

# **INFO-F-102 – Fonctionnement des ordinateurs**

## **Notes de cours**

Gilles GEERAERTS

Année académique 2018–2019

Document mis en page à l'aide de  $\text{\LaTeX}$ , en utilisant la classe KOMA script. Version du 24 octobre 2018.

# Table des matières

<b>I. Notions de base</b>	<b>13</b>
<b>1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur ?</b>	<b>15</b>
1.1. Définitions et étymologie . . . . .	15
1.2. Un premier modèle : l'architecture de VON NEUMANN . . . . .	17
1.3. Une vision différente : structure en niveaux et traductions . . . . .	22
<b>2. Leçons 2 à 4 – Représentation de l'information</b>	<b>27</b>
2.1. Unités de quantité d'information . . . . .	28
2.2. Écriture des nombres dans différentes bases . . . . .	29
2.2.1. Changements de base . . . . .	32
2.2.2. Opérations en base 2 . . . . .	36
2.2.3. Représentation des entiers non-signés . . . . .	39
2.2.4. Représentation des nombres entiers signés . . . . .	39
2.2.5. Représentation des nombres « réels » . . . . .	43
2.3. Représentation des caractères . . . . .	47
2.4. Représentation d'images . . . . .	50
2.5. Représentation des instructions . . . . .	52
2.6. Détection et correction d'erreurs . . . . .	54
2.6.1. Bit de parité . . . . .	54
2.6.2. Code de HAMMING . . . . .	55
2.6.3. Applications des codes correcteurs d'erreur . . . . .	58
2.7. Conclusion : sémantique d'une représentation binaire . . . . .	59
<b>3. Leçon 5 &amp; 6 – Organisation de l'ordinateur</b>	<b>63</b>
3.1. Mémoire primaire . . . . .	63
3.1.1. Structure et adresses . . . . .	64
3.1.2. Mémoire cache . . . . .	65
3.1.3. Ordre des octets dans les mots . . . . .	66
3.1.4. Réalisation de la mémoire primaire . . . . .	68
3.2. Le processeur . . . . .	72
3.2.1. Composants du processeur . . . . .	72
3.2.2. Le chemin des données – <i>datapath</i> . . . . .	74
3.2.3. Exécution des instructions . . . . .	75
3.2.4. Machine à pile . . . . .	76
3.2.5. Choix du jeu d'instructions machine . . . . .	78
3.2.6. Techniques pour améliorer l'efficacité des processeurs . . . . .	80

## Table des matières

3.3. Les périphériques . . . . .	87
3.3.1. Mémoire secondaire . . . . .	87
3.3.2. Les périphériques d'entrée/sortie . . . . .	91
<b>II. Les portes logiques</b>	<b>93</b>
<b>4. Leçons 6 à 9 – Niveau 0 : portes logiques</b>	<b>95</b>
4.1. L'algèbre Booléenne . . . . .	95
4.2. Les circuits logiques . . . . .	102
4.2.1. Réalisation des portes logiques . . . . .	102
4.2.2. Réalisation d'un circuit logique . . . . .	103
4.3. Circuits pour réaliser l'arithmétique binaire . . . . .	105
4.3.1. Circuit additionneur . . . . .	105
4.3.2. Demi-additionneur . . . . .	106
4.3.3. Décalage . . . . .	109
4.3.4. Décodeur . . . . .	111
4.3.5. Le « sélecteur » . . . . .	113
4.3.6. ALU simplifiée (1 bit) . . . . .	115
4.3.7. ALU $n$ bits . . . . .	117
4.4. Circuits pour réaliser des mémoires . . . . .	119
4.4.1. Mémoire élémentaire . . . . .	119
4.4.2. Bascules . . . . .	120
4.4.3. Flip-flops . . . . .	124
4.4.4. Un circuit de mémoire complet . . . . .	125
<b>III. Le micro-langage</b>	<b>131</b>
<b>5. Leçons 10 à 12 – La micro-architecture</b>	<b>133</b>
5.1. Exemple de langage machine : l'IJVM . . . . .	133
5.1.1. Exemples de programmes en IJVM . . . . .	134
5.1.2. Représentation d'un programme IJVM en mémoire . . . . .	134
5.2. Exemple de <i>datapath</i> : le MIC 1 . . . . .	135
5.2.1. Cycle d'exécution . . . . .	138
5.2.2. Communication avec la mémoire . . . . .	139
5.2.3. Micro-instructions . . . . .	140
5.3. Réalisation de l'IJVM à l'aide du MIC1 . . . . .	140
5.3.1. Interpréteur . . . . .	141
5.3.2. Manipulation du <i>stack</i> . . . . .	141
5.3.3. Opérations POP et BIPUSH . . . . .	141
5.3.4. Opérations IADD et ISUB . . . . .	142
5.3.5. Opérations de saut : GOTO et IFEQ . . . . .	143

<b>IV. Le langage Machine</b>	<b>145</b>
<b>6. Leçon 13 – Langage machine</b>	<b>147</b>
6.1. Caractéristiques . . . . .	147
6.1.1. Modèle mémoire . . . . .	147
6.1.2. Registres . . . . .	148
6.2. Introduction et exemple du Pentium IV . . . . .	148
6.3. Types de données . . . . .	148
6.4. Instructions . . . . .	149
6.5. Adressage . . . . .	150
6.6. Types d'instructions . . . . .	151
6.7. Modes d'exécution . . . . .	153
6.8. Exemple de programme en langage machine . . . . .	154
<b>7. Leçon 14 – Interruptions et traps</b>	<b>157</b>
7.1. Les traps . . . . .	157
7.2. Les interruptions . . . . .	158
<b>V. Le système d'exploitation</b>	<b>169</b>
<b>8. Leçon 15 – Système d'exploitation</b>	<b>171</b>
8.1. Types d'OS . . . . .	172
8.2. OS et interruptions . . . . .	173
8.3. Appels système . . . . .	173
<b>9. Leçon 16 et 17 – Gestion de la mémoire primaire</b>	<b>175</b>
9.1. Pagination . . . . .	176
9.1.1. Travail du MMU . . . . .	177
9.2. Pagination à la demande . . . . .	178
9.3. Exemple : mémoire virtuelle l'Ultra Sparc III . . . . .	180
<b>10. Leçon 18 – Gestion des processus et de la mémoire secondaire</b>	<b>183</b>
10.1. Gestion des processus . . . . .	183
10.1.1. Cycle de vie d'un processus . . . . .	183
10.1.2. Systèmes en <i>Time sharing</i> . . . . .	185
10.2. Entrées/sorties virtuelles et systèmes de fichiers . . . . .	186
10.2.1. Fichiers . . . . .	186
10.2.2. Répertoires . . . . .	187
10.2.3. Quelques remarques . . . . .	188



# Table des figures

1.1. Lettre de J. PERRET . . . . .	16
1.2. Un exemple de programme écrit en langage Python. . . . .	18
1.3. L'architecture VON NEUMANN, un premier modèle d'ordinateur. . . . .	21
1.4. L'architecture de l'i486 . . . . .	23
1.5. Les 6 couches décrivant le fonctionnement d'un ordinateur. . . . .	26
2.1. Le code Baudot . . . . .	49
2.2. Le code ASCII . . . . .	50
2.3. Le <i>Code Page 737</i> . . . . .	51
2.4. Extrait du standard Unicode, alphabet Tagbanwa [21]. . . . .	52
2.5. Un exemple de contenu de fichier PGM avec l'image qu'il représente. . . . .	53
2.6. Un code QR . . . . .	59
2.7. Une représentation, différentes interprétations . . . . .	61
3.1. Un modèle simple de l'organisation d'un ordinateur. . . . .	63
3.2. Petit et gros boutistes. . . . .	67
3.3. Un tube Williams. . . . .	69
3.4. Deux mémoires à lignes de délai . . . . .	70
3.5. Détail d'une mémoire à tores de ferrite. . . . .	71
3.6. Un exemple de programme en langage machine i486. . . . .	74
3.7. Le chemin des données . . . . .	75
3.8. Un exemple de pile. . . . .	77
3.9. Un exemple de langage machine i486 utilisant le FPU. . . . .	78
3.10. Illustration d'un <i>pipeline</i> à cinq étages. . . . .	81
3.11. Un <i>pipeline</i> superscalaire . . . . .	81
3.12. Un ordinateur CDC 6600. . . . .	82
3.13. L'ILLIAC IV . . . . .	84
3.14. Une Cray 1 . . . . .	85
3.15. L'organisation d'un ordinateur avec 2 CPUs . . . . .	86
3.16. Une carte perforée au format standard IBM . . . . .	88
3.17. Du ruban perforé. . . . .	88
3.18. Trois types de disquettes : de 8, 5,25, et 3,5 pouces. . . . .	90
3.19. Illustration d'un disque dur. . . . .	90
4.1. Quelques portes logiques de base. . . . .	102
4.2. Exemples de transistors. . . . .	102
4.3. Exemple de tubes IBM. . . . .	104
4.4. Le demi-additionneur. . . . .	107

## Table des figures

4.5. Un additionneur complet . . . . .	108
4.6. Additionneur $n$ bits . . . . .	109
4.7. Le circuit de décalage 3 bits. . . . .	111
4.8. Un décodeur 4 bits. . . . .	112
4.9. Des afficheurs 7 segments . . . . .	113
4.10. Le principe d'un sélecteur $n$ bits. . . . .	114
4.11. Une ALU 1 bit avec 4 opérations. . . . .	116
4.12. La représentation d'une ALU. . . . .	118
4.13. Une ALU 4 bits réalisée sur base de 4 ALUs 1 bit . . . . .	118
4.14. Une boucle de portes <b>non</b> qui permet de stocker une valeur binaire. . . . .	121
4.15. Une bascule. . . . .	122
4.16. Une bascule avec horloge. . . . .	124
4.17. Une bascule $D$ . . . . .	124
4.18. Un générateur de pulsation . . . . .	125
4.19. Le circuit logique d'une flip-flop. . . . .	126
4.20. Une flip-flop, avec ses deux entrées $D$ et horloge, et ses deux sorties $Q$ et $\bar{Q}$ . . . . .	126
4.21. Mémoire : lecture du bit $i$ parmi 4 adresses. . . . .	127
4.22. Mémoire : écriture d'un bit $i$ parmi 4 adresses. . . . .	129
4.23. Une mémoire de 4 mots de 3 bits. . . . .	130
5.1. Le chemin des données du Mic1 . . . . .	136
5.2. Le Mic1 complet . . . . .	137
7.1. La boucle d'interprétation du CPU. . . . .	160
7.2. La boucle d'interprétation du CPU avec gestion des interruptions. . . . .	161
7.3. La boucle d'interprétation du CPU avec gestion des interruptions et machine non-interruptible en cas d'interruption. . . . .	162
7.4. Exemple d'interruption (1). . . . .	163
7.5. Exemple d'interruption (2). . . . .	164
7.6. Exemple d'interruption (3). . . . .	165
7.7. Exemple d'interruption (4). . . . .	166
7.8. Exemple d'interruption (5). . . . .	167

# Liste des tableaux

2.1. Comparaison des différentes représentations (sur $n$ bits). . . . .	43
4.1. Les 4 opérations de notre exemple d'ALU, ainsi que les entrées qui y correspondent. . . . .	115
5.1. Les opcodes IJVM que nous utiliserons. La liste complète peut être trouvée dans [22]. . . . .	135



# Au lecteur

Ce document constitue les notes du cours INFO-F102 « Fonctionnement des Ordinateurs ».

La rédaction de ce document a commencé de manière un peu précipitée en septembre 2017, car la maison qui éditait l'ouvrage servant de livre de référence [22] nous a annoncé de manière brutale qu'elle mettait fin à cette édition fin août 2017. Une première version du document a donc été compilée à la hâte pour l'année académique 2017–2018, avec l'espoir de l'améliorer au cours du temps.

La version que vous tenez en main est la seconde version de ces notes, valable pour l'année académique 2018–2019. Par rapport à l'année dernière, les deux premières parties (Notions de base et Portes logiques) ont été entièrement révisées et sont maintenant complètes. Les parties restantes sont toujours dans un état moins satisfaisant : leur style est souvent très elliptique, et l'auteur de ces lignes s'excuse d'avance pour la piètre qualité de la rédaction...

Dans les parties I et II, les conventions typographiques suivantes ont été adoptées pour les exemples et les concepts les plus importants :



Ceci est un exemple illustrant une notion exposée par ailleurs.



Ceci est une définition ou un concept particulièrement important.

Un document décrivant en détail les objectifs pédagogiques du cours et son organisation pratique est disponible sur l'UV. De nombreuses autres ressources (comme les fichiers des circuits du Chapitre 4) y sont également présentes, et constituent un complément indispensable à ces notes.

Bruxelles, septembre 2018



**Première partie**

**Notions de base**



# 1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur ?

## 1.1. Définitions et étymologie

L'objet de ce cours est, comme son titre l'indique, d'expliquer comment *fonctionne* un ordinateur. Le but ici est de dégager une série de principes de base qui régissent et permettent d'expliquer le fonctionnement des ordinateurs tels que nous les connaissons aujourd'hui, mais aussi tels qu'ils ont toujours existé (et, espérons le, tels qu'ils existeront dans le futur). Nous allons donc commencer par nous demander ce qu'est un ordinateur.

Commençons par observer que le mot « ordinateur » est un mot relativement neuf dans l'acception qui nous intéresse ici. Il a été proposé par Jacques PERRET<sup>1</sup>, qui avait été consulté par le directeur d'IBM France en 1955. IBM cherchait à l'époque un nom français pour ces nouvelles machines que l'on nommait *computer* en anglais. Jacques PERRET répond par une lettre du 16 avril 1955 restée célèbre (voir FIGURE 1.1).

En ce qui concerne maintenant le *sens* du mot, Wikipedia [12] nous donne la définition suivante :

Un ordinateur est un système de traitement de l'information *programmable* [...] et qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en *programmes*, qui lui font exécuter des *opérations logiques et arithmétiques*.

Le Petit Robert, pour sa part, nous donne la définition :

Machine électronique de *traitement numérique* de *l'information*, exécutant à grande vitesse les instructions d'un *programme enregistré*.

Ces deux définitions mettent en avant plusieurs concepts essentiels :

1. L'ordinateur est une machine qui *traite l'information*. Il reçoit de l'information et en produit, sur base de l'information reçue. Par exemple, un ordinateur peut être utilisé pour calculer les racines réelles d'une équation du second degré à une inconnue sur base des coefficients  $a$ ,  $b$  et  $c$  de cette équation (c'est l'exemple que nous développons ci-dessous); ou pour élaborer la fiche de paye d'un travailleur sur base de ses prestations; ou pour afficher une vidéo sur base de données reçues *via Internet*,...
2. Pour traiter l'information, l'ordinateur *exécute un programme*, qui est une séquence *d'instructions*, indiquant chacune à traitement spécifique à exécuter. Cela signifie que

---

1. Philologue français né le 6 décembre 1906 et mort le 29 mars 1992, il est professeur à la Faculté de lettres de Paris (France).

1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur?

UNIVERSITÉ DE PARIS  
—  
FACULTÉ  
DES  
LETTRES

Paris, le 16 IV 55

Cher monsieur,

Que diriez-vous d'ordinateur? C'est un mot couramment formé, qui se trouve bien dans le *Littérature* comme si j'y désignais Dieu qui mit de l'ordre dans le monde. Un mot de ce genre a l'avantage de donner directement un verbe ordiner, un nom d'action ordination. L'inconvénient est que ordination désigne une cérémonie religieuse ; mais les deux champs de signification (religion et computabilité) sont si étendus et la cérémonie d'ordination comme, je crois, du si peu de personnes que l'inconvénient est presque mineur. D'ailleurs votre machine serait ordinateur (ut non ordination) et ce mot est tout à fait parti de l'usage théologique.

L. 1698. — Paris, J.A.C.

FIGURE 1.1. – *Que diriez-vous d'ordinateur?* La proposition de Jacques PERRET pour nommer cette nouvelle machine...

l'ordinateur a été conçu de manière à être capable de réaliser certains traitements correspondants à des instructions pré-définies, mais qu'on peut utiliser ces instructions de base comme des briques pour construire des traitements plus complexes appelés *programmes*.

Par exemple, supposons que nous disposons d'un ordinateur capable d'exécuter les instructions `add` et `idiv` qui calculent respectivement la somme de deux nombres et la division (entière) d'un nombre par un autre. Nous pouvons imaginer qu'il est possible de construire, sur base de ces deux instructions, un programme qui calcule la (partie entière de la) moyenne de deux nombres, en utilisant d'abord `add` pour faire la somme de ces deux nombres, puis `idiv` pour diviser le résultat par 2. Naturellement, la somme doit être effectuée avant la division : l'ordre (la séquence) des instructions est donc important. Ce faisant, nous avons esquissé l'écriture d'un programme, qui calculerait une moyenne en utilisant `add` et `idiv`.

