un peu de temps pour atteindre un état stable. Supposons que ce temps t est constant, quelle que soit la porte élémentaire. Calculer le temps nécessaire en fonction de t pour stabiliser la sortie

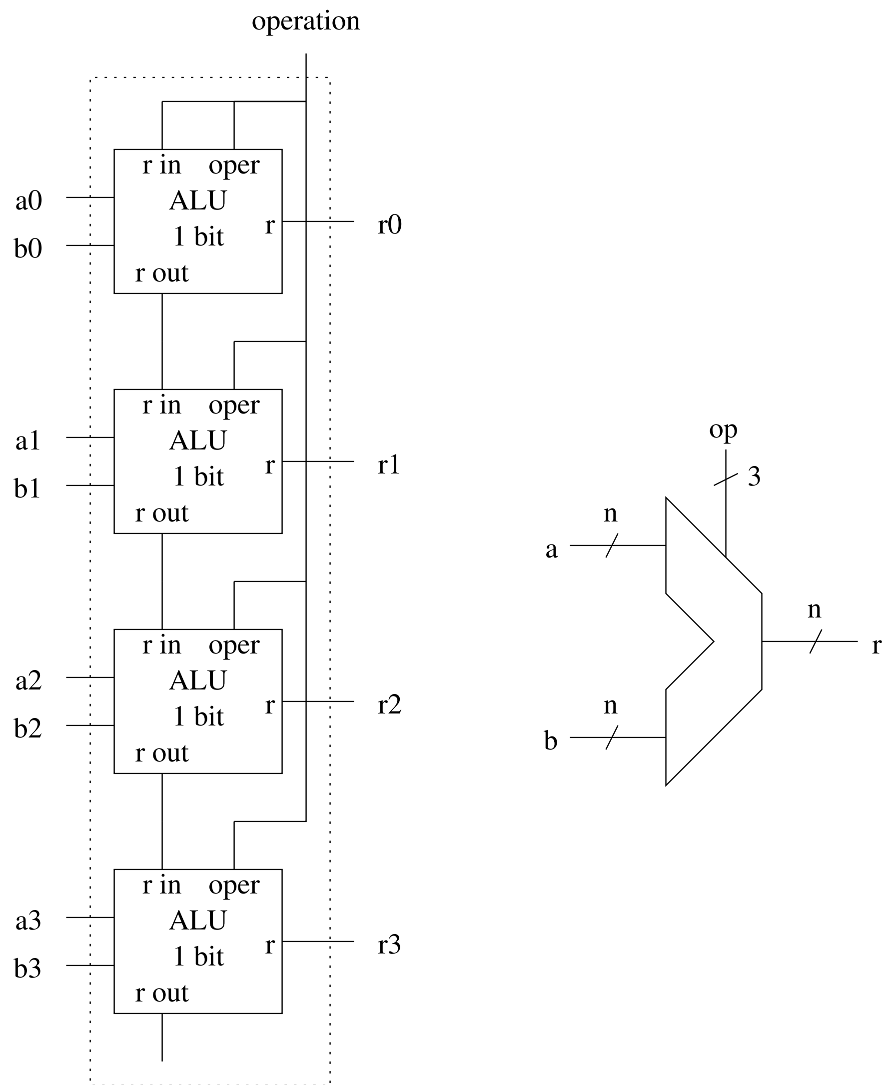


FIGURE 10.12 – Une unité de calcul 4 bits

d'un demi-additionneur, d'un additionneur complet 1 bit, d'un additionneur complet 8 bits. Refaire la même question en échangeant les entrées a et r de l'additionneur.

Exercice 10.7 : Dessiner un multiplexeur *1 parmi 5*.

Exercice 10.8 : Combien faut-il de multiplexeurs élémentaires *1 parmi 2* pour contruire un multiplexeur *1 parmi* n quand n n'est pas une puissance de deux ?

Exercice 10.9 : Dessiner un multiplexeur *1 parmi 64* (heureusement qu'on peut utiliser le multiplexeur *1 parmi 8* ; on peut même définir des multiplexeurs *1 parmi 16* et *1 parmi 32* pour simplifier le travail)

Exercice 10.10 : (à faire) Un circuit *décodeur élémentaire* possède une entrée a et deux sorties s_0 et s_1 : quand $a = 0$, $s_0 = 1$ et $s_1 = 0$. Quand $a = 1$ c'est le contraire. : dessiner un circuit décodeur.

Exercice 10.11 : (à faire) Un décodeur général possède n entrées et 2^n sorties ; si les entrées codent le nombre a en binaire, alors toutes les sorties sont à 0, sauf la sortie s_a qui est à 1. Dessiner le circuit d'un décodeur à quatre sorties. Dessiner le circuit d'un décodeur à 3 entrées.

Exercice 10.12 : A partir du schéma de l'unité de calcul de la figure 10.11, déterminer ce qui se produit quand on envoie les codes opération 100, 110 et 111.

Exercice 10.13 : (à ne pas faire) Modifier le schéma de l'alu 1 bit pour permettre les décalages de mots. On peut le faire, grace aux retenues entrante et sortante, sans modifier le schéma de l'alu 4 bits et en n'utilisant que deux multiplexeurs *1 parmi 2*.

Chapitre 11

Circuits séquentiels

Dans le chapitre précédent, nous avons étudié des circuits qui permettent de calculer des fonctions logiques : les valeurs en sortie ne dépendent que des valeurs présentes en entrée à l'instant présent.

Pour fabriquer des mémoires, il faut des circuits différents, dans lesquels il y a des boucles : l'état du circuit va alors dépendre des valeurs qui s'y sont trouvées auparavant.

11.1 Les boucles d'inverseurs

Nous allons commencer par examiner deux circuits élémentaires dans lesquels il n'y a pas d'entrées et qui ne sont réalisés qu'avec des boucles d'inverseurs.

La première boucle représentée sur la figure 11.1 est instable : s'il y a 0 en entrée de l'inverseur il va y avoir un 1 en sortie. Mais comme la sortie est renvoyée en entrée elle va aussitot changer de valeur pour passer à 1 et ainsi de suite. Le circuit est instable ; la valeur de la sortie q change sans arrêt.

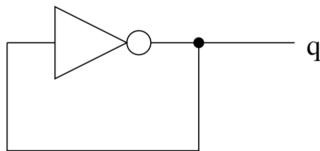


FIGURE 11.1 – Un inverseur bouclé est instable ; la valeur de q change sans arrêt.

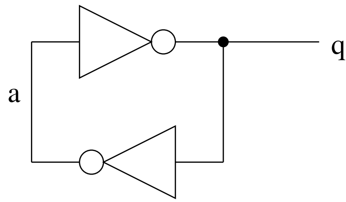


FIGURE 11.2 – Deux inverseurs bouclés sont stables.

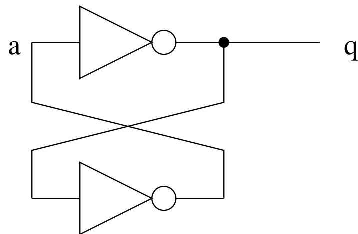


FIGURE 11.3 – Le circuit de la figure 11.2 dessiné autrement.

Le circuit représenté dans la figure 11.2 est stable : quelle que soit la valeur de a, la sortie de l'inverseur du haut vaut $\overline{a}$ et donc la sortie de l'inverseur du bas vaut $\overline{\overline{a}} = a$. Le point important est que nous ignorons la valeur de a : le circuit produit 1 ou 0, suivant la valeur qui s'est trouvée dans a au départ.

L'usage veut qu'on représente les circuits dans le même sens, comme sur la figure 11.3, qui contient les mêmes éléments que la figure 11.2 : ici les deux inverseurs ont leur entrée sur la gauche et leur sortie sur la droite.

L'inconvénient d'un tel circuit est que nous ne pouvons pas modifier la valeur mémorisée. Nous allons résoudre ce problème dans la section suivante.

11.2 La bascule RS

Si on remplace les inverseurs des boucles précédentes par des circuits *Non-Et* ou *Non-Ou*, nous pouvons modifier la valeur mémorisée dans la boucle. Rappelons tout d'abord la table de vérité du *Non-Ou* vue au chapitre précédent :

a	b	$\overline{a+b}$
0	0	1
0	1	0
1	0	0
1	1	0

Avec des circuits *Non-Ou*, on a la classique bascule RS, présentée dans la figure 11.4. Quand les deux signaux R et S sont à *0*, les circuits *Non-Ou* inversent les signaux a et $\overline{a}$ comme le faisaient les inverseurs de la figure 11.3 et la valeur de q reste stable.

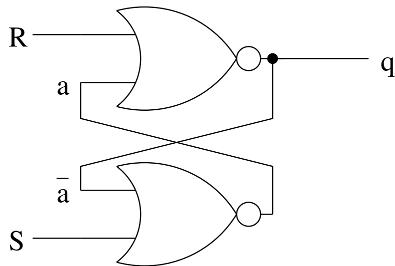


FIGURE 11.4 – Une bascule RS avec deux *Non-Et*. Quand les valeurs de R et S sont fixées à 0, ce circuit est équivalent à celui de la figure 11.3.

Regardons maintenant ce qui se passe quand le signal R passe à *1*, alors que le signal S reste à 0 : quelle que soit la valeur de a, la sortie du circuit du haut q prend la valeur 0. Cette valeur est inversée par le circuit du bas dont la sortie prend la valeur 1. Maintenant, quand R et S repassent à 0, q conserve la valeur 0.

En revanche, quand la sortie R reste à 0 et que c'est la valeur de S qui prend la valeur 1, la sortie a du circuit du bas prend la valeur 0 et celle du circuit du haut prend la valeur 1. Quand maintenant S repasse à 0, la sortie q conserve sa valeur 1.

On a donc construit un circuit dans lequel on peut positionner une valeur de sortie avec l'aide des signaux S et R (comme *Set* et *Reset* : mettre à 1 ou remettre à 0), puis conserver cette valeur tant que les signaux R et S restent à 0.

Quand les deux signaux R et S passent tous les deux à 1 en même temps, la bascule entre dans un état instable, de la même manière que l'inverseur bouclé que nous avons vu en premier. Si les signaux repassent à 0 en même temps, elle convergera vers un des deux états stables mais nous ne pouvons pas prévoir lequel ; pour cette raison, il faut éviter de mettre les deux signaux R et S à 1 en même temps ; on appelle cela l'état *interdit*.

11.3 La bascule D

La bascule RS n'est pas très pratique à manipuler directement avec ses deux signaux en entrée et son état interdit mais elle sert de brique pour fabriquer une cellule mémoire élémentaire de 1 bit, qu'on appelle la *bascule D*.

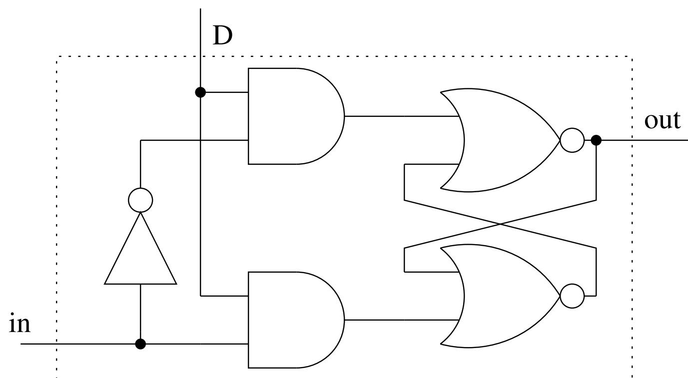


FIGURE 11.5 – La constitution d'une bascule D : cellule élémentaire de mémorisation d'un bit.

Sur la figure 11.5, on a la construction d'une telle bascule. Le signal *in* est le signal à mémoriser et il est transformé avec un inverseur, de manière à éviter l'état interdit. Le signal *in* et son inverse sont ensuite combinés avec le signal D dans deux *Et*; si $D = 0$, la sortie des *Et* vaudra 0 et la boucle des *Non-ou* de droite conserve la valeur ; en revanche, quand $D = 1$, c'est le signal *in* et son inverse qu'on va trouver en sortie des *Et* et la valeur de la boucle de *Non-ou* de droite sera positionnée comme dans la bascule RS.

Donc, quand D vaut 0, la bascule présente sur la sortie *out* la dernière valeur mémorisée. Quand D passe à 1, la bascule lit la valeur du signal *in* et l'utilise comme nouvelle valeur à mémoriser ; cette valeur sera celle du signal *out* quand D repassera à 0.

Souvent pour contrôler la mémorisation d'une bascule, on utilise une *horloge* : il s'agit un signal dont la valeur oscille régulièrement entre 0 et 1. Pour être certain que la valeur soit lue au bon moment, on peut facilement utiliser des bascules D qui seront activée par le changement du signal de l'horloge. On parle de *front montant* pour le passage de 0 à 1 et de *front descendant* pour le passage de 1 à 0.

11.4 Construction d'un mot mémoire avec des bascules D

La bascule D ne permet de mémoriser qu'un seul bit alors que les mots de la mémoire de nos ordinateurs contiennent des valeurs représentées avec plusieurs chiffres. On construit facilement ceci en regroupant plusieurs bacsules D.

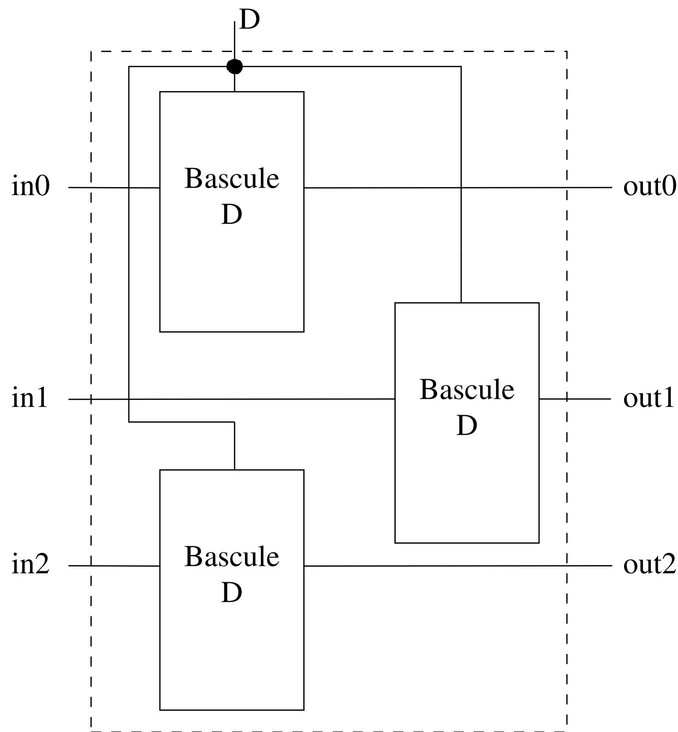


FIGURE 11.6 – La construction d'un mot de 3 bits à partir de 3 bascules D.

La figure 11.6 présente une construction possible pour un mot mémoire qui contient cette fois trois bits, au lieu d'un seul : on a trois bascules D, avec chacune son signal en entrée et son signal ‘en sortie ; les trois bascules sont commandées par le même signal, donc les trois bascules vont mémoriser le signal sur leur entrée au même instant.

Il est élémentaire d'imaginer comment, sur ce modèle, on peut construire des mots de tailles plus raisonnables, comme 8, 16, 32 ou 64 bits : il suffit de multiplier, sur le même modèle, le nombre de bascules D et de signaux en entrée et en sortie.

11.5 Lecture dans une mémoire de 4×1 bit

Quand on a plusieurs mots mémoire, il est nécessaire de sélectionner l'un de ces mots pour ne lire qu'une seule valeur. Pour cette sélection, on attribue un numéro à chaque case de la mémoire et on transmet, pour lire ou écrire dans une case, le numéro de la case qu'on veut sélectionner. C'est ce numéro qu'on appelle une *adresse*.

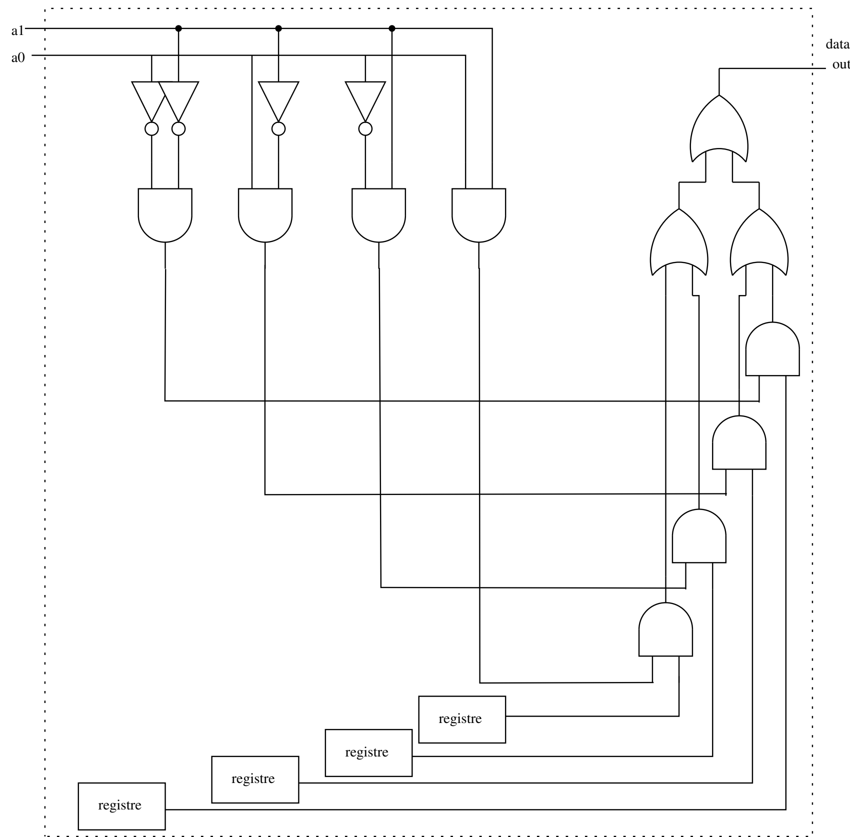


FIGURE 11.7 – La lecture dans une mémoire 4×1 bit.

La figure 11.7 présente une mémoire de quatre mots de 1 bit. Pour simplifier le dessin, on n'a représenté que la partie qui sert à lire le contenu de la mémoire ; l'écriture sera présentée dans la figure suivante.

La sélection du mot intéressant se fait dans la partie en haut à gauche de la figure. qui consitue un *décodeur* d'adresse : les deux signaux a_0 et a_1 représentent un nombre entre 0 et 3. Les quatre inverseurs et les quatre *Et* combinent ces deux valeurs de manière à ce qu'un seul des quatre signaux en sortie vale 1 à un moment donné, en fonction des valeurs de a_0 et a_1.