3. Le *programme de l'ordinateur peut être modifié*. L'ordinateur n'a pas été conçu dans le but d'exécuter un et un seul programme, mais son utilisateur a la faculté de modifier le programme, soit en acquérant des programmes écrits par des tiers (comme des applications achetées et téléchargées sur Internet, par exemple), soit en écrivant lui-même de nouveaux programmes.
4. Enfin, *l'information* que l'ordinateur traite à l'aide des programmes qu'il exécute est *représentée de manière numérique*, et est traitée comme telle. Cela signifie que toute information, quelque soit sa nature (nombres, texte, images, sons, vidéos,...) doit être exprimée sous forme de nombres pour pouvoir être traitée par l'ordinateur. En l'occurrence, tous les ordinateurs modernes représentent l'information sous forme de nombres *binaires*, c'est-à-dire formés de séquences de 0 et de 1 uniquement, comme nous le verrons en détails dans le Chapitre 2. Cela signifie également que les *traitements* exécutés (et exprimés à l'aide des instructions connues de l'ordinateur) réalisent eux aussi des opérations sur des nombres (ce sont des *opérations logiques et arithmétiques*).

Ces éléments constituent les caractéristiques généralement retenues pour définir un *ordinateur* : (1) la possibilité d'exécuter un programme arbitraire, qui peut être modifié, et est donc stocké dans la mémoire de l'ordinateur au même titre que les données qui sont traitées<sup>2</sup>; et (2) la représentation et le traitement numérique de l'information (y compris du programme à exécuter).

## 1.2. Un premier modèle : l'architecture de von Neumann

Maintenant que nous savons *ce que fait* un ordinateur, nous devons répondre à la question du *comment*? Dans une première approche, nous allons partir d'un exemple de programme, emprunté au cours de Programmation (INFO-F-101). Il s'agit d'un exemple écrit en Python,

---

2. Il doit également être possible d'écrire des programmes complexes, comprenant des boucles et des sauts. La notion précise est la notion de machine Turing-complète, telle que définie par le mathématicien anglais Alan TURING [23]. Nous ne développerons pas cette notion en détail ici, le lecteur intéressé peut consulter une introduction à la calculabilité comme l'excellent ouvrage de Pierre WOLPER [25].

## 1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur?

```
1 """
2     Calcul des racines éventuelles d'une équation du second degré
3 """
4 __author__="Thierry Massart"
5 __date__="22 août 2012"
6
7 from math import sqrt
8 a = float(input("valeur de a : "))
9 b = float(input("valeur de b : "))
10 c = float(input("valeur de c : "))
11
12 delta = b**2 - 4*a*c
13 if delta < 0:
14     print(" pas de racines réelles")
15 elif delta == 0:
16     print("une racine : x = ", -b/(2*a))
17 else:
18     racine = sqrt(delta)
19     print("deux racines : x1 = ", \
20           (-b + racine)/(2*a), " x2 = ", (-b - racine) / (2*a))
```

FIGURE 1.2. – Un exemple de programme écrit en langage Python.

et qui calcule les éventuelles racines réelles d'une équation du second degré à une inconnue. Il est donné à la FIGURE 1.2.

Cet exemple est fort utile car il illustre bien les différentes ressources dont l'ordinateur a besoin pour exécuter un programme. En voici la liste :

- L'ordinateur doit disposer d'un moyen d'*obtenir des données en entrée* (c'est-à-dire les données à traiter). Dans notre exemple, le mot-clé `input` interroge l'utilisateur qui entre les données *via* le clavier. Ces données sont stockées dans des variables `a`, `b` et `c`.
- L'ordinateur doit être capable de stocker des données de manière temporaire, c'est-à-dire durant la durée d'exécution d'un programme. Ces données temporaires correspondent en général aux données d'entrée, ou aux résultats de calculs intermédiaires. Dans notre exemple, les variables `a`, `b`, `c` et `delta` représentent ces stockages temporaires. Notez bien que `delta` n'est pas une donnée d'entrée du programme, mais bien une valeur intermédiaire du calcul.
- L'ordinateur doit pouvoir communiquer ses résultats au monde extérieur. Dans notre exemple, le mot-clé `print` permet d'afficher des messages ainsi que certaines variables
- L'ordinateur doit avoir la capacité d'effectuer certaines opérations de base, sans que l'utilisateur n'ait besoin d'écrire un programme pour expliciter à l'ordinateur le sens de ces opérations de base. C'est le cas, par exemple des opérations arithmétiques : elles

## 1.2. Un premier modèle : l'architecture de VON NEUMANN

sont considérées comme « connues » et peuvent être utilisées dans un programme sans qu'il soit nécessaire d'expliciter le calcul du +, du -, etc. Dans notre exemple, ces opérations interviennent dans le calcul du  $\Delta$ .

L'ensemble des opérations de base connues de l'ordinateur, et qu'on peut utiliser pour écrire un programme, composent ce qu'on appelle un *langage*. Le langage principal d'un ordinateur est ce qu'on appelle le *langage machine*. C'est ce langage qui opère directement sur la représentation binaire des informations. Notre exemple, lui, est écrit dans une langage différent : le langage Python. Dans ce langage (et donc dans notre exemple), on ne voit pas la représentation binaire des informations apparaître explicitement<sup>3</sup>. Au contraire, Python permet d'exprimer les opérations sur des données plus *abstraites*, et donc plus facilement compréhensible par un être humain, comme le texte qui est affiché pour présenter le résultat. Le langage Python offre donc une facilité accrue par rapport au langage machine, mais afin qu'il puisse être exécuté par l'ordinateur, il devra impérativement être traduit en langage machine.

- Bien que cela apparaisse de manière moins évidente sur notre exemple, l'ordinateur doit également être capable de *stocker le programme à exécuter* (puisque celui-ci peut être modifié) pour pouvoir s'y référer, et ainsi avoir la capacité d'accéder à la bonne instruction à exécuter. Pour ce faire, il faut aussi disposer d'un moyen retenant l'avancement de l'exécution dans le programme, et de modifier cet état d'avancement. Cette modification peut consister soit à passer à l'instruction suivante dans le programme, soit à désigner une instruction arbitraire comme étant celle à exécuter en prochain lieu, éventuellement selon certaines conditions (c'est ce que fait l'instruction `if` dans notre exemple).

Sur base de ces besoins, on peut proposer un premier modèle *abstrait* de ce qui est nécessaire pour concevoir une telle machine. Notre modèle est présenté à la FIGURE 1.3. Il s'agit d'une conception de l'ordinateur qui a été proposée par John VON NEUMANN<sup>4</sup> en 1945 [11], et sur lequel tous les ordinateurs modernes se basent. Il est bien entendu que ce modèle est une simplification (sans doute un peu grossière) de la réalité, mais elle est suffisante, en première approximation, pour expliquer les principes généraux de fonctionnement de l'ordinateur. Nous raffinerons cette figure dans la suite. Notons que ce modèle est souvent appelé *architecture de von Neumann* ou *architecture à programme enregistré (stored program architecture* en anglais). On utilise ici le terme architecture, car le modèle explique quels sont les différents composants de l'ordinateur et comment ils sont utilisés, interconnectés, pour bâtir la machine qu'est l'ordinateur.

En analysant cette figure, nous pouvons constater :

- que l'**ORDINATEUR** (cadre gras) communique avec le monde extérieur à travers des

3. Par exemple, la valeur 2 utilisé dans la dernière ligne du programme n'est pas exprimée en binaire, sans quoi on aurait écrit 10 (voir Chapitre 2)

4. Mathématicien hongrois, né à Budapest le 28 décembre 1903, mort à Washington, le 8 février 1957.



John VON NEUMANN.

## 1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur ?

- dispositifs appelés *périphériques* d'entrée/sortie (par exemple : le clavier, la souris, l'écran, etc) ;
- que l'ordinateur est composé essentiellement de deux composants : le processeur (ou en anglais, *Central Processing Unit*, CPU) et la mémoire. Ce sont les deux cadres aux bords arrondis sur la figure ; et enfin
- que le processeur est lui-même composé d'une unité de contrôle, d'une unité arithméticologique (en anglais *Arithmetic and Logic Unit*, ALU) et de registres.

Détaillons maintenant les rôles de ces différents composants, pour l'exécution d'un programme. Tout d'abord, le programme et ses données sont stockées dans la *mémoire*, qui, comme son nom l'indique, est un composant servant à mémoriser (stocker) l'information. Comme cette mémoire est modifiable, on peut également modifier le programme ou l'appliquer à d'autres données ; ce qui est essentiel dans notre définition d'ordinateur (cela explique le nom d'*architecture à programme enregistré*, car le programme est enregistré dans la mémoire).

Ensuite, le *processeur* a pour tâche d'exécuter le programme, instruction par instruction. Pour ce faire, il interroge la mémoire qui lui transmet la prochaine instruction à exécuter ; puis, il analyse et exécute celle-ci, et recommence *ad infinitum*. Le processeur a donc un comportement *cyclique* que l'on représente par la *boucle d'interprétation du processeur* parfois appelée également *fetch–decode–execute*, du nom des trois étapes principales (en anglais) :

1. *fetch* : aller chercher la prochaine instruction à exécuter, en mémoire. Cela signifie, comme nous l'avons déjà dit, que le processeur doit avoir un moyen, d'identifier cette instruction parmi d'autres en mémoire. C'est un des rôles (mais pas le seul) des registres.
2. *decode* : il s'agit ici d'analyser la représentation binaire de l'instruction (en *langage machine*) pour en déduire l'opération à effectuer.
3. *execute* : exécuter l'instruction à proprement parler. Les éléments du CPU en charge de l'exécution d'une instruction sont les registres et l'ALU. D'une part, les registres sont des zones de mémoire de très petite capacité mais d'accès très rapide, dans lesquels les données nécessaires à l'exécution de l'instruction, ainsi que son résultat, seront stockés. D'autre part, l'ALU est un circuit électronique du processeur qui peut appliquer une opération arithmétique ou logique à deux données provenant des registres. Par exemple, si l'on souhaite faire la somme de deux nombres, il faudra placer les deux valeurs dans des registres, transmettre ces valeurs à l'ALU qui effectuera la somme, puis placer le résultat dans un registre. Nous décrirons ce procédé plus en détail à la Section 3.2, et surtout au Chapitre 5.

L'effet d'une instruction peut également être d'écrire ou de lire une valeur en mémoire (depuis ou vers les registres), ou encore de communiquer avec les périphériques.

La coordination de ces trois étapes est assurée par l'unité de contrôle du CPU. Ainsi, l'unité de contrôle doit s'assurer que ce sont les bons registres qui transmettent leur données à l'ALU, que l'ALU applique la bonne opération aux données (il faut choisir l'opération à effectuer

## 1.2. Un premier modèle : l'architecture de VON NEUMANN

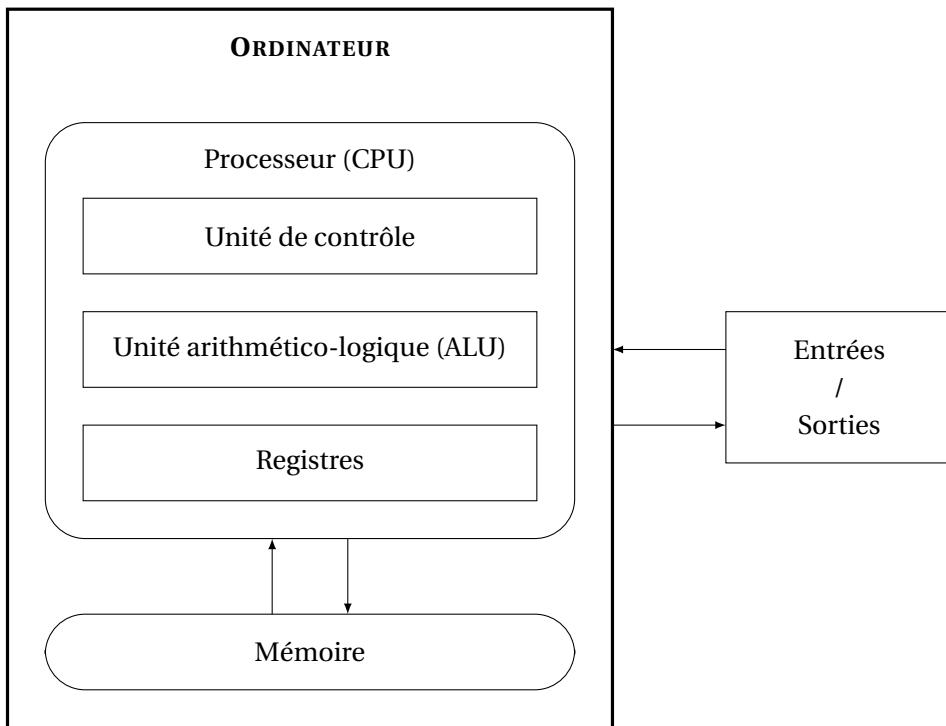


FIGURE 1.3. – L'architecture VON NEUMANN, un premier modèle d'ordinateur.

parmi une palette d'opérations disponibles), que le résultat calculé par l'ALU est placé dans le bon registre de sortie, *etc.*

Dans le Chapitre 3, nous expliquerons plus en détail le fonctionnement de cette boucle. Dans le Chapitre 7, nous raffinerons cette boucle pour introduire un mécanisme d'*interruption*, qui est essentiel au bon fonctionnement des ordinateurs modernes.



Dans ces notes de cours, nous utiliserons souvent la famille de processeurs x86 de la firme Intel comme exemples pour illustrer les notions relatives aux processeurs. On retrouve aujourd'hui ces processeurs dans l'immense majorité des ordinateurs personnels.

En particulier, nous considérerons le processeur i486. Il s'agit d'un processeur qui a été commercialisé de 1989 à 2007, et il a des performances relativement modestes relativement aux processeurs récents. Néanmoins, il possède la plupart des caractéristiques des processeurs modernes, tout en restant relativement simple. Il constitue donc un excellent exemple pédagogique.

## 1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur ?



L'architecture de l'i486 est montrée à la FIGURE 1.4. Comme on peut le voir, elle est bien plus complexe que notre simple modèle de la FIGURE 1.3! Mais nous pouvons déjà reconnaître certains éléments :

- En jaune, sur la gauche de la figure, nous retrouvons l'ALU (avec le *shifter* qui peut également être vu comme un circuit réalisant des opérations arithmétiques), et les registres servant à contenir des données (l'ensemble de ces registres est appelé *register file*). L'i486 possède au total 32 registres qui ont des fonctions différentes. En particulier, les 4 registres appelés `eax`, `ebx`, `ecx` et `edx` servent à stocker les données sur lesquelles opèrent les instructions. Ce sont des registres de 32 bits. L'i486 possède également 8 registres spécialisés de 80 bits, qui permettent d'effectuer des opérations arithmétiques sur des nombres décimaux (dont nous parlerons dans la Section 2.2.5). Ces opérations un peu spéciales ne sont pas réalisées par l'ALU, mais par le FPU (pour *floating point unit*), en orange, sur la gauche de la figure.
- Les flèches en noires à droite de l'image représentent les communications du processeur avec le monde extérieur, c'est-à-dire avec la mémoire (deux flèches du haut) et les périphériques.
- On reconnaît également un bloc chargé du décodage des instructions (dans le bas de la figure) et un bloc chargé de la lecture en mémoire (*fetch*), appelé ici *prefetcher*.

### 1.3. Une vision différente : structure en niveaux et traductions

Maintenant que nous avons identifié les différents composants d'un ordinateur et leurs fonctions, nous pouvons approfondir la question du *comment*? posée au début de la section précédente. Nous consacrerons évidemment tout le cours à expliquer *comment* les différents composants que nous avons identifiés remplissent leur fonction, mais nous pouvons déjà donner une vue globale qui correspondra au plan de l'exposé.

**Différents niveaux** Partons du CPU. Celui-ci est composé de circuits électroniques que nous appellerons *circuits logiques*, et que nous étudierons dans quelques leçons. C'est à l'aide de ces circuits logiques que le CPU doit donc interpréter le langage machine. Comme nous le verrons plus tard, l'interprétation de chaque instruction en langage machine n'est pas immédiate : le langage machine est d'abord traduit dans une langue plus simple encore, le *microlangage*, lequel est enfin exécuté à l'aide des circuits logiques. Tout comme un programme (en langage machine) se compose d'une série d'instructions (appelées *instructions machine*), chaque instruction machine est traduite<sup>5</sup> en une série de micro-instructions (les

5. À l'aide d'une boucle similaire à la boucle *fetch-decode-execute*.