Les quatre bascules D se trouvent en bas de la figure, avec l'étiquette *registre*. Comme on se contente, pour le moment de lire dans la mémoire, on a omis de dessiner le signal en entrée et la signal de contrôle de ces bascules.

Sur la droite de la figure, les circuits *Et* permettent de mettre à 0 les signaux qui sortent de toutes les bascles, sauf celle indiquée par la sortie du décodeur

d'adresse. Finalement, les circuits *Ou* servent a présenter cette valeur sélectionnée sur la sortie (unique) du circuit.

11.6 Ecriture dans une mémoire de 4×1 bit

L'écriture dans une mémoire fonctionne sur le même principe.

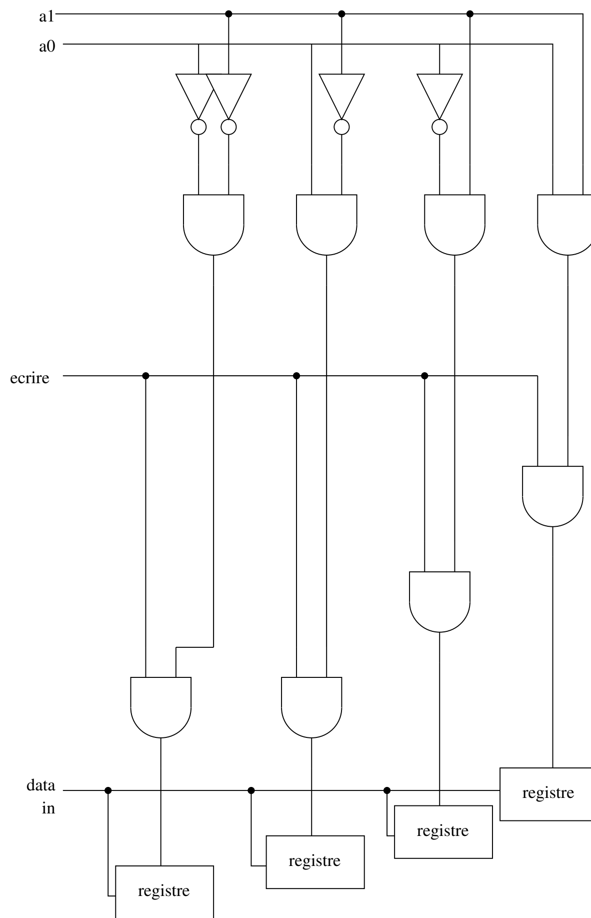


FIGURE 11.8 – L'écriture dans une mémoire 4×1 bit.

La figure 11.8 présente de grandes similarités avec la figure précédente. En haut, le décodeur d'adresse est le même que pour la lecture mais cette fois le signal (qui sert toujours à choisir à partir de l'adresse laquelle des bascules D activer) est combiné avec le signal *écrire* dans quatre *Et* supplémentaires.

Sur les bascules D, on a cette fois omis de représenter le signal en sortie pusiqu'on se contente de considérer l'écriture dans la mémoire. Le signal en entrée est envoyé sur chacune des bascules mais quel que soit ce signal, l'état des bascules ne change pas tant que le signal *écrire* est à 0.

Quand le signal *écrire* passe à 1, il est combiné avec les quatre sorties du décodeur d'adresse dont un seul est à 1 (toujours celui indiqué par la valeur de a_0 et a_1 et donc c'est la bascule D d'adresse a_1a_0 va mémoriser la valeur présente en entrée.

11.7 Lire et écrire dans une mémoire $x \times y$ bits

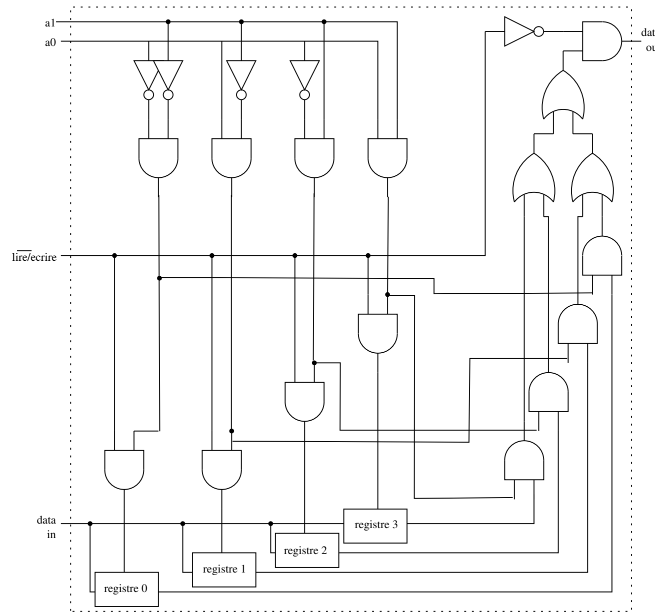


FIGURE 11.9 – Lecture et écriture dans une mémoire 4×1 bit.

La combinaison des deux figures 11.7 et 11.8 donne le schéma de la figure 11.9. On a ici représenté un circuit de mémoire qui permet de stocker quatre mots de 1 bits. On peut, à l'aide des signaux a_0 et a_1 sélectionner l'un des quatre mots et le signal $\overline{\text{lire}}$/écrire de commande permet soit de présenter en sortie la valeur mémorisée dans la bascule D sélectionnée, soit de mémoriser dans cette bascule la valeur présente en entrée (voir exercice).

Pour construire des mémoires avec plus de mots, il y a peu de modifications à effectuer dans le circuit. Pour une mémoire de $2^n \times 1$ bit, le circuit décodeur aura n bits d'adresse en entrée et 2^n signaux en sortie, Ces signaux serviront

à sélectionner une seule parmi les 2^n bascules D. Enfin, il faudra augmenter la taille de l'arbre de *Et* et *Ou* en sortie.

Pour construire des mémoires dont les mots contiennent m bits, avec $m > 1$, la modification est directe. Le signal en entrée et le signal en sortie seront remplacés par m signaux différents. Chaque bascule D du dessin original sera remplacée par un groupe de m bascules D qui seront controlées par le même signal D et dont l'entrée sera l'un des signaux en entrées. De même, il faudra dupliquer l'arbre de *Et* et de *Ou* pour chacun des signaux de sortie.

D'une façon générale, la capacité des circuits mémoires est annoncée sous la forme $x \times y$: le nombre de bits mémorisables dans le circuit est dans tous les cas de $x \times y$; x est le nombre de mots et y le nombre de bits dans chaque mot. Le circuit intégré aura $\log_2(x)$ fils d'adresse et y fils de données en entrée et en sortie. Par exemple, deux circuits $1k \times 8$ bits et $8k \times 1$ bits permettent tous les deux de mémoriser 8 kilo-bits mais le circuit $1k \times 8$ bits aura 10 signaux d'adresse et 8 signaux d'entrée et de sortie, alors que le circuit $8k \times 1$ bit aura 13 signaux d'adresse pour un seul signal d'entrée et de sortie.

Les mémoires telles que nous les avons vues ici sont une vue simplifiée par rapport à la variété des dispositifs de mémorisation qu'on rencontre dans les ordinateurs réels. Elle est proche des mots de mémoire utilisés dans les processeurs, dans l'unité de calcul ou l'unité de contrôle, qu'on appelle souvent des *registres*.

11.8 Catégories de mémoire

Cette section fait un rapide survol des catégories les plus communes de mémoire.

11.8.1 Mémoire RAM

D'une façon générale, on tend à désigner la mémoire sous le nom de *RAM*, comme *Random Access Memory*, c'est à dire « mémoire à accès aléatoire », pour la différencier des valeurs mémorisées sur les bandes magnétiques, les disques, les CDROM et les DVD, pour lesquels il est bien plus rapide de lire ou d'écrire des données en séquence, en les adressant dans l'ordre, plutôt que d'accéder à des adresses quelconques dans le désordre.

11.8.2 Mémoire statique

La mémoire statique est le plus souvent construite à partir de bascules D, comme celles que nous avons vues dans la section précédente. Ce sont des mémoires rapides, plutôt chères, avec des capacités réduites par rapport aux mémoires dynamiques que nous présentons dans la section suivante.

En pratique, elles sont en général utilisées dans les PCs pour réaliser les mémoires *cache* de niveau 2, qui contiennent une copie des zones de la mémoire principale récemment accédées. En 2007, la taille des mémoires caches d'un processeur ordinaire est de l'ordre de quelques Méga-octets.

11.8.3 Mémoire dynamique

La mémoire centrale de nos ordinateurs est principalement consituée de mémoire *dynamique*, qui utilise une autre technique de réalisation que celle présentée dans ce chapitre : chaque bit de mémoire est constitué d'un condensateur qui est chargé pour représenter 1 et déchargé pour représenter 0. Comme la charge d'un condensateur s'atténue avec le temps, il est nécessaire de temps en temps de repasser sur toute la mémoire, d'en lire le contenu et de le ré-écrire. Cette opération est effectuée de l'ordre d'un millier de fois par seconde. On appelle cela *rafraichir* la mémoire.

Comme cette technique est beaucoup plus économique en silicium que celle des ram statiques, les ram dynamiques ont des capacités plus importantes. Elles se présentent le plus souvent sous forme de *barrettes*, des petits circuits imprimés, sur lesquels sont placés huit circuits de mémoire 1 bit pour réaliser des mémoires huits bits. La capacité ordinaire d'une mémoire dynamique en 2009 est de l'ordre du Giga Octet.