### 1.3. Une vision différente : structure en niveaux et traductions

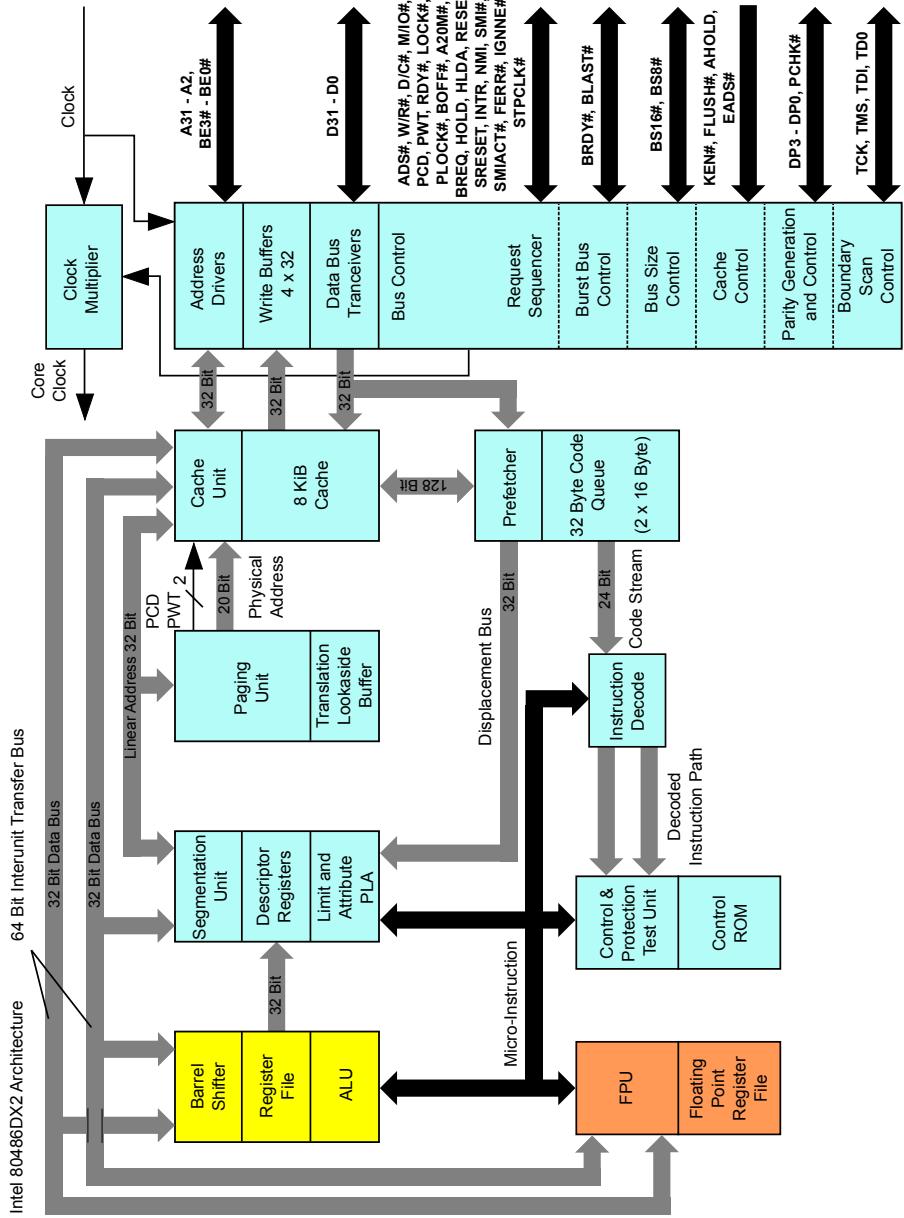


FIGURE 1.4. – L'architecture de l'i486 (sous sa variante 80486DX2), telle que publiée par Intel [9].

Source : Appaloosa ([https://commons.wikimedia.org/wiki/File:80486DX2\\_arch.svg](https://commons.wikimedia.org/wiki/File:80486DX2_arch.svg)), « 80486DX2 arch », 80486DX2 arch, <https://creativecommons.org/licenses/by-sa/3.0/legalcode>

## 1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur ?

instructions du micro-langage). Nous avons donc identifié les trois « couches » les plus basses de l'ordinateur, à savoir (et en partant de la couche la plus basse) :

0. les circuits logiques (qui constituent le véritable matériel constitutif de l'ordinateur) ;
1. le micro-langage ; et
2. le langage machine, qui est le langage du CPU.

Durant ce cours, nous nous concentrerons essentiellement sur ces trois couches.

Mais tout qui a un jour utilisé un ordinateur sait pertinemment bien que celui-ci peut être utilisé sans interagir directement avec le processeur, à l'aide du langage machine. L'exemple de la FIGURE 1.2 est écrit, comme nous l'avons déjà dit, en Python, un langage plus abstrait (on dit aussi : « de plus haut niveau ») que le langage machine, ce qui est une facilité non-négligeable. En effet, un même programme en langage Python peut être exécuté par des ordinateurs différents qui ont potentiellement des caractéristiques (nombre de registres, langage machine) différentes. Cela n'a pas d'importance pour le programmeur qui a écrit son programme en Python, car ce langage ne demande pas et ne permet pas d'interagir directement avec les composants matériels de base (ALU, registres) de la machine. Au contraire, si on souhaite exécuter un programme en Python sur un ordinateur donné, il faut disposer d'un *interpréteur*, qui est lui-même un programme qui *traduit* le programme en langage Python vers une séquence d'instruction machine propre à l'ordinateur sur lequel le programme Python doit être exécuté.

Par ailleurs, tout ordinateur personnel, tout *smartphone* est aujourd'hui livré avec un programme de base, permettant d'interagir facilement avec lui, et appelé « système d'exploitation », ou *operating system*, ou encore OS (par exemple, Windows, Linux, MacOS, Android, iOS, etc). Tous ces programmes sont *in fine* exécutés par le processeur et doivent donc être *traduits* en langage machine. Au final, nous avons 3 couche supplémentaires qui s'empilent au-dessus du langage machine :

3. le système d'exploitation, dont nous parlerons brièvement à la fin du cours, mais qui fera surtout l'objet d'un cours de deuxième année;
4. l'assembleur, qui est un langage intermédiaire entre le langage machine et les programmes écrits dans un langage de haut niveau. L'assembleur sera étudié dans le cours de langages de programmation, au second quadrimestre;
5. les langages de haut niveau, qui sont ceux à l'aide desquels on écrit les applications que l'on utilise quotidiennement (traitement de texte, navigateur web, etc). Le cours de programmation vous enseigne un de ces langages (python) et le cours de langages de programmation (second quadrimestre) vous en enseignera d'autres.

De bout en bout, l'exécution d'un programme de haut niveau peut donc se résumer comme suit : le programme de haut niveau (niveau 5) est traduit en assembleur (niveau 4). L'assembleur est traduit en un langage qui est essentiellement du langage machine augmenté d'appels aux services prodigués par le système d'exploitation (niveau 3). Ces « appels systèmes » sont eux-mêmes traduits en langage machine, et on obtient alors, au niveau 2, un programme entièrement en langage machine. L'exécution de ce programme machine par le CPU consiste en une traduction vers le micro-langage interne au CPU (niveau 1), dont le résultat est finalement exécuté à l'aide de circuits logiques (niveau 0).

**Interprétation et Compilation** On voit donc qu'on a affaire à une chaîne de traductions successives. Celles-ci peuvent être effectuées de deux façons différentes :

- soit un programme d'un certain niveau est intégralement traduit en un programme du niveau inférieur, avant le début de l'exécution. On parle alors de *compilation*. Le résultat de la traduction peut être stocké et réutilisé, mais ne sera plus modifié durant l'exécution. La compilation est réalisée à l'aide d'un programme appelé *compilateur*.

C'est le cas des applications que nous téléchargeons sur Internet (que ce soit pour un ordinateur de bureau ou pour un *smartphone*) : elles ont été écrites dans un langage de haut niveau par leurs concepteurs, puis *compilées* en langage machine. C'est le résultat de cette compilation qui est distribué et téléchargé par les utilisateurs finaux;

- soit un programme d'un certain niveau est traduit vers le niveau inférieur, instruction par instruction, au moment de l'exécution. On parle alors d'*interprétation*. C'est ce qui se passe, par exemple, quand on traduit le langage machine en micro-code, ou quand le processeur exécute le langage machine.

De manière générale, un *interpréteur* est un programme écrit dans un langage de niveau  $i$  et qui interprète un programme écrit dans un langage de niveau  $j > i$ , instruction par instruction, c'est-à-dire en exécutant continuellement une boucle qui consiste à : (1) lire la prochaine instruction à exécuter; (2) analyser et exécuter cette instruction; (3) passer à l'instruction suivante. Le CPU peut donc être vu comme la réalisation d'un interpréteur pour le langage machine, réalisé à l'aide de micro-code.

La hiérarchie de niveaux que nous venons de décrire, ainsi que les différents types de traductions entre ces niveaux, sont présentés à la FIGURE 1.5.

**Matériel et logiciel** Notons finalement que chacun des 6 niveaux peut théoriquement être réalisé soit à l'aide *matériel* soit à l'aide de *logiciel*. Élucidons ces deux termes :

- le *matériel* (*hardware* en anglais) est l'ensemble des dispositifs physiques qui constituent l'ordinateur; tandis que
- un *logiciel* (*software* en anglais) est une représentation informatique des informations nécessaires à un traitement réalisé par l'ordinateur, à savoir, la séquence d'instructions à exécuter, et les données.

En pratique, les niveaux 0,1 et 2 sont réalisés à l'aide de matériel, et les niveaux 3, 4 et 5 à l'aide de logiciel. Le logiciel et le matériel ont chacun leurs avantages et leurs inconvénients. Le logiciel est plus flexible, dans le sens où on peut le modifier et le remplacer, mais plus lent. Le matériel est plus rapide, mais n'est pas modifiable, en général. Il est par exemple possible de changer d'interpréteur Python (niveau 5) si jamais une nouvelle version de ce langage venait à être proposée. Par contre, le processeur, qui exécute le langage machine, doit être très rapide, et on ne peut pas modifier la traduction du langage machine en micro-langage (niveau 2 vers niveau 1).

## 1. Leçon 1 – Introduction : Qu'est-ce qu'un ordinateur ?

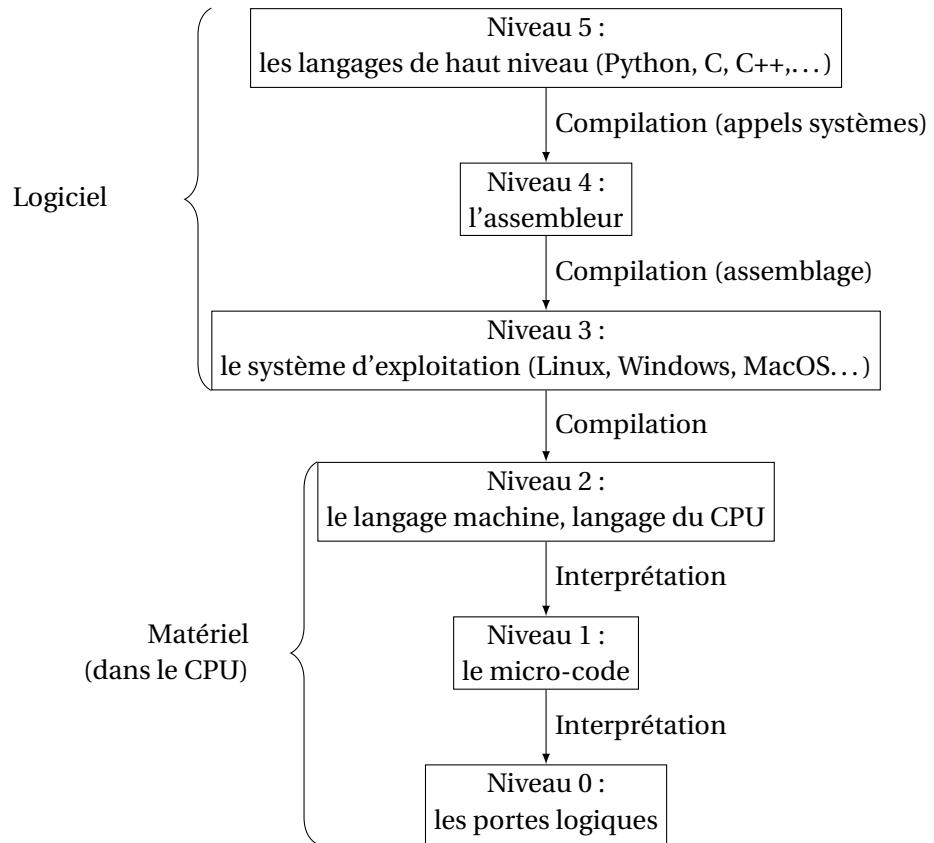


FIGURE 1.5. – Les 6 couches décrivant le fonctionnement d'un ordinateur.



## 2. Leçons 2 à 4 – Représentation de l'information

Avant de pouvoir étudier la manière dont l'ordinateur  *traite* l'information, nous devons fixer la manière dont cette information est représentée. L'objet de ces leçons est d'étudier la représentation *binaire*, qui est aujourd'hui<sup>1</sup> la représentation universellement utilisée en informatique pour toutes les informations que les ordinateurs manipulent.



La représentation *binaire* signifie que toute l'information sera représentée à l'aide de **deux symboles uniquement**. Ces deux symboles sont en général les chiffres 0 et 1, mais on peut également considérer que ce sont les valeurs logiques *faux* et *vrai*.

Dans ce chapitre, nous développerons des techniques permettant de représenter tout type d'information (nombres, texte, images,...) à l'aide d'un représentation binaire, et de manipuler cette représentation. Nous verrons également comment concevoir des techniques de détection et de correction d'erreur sur cette représentation. Mais avant d'aborder cette matière, il est important de bien comprendre la différence entre une information, et sa représentation. L'exemple suivant illustre cette différence.



Considérons la quantité 17, nous pouvons en trouver plusieurs représentations :

- la représentation usuelle, dite « en base  $10^2$  » : 17,
- en base 2 : 10001,
- en chiffres romains : XVII,
- en utilisant des marques de dénombrement :
- ...

On voit donc qu'une même information, qu'un même concept admet plusieurs représentations. Le choix de la représentation binaire comme représentation universelle en informatique s'explique par le fait qu'on peut aisément représenter deux valeurs différentes à l'aide d'états différents de composants électriques comme les transistors. Ces même transistors, assemblés en *portes logiques*, permettent également de manipuler l'information binaire,

1. Il y a eu quelques expériences utilisant des représentations *ternaires*. On peut citer les ordinateurs soviétiques Setun (1959–1965) [13].

2. Les notions de base et de changement de base seront détaillées dans la suite.

## 2. Leçons 2 à 4 – Représentation de l'information

comme nous le verrons dans le Chapitre 3.

### 2.1. Unités de quantité d'information

Commençons par fixer des unités permettant de mesurer la *quantité* d'information. Nous utiliserons la taille de la représentation binaire de l'information comme mesure :



Chaque chiffre (0 ou 1) d'une représentation binaire est appelé **bit** (contraction de l'anglais *binary digit*, ou chiffre binaire). Une séquence de 8 bits est appelée un **octet** (ou **byte** en anglais).

Les symboles associés à ces unités sont les suivants :

Unité	Symbole
bit	B
octet	o
byte	b

Nous mesurerons donc l'information en terme de bits ou d'octets (bytes). Naturellement, les quantités d'information manipulées usuellement s'expriment en milliers, millions,... d'octets, il faut donc fixer des noms pour ces unités. Les préfixes officiellement reconnus par l'industrie [15] sont similaires aux kilo-, méga-,... du système SI. Ces préfixes ne se réfèrent pas à des puissances de 10 comme dans le système SI mais à des puissances de 2 :

Nom	Abréviation	Quantité
kibioctet	kio	$2^{10} = 1024 \approx 10^3$
mébioctet	Mio	$2^{20} = 1\ 048\ 576 \approx 10^6$
gibioctet	Gio	$2^{30} = 1\ 073\ 741\ 824 \approx 10^9$
tebioctet	Tio	$2^{40} = 1099\ 511\ 627\ 776 \approx 10^{12}$

La définition de ces préfixes est relativement récente (elle date du début des années 2000). En pratique, on est souvent confronté à l'utilisation des préfixes traditionnels kilo-, méga-, tera-, etc du système SI, qui sont, pour rappel :

Nom	Abréviation	Quantité
kiloctet	ko	$10^3$ octets
mégoctet	Mo	$10^6$ octets
gigaoctet	Go	$10^9$ octets
teraoctet	To	$10^{12}$ octets

On voit donc qu'un mégoctet est un petit peu plus petit qu'un mébioctet... Force est de reconnaître qu'une certaine confusion règne! Cette confusion a d'ailleurs donné lieu à des procès retentissants, notamment aux États-Unis, où des consommateurs ont entamé une *class*

## 2.2. Écriture des nombres dans différentes bases

*action* contre plusieurs fabricants de disques durs et mémoires, les accusant de tromperie sur la capacité de stockage des produits<sup>3</sup>.

### 2.2. Écriture des nombres dans différentes bases

Nous allons maintenant rappeler les techniques permettant de représenter des nombres dans différentes *bases*, car ces techniques serviront à expliquer les différents encodages binaires des nombres entiers et rationnels.

La représentation usuelle des nombres est une représentation *décimale* et *positionnelle*. Cela signifie :

1. que les nombres sont représentés par une séquence de *chiffres* de 0 à 9 inclus. Il y a bien *dix* chiffres différents utilisés, d'où le terme *décimal*; et
2. que la *position* de chaque chiffre dans la séquence permet d'associer ce chiffre à une *puissance de 10*.

Dans cette représentation usuelle, 10 est appelé la *base*. On parle aussi de représentation en base 10.



Par exemple, la représentation :

1234,56

signifie que le nombre est composé de :

- 4 unités, ou  $4 \times 10^0$ ;
- 3 dizaines, ou  $3 \times 10^1$ ;
- 2 centaines, ou  $2 \times 10^2$ ;
- 1 millier, ou  $1 \times 10^3$ ;
- 5 dixièmes, ou  $5 \times 10^{-1}$ ; et
- 6 centièmes, ou  $6 \times 10^{-2}$ .

On voit donc que les puissances de 10 associées aux différentes positions (aux différents chiffres) vont décroissantes quand on lit le nombre de gauche à droite, et que la puissance  $10^0$  correspond au chiffre qui se trouve juste à gauche de la virgule.

On note également que la base donne le nombre de chiffres que l'on peut utiliser pour représenter les nombres. En base 10, on peut utiliser 10 chiffres différents, à savoir les chiffres de 0 à 9 inclus.

Ces concepts se généralisent dans n'importe quelle base. Fixons à partir de maintenant une base  $b \geq 2$ . Dans cette base, le nombre :

$$d_n \cdots d_0, d_{-1} \cdots d_{-k}$$

---

3. Voir par exemple : [https://en.wikipedia.org/wiki/Binary\\_prefix#Inconsistent\\_use\\_of\\_units](https://en.wikipedia.org/wiki/Binary_prefix#Inconsistent_use_of_units).

## 2. Leçons 2 à 4 – Représentation de l'information

représente la valeur :

$$N = d_n \times b^n + d_{n-1} \times b^{n-1} + \cdots + d_0 \times b^0 + d_{-1} \times b^{-1} + \cdots + d_{-k} \times b^{-k} \quad (2.1)$$

$$= \sum_{i=-k}^n d_i \times b^i, \quad (2.2)$$

où tous les  $d_i$  sont des chiffres dans  $\{0, \dots, b-1\}$ .



Si on fixe  $b = 2$ , le nombre 1001,101 représente :

$$\begin{aligned} & 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 8 + 0 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8} \\ &= 9,625. \end{aligned}$$

La dernière valeur est bien entendu donnée en base 10.

Symétriquement, la valeur  $-4,5$  (en base 10) est représentée par :

$$-100,1$$

en base 2. Notons au passage que cette dernière représentation n'est pas *stricto sensu* une représentation *binaire*, étant donné que les symboles , (virgule) et – (moins unaire) sont utilisés, en plus du 0 et du 1. Nous verrons plus tard comment une telle valeur peut être représentée uniquement en utilisant des 0 et des 1.

Afin d'éviter toute confusion dans la suite, nous noterons, quand c'est nécessaire, la base en indice des nombres. Par exemple,  $101_2$  signifie que le nombre 101 doit être interprété comme étant en base 2 (c'est-à-dire  $5_{10}$ ). Notons que dans certains langages de programmation, comme le C [1], on utilise d'autres conventions pour expliciter certaines bases : les nombres en base 16 sont préfixés par `0x`, et les nombres en base 2 par `0b`. Enfin, notons qu'en base 2, nous grouperons souvent les bits par paquets de 4, en introduisant un petit espace, et ce, uniquement dans un souci de lisibilité. Nous écrivons ainsi 1011 1001 au lieu de 10111001.

En informatique, les bases usuelles sont la base 2, la base 8 (on parle d'octal) et la base 16 (on parle d'hexadécimal). Pour la base 16, on est confronté au problème suivant : les chiffres utilisés devraient être 0, 1, ..., 10, 11, 12, 13, 14 et 15. Mais les « chiffres » de 10 à 15 demandent deux symboles pour être représentés, ce qui risque de porter à confusion. Par exemple, doit-on interpréter  $10_{16}$  comme le chiffre 10 (et alors  $10_{16} = 10_{10}$ ) ou comme les chiffres 1 et 0 (et donc  $10_{16} = 1 \times 16 + 0 \times 1 = 16_{10}$ ) ? Pour éviter ce problème, on remplace les « chiffres » de 10 à 15 par les lettres de a à f, selon la correspondance suivante :

Lettre :	a	b	c	d	e	f
Valeur :	10	11	12	13	14	15

## 2.2. Écriture des nombres dans différentes bases



En base 16 :

$$\begin{aligned} a5f.b &= 10 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 + 11 \times 16^{-1} \\ &= 2655,6875_{10}. \end{aligned}$$

Sur base de ces définitions, nous pouvons déjà faire quelques remarques utiles.

**Chiffres de poids fort, de poids faible** Dans la représentation positionnelle usuelle, le chiffre le plus à gauche est associé à une puissance plus élevée de la base. On parle donc de chiffre de « poids fort » ; le chiffre étant associé à la puissance la plus faible est appelé chiffre de « poids faible ». En particulier, en binaire, on parle de *bit de poids fort* et *bit de poids faible*.

Comme nous l'avons dit, le bit de poids fort est, par convention à gauche dans la représentation (positionnelle) usuelle. Mais quand on s'intéresse à des valeurs qui ne sont plus écrites sur papier, mais bien sont stockées ou manipulées par des circuits électroniques (où les notions de « gauche » et « droite » n'ont pas forcément de sens), il est parfois utile de préciser explicitement quel est le bit de poids fort ou le bit de poids faible.

**Ajouts de zéros** Dans toutes les bases, et à condition d'utiliser la représentation positionnelle usuelle (avec le chiffre de poids fort à gauche), on peut toujours ajouter des 0 à gauche de la partie entière ou à droite de la partie décimale, sans modifier la valeur du nombre représentée. Par exemple,  $13,5_{10} = 0013,5000_{10}$ . Par contre, on ne peut pas ajouter de zéros à d'autres endroits sans modifier la valeur du nombre. Par exemple,  $101_2 \neq 101000_2$  (voir aussi la remarque suivante).

**Multiplier ou diviser par la base** Dans toutes les bases, on peut aisément multiplier ou diviser par la base en déplaçant la position de la virgule :



Dans toute base  $b$ , déplacer la virgule de  $k$  positions vers la **droite** (ou ajouter  $k$  zéros à droite dans le cas d'un nombre entier) revient à **multiplier** par  $b^k$ . Déplacer la virgule de  $k$  positions vers la **gauche** revient à **diviser** par  $b^k$ . Si on souhaite faire une division entière par  $b^k$ , le **reste** est constitué des  $k$  chiffres de poids faible, et on obtient le **quotient** en retranchant ces  $k$  chiffres de poids faible.

Nous utiliserons les opérateurs  $\div$  et  $\text{mod}$  pour noter respectivement la division entière et le reste de la division. Voici quatre exemples qui illustrent ces remarques :

## 2. Leçons 2 à 4 – Représentation de l'information



$$\begin{aligned}
 101,111_2 \times 4_{10} &= 10111,1_2 \\
 101,111_2 / 2_{10} &= 10,1111_2 \\
 1011 1011_2 \div 100_2 &= 1011 1011_2 \div 2^2 \\
 &= 10 1110 \\
 1011 1011_2 \bmod 100_2 &= 11_2.
 \end{aligned}$$

**Combien de nombres représentables sur  $n$  chiffres ?** Enfin, l'observation suivante aura toute son importance dans la suite. Fixons une base  $b$  et considérons des nombres entiers positifs sur  $n$  chiffres. Nous avons  $b$  choix pour chacun de ces  $n$  chiffres, soit un total de  $b^n$  nombres différents que l'on peut représenter sur  $n$  chiffres. Ces nombres sont tous les nombres de :

$$\underbrace{0 \cdots 0}_{n \text{ chiffres}}$$

à :

$$\underbrace{(b-1) \cdots (b-1)}_{n \text{ chiffres}} = b^n - 1.$$



En base  $b$ , il y a  $b^n$  nombres (entiers positifs) représentables sur  $n$  chiffres : les nombres de 0 à  $b^n - 1$ .

Notons que quand nous souhaiterons représenter des nombres négatifs de manière purement binaire (donc, sans utiliser le symbole  $-$ ), nous aurons besoin d'utiliser certaines de ces représentations pour « encoder » des nombres négatifs.



Il y a donc  $2^n$  nombres binaires sur  $n$  bits. Par exemple, il y a  $2^{32} = 4\,294\,967\,296_{10}$  (soit à peu près 4 milliards) nombres binaires différents sur 32 bits.

### 2.2.1. Changements de base

Comment peut-on, étant donné un nombre  $N$  donné dans une certaine base, trouver sa représentation dans une autre base  $b$ ? Répondre à cette question revient à calculer les valeurs  $d_i$  de l'équation (2.2) dans la base  $b$ .

**Cas des nombres entiers** Nous allons commencer par supposer que  $N$  est un nombre entier. Observons d'abord que si  $N < b$ , on exprime le nombre à l'aide du chiffre correspondant

## 2.2. Écriture des nombres dans différentes bases

dans la nouvelle base<sup>4</sup>. Autrement, division  $N$  par  $b$  (il s'agit de la division entière), ce qui nous donne un quotient  $q_0 = N \div b$  et un reste  $r_0 = N \bmod b$ . La relation existant entre ces valeurs est :

$$N = q_0 \times b + r_0. \quad (2.3)$$

Comme nous avons supposé que  $N \geq b$ , nous savons que  $q_0 \neq 0$ . Recommençons l'opération en divisant  $q_0$  par  $b$ , obtenons à nouveau un quotient que nous appelons  $q_1$  et un reste que nous appelons  $r_1$ . Nous avons donc la relation :

$$q_0 = q_1 \times b + r_1,$$

En combinant cette équation avec (2.3), nous obtenons :

$$\begin{aligned} N &= (q_1 \times b + r_1) \times b + r_0 \\ &= q_1 \times b^2 + r_1 \times b + r_0. \end{aligned}$$

Si  $q_1 > 0$ , on recommence en divisant  $q_1$  par  $b$ , pour obtenir un nouveau quotient  $q_2$  et un nouveau reste  $r_2$  tels que :

$$\begin{aligned} N &= (q_2 \times b + r_2) \times b^2 + r_1 \times b + r_0 \\ &= (q_2 \times b^3) + r_2 \times b^2 + r_1 \times b + r_0. \end{aligned}$$

Si nous continuons ce développement en réalisant  $k+1$  divisions, nous obtenons une suite de restes  $r_0, r_1, \dots, r_k$  tels que :

$$N = (q_k \times b^{k+1}) + r_k \times b^k + \dots + r_2 \times b^2 + r_1 \times b + r_0.$$

Supposons que nous avons continué ce développement jusqu'à ce que  $q_k = 0$  (ce qui finira toujours par arriver étant donné qu'on est parti d'un  $N$  fixé). On a alors :

$$\begin{aligned} N &= (q_k \times b^{k+1}) + r_k \times b^k + \dots + r_2 \times b^2 + r_1 \times b + r_0 \\ &= (q_k \times b^{k+1}) + r_k \times b^k + \dots + r_2 \times b^2 + r_1 \times b^1 + r_0 \times b^0 \\ &= (q_k \times b^{k+1}) + \sum_{i=0}^k r_i \times b^i \\ &= (0 \times b^{k+1}) + \sum_{i=0}^k r_i \times b^i \\ &= \sum_{i=0}^k r_i \times b^i. \end{aligned} \quad (2.4)$$

---

4. Par exemple,  $10_{10}$  est exprimé par  $a$  en base 16.

## 2. Leçons 2 à 4 – Représentation de l'information

On peut maintenant comparer cette dernière équation (2.4) à l'équation (2.2), et constater que les restes  $r_i$  que nous avons calculés en effectuant des divisions successives par  $b$  correspondent aux  $d_i$  que nous cherchons pour représenter le nombre  $N$  en base  $b$ . Attention tout de même que le premier reste calculé ( $r_0$ ) est le chiffre de poids faible! Pour résumer :



Pour exprimer un nombre  $N$  en base  $b$ , on divise  $N$  par  $b$  de manière répétée. La séquence des restes calculés donne la représentation de  $N$  en base  $b$  du chiffre de poids faible au chiffre de poids fort (autrement dit : « à l'envers »).

**Nombres réels** Nous pouvons maintenant généraliser ce principe aux nombres réels<sup>5</sup>. Commençons par observer qu'un nombre rationnel qui possède une expression finie dans une base ne possède pas forcément une expression finie dans toutes les bases. Prenons par exemple le nombre  $\frac{1}{3}$ . Nous savons qu'il n'est possible de l'exprimer de manière finie en base 10, mais bien de manière infinie périodique :

$$\frac{1}{3} = 0,3333_{10}\dots$$

Nous notons cette représentation infinie périodique en soulignant la partie qui se répète infiniment souvent :

$$\frac{1}{3} = 0,\underline{\underline{3}}.$$

Par contre, comme  $\frac{1}{3} = 3^{-1}$ , on peut exprimer  $\frac{1}{2}$  de manière finie en base 3 :

$$\frac{1}{3} = 0,1_3.$$

Voyons maintenant comment représenter un nombre  $N < 1$  dans une base  $b$  arbitraire. Notons que nous supposons que  $N < 1$  car on peut traiter séparément la partie entière et la partie fractionnelle d'un nombre. Pour exprimer  $N$  en base  $b$ , nous allons procéder par *multiplication successive* par  $b$ . Supposons que  $N$  est de la forme :

$$N = 0, \alpha_0$$

où  $\alpha_0$  représente la séquence de chiffres de la partie fractionnaire du nombre. En multipliant  $N$  par  $b$  on obtient un nouveau nombre de la forme :

$$N \times b = \beta_1, \alpha_1$$

---

5. En pratique, sur un ordinateur, nous ne pourrons représenter et manipuler de manière précise que les nombres qui possèdent une représentation finie en base 2, ce qui exclut d'office tous les nombres irrationnels, comme  $\pi$  ou  $\sqrt{2}$ , par exemple. Mais cela n'empêche que même les nombres irrationnels possèdent aussi une représentation (infinie non-périodique) dans d'autres bases que la base 10.

## 2.2. Écriture des nombres dans différentes bases

où  $\beta_1$  est la partie entière, et  $\alpha_1$  la partie décimale. Comme nous avons supposé que  $N < 1$ , nous savons que  $0 \leq \beta_1 \leq b - 1$ . Considérons le nouveau nombre  $0, \alpha_1$ , et appelons-le  $N_1$ . Nous avons donc :

$$\begin{aligned} N_1 &= 0, \alpha_1 \\ N &= \frac{\beta_1}{b} + \frac{N_1}{b}. \end{aligned} \tag{2.5}$$

Multipliions maintenant  $N_1$  par  $b$ , nous obtenons un nombre de la forme :

$$N \times b = \beta_2, \alpha_2$$

avec  $0 \leq \beta_2 \leq b - 1$ . Nous pouvons à nouveau poser  $N_2$  et obtenir :

$$\begin{aligned} N_2 &= 0, \alpha_2 \\ N_1 &= \frac{\beta_2}{b} + \frac{N_2}{b}. \end{aligned} \tag{2.6}$$

En combinant (2.6) et (2.5), nous obtenons :

$$\begin{aligned} N &= \frac{\beta_1}{b} + \frac{\frac{\beta_2}{b} + \frac{N_2}{b}}{b} \\ &= \frac{\beta_1}{b} + \frac{\beta_2}{b^2} + \frac{N_2}{b^2} \\ &= \beta_1 \times b^{-1} + \beta_2 \times b^{-2} + N_2 \times b^{-2}. \end{aligned}$$

Nous pouvons continuer ce développement, en multipliant successivement tous les  $N_i = 0, \alpha_i$  par  $b$ , pour obtenir une suite aussi longue que souhaitée de  $\beta_i$  tels que :

$$\begin{aligned} N &= \beta_1 \times b^{-1} + \beta_2 \times b^{-2} + \cdots + \beta_k \times b^{-k} + N_k \times b^{-k} \\ &= \sum_{i=1}^k \beta_i b^{-i} + N_k \times b^{-k}. \end{aligned} \tag{2.7}$$

En comparant (2.7) et (2.2), on voit que la séquence des  $\beta_i$  ainsi calculée correspond à la séquence des  $d_{-i}$  nécessaires pour exprimer  $N$  en base  $b$ . Ce développement sera arrêté soit lorsque  $N_i = 0$ , soit dès qu'on détecte deux valeurs  $N_j, N_i$  ( $i \neq j$ ) telles que  $N_j = N_i$ , ce qui signifie que l'expression de  $N$  en base  $b$  est infinie périodique.

## 2. Leçons 2 à 4 – Représentation de l'information



Considérons  $N = -24,42_{10}$  à exprimer en base 2. Nous allons commencer par exprimer 24 en base 2, en divisant successivement 24 par 2. On commence par  $24/2 = 12$  avec un reste de 0. Autrement dit  $q_0 = 12$  et  $r_0 = 0$ . On poursuit en divisant  $q_0$  par 2, etc :

$i$	$q_i$	$r_i$
0	12	0
1	6	0
2	3	0
3	1	1
4	0	1

On lit maintenant la séquence des restes de haut en bas, en on obtient :  $24_{10} = 11000_2$ .

Pour la partie fractionnaire, on procède par multiplication successive par 2. On commence par  $0,42 \times 2 = 0,84$ . Autrement dit,  $\beta_1 = 0$  et  $N_1 = 0,84$ . On multiplie à nouveau  $N_1$  par 2, soit  $2 \times 0,84 = 1,68$ . On obtient donc :

$i$	$\beta_i$	$N_i$	$i$	$\beta_i$	$N_i$	$i$	$\beta_i$	$N_i$
1	0	0,84	8	1	0,52	15	0	0,56
2	1	0,68	9	1	0,04	16	1	0,12
3	1	0,36	10	0	0,08	17	0	0,24
4	0	0,72	11	0	0,16	18	0	0,48
5	1	0,44	12	0	0,32	19	0	0,96
6	0	0,88	13	0	0,64	20	1	0,92
7	1	0,76	14	1	0,28	21	1	0,84

On voit donc que  $N_{21} = N_1$ , la suite du développement se poursuivra donc *ad infinitum* de manière cyclique. Donc :

$$0,42_{10} = 0,0\underline{110\ 1011\ 1000\ 0101\ 0011}_2$$

et finalement :

$$-24,42_{10} = -1\ 1000,\underline{0110\ 1011\ 1000\ 0101\ 0011}_2.$$

### 2.2.2. Opérations en base 2

Comme la base 2 est celle qui nous sera la plus utile dans la suite, nous allons maintenant nous intéresser aux différentes opérations que nous pouvons appliquer aux nombres dans cette base.