11.8.4 Mémoire ROM et variantes

On trouve dans nos ordinateurs des mémoires dans lesquelles il est impossible d'écrire. On les appelle des ROM (prononcer comme *Rome*), comme *Read Only Memory* c'est à dire « mémoire en lecture seule ». Ce ne sont pas à proprement parler des circuits de mémorisation, puisque les sorties ne dépendent que des entrées mais c'est une façon pratique de considérer certains circuits combinatoires.

Certaines ROM sont en fait des PROM, comme *Programmable ROM* : chaque bit de la mémoire est controlé par un fusible, qu'on peut faire fondre (une seule fois !) pour modifier la valeur contenue dans la mémoire, dans une machine spéciale qu'on appelle un *programmateur de PROM*. Cela permet d'avoir un circuit bon marché, parce que produit en grande quantité, que chacun peut adapter à son application.

Certaines PROM sont en fait des EPROM (prononcer comme *euprome*), des *Erasable PROM*, c'est à dire des PROM effaçables. Ces mémoires sont programmables comme des PROM mais en les exposant à des rayons ultra-violets, on peut en effacer le contenu et les reprogrammer dans un programmateur d'EPROM. Quand vous démontez un vieil ordinateur, si vous voyez un circuit intégré placé sur un support d'où on peut l'extraire, avec une étiquette en papier qui couvre le dessus, il s'agit probablement d'une EPROM ; en détachant l'étiquette vous verrez apparaitre une petite fenêtre qui est celle utilisée pour

effacer la mémoire en exposant le circuit intégré aux ultra-violets. Le principal avantage par rapport aux PROM, c'est qu'on peut réutiliser le circuit, ce qui facilite la mise au point du programme placé dans l'EPROM.

D'autres PROM sont en fait des EEPROM (prononcer comme *eu-euprome*), des *Electrically Erasable PROM*, c'est à dire des PROM effaçable électriquement. On peut effacer le contenu de ces mémoires en utilisant des signaux de commandes et des tensions électriques particulières. Le principal intérêt des EEPROM est qu'il n'y a pas besoin d'extraire le circuit intégré pour changer son contenu. Le BIOS d'une carte mère peut être stocké dans une EEPROM, ce qui permet sa mise à jour sans démonter l'ordinateur.

Une catégorie spécifique d'EEPROM est la mémoire *Flash*. A la différence des EEPROM ordinaires, on y efface les données par bloc et non par octet. La mémoire flash présente l'avantage d'être bon marché, d'avoir une bonne capacité et d'être rapide d'accès en lecture. C'est elle qu'on trouve dans les cartes mémoires des appareils photos, dans les baladeurs MP3 et dans les porte-clefs USB. La mémoire flash présente l'inconvénient, par rapport aux mémoires ordinaires (et aux disques durs) de s'altérer après quelques milliers ou dizaines de milliers d'écritures.

A coté de ces types de mémoire, on rencontre aussi des mémoires, dites parfois NVRAM (comme *Non Volatile RAM*) ou Mémoire CMOS (sur les PC) qui sont en réalité des mémoires statiques ordinaires alimentées par des batteries. C'est notamment là que sont stockés, sur les PCs, les paramètres du BIOS. Quand on a oublié le mot de passe du BIOS, il suffit de retirer la batterie de la carte mère pour que le contenu du BIOS soit réinitialisé. [Sur certains PC récents (en 2013), les paramètres du BIOS sont stockés dans une mémoire Flash ; retirer la batterie de la carte mère ne réinitialise pas la mémoire ; il faut utiliser une autre méthode ; en général, on peut l'effacer en déplaçant un jumper sur la carte.]

11.9 Exercices

Exercice 11.1 : Montrer que si les signaux R et S d'une bascule RS passent tous les deux à 1 au même instant, la bascule RS sera dans un état imprévisible après qu'ils soient revenus à 0.

Exercice 11.2 : Dessiner un circuit de mémorisation équivalent à la bascule RS avec des circuits *Non-Et* à la place du circuit *Non-Ou*. Comment le commande-t-on ?

Exercice 11.3 : La figure 11.9 contient un circuit *Non* et un circuit *Et* en plus de la juxtaposition des figures 11.7 et 11.8. Deviner et expliquer leur rôle.

Exercice 11.4 : Dessiner en détail une mémoire 2×2 bits.

Exercice 11.5 : A l'aide d'un site comme `www.rue-montgallet.com`, trouver l'ordre de grandeur du nombre de bits qu'on peut stocker pour environ 100 euros dans une mémoire vive, dans une clef USB, sur un disque dur.

Chapitre 12

Automates

Avec les circuits logiques combinatoires, nous pouvons calculer des fonctions logiques quelconques ; avec les circuits logiques séuentiels, nous pouvons mémoriser des valeurs. En combinant ces deux types de circuits, nous sommes en mesure de fabriquer des *automates* simples qui réaliseront des suites complexes d'action.

Nous nous intéresserons ici à une classe restreinte d'automates, qu'on appelle des *automates finis déterministes* que nous définissons et utilisons de façon informelle ici.

12.1 Les automates : un exemple simple

On peut décrire les automates qui nous intéressent par un graphe, composé d'états et de transitions. Ainsi, la figure 12.1 présente un automate élémentaire à quatre états et quatre transitions.

Les états sont indiqués par des cercles ; ils sont reliés entre eux par des *transitions*, représentées par des flèches. A un moment donné, l'automate se trouve dans un des états (au départ, dans l'état 0). Une horloge externe bat la mesure ; à chaque battement de l'horloge, l'automate prend une des transitions qui part de l'état courant et arrive dans un autre état. Ainsi, l'automate simple de la figure 12.1 va passer successivement dans les états 0, 1, 2 puis 3, pour finalement revenir dans l'état 0 et recommencer la boucle. Il ne s'arrête jamais.

Avec chaque transition, nous avons indiqué une action ; dans notre exemple, écrire une valeur, 0 ou 1. Notre automate va donc produire une suite $00110011\cdots$ de bits tant qu'il fonctionnera.

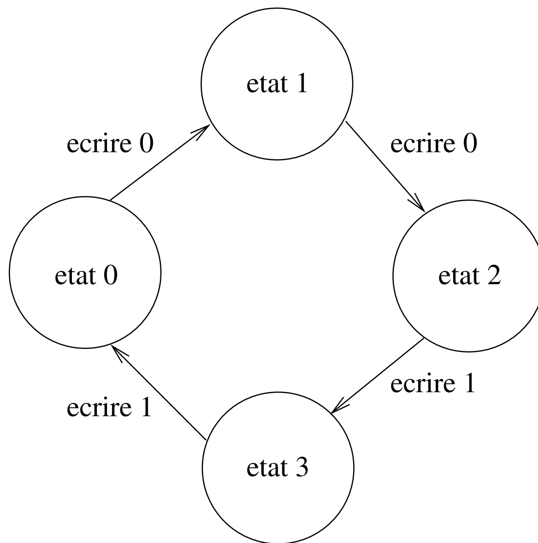


FIGURE 12.1 – Un automate simple qui produit 0 0 1 1 0 0 1 1 0 0 1 1 ...
.

12.1.1 Du graphe à la réalisation

On peut réaliser les automates d'une façon systématique avec un registre pour représenter l'état courant et un bloc de logique combinatoire pour représenter les transitions et les signaux à produire en sortie. Dans notre exemple, l'automate a quatre états et il faudra donc deux bits pour le représenter ; appelons-les e_1 et e_0.

La sortie, que nous notons *out* dépend de l'état courant, en fonction de la table de vérité suivante, où nous avons ajouté pour faciliter la lecture une colonne à gauche qui contient le numéro de l'état.

état courant	e_1	e_0	out
0	0	0	0
1	0	1	0
2	1	0	1
3	1	1	1

On voit sur la table de vérité que *out* est simplement égal à e_1, De même, l'état d'arrivée des transitions peut se calculer simplement comme une fonction logique de e_1 et de e_0. Notons ces valeurs qui représenteront le prochain état courant comme s_1 et s_0. Les trois tables de vérité pour calculer *out*, s_0 et s_1 peuvent se combiner dans la table suivante :

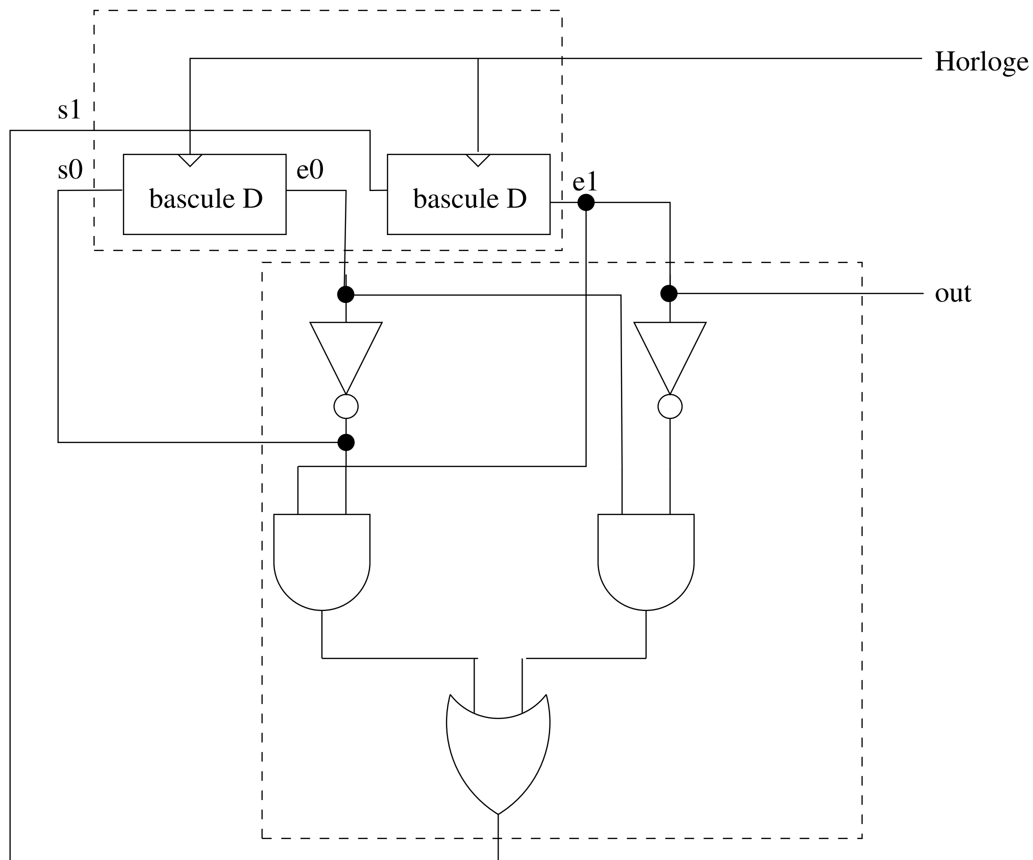


FIGURE 12.2 – Une réalisation de l'automate de la figure 12.1.

.

état courant	état suivant	e_1	e_0	out	s_1	s_0
0	1	0	0	0	0	1
1	2	0	1	0	1	0
2	3	1	0	1	1	1
3	0	1	1	1	0	0

De ces tables de vérité, nous déduisons que $s_1 = e_0.\overline{e_1} + \overline{e_0}.e_1$ et que $s_0 = \overline{e_0}$.

La réalisation de l'automate tel qu'il est représenté sur la figure 12.2 est maintenant directe : pour représenter l'état courant, nous avons besoin d'un registre de deux bits, que nous réalisons avec les deux bascules D en haut à gauche du schéma ; les bascules D sont commandées par l'horloge, ce qui assurera que l'automate change d'état à chaque battement d'horloge.

La valeur *out* présentée en sortie et l'état suivant sont calculés par le bloc de logique combinatoire qui occupe le bas du schéma. Les deux valeurs pour représenter le nouvel état sont fournies en entrées aux bascules D de manière à être mémorisées au prochain battement de l'horloge.

12.2 Réalisation d'un automate avec une entrée

On peut construire avec la même méthode un automate qui traite des données en entrée. Un exemple simple est une machine qui reçoit un nombre représenté en binaire, bit par bit de droite à gauche et multiplie ce nombre par 3.

Commençons par considérer la multiplication d'un bit par 3. Bien sur $3 \times 0 = 0$ et $3 \times 1 = 3$. Comme nous traitons des nombres en binaires, le cas $3 \times 1 = 3_{10} = 11_2$ doit produire un résultat de 1 (pour le bit de droite) et une retenue de 1 (pour le bit de gauche. Cette retenue doit être ajoutée au résultat de la multiplication du bit suivant, quel que soit ce bit. Cela représente les deux cas suivant : $3 \times 0 + 1 = 1$ et retenue$= 0$) quand le bit d'entrée vaut 0 et $3 \times 1 + 1 = 4_{10} = 100_2 = 0$ et retenue$= 10_2$ quand le bit d'entrée vaut 1.

Il faut donc aussi traiter le cas où la retenue vaut $10_2 = 2_{10}$. Cela fait : $3 \times 0 + 10_2 = 10_2 = 0$ et retenue $= 1$ d'une part et $3 \times 1 + 10_2 = 101_2 = 1$ retenue $= 10_2$ d'autre part.

La retenue ne peut donc prendre que les valeurs $0, 1$ et 10_2 et nous avons ainsi examiné tous les cas possibles, résumés dans la table suivante :

$3 \times$ bit + retenue	résultat	bit produit	retenue
$3 \times 0 + 0$	0	0	0
$3 \times 0 + 1$	1	1	0
$3 \times 0 + 2$	$2_{10} = 10_2$	0	1
$3 \times 1 + 0$	$3_{10} = 11_2$	1	1
$3 \times 1 + 1$	$4_{10} = 100_2$	0	10
$3 \times 1 + 2$	$5_{10} = 101_2$	1	10

Il faut mémoriser la retenue de l'opération précédente : pour cela nous utilisons les états de l'automate : chaque état permet de se souvenir de la valeur de la retenue ; à partir d'un état partent deux transitions, étiquetées par 0 et 1 : suivant la valeur du bit en entrée, l'automate prend l'une ou l'autre des transitions pour aller vers un nouvel état qui mémorise la nouvelle valeur de la retenue ; avec chaque transition, il y a aussi la production du bit de résultat correct. L'automate est représenté sur la figure 12.3.

Pour faire fonctionner l'automate, on se place dans l'état de départ (l'état a), puis on effectue son travail : (1) lire le bit suivant du nombre à multiplier, (2)

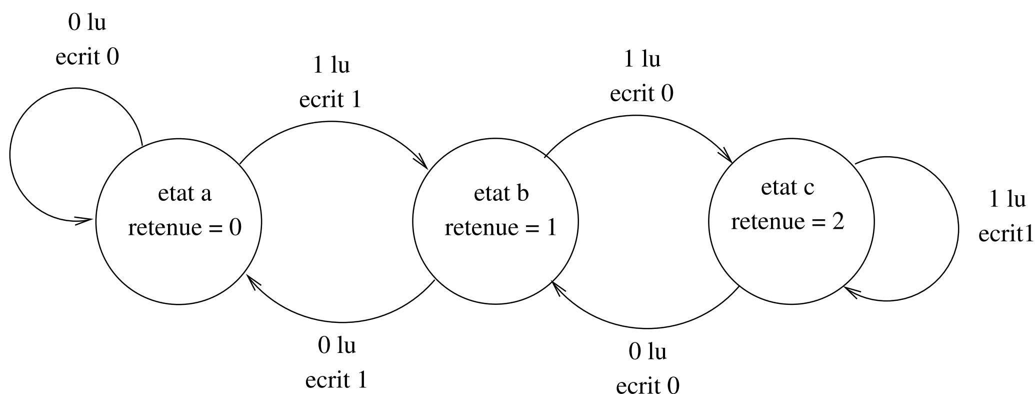


FIGURE 12.3 – L'automate de multiplication par 3. Chaque transition est étiquetée par le bit lu et est accompagnée d'une valeur à produire. Chaque état correspond à une valeur de la retenue.

choisir la transition qui y correspond depuis l'état courant, (3) écrire la valeur indiquée sur la transition, (4) l'état d'arrivée de la transition est le nouvel état courant, recommencer à l'étape (1).

Si l'automate reçoit en entrée les bits 110010 qui correspondent au nombre $010011_2 = 19_{10}$, il va passer par les états a, b, c, b, a, b, a et produire les bits 100111 qui correspondent au nombre $111001_2 = 71_8 = 57_{10}$. Le travail sur cet exemple peut se résumer dans la table suivante :

état	a		b		c		b		a		b		a
bit lu		1		1		0		0		1		0	
bit écrit		1		0		0		1		1		1	

Une fois que l'automate est dessiné, nous sommes en mesure de construire la table de vérité qui y correspond : à chaque combinaison de l'état courant (représenté sur deux bits e_1 et e_0) et du bit entrant i, on fait correspondre le nouvel état (représenté sur deux bits n_1 et n_0) et le bit de sortie o comme dans la table de vérité qui suit :

état courant	e_1	e_0	i	etat suivant	n_1	n_0	o
0	0	0	0	0	0	0	0
0	0	0	1	1	0	1	1
1	0	1	0	0	0	0	1
1	0	1	1	2	1	0	0
2	1	0	0	1	0	1	0
2	1	0	1	2	1	0	1
3	1	1	0	?	?	?	?
3	1	1	1	?	?	?	?

Comme on peut le voir, nous avons complété la table avec les lignes qui rendent

compte de l'état 3, que nous n'utilisons pas, pour lequel les valeurs de sorties sont quelconques ; nous pouvons, si c'est utile, choisir d'utiliser ces valeurs pour simplifier les expressions dans notre table de Karnaugh.

On peut donc calculer les fonctions qui permettent de calculer n_1, n_0 et o comme :

$$\begin{aligned}
n_1 = &\quad \overline{e1}.e0.i + e1.\overline{e0}.i \\
n_0 = &\quad \overline{e1}.\overline{e0}.i + e1.\overline{e0}.\overline{i} \\
o = &\quad \overline{e1}.\overline{e0}.i + \overline{e1}.e0.\overline{i} + e1.\overline{e0}.i
\end{aligned}$$

On utilise les tables de Karnaugh pour simplifier ces expressions et on trouve pour n_1 :

	$e_1 e_0$	$e_1 \overline{e_0}$	$\overline{e_1}.\overline{e_0}$	$\overline{e_1} e_0$
i	?	V		V
$\overline{i}$	?			

donc $n_1 = e_1.i + e_0.i$.

	$e_1 e_0$	$e_1 \overline{e_0}$	$\overline{e_1}.\overline{e_0}$	$\overline{e_1} e_0$
i	?		V	
$\overline{i}$	?	V		

donc $n_0 = e_1.\overline{i} + \overline{e_1}.\overline{e_0}.i$.