**Opérations arithmétiques** On peut utiliser l'addition, la soustraction, la multiplication et la division euclidiennes sur les nombres dans n'importe quelle base, en particulier la base 2. Pour l'addition, on procède en base 2 comme en base 10, en faisant d'abord la somme des unités, puis des chiffres correspondant au poids  $2^1$  (au lieu de  $10^1$  en base 10),  $2^2$ , etc... Il faut simplement être attentif au fait qu'un report a lieu dès que la somme dépasse  $2_{10} = 10_2$ . Par exemple :  $1 + 0$  donne une somme de 1, et un report de 0. Par contre,  $1 + 1 = 10_2$  donne une somme de 0 et un report de 1. De même,  $1 + 1 + 1 = 11_2$  donne une somme de 1 et un report de 1. Par exemple :



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

Dans le chapitre 4, nous utiliserons ces concepts pour construire un circuit qui réalise ces additions.

**Opérations logiques** En base 2, on peut utiliser une série d'opérations qui n'ont pas d'équivalent naturel dans les autres bases. Ce sont les opérations « logiques », que l'on peut appliquer en considérant que le 0 représente la valeur logique « faux », et que le 1 représente la valeur logique « vrai ».

On définit les opérateurs binaires<sup>6</sup> **et** (noté  $\wedge$ ), **ou** (noté  $\vee$ ), et **ou exclusif** (noté XOR); ainsi que l'opérateur unaire **non** (noté  $\neg$ ). Pour définir leur effet sur deux nombres binaires, on commence par les définir sur deux bits (ou un bit dans le cas du  $\neg$ ). Pour ce faire, on utilise une *table de vérité*, qui donne, pour chaque valeur possible des opérandes (de l'opérande), la valeur renvoyée par l'opérateur. Voici les tables de vérité de ces opérateurs :



$x$	$y$	$x \wedge y$	$x$	$y$	$x \vee y$	$x$	$y$	$x \text{XOR } y$	$x$	$\neg x$
0	0	0	0	0	0	0	0	0	1	0
0	1	0	0	1	1	0	1	1	1	0
1	0	0	1	0	1	1	0	1	0	1
1	1	1	1	1	1	1	1	0	0	0

On remarquera que ces opérateurs expriment l'idée intuitive qu'on peut s'en faire à la lecture de leur nom. L'opération  $x \wedge y$  vaut 1 si et seulement  $x$  et  $y$  sont vraies (soit  $x = 1$  et  $y = 1$ ). L'opération  $x \vee y$  vaut 1 si et seulement si  $x$  ou  $y$  est vraie. L'opération  $x \text{XOR } y$  vaut 1 si

6. Attention! Ici, « binaire » signifie que l'opérateur porte sur deux opérandes, comme l'addition par exemple. Au contraire, un opérateur unaire porte sur une seule opérande (comme le *moins* dans  $-5$  par exemple).

## 2. Leçons 2 à 4 – Représentation de l'information

et seulement si soit  $x$  est vraie soit  $y$  est vraie mais pas les deux en même temps (si  $x$  est vraie, cela exclut donc le fait que  $y$  soit vraie et vice-versa). Finalement,  $\neg x$  remplace la valeur de vérité de  $x$  par son opposé (vrai par faux et faux par vrai).

On peut maintenant généraliser ces opérations à n'importe quel nombre binaire, en appliquant les opérations *bit à bit* : étant donné deux nombres sur  $n$  bits  $A = a_{n-1}a_{n-2}\cdots a_0$  et  $B = b_{n-1}b_{n-2}\cdots b_0$ , on applique l'opération sur chaque paire de bits  $a_i, b_i$  pour obtenir le bit  $c_i$  du résultat. Par exemple,  $A \wedge B$  est le nombre  $C = c_{n-1}c_{n-2}\cdots c_0$ , où  $c_{n-1} = a_{n-1} \wedge b_{n-1}$ ,  $c_{n-2} = a_{n-2} \wedge b_{n-2}, \dots$  et  $c_0 = a_0 \wedge b_0$ . Et ainsi de suite pour les autres opérations.

**Masques** Ces opérations logiques permettent de réaliser certaines manipulations des valeurs binaires, appelées *masques* ou *masquages*. On observe que  $x \wedge 0 = 0$  et  $x \wedge 1 = x$ , quelque soit la valeur de  $x$  (sur un bit). De ce fait, le  $\wedge$  peut être utilisé pour *masquer* certains bit d'un nombre, c'est-à-dire remplacer ces bits par des 0 tout en conservant les autres. Il suffit de prendre la conjonction de ce nombre avec une valeur binaire comprenant des 0 à toutes les positions qu'on veut masquer, et des 1 partout ailleurs.



Supposons qu'on souhaite masquer les 4 bits de poids fort d'un nombre  $N$  de 8 bits. On peut faire :  $N \wedge 0000\ 1111$ . Si  $N = 1010\ 1010$ , on a bien  $N \wedge 0000\ 1111 = 0000\ 1010$ .

Les masques ont plusieurs utilités, nous en verrons certaines dans ce cours, notamment dans le Chapitre 9 consacré à la gestion de la mémoire à l'aide de la pagination. On peut déjà observer que les masques permettent de calculer le reste d'une division entière par une puissance de 2. En effet, nous avons déjà vu plus haut que la reste de la division entière de  $N$  par  $2^k$  est le nombre composé des  $k$  bits de poids faible de  $N$ . On peut donc masquer les autres bits pour ne retenir que ces  $k$  bits.



Par exemple :

$$\begin{aligned} 1010\ 0101_2 \bmod 8_{10} &= 1010\ 0101_2 \bmod 2^3 \\ &= 1010\ 0101_2 \wedge 0000\ 0\underbrace{111}_3 \\ &= 0000\ 0101_2 \\ &= 101_2. \end{aligned}$$

**Décalages, division et multiplications** Une autre opération utile (mais qui n'est pas réservée aux nombres binaires) est l'opération de *décalage de  $n$  positions* (où  $n$  est un naturel). Il y a deux types de décalage :

## 2.2. Écriture des nombres dans différentes bases

- le décalage vers la gauche de  $n$  positions, d'un nombre  $b$  est noté  $b << n$ , et consiste à ajouter  $n$  zéros à la droite (chiffres de poids faible) du nombre (quitte à supprimer les  $n$  chiffres de poids forts pour garder le même nombre de bits). Les chiffres constituant le nombre initial sont donc bien décalés vers la gauche.
- le décalage vers la droite de  $n$  positions, d'un nombre  $b$  est noté  $b >> n$ , et consiste à supprimer les  $n$  chiffres de poids faibles du nombre  $n$  (quitte à ajouter des 0 à gauche pour garder le même nombre de bits)



Par exemple, en binaire :  $0010\ 1101 >> 3 = 0000\ 0101$ . De même  $0010\ 1101 << 2 = 1011\ 0100$ .

Comme nous l'avons déjà vu plus haut, ces opérations permettent d'effectuer des multiplications (décalage vers la gauche de  $k$  positions) et des divisions (décalage vers la droite de  $k$  positions) par  $2^k$  (ou, de manière générale, par  $b^k$  dans n'importe quelle base  $b$ ).



Par exemple :

- $45_{10} \times 10_{10} = 45_{10} \times 10^1_{10} = 45_{10} << 1 = 450_{10}$
- $531_{10} \div 10_{10} = 531 >> 1 = 53_{10}$
- $100010_2 \div 4_{10} = 100010 \div 2^2 = 100010 >> 2 = 1000_2$

### 2.2.3. Représentation des entiers non-signés

Si nous devons manipuler des nombres entiers non-négatifs<sup>7</sup> uniquement, on peut se contenter d'exprimer ce nombre en base 2. Cette représentation n'utilisera que les symboles 0 et 1 et constitue donc bien une représentation binaire. Ces nombres sont souvent appelés les nombres « non signés » en informatique, car il n'est pas nécessaire d'utiliser de symbole pour représenter leur signe. Dans les langages comme C ou C++, par exemple, on trouve le type `unsigned int` [1, 18].

Si on considère une représentation sur un nombre  $n$  fixé de bits, cette technique permet de représenter tous les nombres de 0 (représenté par  $\underbrace{0 \cdots 0}_n$ ) à  $2^{n-1}$  (représenté par  $\underbrace{1 \cdots 1}_n$ ).

### 2.2.4. Représentation des nombres entiers signés

Voyons maintenant comment nous pouvons incorporer les nombres négatifs dans notre représentation. On ne peut pas se contenter de convertir les nombres vers la base 2, car le signe – qui est utilisé pour signaler qu'un nombre est négatif n'est pas directement manipulable par un ordinateur qui manipule des valeurs en binaire. Il faut donc trouver une technique

7. On se souviendra qu'un nombre positif est un nombre  $> 0$  et qu'un nombre négatif est un nombre  $< 0$ . Les nombres  $\geq 0$  sont donc les nombres « non-négatifs ».

## 2. Leçons 2 à 4 – Représentation de l'information

pour représenter ce signe à l'aide de 0 et de 1 uniquement. Nous allons étudier quatre techniques différentes. Nous présenterons ces techniques en supposant que l'on considère une représentation sur  $n$  bits (par exemple,  $n = 8$ ).

**Bit de signe** La première technique est la plus simple. Elle n'a que peu d'intérêt pratique, mais nous l'étudions quand même car elle sans doute la première idée à laquelle on pourrait songer, et qu'il est dès lors utile de la comparer à celles qui sont utilisées en pratique. Elle consiste à réserver le bit de poids fort pour représenter le signe du nombre (1 signifiant que le nombre est négatif), qui sera représenté en valeur absolue sur les  $n - 1$  bits de poids faible.



Avec le bit de signe sur  $n = 8$  bits, le nombre 5 est représenté par : 0000 0101. Le nombre  $-5$  est représenté par 1000 0101.

Cette technique présente plusieurs inconvénients :

- la valeur 0 possède deux représentations : 0…0 et 10…0 (qui représente  $-0$ ) ;
- il est difficile d'effectuer des opérations sur cette représentation. En particulier, effectuer la somme (selon la procédure usuelle) d'un nombre positif et d'un nombre négatif ne donne pas le résultat attendu…

Les valeurs représentables à l'aide du bit de signe, et sur  $n$  bits vont de  $-2^{n-1} + 1$  (représentée par 1…1) à  $2^{n-1} - 1$  (représentée par 01…1). Cela fait au total  $2^n - 1$  valeurs différentes, alors qu'il existe  $2^n$  représentations. La différence provient du fait que la zéro a deux représentations distinctes.

**Complément à 1** Le complément à 1 d'un nombre binaire  $N$  est le nombre  $\bar{N}$  qu'on obtient en inversant tous les bits de  $N$ . On peut utiliser le complément à 1 pour signaler qu'un nombre est négatif. Plus précisément, si on a une représentation sur  $n$  bits, on représente tous les nombres entiers non-négatifs par leur représentation binaire habituelle, et tous les nombres non-positifs par le complément à 1 de leur valeur absolue. Afin de pouvoir distinguer les nombres positifs des nombres négatifs, on limite les valeurs qu'on peut représenter : si on a une représentation sur  $n$  bits, on se limite aux nombres qui, en valeur absolue, tiennent sur  $n - 1$  bits. Ainsi, le bit de poids fort sera toujours égal à 0 pour les nombres positifs, et à 1 pour les nombres négatifs.



Le nombre 5 est représenté par 0000 0101. La représentation de  $-5$  est obtenue en inversant tous les bits de la représentation de 5, soit 1111 1010. Sur 8 bits, la valeur 128, par exemple, n'est pas représentable, elle s'exprime en binaire par un nombre de 8 bits au moins : 1000 0000, et cette représentation s'interprète, en complément à 1, comme  $-0111\ 1111_2 = -127$ .

L'avantage du complément à 1 sur le bit de signe est qu'il permet de faire des additions de manière relativement naturelle : on peut faire la somme usuelle de deux nombres (positifs ou

## 2.2. Écriture des nombres dans différentes bases

négatifs) en complément à 1 et obtenir la réponse correcte, à condition d'ajouter le dernier report au résultat.



Nous donnons deux exemples sur 4 bits.

Considérons  $3 - 2 = 3 + (-2)$ . En binaire, avec le complément à 1, on obtient  $0011 + 1101$ . En faisant la somme euclidienne, on obtient  $1\ 0000$ , soit, sur 4 bits,  $0000$  avec un dernier report de 1. On ajoute ce report aux 4 bits de poids faible, et on obtient :  $0001$ .

Considérons  $2 - 4 = 2 + (-4)$ . En binaire, avec le complément à 1, on obtient  $0010 + 1011$ . En faisant la somme euclidienne, on obtient  $1101$  (le dernier report est égal à 0), ce qui est bien la représentation, en complément à 1, de  $-2$ .

Malheureusement, comme avec le bit de signe, 0 possède toujours deux représentations en complément à 1 :  $0 \cdots 0$  et  $1 \cdots 1$ . De ce fait, on ne peut, sur  $n$  bits, représenter que les nombres de  $-2^{n-1} + 1$  à  $2^{n-1} - 1$ , soit à nouveau  $2^n - 1$  valeurs différentes.

**Complément à 2 :** Cette représentation est celle qui est utilisée en pratique pour les nombres entiers sur les processeurs modernes.

Le complément à 2 d'un nombre  $N$  est le nombre  $\bar{N}^2 = 2^n - N$  (où  $n$  est toujours le nombre de bits de la représentation). La représentation des nombres en complément à 2 suit le même principe que le complément à 1 : les nombres non-négatifs sont représentés par leur encodage binaire usuel; et les nombres négatifs sont représentés par le complément à 2 de leur valeur absolue.



Voici 3 exemples sur  $n = 8$  bits :

- 1 est représenté par  $00000001$ ;
- $-1$  est représenté par la représentation de  $2^8 - 1$ , soit  $11111111$ ;
- $-5$  est représenté par la représentation de  $2^8 - 5$ , soit  $11111011$ .

La technique du complément à 2 possède de nombreux avantages. Tout d'abord, 0 ne possède plus qu'une seule représentation, à savoir  $0 \cdots 0$ . Ensuite, on peut faire usage de l'addition usuelle sur tous les nombres positifs ou négatifs en complément à deux (il n'est pas nécessaire d'utiliser le report circulaire comme dans le cas du complément à 1, celui-ci peut être oublié).



Voici un exemple d'addition sur 4 bits. Considérons la somme  $7 - 5 = 7 + (-5)$ . En binaire et avec le complément à 2,  $-5$  est représenté par la représentation « classique » de  $2^4 - 5 = 16 - 5 = 11_{10}$ , soit  $1011_2$ . On a donc :  $0111_2 + 1011_2 = 1\ 0010_2$ , que l'on tronque sur 4 bits (on oublie systématiquement le dernier report) pour obtenir  $0010_2$ , soit  $2_{10}$ .

## 2. Leçons 2 à 4 – Représentation de l'information

Par ailleurs, comme zéro n'a plus qu'une seule représentation, on peut maintenant représenter  $2^n$  valeurs différentes sur  $n$  bits. La plus petite valeur représentable est maintenant  $-2^{n-1}$  (représentée par  $10\cdots 0$ ), et la plus grande est  $2^{n-1} - 1$  (représentée par  $01\cdots 1$ ). On a donc gagné une valeur dans les négatifs par rapport au complément à 1.

Enfin, remarquons que  $\overline{N}^2$  peut être calculé plus facilement grâce à  $\overline{N}$ , le complément à 1. En effet :

**Théorème 1** Pour tout  $N$  :  $\overline{N}^2 = \overline{N} + 1$ .

**Preuve.** Nous considérons une représentation des nombres sur  $n$  bits. Nous savons que la somme d'un nombre avec son complément à 1 donne :

$$\begin{aligned} N + \overline{N} &= 1\cdots 1 \\ &= 2^n - 1. \end{aligned}$$

Donc, en ajoutant 1 de part et d'autre de cette équation, nous avons :

$$\begin{aligned} N + \overline{N} + 1 &= 2^n - 1 + 1 \\ &= 2^n. \end{aligned}$$

En retranchant  $N$  de deux côtés, nous avons :

$$\begin{aligned} 2^n - N &= N + \overline{N} + 1 - N \\ &= \overline{N} + 1. \end{aligned}$$

Or,  $2^n - N = \overline{N}^2$ , par définition. La dernière équation prouve donc que  $\overline{N}^2 = \overline{N} + 1$ . □

**Excès à  $K$**  La technique de l'excès à  $K$  est à nouveau une idée simple : elle consiste à fixer une valeur  $K$  (appelée *bias*) suffisamment grande, et à représenter tous les nombres  $N$  (positifs ou négatifs) par la représentation binaire de  $N + K$  (nécessairement positif). De ce fait, sur  $n$  bits, toutes les valeurs entre  $-K$  (représentée par  $0\cdots 0$ ) et  $2^n - 1 - K$  (représentée par  $1\cdots 1$ ) sont représentables. On choisit souvent  $K$  égal à  $2^{n-1}$  de manière à répartir les valeurs positives et négatives représentables de manière équitable, mais ce n'est pas obligatoire (par exemple, dans la norme IEEE754 que nous verrons plus tard, ce n'est pas le cas).