	$e_1 e_0$	$e_1 \overline{e_0}$	$\overline{e_1}.\overline{e_0}$	$\overline{e_1} e_0$
i	?	V	V	
$\overline{i}$	?			V

donc $o = \overline{e_0}.i + e_0.\overline{i}$.

Il suffit maintenant de réaliser le circuit comme sur la figure 12.4 pour avoir une implémentation de l'automate. Le circuit à proprement parler est à l'intérieur du rectangle pointillé ; il y a deux signaux en entrées et un signal en sortie

En entrée l'horloge sert à cadencer le fonctionnment. Sur le signal i sont présentés, à chaque battement de l'horloge, la valeur du bit suivant du nombre à multiplier. En sortie, le signal o contiendra la valeur du bit suivant du résultat de la multiplication du nombre en entrée.

Lors du battement de l'horloge, la valeur du bit présent sur i est mémorisé dans la bascule D du bas, alors que les deux bascules du haut prennent la valeur du prochain état destination calculé par la dernière transition.

Ces valeurs sont alors envoyées dans le bloc de logique combinatoire qui occupe toute la partie droite du circuit. Il produit la valeur du prochain bit du résultat (sur o) et le prochain état (sur n_0 et n_1) qui sera mémorisés lors du prochain battement de l'horloge.

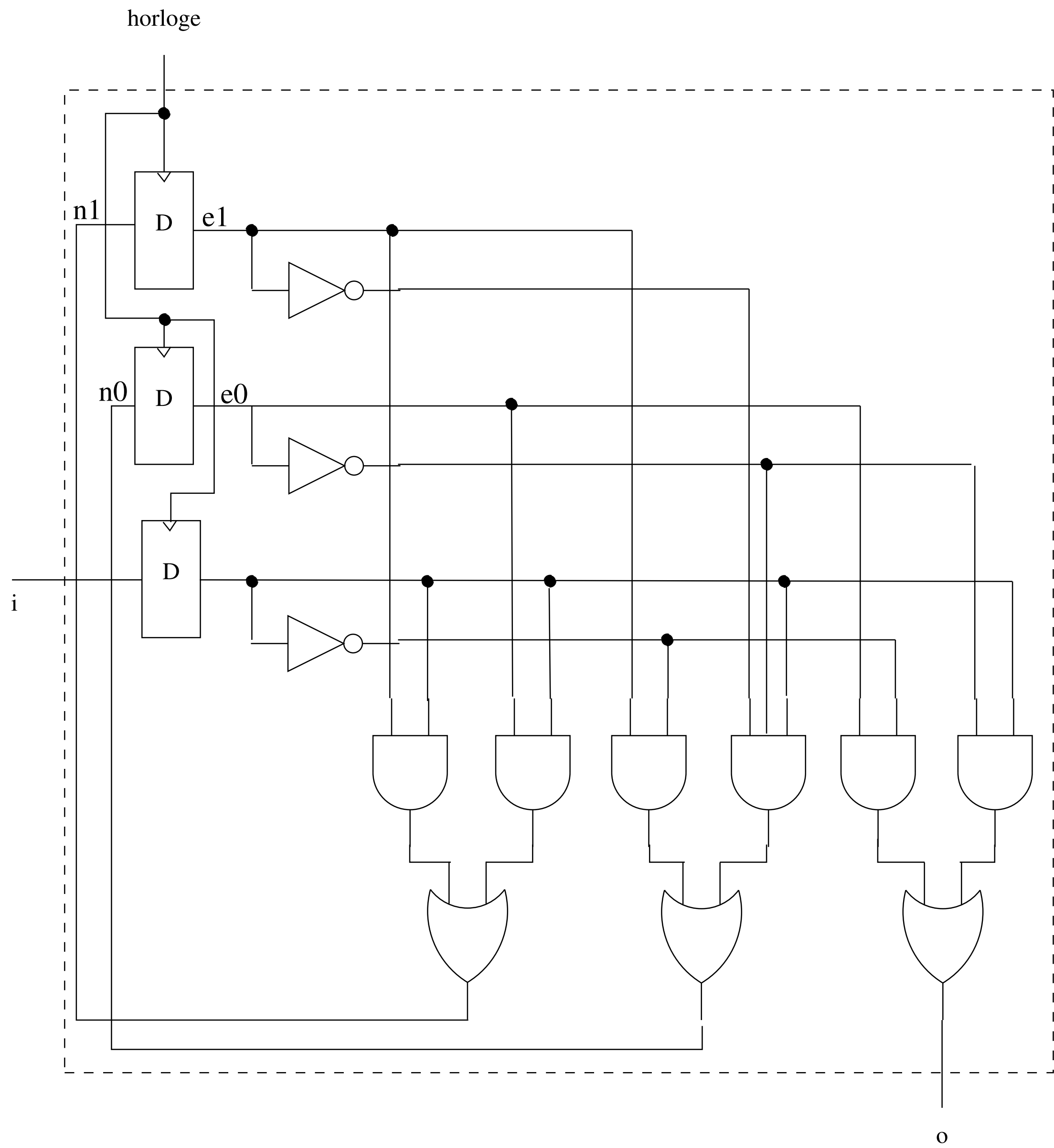


FIGURE 12.4 – Une réalisation de l'automate pour multiplier (la représentation d')un nombre (en binaire) par 3.

12.3 Un automate général

On peut réaliser sur le même mode un automate quelconque : un registre contient l'état courant ; un bloc de logique calcule les signaux de sortie et l'état suivant à partir de l'état courant et des signaux en entrée ; une horloge assure la transition d'un état au suivant. On a représenté l'architecture générale des automates sur la figure 12.5.

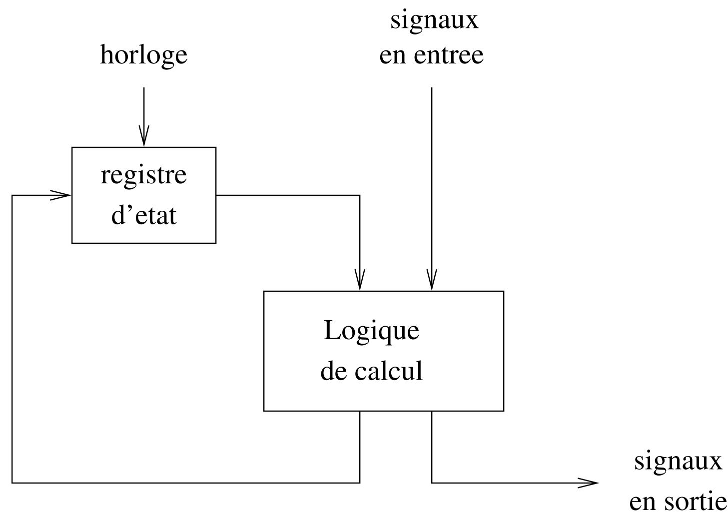


FIGURE 12.5 – L'architecture générale d'un automate

12.4 Exercices

Exercice 12.1 : Réaliser un automate qui produise la séquence de valeurs 11001 en utilisant la même technique que pour le premier automate simple.

Exercice 12.2 : Dessiner le graphe d'un automate qui réalise une multiplication par 5 sur le modèle de celui de la multiplication par 3. Quelle sera l'équation qui permettra de calculer la sortie en fonction des bits du numéro de l'état ?

Exercice 12.3 : L'automate de la figure 12.6 lit un nombre binaire bit par bit et détermine si c'est un multiple de 3. Dessiner le circuit qui lui correspond.

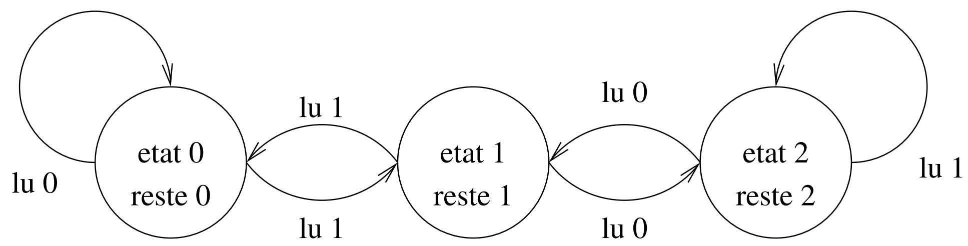


FIGURE 12.6 – Quand l'automate est dans l'état 0, le nombre lu jusqu'à maintenant est un multiple de 3.

Chapitre 13

L'ordinateur en papier

Voici un ordinateur qui fonctionne sans électricité ; il suffit d'un papier, d'un crayon et d'une gomme (et d'un peu de patience) pour le faire fonctionner. (Ce chapitre est une retranscription du support de cours conçu par Vincent LESBROS en octobre 1990.)

Son schéma est réduit au minimum mais le principe de fonctionnement est inspiré de celui de petits processeurs séquentiels qui existent réellement.

13.1 Anatomie

L'ordinateur en papier possède une mémoire (la grille carrée) dont chaque case est numérotée. Chaque case peut contenir une valeur numérique (notée en base seize ou système de numération hexadécimal, avec deux chiffres) variant de 00 à FF.

La mémoire contient deux registres :

— Le *Registre de Sélection* (RS) permet de désigner une case de la mémoire par son numéro (ou adresse).

— Le *Registre Mot* (RM) permet d'échanger des valeurs entre la mémoire et l'unité centrale.

La mémoire est de 256 cases, donc les adresses des cases vont de 00 à FF (et peuvent être également contenues dans des cases de la mémoire pour constituer un *pointeur*).

L'unité centrale comporte une unité de contrôle et une Unité Arithmétique et Logique (UAL). L'ordinateur est doté en outre des indispensables moyens de communication avec l'extérieur : une entrée pour saisir des données et une sortie pour exprimer les résultats des traitements.