Voici trois exemples sur 8 bits avec  $K = 2^{n-1} = 2^7 = 128_{10}$  :

- 1 est représenté par la représentation binaire de  $1 + 2^7 = 129$ , soit  $1000\ 0001$ ;
- $-1$  est représenté par la représentation de  $2^7 - 1$ , soit  $0111\ 1111$ ;
- $-5$  est représenté par la représentation de  $2^7 - 5$ , soit  $0111\ 1011$ .

## 2.2. Écriture des nombres dans différentes bases

TABLE 2.1. – Comparaison des différentes représentations (sur  $n$  bits).

Représentation	Valeur représentée				
	Non signé	Bit Signe	Cpl. 1	Cpl. 2	Excès $K$
00…00	0	0	0	0	$-K$
00…01	1	1	1	1	$-K+1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
01…11	$2^{n-1}-1$	$2^{n-1}-1$	$2^{n-1}-1$	$2^{n-1}-1$	$2^{n-1}-1-K$
10…00	$2^{n-1}$	0	$-2^{n-1}+1$	$-2^{n-1}$	$2^{n-1}-K$
10…01	$2^{n-1}+1$	-1	$-2^{n-1}+2$	$-2^{n-1}+1$	$2^{n-1}+1-K$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
11…11	$2^n-1$	$-2^{n-1}+1$	0	-1	$2^n-1-K$

**Comparaison des différentes représentation** Afin de bien comprendre comment les différentes techniques de représentation fonctionnent, il peut être utile de les comparer. C'est l'objet de la TABLE 2.1 : elle montre comment une même représentation binaire sur  $n$  bits (colonne de gauche) représente des valeurs différentes en fonction de la convention utilisée (voir aussi, à ce sujet, la discussion à la fin du présent chapitre).

### 2.2.5. Représentation des nombres « réels »

Nous allons maintenant étudier des techniques permettant de représenter des nombres qui ne sont pas entiers. Comme nous l'avons déjà observé, il est impossible, en toute généralité, de représenter tous les nombres *réels* à l'aide d'un ordinateur. En effet, les  *nombres irrationnels*<sup>8</sup>, comme le nombre  $\pi$ , ont une *partie décimale infinie non-périodique*<sup>9</sup>. Or tout ordinateur ne dispose jamais que d'une quantité finie de mémoire, et la représentation explicite d'un nombre rationnel (dans n'importe quelle base naturelle) demanderait donc une mémoire infinie.

Les nombres que nous allons être en mesure de représenter sont tous des nombres *rationnels*, c'est-à-dire des nombres que nous exprimerons (dans une base fixée) sous la forme  $\alpha,\beta$  : à l'aide d'une partie entière  $\alpha$  et d'une partie fractionnaire  $0,\beta$ , toutes les deux finies. Par exemple, pour le nombre 42,625, nous avons  $\alpha = 42$  et  $\beta = 625$ .

**Une première technique : la virgule fixe** Comme nous l'avons déjà remarqué, si nous pouvons exprimer séparément  $\alpha$  et  $0,\beta$  en binaire, nous ne pouvons pas exprimer explicitement la virgule, et nous devons donc trouver une manière de contourner ce problème. Une première technique, qui est simple mais qui a ses limites, consiste à fixer arbitrairement la position de la virgule au sein d'une représentation de taille fixée. Par exemple, si on considère des représentations sur  $n = 32$  bits, on pourrait décider que les 16 bits de poids fort

8. Il s'agit des nombres réels qui ne sont pas rationnels.

9. C'est-à-dire qu'elle ne peut pas être exprimée sous la forme d'un préfixe fini suivi d'une répétition infinie d'une séquence finie, car il est connu que tout nombre qui peut être exprimé de cette manière est rationnel.

## 2. Leçons 2 à 4 – Représentation de l'information

représentent la partie entière  $\alpha$ , et que les 16 bits de poids faible représentent la partie fractionnaire de  $0.\beta$ .



Avec la convention ci-dessus, le nombre 42,625 est représenté par :

0000 0000 0010 1010	1010 0000 0000 0000
---------------------	---------------------

En effet,  $42,625_{10} = 10\ 1010,101_2$ . On remarque que des zéros ont été ajoutés pour compléter les 32 bits : *à gauche* de la partie entière et *à droite* de la partie décimale.

Le problème de cette technique est qu'elle limite de manière excessive les nombres qu'on peut représenter, comme le montre l'exemple suivant :



En suivant la convention donnée ci-dessus, le nombre  $2^{-17}$  ne peut pas être représenté. En effet :

$$2^{-17} = 0,0000\ 0000\ 0000\ 0000_2,$$

et donc, la partie  $\beta$  n'est pas représentable sur les 16 bits alloués dans notre représentation. On a affaire à un nombre *trop petit*, on parle d'*underflow*.

De même, le nombre  $2^{16}$  ne peut pas être représenté car :

$$2^{16} = 1\ 0000\ 0000\ 0000\ 0000_2.$$

On a ici affaire à un *overflow*.

Ces deux exemples sont un peu frustrants. On a en effet une représentation sur 32 bits, qui permettrait aisément de représenter tant  $2^{-17}$  que  $2^{16}$ , si on avait l'opportunité de déplacer la virgule. En effet, tant la partie entière de  $2^{-17}$  que la partie décimale de  $2^{16}$  se réduisent à un seul 0, et ne nécessitent donc certainement pas 16 bits.

**La virgule flottante : IEEE754** Pour remédier à ce problème, on peut utiliser une technique dite de *virgule flottante*, où la position de la virgule n'est pas spécifiée *a priori*, mais où la représentation binaire du nombre (tant sa partie entière que sa partie décimale) s'accompagne d'une information qui indique où positionner la virgule. Une telle technique s'inspire de la *notation scientifique* des nombres, qui consiste à exprimer tous les nombres (en base 10) sous la forme :

$$0,f \times 10^x.$$

## 2.2. Écriture des nombres dans différentes bases



Voici trois nombres et leur représentations « scientifiques » respectives :

$$\begin{aligned} 42,625 &= 0,42625 \times 10^2 \\ 0,00034 &= 0,34 \times 10^{-3} \\ 25 &= 0,25 \times 10^2. \end{aligned}$$

On voit bien sur ces exemples que l'exposant  $x$  indique la position de la virgule, où, pour être plus précis, le nombre de décalages (vers la droite pour un exposant positif, vers la gauche pour un exposant négatif) de la virgule qu'il faut affecter au nombre 0, ... pour retrouver le nombre d'origine.

Ce principe se retrouve dans le standard industriel IEEE754-2008<sup>10</sup>, dont nous allons maintenant étudier une partie, à titre exemplatif [5]. Nous allons nous concentrer sur la représentation des nombres sur 32 bits. Dans cette norme<sup>11</sup>, les 32 bits sont répartis entre :

- un bit de signe  $s$ , le bit de poids fort;
- suivi de 8 bits représentant un *exposant*  $e$ , exprimé en *excès à 127*;
- suivis de 23 bits de *signifiant*  $f$ .

Une telle représentation est l'encodage du nombre suivant :

$$(-1)^2 \times 1,f \times 2^e.$$

On voit donc que le bit  $s$  se comporte bien comme un bit de signe (le nombre représenté est négatif si et seulement si  $s = 1$ ). On suppose qu'on a préalablement exprimé le nombre sous la forme  $1,f$ , et seuls les bits de  $f$  sont effectivement stockés dans la représentation, ce qui permet d'économiser un bit. Enfin, l'exposant  $e$  est une puissance de 2 et non pas de 10 comme dans la représentation scientifique, ce qui est logique étant donné que nous utilisons une représentation binaire. L'exposant indique donc bien un décalage à affecter à la virgule.

10. L'IEEE est l'*Institute of Electrical and Electronics Engineers*, un association à but non-lucratif américaine regroupant des centaines de milliers de professionnels de l'électronique et de l'informatique. Elle conçoit et publie des normes qui peuvent ensuite être adoptée et mise en pratique par l'industrie. La norme IEEE754-2008 est la révision en 2008 de la norme 754 qui définit une représentation en virgule flottante adaptée aux ordinateurs. Le groupe de travail qui conçoit et publie cette norme possède une page web qu'on peut consulter pour plus d'information : <https://standards.ieee.org/develop/wg/754.html>.

11. Le site web <http://babbage.cs.qc.cuny.edu/IEEE-754/> implémente un convertisseur automatique qu'on peut utiliser pour se familiariser avec cette norme.

## *2. Leçons 2 à 4 – Représentation de l'information*



Considérons à nouveau le nombre  $42,625_{10}$ . Pour trouver sa représentation IEEE754, nous commençons par l'exprimer en binaire :

$$42,625_{10} = 10\ 1010,101_2.$$

Nous normalisons ensuite cette représentation en déplaçant la virgule de 5 positions vers la gauche pour obtenir un nombre de la forme  $1, f \times 2^e$  :

$$42,625_{10} = 1,0101\ 0101 \times 2^5.$$

À noter que l'exposant est positif pour maintenir l'égalité. Nous pouvons maintenant trouver aisément les différents composants de la représentation :

- le signe  $s = 0$ , car le nombre est positif;
  - l'exposant  $e = 5$ , que nous devons représenter en excès à 127 sur 8 bits. Cela revient à représenter  $5 + 127 = 132$  en binaire, soit 1000 0100;
  - enfin, le significant est la partie après la virgule :  $f = 01010101$ .

Nous avons donc la représentation :

0	1000 0100	0101 0101 0...0
---	-----------	-----------------

Remarquons que les 0 ont été ajoutés dans les bits de poids faible du signifiant, afin de ne pas changer sa valeur (ajouter des zéros dans les bits de poids forts reviendrait à insérer des zéros juste à droite de la virgule). .../.



Comme ces nombres sont relativement longs à écrire, il est souvent pratique d'utiliser une représentation en hexadécimal pour la totalité de l'encodage binaire :

$$\begin{array}{ccccccccc}
 0100 & 0010 & 0010 & 1010 & 1000 & 0000 & 0000 & 0000 \\
 & & & & = & & & \\
 4 & 2 & 2 & a & 8 & 0 & 0 & 0
 \end{array}$$

soit : 422a8000.

Un problème que nous devons encore résoudre est la représentation de 0. En effet, 0 ne peut pas s'exprimer sous la forme  $1,f$ , il faut donc fixer une représentation spéciale. La norme IEE754 en retient deux :  $10\cdots0$ , soit  $00\cdots0$ .

Notons enfin que la norme admet également des valeurs spéciales, comme  $\infty$  et *Not a Number* (ou **NaN**). Ces valeurs sont utilisées pour certains résultats des opérations arithmétiques. Par exemple pour une division par 0, comme on peut le voir dans le tableau suivant :

$x/y$	$y \neq 0, \infty, \text{NaN}$	$y = \infty$	$y = 0$	$y = \text{NaN}$
$x \neq 0, \infty, \text{NaN}$	valeur la plus proche de $x/y$	0	$\infty$	$\text{NaN}$
$x = \infty$	$\infty$	$\text{NaN}$	$\infty$	$\text{NaN}$
$x = 0$	0	0	$\text{NaN}$	$\text{NaN}$
$x = \text{NaN}$	$\text{NaN}$	$\text{NaN}$	$\text{NaN}$	$\text{NaN}$

En pratique, la manipulation de cette représentation demande des circuits spéciaux, que l'on trouve sur la plupart des processeurs modernes.



Sur le processeur Intel 486 [9], il est possible de manipuler des données en virgule flottante selon la norme IEE754 (originelle), sur 32, 64 ou même 80 bits. Ces données doivent être chargées dans des registres spéciaux de 80 bits appelés  $st0, \dots, st7$ . Des instructions dédiées comme `fadd`, `fsub`, `fdiv`, etc implémentent les opérations arithmétiques sur ces registres. Ces opérations ne sont pas réalisées par l'ALU, mais par un circuit dédié du processeur : le FPU (*floating point unit*), qui était un circuit séparé sur les processeurs Intel précédent le 486 (par exemple, pour le 386, il fallait acheter séparément un co-processeur appelé 387 pour disposer d'un FPU).

## 2.3. Représentation des caractères

Maintenant que nous avons étudié des techniques pour représenter et manipuler des  *nombres*  en binaire, intéressons-nous à d'autres types d'informations. Nous commencerons par le  *texte* , c'est-à-dire des séquences de caractères. L'idée générale de la représentation (en binaire) des caractères consiste à faire correspondre à chaque caractère un et un seul nombre naturel, et à représenter le caractère par la représentation binaire de ce naturel. Comme il n'existe pas de manière universelle et naturelle d'associer un nombre à chaque caractère, il faut fixer une table, qui fait correspondre chaque caractère à un nombre.

**Codes historiques : Émile Baudot et les télescripteurs** Le besoin de représenter et transmettre de l'information de manière mécanique est une préoccupation qui a reçu une première réponse lors de l'introduction du télégraphe, où le fameux code Morse était utilisé pour représenter les lettres par une série de traits et de points. Au dix-neuvième siècle, un français, Jean-Maurice Émile BAUDOT<sup>12</sup> invente le premier code pratique permettant de représenter tout l'alphabet latin (ainsi que des symboles de ponctuation usuels) sur 5 bits. Ce code est présenté à la FIGURE 2.1.



12. Ingénieur français, né le 11 septembre 1845, mort le 28 mars 1903.

## 2. Leçons 2 à 4 – Représentation de l'information

La machine de BAUDOT comprenait un clavier de 5 touches, avec lequel on entrait les caractères à transmettre, selon le code que nous avons présenté. Le récepteur disposait d'une machine imprimant automatiquement les caractères transmis sur un ruban de papier. La cadence à laquelle les caractères pouvaient être transmis (par exemple, 10 caractères par minute) était mesurée en *bauds*, une unité que l'on connaît encore aujourd'hui, bien qu'elle soit tombée en désuétude. Le système de BAUDOT sera plus tard amélioré pour utiliser du ruban perforé pour stocker les informations à transmettre. Cela donnera lieu aux *téléscripteurs* (ou *teletypes* en anglais, abbrévié TTY), sortes de machines à écrire qui ont la capacité d'envoyer et de recevoir du texte en utilisant le code de BAUDOT, sur des lignes téléphoniques. Ces systèmes ont été largement utilisés pour la transmission d'information, notamment par la presse, jusque dans les années 1980.

**Le code ASCII** L'évolution la plus marquante des développements historiques que nous venons de présenter est le code ASCII (*American Standard Code for Information Interchange*, présenté en 1967), qui est encore en usage aujourd'hui. Il comprend 128 caractères encodés sur 7 bits, et est présenté à la FIGURE 2.2.



Avec le code ASCII la chaîne de caractères INFO-F-102 est représentée par (en hexadécimal) :

49 4E 46 4F 2D 46 2D 31 30 32,

ou, en binaire, par :

100 1001 100 1110 100 0110 100 1111 010 1101 100 0110 010 1101  
011 0001 011 0000 011 0010.

Ce code, bien adapté à la langue anglaise, ne permet pas de représenter les caractères accentués. C'est pourquoi plusieurs extensions ont vu le jour, sur 8 bits, où les 128 caractères supplémentaires permettaient d'encoder les caractères propres à une langue choisie. Par exemple sur l'IBM PC, ces extensions étaient appelées *code pages*. En voici deux exemples :

- le *code page* 437 : le jeu de caractère ASCII standard auxquels on a ajouté les caractères accentués latin ;
- le *code page* 737 : le code ASCII standard plus les caractères grecs, voir FIGURE 2.3.

L'utilisation de ces *code pages* supposait que l'utilisateur connaissaient le code qui avait été utilisé pour représenter le texte source, et qu'il disposait des moyens logiciels et matériels pour afficher les caractères correspondants. En pratique, cela se révélait souvent fastidieux, et donnait lieu à des surprises si on se trompait de *code page*...

(No Model.)

11 Sheets—Sheet 6.

J. M. E. BAUDOT.  
PRINTING TELEGRAPH.

No. 388,244.

Patented Aug. 21, 1888.

*Fig. 24.*

	1	2	3	4	5
A	+	-	-	-	-
B	-	-	+	+	-
C	+	-	+	+	-
D	+	+	+	+	-
E	-	+	-	-	-
F	-	+	+	+	-
G	-	+	-	+	-
H	+	+	-	+	-
I	-	+	+	-	-
J	+	-	-	+	-
K	+	-	-	+	+
L	+	+	-	+	+
M	-	+	-	+	+
N	-	+	+	+	+
O	+	+	+	-	-
P	+	+	+	+	+
Q	+	-	+	+	+
R	-	-	+	+	+
S	-	-	+	-	-
T	+	-	+	-	+
U	+	-	+	-	-
V	+	+	+	-	+
W	-	+	+	-	+
X	-	+	-	-	+
Y	-	-	+	-	-
Z	+	+	-	-	+
t	+	-	-	-	+
ñ	-	-	-	+	-
ñ	-	-	-	-	-

INVENTOR:

Jean Maurice Emile Baudot

FIGURE 2.1. – Le code Baudot, un système historique de représentation binaire des caractères (on peut associer le + à 1 et le - à 0), breveté en 1882 en France. On voit ici un extrait du brevet américain (1888).

## 2. Leçons 2 à 4 – Représentation de l'information

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

FIGURE 2.2. – Le code ASCII. Chaque caractère est représenté par une valeur hexadécimale sur 2 chiffres : le chiffre de poids fort est donné par la ligne, le chiffre de poids faible par la colonne.

**Unicode** Plus récemment, le projet *Unicode*<sup>13</sup> a vu le jour, et s'est donné pour objectif de créer une norme reprenant la plupart des systèmes d'écriture utilisés dans le monde, et de leur attribuer un encodage. La version en cours d'Unicode est la version 11 et elle comprend 137 439 caractères. La norme Unicode associe plusieurs encodages à chaque caractère. L'encodage le plus courant est appelé UTF-8 : il s'agit d'un encodage à longueur variable car chaque caractère est encodé sur 1, 2, 3 ou 4 octets. Tous les caractères encodés sur 1 octet sont compatibles avec le standard ASCII. Les encodages sur 2, 3 et 4 octets sont utilisés pour représenter d'autres caractères, aussi exotiques soient-ils... Par exemple, la FIGURE 2.4 présente l'encodage Unicode de l'alphabet Tagbanwa, en usage aux Philippines [21, 20].

La norme Unicode est devenue aujourd'hui bien adoptée, notamment par les sites et navigateurs web.

## 2.4. Représentation d'images

Il existe de nombreux formats permettant de stocker et manipuler des images, les plus utilisés étant sans doute le JPEG<sup>14</sup> [8], le PNG<sup>15</sup> [7] ou le TIFF<sup>16</sup> (ou pourrait encore citer des formats historiques comme le GIF ou le BMP...) Ces formats sont utilisés par les sites webs, par les appareils photos numériques, etc. Tous ces formats représentent des images de type *bitmap*<sup>17</sup>, et suivent le même principe de base. Une image y est vue comme une matrice de points, appelés *pixels* (contraction de l'anglais *picture elements*), qui sont indivisibles, et monochromatiques. Ce sont les éléments de base constitutifs d'une image. La couleur de

13. Le consortium qui conçoit et publie Unicode possède un site web <https://unicode.org/>.

14. Voir <https://jpeg.org/>

15. Voir : <http://www.libpng.org/pub/png/>

16. Voir : <https://www.loc.gov/preservation/digital/formats/fdd/fdd000022.shtml>

17. Notons qu'il existe d'autres formats, comme le SVG, qui représentent les images sous formes de points connectés par des lignes ou des courbes décrites de manière mathématique. Ces formats ont l'avantage de permettre des agrandissements sans perte de qualité.

## 2.4. Représentation d'images

Only the upper half (128–255) of the table is shown, the lower half (0–127) being plain ASCII.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
8-	A 391	B 392	Γ 393	Δ 394	Ε 395	Ζ 396	Η 397	Θ 398	Ι 399	Κ 39A	Λ 39B	Μ 39C	Ν 39D	Ξ 39E	Ο 39F	Π 3A0
9-	P 3A1	Σ 3A3	Τ 3A4	Υ 3A5	Φ 3A6	Χ 3A7	Ψ 3A8	Ω 3A9	α 3B1	β 3B2	γ 3B3	δ 3B4	ε 3B5	ζ 3B6	η 3B7	θ 3B8
A-	λ 3B9	κ 3BA	λ 3BB	μ 3BC	ν 3BD	ξ 3BE	ο 3BF	ρ 3C0	π 3C1	σ 3C3	ς 3C4	τ 3C5	υ 3C6	φ 3C7	ψ 3C8	
B-	β 2591	β 2592	β 2593	β 2502	β 2524	β 2561	β 2562	β 2556	β 2555	β 2563	β 2551	β 2557	β 255D	β 255C	β 255B	β 2510
C-	λ 2514	λ 2534	τ 252C	τ 251C	– 2500	+ 253C	 255E	 255F	 255A	 2554	 2569	 2566	 2560	= 2550	 256C	 2567
D-	β 2568	β 2564	τ 2565	τ 2559	τ 2558	τ 2552	τ 2553	τ 256B	τ 256A	τ 2518	τ 250C	τ 2588	τ 2584	τ 258C	τ 2590	τ 2580
E-	ω 3C9	ά 3AC	έ 3AD	ή 3AE	ύ 3CA	ό 3AF	ύ 3CC	ύ 3CD	ύ 3CB	ύ 3CE	ά 386	ή 388	ή 389	ή 38A	ή 38C	ή 38E
F-	Ω 38F	± B1	≥ 2265	≤ 2264	÷ 3AA	≈ 3AB	≈ F7	≈ 2248	≈ B0	≈ 2219	≈ B7	≈ 221A	≈ 207F	≈ B2	≈ 25A0	≈ A0

Retrieved from "[http://en.wikipedia.org/wiki/Code\\_page\\_737](http://en.wikipedia.org/wiki/Code_page_737)"

FIGURE 2.3.– Le Code Page 737 : une extension du code ASCII pour obtenir les caractères grecs. [3]

## 2. Leçons 2 à 4 – Représentation de l'information

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+176x	ꝑ	Ꝓ	ꝓ	Ꝕ	ꝕ	ꝗ	Ꝙ	ꝙ	Ꝛ	ꝛ	Ꝝ	ꝝ	Ꝟ	ꝟ	ꝧ	Ꝩ
U+177x	ꝑ		ꝓ	Ꝕ												

FIGURE 2.4. – Extrait du standard Unicode, alphabet Tagbanwa [21].

chaque *pixel* peut (comme on l'a fait pour les caractères) alors être représentée par un naturel selon un encodage fixé *a priori*. Par exemple, sur 8 bits on pourra avoir une palette de 256 couleurs, ou 256 niveaux de gris... On peut alors représenter l'image comme la suite des encodages binaires de chacun des pixels, la matrice étant lue ligne par ligne (par exemple). Un fichier contenant une image contiendra en général d'autres informations utiles (comme les dimensions de l'image).



Un format de fichier d'image élucidant clairement ces concepts est le format PGM. Il s'agit en fait d'un fichier texte, contenant plusieurs valeurs numériques, stockées en ASCII. La FIGURE 2.5 présente un exemple de fichier PGM, avec l'image représentée.

Le fichier commence par « P2 », qui indique, par convention, que les couleurs représentées sont en fait des niveaux de gris. Ensuite, viennent les dimensions de l'image : 45 pixels de large, par 7 de haut. Puis, vient le nombre de couleur utilisées : ici, on pourra utiliser les nombres de 0 à 14 pour représenter des niveaux de gris qui s'échelonnent du noir (0) au blanc (14). Enfin, vient la suite des valeurs numériques décrivant chaque pixel, énumérés ligne par ligne.

Les formats d'images utilisés en pratique comme JPEG ne se contentent pas de stocker la suite des pixels de l'image, mais appliquent aussi des techniques de compression pour réduire l'espace nécessaire pour stocker cette information [8]. Notons enfin qu'une vidéo n'est jamais qu'une séquence d'image fixes. Les concepts développés ici s'appliquent donc aux différents formats (MPEG, AVI, etc) utilisés pour encoder des images en mouvement.

## 2.5. Représentation des instructions

Il y a encore un type d'*informations* que l'ordinateur doit pouvoir manipuler de manière cruciale : ce sont les instructions (machine) des programmes à exécuter. En effet, notre définition de la notion d'ordinateur insiste sur le fait que le programme doit pouvoir être modifié arbitrairement par l'utilisateur. Le programme doit donc être fourni à l'ordinateur dans un format bien spécifié, afin qu'il puisse être analysé et exécuté (nous pensons ici à la boucle *fetch-decode-execute* du CPU dont nous avons déjà parlé brièvement dans le Chapitre 1).

La manière dont les instructions du langage machine sont représentées dépend du type de processeur qui va les exécuter, il est donc assez difficile de présenter des généralités. Nous pouvons déjà dire que toute représentation binaire d'une instruction sera composée de deux

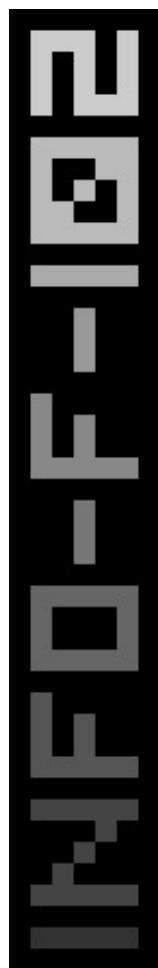


FIGURE 2.5. – Un exemple de contenu de fichier PGM avec l'image qu'il représente...

## 2. Leçons 2 à 4 – Représentation de l'information

parties :

1. l'*opcode* (abréviation anglaise d'*operation code*), qui représente l'instruction à proprement parler, c'est-à-dire le traitement à effectuer (par exemple : faire une addition ou faire une soustraction) ; et
2. éventuellement une ou plusieurs *opérandes*, qui donnent accès aux données à traiter.

Nous discuterons plus en détail du langage machine et de sa représentation dans le Chapitre 6.



Voici deux exemples d'instructions de l'i486 :

1. L'instruction `hlt` (*halt*) qui arrête le processeur est représentée par 8 bits : 1111 0100. Cette instruction très simple ne comprend donc qu'une *opcode* (1111 0100) et ne nécessite pas d'opérande.
2. Pour faire la somme de deux registres de 32 bits, on utilise l'instruction `add` qui tient sur 16 bits et attend deux opérandes : les numéros des registres à sommer; et qui place son résultat dans le premier de ces deux registres. Cette instruction est représentée de la façon suivante (sur 2 octets, avec l'octet de poids fort à gauche, les numéros des bits étant indiqués au-dessus) :

15	8	7	0
0000 0001	11	<i>reg</i> <sub>1</sub>	<i>reg</i> <sub>2</sub>

où *reg*<sub>1</sub> et *reg*<sub>2</sub> représentent les registres à sommer, sur 3 bits (par exemple, `eax` est représenté par 000 et `ebx` par 001). Dans cet exemple, l'opérande tient sur dix bits (0000 0001 11).

## 2.6. Détection et correction d'erreurs

Lorsque les informations sont stockées ou transmises, il arrive parfois que des erreurs soient introduites (par exemple, sur un réseau non-fiable). Il est alors utile de développer des techniques de représentation de l'information qui permettent au récepteur de déetecter la présence d'erreurs éventuelles (auquel cas, il peut demander une re-transmission), voire de les corriger par lui-même. Nous allons maintenant étudier deux de ces techniques. Elles sont toutes les deux basées sur le même principe : introduire de la redondance dans l'information transmise, de manière à pouvoir détecter un nombre restreint d'erreurs, voire reconstituer l'information d'origine.

### 2.6.1. Bit de parité

Cette technique est sans doute la première à avoir été mise en œuvre. Elle consiste à ajouter, à toute information transmise, un bit (appelé *bit de parité*) qui indique si le nombre de bits à 1

dans l'information à représenter est impair (bit de parité à 1) ou pair (bit de parité à 0). Ainsi, si un seul bit est modifié, la parité change nécessairement, et cette erreur peut être détectée (mais pas corrigée car on n'a pas de moyen d'identifier le bit qui a été modifié).

Plus précisément, étant donnée un information binaire

$$b_{n-1} \cdots b_0,$$

on lui associe le bit de parité (ou bit de contrôle)  $b_c$  calculé de la manière suivante :

$$b_c = b_{n-1} \text{XOR } b_{n-2} \cdots \text{XOR } b_0.$$

On peut vérifier que  $b_c = 1$  si et seulement si le nombre de bits à 1 parmi  $b_{n-1}, \dots, b_0$  est impair.



Fixons  $n = 7$ . Si l'information considérée est 011 1111 (ce qui correspond par exemple au caractère ASCII ?, voir FIGURE 2.2), on a le bit de contrôle  $b_c = 0$ . On obtient alors l'octet 0011 1111, en supposant que le bit de contrôle est le bit de poids fort.

Supposons maintenant que la donnée soit modifiée en 0011 **0**111 (le bit  $b_3$  a été inversé), suite à une erreur. On peut calculer la parité de l'information reçue  $0\text{XOR}1\text{XOR}1\text{XOR}0\text{XOR}1\text{XOR}1\text{XOR}1 = 1$ , et constater que ce n'est pas le bit de contrôle reçu. On a donc détecté une erreur. Si, par exemple, l'information a été transmise sur un réseau, on peut demander une retransmission.

Supposons maintenant qu'une erreur ait lieu sur le bit de parité : la valeur 1011 1111 est reçue. À nouveau, une erreur est détectée, même si l'information à proprement parler est correcte.

Supposons maintenant que *deux* erreurs ont lieu : 0011 **0**011. On voit que le bit de contrôle calculé est 0, ces deux erreurs ne seront donc pas détectées.

On peut facilement se convaincre que la technique du bit de parité permet de *détecter* un nombre *impair* d'erreurs, mais ne permet d'en corriger aucune car rien ne permet d'identifier le bit qui a été altéré.

## 2.6.2. Code de Hamming

Le code de HAMMING est en fait une *famille* de codes dont la version de base (appelée Hamming(7,4) et que nous allons étudier ici) a été introduite dans les années 1950 [4] par Richard HAMMING<sup>18</sup>, pour détecter et corriger des erreurs qui avaient lieu lors de la lecture de cartes et de rubans perforés (voir Chapitre 3).

Ce système est plus puissant que le bit de parité : il permet de *corriger* les erreurs de transmission, à condition que celles-ci affectent au maximum 1 bit. Naturellement, pour arriver à ce résultat, il faudra utiliser plus d'un bit de contrôle.

<sup>18</sup>. Né le 11 février 1915 et mort le 7 janvier 1998, Richard HAMMING est un mathématicien américain, récipiendaire du prix Turing en 1968, qui a travaillé entre autres pour le projet Manhattan et aux laboratoires des Bell Labs.

## 2. Leçons 2 à 4 – Représentation de l'information

Pour expliquer le code de HAMMING, nous allons changer quelque peu nos conventions et numérotter les bits à partir de 1. Nous allons considérer, pour l'exemple, des représentations (information et bits de contrôle) sur 7 bits, mais ces principes se généralisent aisément. Sur ces 7 bits, tous les bits dont le numéro est une puissance de 2 (donc, les bits 1, 2 et 4) sont des bits de contrôle ( $C_i$ ), les 4 bits restants sont des bits de données ( $D_i$ ) :

1	2	3	4	5	6	7
$C_1$	$C_2$	$D_3$	$C_4$	$D_5$	$D_6$	$D_7$

Chacun des bits de contrôle sera en fait un bit de parité, mais ne portera pas sur *tous* les bits de données (sans quoi il serait inutile d'avoir plusieurs bits de contrôle qui auraient tous la même valeur). Au contraire, nous allons associer les bits de contrôle à des bits de données bien choisis, de manière à pouvoir reconstituer le numéro du bit erroné en cas d'erreur. Nous effectuons cette association de la manière suivante



Dans le code de HAMMING, le bit de contrôle  $C_i$  est un bit de parité pour tous les bits de données  $S_j$ , tels que le bit de poids  $i$  est à 1 dans l'expression binaire de  $j$ .

Quand un bit  $C_i$  est un bit de parité pour un bit de donnée  $D_j$ , nous dirons que «  $C_i$  vérifie  $D_j$  ».

Élucidons cette définition dans le cas où  $n = 7$ . Les représentations binaires de numéros des bits de donnée sont les suivants  $011 = 3_{10}$ ,  $101 = 5_{10}$ ,  $110 = 6_{10}$  et  $111 = 7_{10}$ . Donc, le bit de donnée 3 est vérifié par les bits de contrôle 2 et 1 (car en effet  $1 + 2 = 3$ ) ; le bit de donnée 5 est vérifié par 4 et 1 ( $4 + 1 = 5$ ) ; le bit de donnée 6 est vérifié par 4 et 2 ( $4 + 2 = 6$ ) ; et le bit de donnée 7 est vérifié par 4, 2 et 1 ( $4 + 2 + 1 = 7$ ). On peut donc observer les deux propriétés suivantes, qui découlent de la définition des bits de contrôle et qui seront cruciales dans la suite pour *corriger* erreurs :

1. quand on fait la somme des numéros des bits de contrôle qui vérifient un bit de donnée  $D_j$ , on retrouve cette valeur  $j$ ; et
2. chaque bit de donnée est vérifié par au moins 2 bits de contrôle. Cela provient du fait que les numéros de bits de données ne sont *pas* des puissances de 2, leur représentation en binaire comprend donc au moins deux bits à 1.

On peut ré-exprimer la relation entre les bits de contrôle et les bits de données de la façon suivante, ce qui nous permet de calculer directement les bits de contrôle :

- le bit de contrôle  $C_1$  vérifie les bits de données  $D_3$ ,  $D_5$  et  $D_7$ ;
- le bit de contrôle  $C_2$  vérifie les bits  $D_3$ ,  $D_6$  et  $D_7$ ; et
- le bit de contrôle  $C_4$  vérifie les bits  $D_5$ ,  $D_6$  et  $D_7$ .