L'unité de contrôle contient :

— une horloge (non figurée) qui cadence le déroulement des opérations

— le registre (PC) *Program Counter* qui désigne l'adresse en mémoire de la

prochaine instruction du programme à exécuter.

— le séquenceur a accès au *Registre Instruction* (RI), lui-même constitué de deux registres (OP) et (AD) contenant respectivement le *code opératoire* de l'instruction en cours et l'adresse de la donnée à traiter.

L'unité arithmétique et logique est capable d'effectuer des additions, des soustractions et des opérations booléenne.

L'opération à faire s'inscrit dans la case trapézoïdate, les opérandes sont l'*Accumulateur* (le registre (A)) et le registre mot (RM) ; les résultats des opérations sont toujours rendus dans l'accumulateur.

Les différents registres sont des cases pouvant contenir une valeur de huit bits (deux chiffres hexadécimaux) comme les cases de la mémoire. Les registres sont reliès entre eux par des lignes (les bus) et les transferts de valeur par ces lignes sont gouvernés par le séquenceur (suivant l'instruction en cours).

13.2 Fonctionnement

Cette petite machine universelle ne peut fonctionner sans programme : il faut inscrire dans la mémoire une suite d'instructions à effectuer (ou exécuter) et placer dans le registre (PC) l'adresse de la première instruction.

Pour écrire le programme, reportez-vous aux rubriques suivantes : *langage d'assemblage* et *modes d'adressage*.

Quand le programme est écrit, faites *tourner* le programme en répétant le cycle. Le cycle est constitué de trois phases : La première phase est la recherche d'instruction, la deuxième phase contient le décodage de l'instruction et sa réalisation proprement dite (recherche de l'opérande et calcul éventuel), la troisième phase (qui peut être omise dans certains cas) permet de pointer vers l'instruction suivante.

La rubrique *cycle* vous indique pour chaque phase la séquence de microcode à effectuer. (Les microcodes sont notés dans des cercles.) Pour la deuxième phase, la séquence de microcode dépend du code opératoire inscrit dans le registre OP.

La rubrique *les microcodes* vous donne, pour chaque microcode, les choses à faire avec votre crayon et votre gomme.

13.3 Le langage d'assemblage

Mnémoniques	Code opératoire	description
arithmétique		
ADD $\sharp$	20	$A \leftarrow A + V$
ADD α	60	$A \leftarrow A + (\alpha)$
ADD $*\alpha$	E0	$A \leftarrow A + *(\alpha)$
SUB $\sharp$	21	$A \leftarrow A - V$
SUB α	61	$A \leftarrow A - (\alpha)$
SUB $*\alpha$	E1	$A \leftarrow A - *(\alpha)$
logique		
NAND $\sharp$	22	$A \leftarrow \neg[A \& V]$
NAND α	62	$A \leftarrow \neg[A \& (\alpha)]$
NAND $*\alpha$	E2	$A \leftarrow \neg[A \& *(\alpha)]$
transferts		
LOAD $\sharp$	00	$A \leftarrow V$
LOAD α	40	$A \leftarrow (\alpha)$
LOAD $*\alpha$	C0	$A \leftarrow *(\alpha)$
STORE α	48	$(\alpha) \leftarrow A$
STORE $*\alpha$	C8	$*(\alpha) \leftarrow A$
Entrées / Sorties		
IN α	49	$(\alpha) \leftarrow$ Entrée
IN $*\alpha$	C9	$*(\alpha) \leftarrow$ Entrée
OUT α	41	Sortie $\leftarrow (\alpha)$
OUT $*\alpha$	C1	Sortie $\leftarrow *(\alpha)$
Branchement inconditionnel		
JUMP α	10	PC $\leftarrow \alpha$
conditinonels		
BRN α	11	si $A < 0$ alors PC $\leftarrow \alpha$
BRZ α	12	si $A = 0$ alors PC $\leftarrow \alpha$

Légende :

La flèche $\leftarrow$représente l'affectation.
(α) représente le contenu de la case d'adresse alpha.
$*(\alpha)$ représente le contenu de la case dont l'adresse est dans la case d'adresse alpha.
V est la valeur donnée immédiatement.
$\neg$ représente le non logique.
& représente le *et* logique.
Les crochets [ ] groupent les termes sur lesquels s'applique une opération.

Chaque instruction est codée sur deux octets, le premier est le code opératoire (dans un rectangle), le second est la valeur ou l'adresse de l'opérande.

Par exemple :
LOAD 1F (charger le contenu de la case 1F dans A)
s'encode avec : 40 1F dans deux cases consécutives.
LOAD ♯00 (mettre 00 dans A)
s'encode avec : 00 00, toujours sur deux octets.

13.4 Les modes d'addressage

Les modes d'adressages correspondent aux différentes façons d'accéder aux données à traiter.

Adressage immédiat

Dans l'adressage immédiat, on donne directement la valeur avec laquelle l'opération est à effectuer.

Exemples

Mnémoniques	hexadécimal	description
LOAD ♯FF	00 FF	Charge l'accumulateur avec la valeur FF.

Mnémoniques	hexadécimal	description
ADD ♯01	20 01	Incrémente l'accumulateur (c'est à dire ajoute 1 à son contenu)

Adressage absolu

L'adressage absolu permet de désigner une valeur par son adresse en mémoire (ce mode est qualifié d'*absolu* par opposition aux modes dits *relatifs* qui spécifient une adresse en mémoire grâce à un déplacement par rapport à une adresse de référence ; ici, la référence est le début de la mémoire).

Exemple

Mnémoniques	hexadécimal
LOAD 22	40
	22
ADD 20	60
	20

Ce programme charge l'accumulateur avec la valeur contenue dans la case mémoire d'adresse 22 puis ajoute à cette valeur le contenu de la case mémoire 20 ; le résultat est dans l'accumulateur.

Adressage indirect

L'adressage indirect permet de faire une *indirection*, c'est à dire de désigner la valeur à traiter en spécifiant l'adresse d'une case qui contient l'adresse de la valeur.

Une case mémoire contenant l'adresse d'une autre case mémoire est appelée *pointeur*. Ce mode permet de spécifier une valeur en donnant l'adresse du pointeur vers la valeur.

Exemple

Mnémoniques	hexadécimal
SUB * 1F	E1
	1F

Par cette instruction, l'accumulateur va être décrémenté de la valeur dont l'adresse est dans la case mémoire d'adresse 1F. Si, avant cette instruction, l'accumulateur contenait 10, la case mémoire 1F contenait AF et la case AF contenait 2, alors l'accumulateur contiendra 0E après l'exécution de l'instruction (10 − 02 = 0E).

13.5 Les microcodes

Cette rubrique contient la descrption des opérations à effectuer pour chacun des microcodes.

Transferts

Gommez le contenu du registre à gauche de la flèche et y inscrire le contenu de celui de droite.

① (RS) ←(PC)

② (PC) ←(RM) ; ne pas faire la phase III.

③ (A) ←(RM)

④ (RM) ←(A)

⑤ (OP) ←(RM)

⑥ (AD) ←(RM)

⑦ (RS) ←(AD)

⑧ (RM) ←(Entrée)

⑨ (Sortie) ←(RM)

Préparation du calcul

(10) Écrire + dans UAL (la case trapézoïdale)

(11) Écrire − dans UAL (la case trapézoïdale)

(17) Écrire NAND dans UAL (la case trapézoîdale)

Calculer

(12) Effectuer l'opération affichée sur l'UAL en prenant le contenu de (A) comme premier opérande et le contenu de (RM) comme second. Ecrivez le résultat dans l'accumulateur (A).

Commandes de la mémoire

(13) Lecture de la mémoire :
Décodez l'adresse contenue dans le registre de sélection (RS) en prenant le chiffre gauche (ou Haut) pour numéro de colonne et le droit (ou Bas) pour numéro de ligne. Reportez le contenu de la case mémoire ainsi désignée dans le registre mot (RM).

(14) Ecriture en mémoire :
Décodez l'adresse contenue dans le registre de sélection (RS) de la même façon que pour la lecture. Reportez le contenu du registre mot (RM) dans la case mémoire.

Divers

(15) Incrémenter le contenu de (PC) le pointeur ordinal pour pointer sur l'instruction ou la valeur suivante.

(16) Attendre qu'une valeur soit écrite dans l'Entrée.

13.6 Le cycle

Phase I : Recherche de l'instruction

(1) (13) (5) (15) passer à la phase II

Phase II Décodage de l'instruction, recherche d'opérande et calcul

Valeur de (OP)	microcodes
00	(1) (13) (3)
10	(1) (13) (2) puis passer à la phase I
11	Si (A) <0 (1) (13) (2) puis phase I
12	Si (A) $=0$ (1) (13) (2) puis phase I
20	(10) (1) (13) (12)
21	(11) (1) (13) (12)
22	(17) (1) (13) (12)
40	(1) (13) (6) (7) (13) (3)
41	(1) (13) (6) (7) (13) (9)
48	(1) (13) (6) (7) (4) (14)
49	(1) (13) (6) (7) (16) (8) (14)
60	(10) (1) (13) (6) (7) (13) (12)
61	(11) (1) (13) (6) (7) (13) (12)
62	(17) (1) (13) (6) (7) (13) (12)
C0	(1) (13) (6) (7) (13) (6) (7) (13) (3)
C1	(1) (13) (6) (7) (13) (6) (7) (13) (9)
C8	(1) (13) (6) (7) (13) (6) (7) (4) (14)
C9	(1) (13) (6) (7) (13) (6) (7) (16) (8) (14)
E0	(10) (1) (13) (6) (7) (13) (6) (7) (13) (12)
E1	(11) (1) (13) (6) (7) (13) (6) (7) (13) (12)
E2	*A faire* (voir exercice)

Phase III

(15) puis phase I

13.7 Exemple de programme : un boot-strap

Le *boot-strap* est le programme de lancement de l'ordinateur ; il est implanté à une adresse qui est placée dans le PC à chaque démarrage, c'est donc le premier programme à être exécuté et il est chargé de faire le nécessaire pour la mise en place et le fonctionnement du système d'exploitation.

Le programme qui suit permet de charger en mémoire votre propre programme et de l'exécuter.

Mode d'emploi :

— Ecrivez le programme à partir de l'adresse 00

— Allumez l'ordinnateur : (PC) ← 00

— Entrez l'adresse de début de votre programme

— Entrez la taille (en nombre d'octets) utilisée par votre programme, puis entrez tour à tour chaque code constituant votre programme.

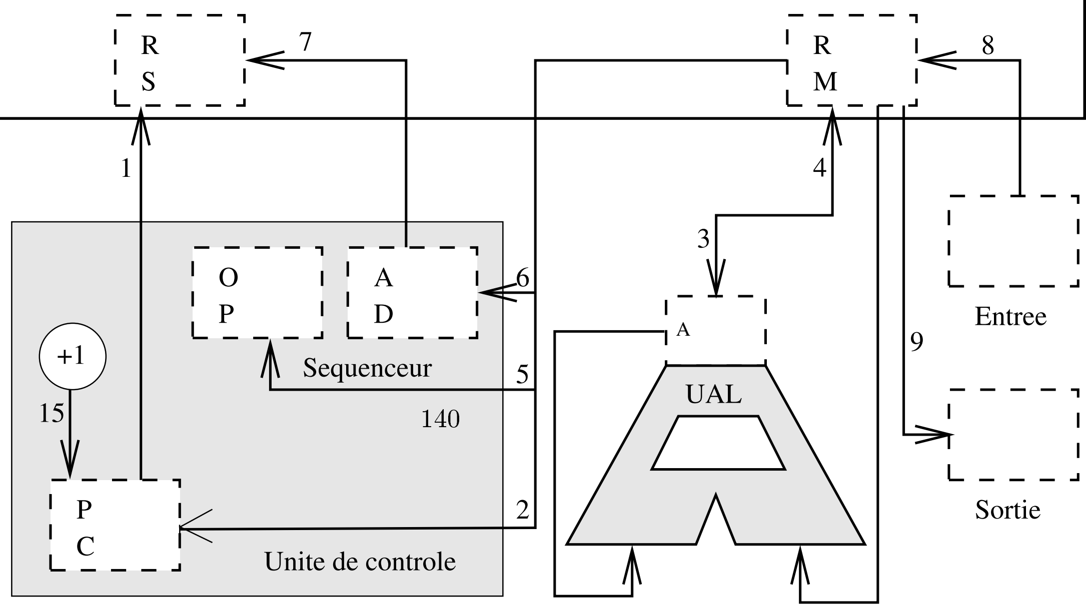
adresse	*code*	*mnémonique*	*commentaire*
00	49	IN 20	Entrez l'adresse de chargement du programme
01	20		
02	49	IN 22	Entrez la taille du programme
03	22		
04	40	LOAD 20	A ←adresse du programme
05	20		
06	48	STORE 21	Adresse courante ←A
07	21		
08	C9	IN * 21	Entrez une instruction ou une valeur
09	21		
0A	40	LOAD 22	A ←Nb. d'octets restant à charger
0B	22		
0C	21	SUB ♯1	Décrémenter le Nb. d'octets
0D	01		
0E	12	BRZ 1F	Exécuter le programme si tout est chargé
0F	1F		
10	48	STORE 22	Ranger le nouveau Nb. d'octets
11	22		
12	40	LOAD 21	Reprendre l'adresse courante dans A
13	21		
14	20	ADD ♯1	l'incrémenter
15	01		
16	10	JUMP 06	Continuer à charger
17	06		
18	1E	00	
1F	10	JUMP ? ?	Brancher sur ...
20	? ?		l'adresse de début
21	? ?		Adresse courante
22	? ?		Taille (ou Nb. d'octets)

Les adresses 00 à 1F sont censées être en mémoire morte et la mémoire vive commence à 20.

13.8 Exercices

Exercice 13.1 : Il manque les microcodes de la phase 2 pour le cas où OP

B \ H	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																



contient E62. Compléter.

Exercice 13.2 : Étant donné la logique qui existe entre les codes opératoires et le microcode, à quelle code pourrait correspondre l'instruction `IN` $\sharp$?

Exercice 13.3 : Étant donné la logique qui existe entre les codes opératoires et le microcode, à quelle opération pourrait correspondre le code 01 ?

Exercice 13.4 : (Piégé) Dans le programme de bootstrap, les adresses 18 à 1E sont remplies de 0. Que se passe-t-il au cours du boot-strap quand l'ordinateur exécute ces instructions ?

Exercice 13.5 : Écrire un programme pour l'ordinateur en papier qui lit deux nombres et affiche leur produit. Comme pour le programme de bootstrap donné dans le chapitre, indiquer les adresses, les codes, les mnémoniques et les commentaires. (on peut calculer le produit avec une suite d'additions simple ; faute de décalage à droite, l'ordinateur en papier ne nous permet pas d'utiliser l'algorithme plus rapide vu en cours).

Exercice 13.6 : Même question que précédemment pour la division, avec des soustractions successives.On pourra s'inspirer de l'algorithme suivant :

Etant donné x *et* y*, calculer* a *et* b *tels que* $a \times x + b = y$
$a \leftarrow 0$
$b \leftarrow y$
tant que $b > x$
    $a \leftarrow a + 1$
    $b \leftarrow b - x$

Exercice 13.7 : Quelles parties de l'ordinateur faut-il modifier pour ajouter une instruction `SHIFT` de décalage à droite ? Décrire ces modifications, choisir un code opératoire, détailler les modifications à apporter au séquenceur.

Exercice 13.8 : (Assez difficile) Dans la mémoire, on trouve de l'adresse 30 à 45 les valeurs suivantes : 10, 3a, 00 31 40 32 60 33 48 32 49 33 40 33 22 ff 12 34 41 32 10 44. Dans PC se trouve 30. Que fera ce programme quand l'ordinateur en papier va tourner et l'exécuter ?

Exercice 13.9 : (Assez facile) Indiquer les mnémoniques qui correspondent aux valeurs suivantes dans la mémoire.

(Difficile) Commenter le programme. Que fait-il ?

adresse	valeur
50	49 70
52	40 70
54	48 71
56	48 72
58	00 5E
5A	48 8d
5C	10 74
5E	40 71
60	48 72
62	40 70
64	48 71
66	00 6C
68	48 8d
6A	10 74
6C	41 71
6E	10 6E
70	00
71	00
72	00
73	00
74	00 00
76	48 73
78	40 71
7A	12 88
7C	21 01
7E	48 71
80	40 73
82	60 72
84	48 73
86	10 78
88	40 73
8a	48 71
8c	10 00

Exercice 13.10 : (Amusant mais difficile) Écrire un programme qui met la plus grande partie possible de la mémoire à 0 (en supposant qu'il n'y a pas de mémoire morte).

Exercice 13.11 : (Long, pas de corrigé) Écrire un programme d'extraction de la racine carrée d'un nombre, avec la méthode de Newton pour l'ordinateur en papier.

Chapitre 14

Projet et évaluation

Ce court chapitre est destiné à vous permettre de donner votre évaluation du cours. Merci de prendre aussi le temps de répondre aux questions suivantes pour m'aider à évaluer la façon dont le cours s'est passé de votre point de vue.

Notez que l'idée est d'essayer d'améliorer les choses, aussi les réponses excessives sont à peu près inutiles. Par exemple avec *« Le chapitre le plus intéressant ? Aucun. Le chapitre le moins intéressant ? Tous. »*, on ne peut pas faire grand chose.

— Le contenu du cours a-t-il correspondu à ce que vous attendiez ? (si non, de quelle manière)
— Qu'est-ce qui vous a le plus surpris (en bien) ?
— Qu'est-ce qui vous a le plus déçu ?
— Quels étaient les chapitres les plus difficiles ?
— Quels étaient les chapitres les plus faciles ?
— Quels étaient les chapitres les plus intéressants ?
— Quels étaient les chapitres les moins intéressants ?
— Que me suggéreriez-vous de modifier dans l'ordre des chapitres ?
— Qu'est-ce qui est en trop dans le cours ?
— Qu'est-ce qui manque dans le cours ?
— Comment étaient les exercices du point de vue quantité (pas assez, trop).
— Comment étaient les exercices du point de vue difficulté (trop durs, trop faciles) ?
— Que donneriez-vous comme conseil à un étudiant qui va suivre le cours ?
— Que me donneriez-vous comme conseil en ce qui concerne le cours ?
— Si vous deviez mettre une note globale au cours, entre 0 et 20, laquelle mettriez-vous ?
— Quelles questions manque-t-il pour évaluer correctement le cours (et bien évidemment, quelle réponse vous y apporteriez) ?

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http ://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom : to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation : a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals ; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under

copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires

to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties : any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts : Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more

than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version :

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in

the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version..

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another ; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this

License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History" ; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sec-

tions with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http ://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be

used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.