Par exemple le bit de contrôle  $C_2$  est un bit de parité pour  $D_3$ ,  $D_6$  et  $D_7$ , donc

$$C_1 = D_3 \text{XOR} D_6 \text{XOR} D_7.$$

Mettons ces idées en pratique sur un exemple :



Supposons que nous ayons la donnée 1011 :

$C_1$	$C_2$	$D_3$	$C_4$	$D_5$	$D_6$	$D_7$
		1		0	1	1

En appliquant le développement ci-dessus, nous obtenons :

$$\begin{aligned} C_1 &= D_3 \text{XOR} D_4 \text{XOR} D_7 = 1 \text{XOR} 0 \text{XOR} 1 = 0; \\ C_2 &= D_3 \text{XOR} D_6 \text{XOR} D_7 = 1 \text{XOR} 1 \text{XOR} 1 = 1; \\ C_4 &= D_3 \text{XOR} D_5 \text{XOR} D_7 = 0 \text{XOR} 1 \text{XOR} 1 = 0. \end{aligned}$$

La représentation de la donnée accompagnée de son code de HAMMING est donc :

$C_1$	$C_2$	$D_3$	$C_4$	$D_5$	$D_6$	$D_7$
0	1	1	0	0	1	1

Cette technique de codage permet non seulement de détecter une erreur mais également de la corriger. Nous pouvons aisément nous convaincre du fait qu'on peut détecter *une* erreur (c'est-à-dire le fait qu'*un* seul bit soit inversé), en constatant que chaque bit de donnée est vérifié par au moins un bit de parité parmi  $C_1, C_2, C_3$ . Or, nous savons déjà que la technique du bit de parité permet de détecter tout changement d'un seul bit.

En ce qui concerne la *correction* de l'erreur, supposons que celle-ci est unique. Nous avons alors deux cas à considérer :

- soit cette erreur a lieu sur un bit de donnée,  $D_j$ . Dans ce cas, *tous* les bits de contrôle qui vérifient  $D_j$  auront une valeur qui ne correspond pas à la donnée vérifiée. Il suffit dès lors de faire la somme des numéros de ces bits de vérification pour retrouver la valeur  $j$  (*cfr.* la première des deux propriétés énoncées ci-dessus).;
- soit cette erreur a lieu sur un bit de contrôle  $C_i$ . Dans ce cas, ce bit sera le seul à poser problème, et nous saurons donc qu'il n'y a pas d'erreur dans les données (sans quoi il y aurait au moins deux bits de contrôle problématiques, selon la seconde propriété énoncée ci-dessus).

## 2. Leçons 2 à 4 – Représentation de l'information



Continuons l'exemple ci-dessus et imaginons qu'il y ait une erreur sur le bit  $D_3$ , soit :

$C_1$	$C_2$	$D_3$	$C_4$	$D_5$	$D_6$	$D_7$
0	1	0	0	0	1	1

Re-faisons le calcul de chacun des bits de contrôle :

**Bit  $C_1$**  : le calcul de  $D_3 \text{XOR } D_5 \text{XOR } D_7$  donne 1, au lieu de 0;

**Bit  $C_2$**  : le calcul de  $D_3 \text{XOR } D_6 \text{XOR } D_7$  donne 0 au lieu de 1; et enfin

**Bit  $C_4$**  : le calcul de  $D_5 \text{XOR } D_6 \text{XOR } D_7$  donne 0, ce qui est bien la bonne valeur.

Il y a donc des problèmes avec les bits de contrôle  $C_1$  et  $C_2$ . On en déduit que le bit erroné est le bit  $D_j$  avec  $j = 1 + 2 = 3$ . On peut donc re-constituer la donnée d'origine en inversant le bit  $D_3$ .

Continuons toujours le même exemple et supposons qu'il y ait une erreur sur le bit de contrôle  $C_2$ , soit :

$C_1$	$C_2$	$D_3$	$C_4$	$D_5$	$D_6$	$D_7$
0	0	1	0	0	1	1

Re-faisons le calcul de chacun des bits de contrôle :

**Bit  $C_1$**  : le calcul de  $D_3 \text{XOR } D_5 \text{XOR } D_7$  donne 0, ce qui est bien la bonne valeur;

**Bit  $C_2$**  : le calcul de  $D_3 \text{XOR } D_6 \text{XOR } D_7$  donne 0 au lieu de 1; et enfin

**Bit  $C_4$**  : le calcul de  $D_5 \text{XOR } D_6 \text{XOR } D_7$  donne 0, ce qui est bien la bonne valeur.

Il y a donc un problème sur le bit  $C_2$  uniquement, c'est donc lui qui est erroné et la donnée est donc transmise correctement.

### 2.6.3. Applications des codes correcteurs d'erreur

Comme nous l'avons déjà évoqué, le bit de parité est une technique qui a largement été utilisée pour détecter des erreurs lors de la transmission de données en ASCII sur des lignes téléphonique ou des réseaux informatiques. Le code de HAMMING a initialement été développé pour détecter et corriger des erreurs lors de la lecture de cartes perforées.

On pourrait croire que les progrès technologiques ont diminué voire supprimé la nécessité des codes correcteurs d'erreur. Il n'en est rien : nous sommes encore aujourd'hui confrontés à une série d'applications où la transmission d'information fait l'objet d'erreurs. Prenons deux exemples :

1. La lecture d'un CD ou d'un DVD est un processus délicat, car le rayon LASER qui doit lire la surface du disque est très fin, et que le disque lui-même est souvent couvert de poussières, de griffes, voire de traces de doigts... De nombreuses erreurs de lecture ont donc lieu en permanence...
2. Avec l'âge des *smartphones*, une nouvelle famille de code barres, appelés codes matri-

## 2.7. Conclusion : sémantique d'une représentation binaire

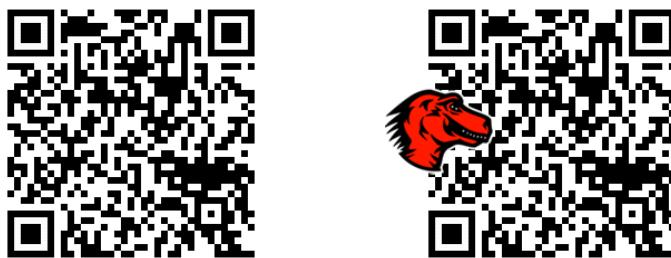


FIGURE 2.6. – Un exemple de code QR : à gauche, le code d'origine, à droite le code dont une partie a été masquée par Mozilla... Il est pourtant toujours possible de lire le message, car les QR codes utilisent les codes de REED et SALOMON pour détecter et corriger les erreurs.

ciels, ont fait leur apparition. Les plus célèbres sont les codes QR (la FIGURE 2.6 donne un exemple) qui ont été inventés dans les années 1990 au Japon pour simplifier la logistique de pièces automobiles, et ont depuis été standardisés [6]. À nouveau, les conditions de lecture de ces codes ne sont pas toujours idéales, et de nombreuses erreurs de lecture ont lieu (c'était d'ailleurs déjà le cas avec les code à barre « classiques » unidimensionnels que l'on retrouve aujourd'hui sur tous les emballages de produits manufacturés).

Ces deux exemples ont en commun la technique utilisée pour détecter et corriger les erreurs, à savoir les codes de REED<sup>19</sup> et SALOMON<sup>20</sup>, introduits en 1960 par ces deux auteurs [16]. Nous n'allons pas expliquer ici comment ils fonctionnent, cela nous emmènerait trop loin... mais ces deux exemples montrent que les codes détecteurs et correcteurs d'erreur restent importants. Ils font d'ailleurs encore l'objet d'une recherche active.

## 2.7. Conclusion : sémantique d'une représentation binaire

Cette section nous a convaincu que tous les types d'information manipulés par l'ordinateur peuvent être représentés uniquement à l'aide de 0 et de 1. Il est maintenant naturel de se poser la question *inverse*, à savoir : « quel est le sens à associer à une représentation binaire donnée? »



La représentation binaire 1000 0001 s'interprète comme  $129_{10}$  si on considère des nombres non-signés; comme  $-1_{10}$  si on considère le bit de signe; comme  $-127_{10}$  si on considère le complément à deux; comme le caractère « ü » si on considère qu'on a affaire à un caractère ASCII étendu...

19. Irving REED, né le 12 novembre 1923, et mort le 11 septembre 2012, est un mathématicien et ingénieur américain, qui a enseigné à l'*University of Southern California*, É-U d'Amérique.

20. Gustave SOLOMON, né le 27 octobre 1930 et mort le 31 janvier 1996 est un mathématicien et ingénieur américain, qui a travaillé pour la *Hughes Aircraft Company*.

## 2. Leçons 2 à 4 – Représentation de l'information

La réponse à cette question est que rien ne permet, *a priori* de décider quelle interprétation donner à une séquence binaire particulière. C'est le contexte, et en particulier le programme qui est exécuté sur ces données, qui leur donne un sens. Nous le prouvons à l'aide du programme C++ de la FIGURE 2.7. Il consiste à placer la même représentation binaire (sur 32 bits) dans trois variables *x*, *y* et *z* différentes, qui ont des *types* différents. Cela signifie que les instructions machines qui seront *in fine* exécutées pour afficher ces variables (instructions std::cout << à la fin du programme) seront différentes. Dans le cas la variable *x*, la représentation binaire sera interprétée comme un entier signé en complément à deux; dans le cas de la variable *y*, la représentation binaire sera interprétée comme un entier non-signé; et dans le cas de la variable *z*, la représentation binaire sera interprétée comme un caractère (les 24 bits de poids fort seront donc ignorés).

La sortie du programme confirme cela :

```
-2147483601  
2147483695  
/
```



## 2.7. Conclusion : sémantique d'une représentation binaire

```
1 #include <iostream>
2
3 int main() {
4     int x ; // x est un entier signé (complément à deux)
5     unsigned int y ; // y est un entier non négatif
6     char z ; // z est un caractère
7
8     // Toutes ces variables tiennent sur 32 bits
9
10    // On place la même valeur de 32 bits dans les 3 variables
11    // avec le bit de poids fort valant 1
12    // 1000 0000 0000 0000 0000 0010 1111
13
14    x = 0b10000000000000000000000000000000101111 ;
15    y = 0b10000000000000000000000000000000101111 ;
16    z = 0b10000000000000000000000000000000101111 ;
17
18    // L'affichage des trois valeurs est différent
19    // L'affichage tient compte du type: ce sont les
20    // instructions qui donnent leur sémantique aux
21    // valeurs représentées en binaire.
22
23    std::cout << x << std::endl ;
24
25    std::cout << y << std::endl ;
26
27    std::cout << z << std::endl ;
28
29 }
```

FIGURE 2.7. – Un programme C++ qui démontre qu'une même représentation binaire est interprétée de façon différente par différentes instructions.



## 3. Leçon 5 & 6 – Organisation de l'ordinateur

Maintenant que la manière dont l'ordinateur représente et manipule les informations nous est connue, nous pouvons repartir du modèle de l'architecture de VON NEUMANN (FIGURE 1.3) et étudier ses différents composants plus en détail. On se souviendra que les composants principaux de l'ordinateur sont :

1. le processeur;
2. la mémoire (primaire, principale);
3. les périphériques, notamment les périphériques d'entrée/sortie, mais également des périphériques de stockage à long terme des données.

Tous ces composants sont connectés entre eux de manière à pouvoir communiquer. Les canaux de communication entre ces composants, ainsi qu'entre les éléments constitutifs du CPU sont appelés des *bus*. Durant ce chapitre, nous allons donc raisonner sur la FIGURE 3.1, qui schématise cette organisation des composants de l'ordinateur.

### 3.1. Mémoire primaire

Nous allons commencer discuter de la mémoire primaire (ou mémoire principale, ou mémoire RAM). Il s'agit du composant de l'ordinateur où sont stockés les programmes en cours d'exécution et les données de ces programmes. Le CPU lit les instructions à exécuter depuis la mémoire primaire, et alimente les registres en données depuis la mémoire. Le contenu de la mémoire peut provenir de différentes sources, notamment des périphériques d'entrée, ou de la mémoire secondaire.

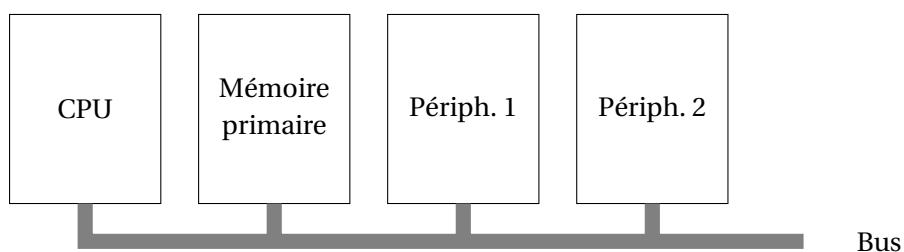


FIGURE 3.1. – Un modèle simple de l'organisation d'un ordinateur.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

#### 3.1.1. Structure et adresses

Indépendamment de la manière dont la mémoire est réalisée physiquement, elle a toujours la même structure : il s'agit d'une succession de *bits*, qui sont groupés par séquences de taille fixée, lesquelles sont stockées dans des *cellules*. Au cours de l'histoire, la taille des cellules a varié d'un ordinateur à l'autre, mais on a aujourd'hui adopté des cellules de 8 bits, ou 1 *octet* (*byte*).

Cette structure de mémoire peut être lue ou écrite : chaque *cellule* possède un identifiant propre qu'on appelle *adresse* et qui est un nombre naturel non-négatif. La première cellule a donc l'adresse 0, la seconde l'adresse 1, etc. La mémoire permet, soit de consulter le contenu d'une cellule à une adresse donnée, soit de stocker une information dans une cellule à une adresse donnée. On n'accède donc en général pas aux bits individuels.

Il arrive souvent que plusieurs cellules mémoires puissent être manipulées en même temps. En effet, chaque processeur admet une taille de données fixées qui est l'unité fondamentale d'information qu'il manipule. On appelle cela un *mot*, et tous les composants de l'ordinateur sont conçus en fonction de la taille d'un mot : les registres ont une capacité d'un mot, l'ALU peut traiter des données d'un mot, les bus peuvent transmettre des données d'un mot, et enfin, la mémoire peut lire et écrire des données d'un mot.

Dans le cas de la mémoire, cela signifie que l'adresse donnée lors d'une opération est celle de la première cellule où effectuer les opérations. Par exemple, avec des mots de 4 octets, écrire à l'adresse 20 signifie que les octets du mot seront écrits dans les cellules<sup>1</sup> 20, 21, 22 et 23.



Le contenu d'une mémoire est structuré en *cellules* ou *cases* de taille fixée (typiquement 8 bits) qui ont chacune une *adresse* unique. La mémoire permet deux types d'opération :

1. la *lecture* : on transmet à la mémoire une adresse, et la mémoire renvoie le mot stocké en mémoire à partir de cette adresse ; et
2. l'*écriture* : on transmet à la mémoire une adresse et une donnée (un mot), et la mémoire stocke la donnée dans les cellules à partir de l'adresse donnée.



Les mots du processeur i486 sont de 4 octets : c'est un processeur 32 bits. Dans la documentation technique de ce processeur, ces mots sont en fait appelés « double mots » (*Dwords*) car l'i486 est compatible avec les premiers processeurs Intel qui avaient des mots de 16 bits. L'appellation « mot » (*word*) a donc été conservée pour désigner les données de 16 bits.

---

1. En plaçant l'octet de poids fort soit dans la cellule 20 soit dans la cellule 23, *cfr.* la discussion sur l'ordre des octets dans les mots donnée plus loin.

### 3.1.2. Mémoire cache

Le principe de la mémoire cache est une famille de mécanismes permettant d'optimiser fortement la vitesse d'accès (par le CPU) aux données présentes en mémoire primaire. Il consiste à insérer, en petite quantité, des mémoires très rapides entre le CPU et la mémoire primaire qui est à la source des données<sup>2</sup>. Quand on a besoin de consulter une donnée à la source, on vérifie d'abord si celle-ci n'est pas présente dans la cache. Si c'est le cas (on parle de *cache hit*), on utilise la copie en cache, ce qui est plus rapide que d'accéder à la source. Si non (*cache miss*), on doit consulter la source, et on en profite pour stocker la donnée lue dans la mémoire cache, afin d'espérer profiter d'un *cache hit* si cette même donnée est consultée dans un futur proche. Pour assurer la rapidité de la mémoire cache, celle-ci est souvent (en tous cas, en partie) présente physiquement dans le CPU, et son accès ne requiert donc pas l'utilisation des bus externes.

Concrètement la mémoire cache peut être vue comme un tableau à deux colonnes : la première indique l'adresse de la case mémoire dont le contenu est copié dans la cache, et la seconde indique ce contenu. Par exemple :

Adresse	Contenu
:	:
00a4 5b11	$d_1$
1b3 2011	$d_2$
:	:

Dans ce cas, la consultation de l'adresse 00a4 5b11 donnera un *cache hit*, et on accèdera directement à la donnée  $d_1$ . La consultation d'une adresse n'apparaissant pas dans la première colonne, donne lieu à un *cache miss*, suite à quoi la donnée lue en mémoire est stockée dans la cache.

Ce mécanisme est efficace si de nombreux *cache hits* ont lieu. Or, la taille de la mémoire cache est forcément beaucoup plus petite que la mémoire source, il faut donc bien sélectionner les informations qu'on prélève dans la mémoire source pour les stocker en cache. Par ailleurs, quand la cache est pleine (toutes les lignes du tableau sont occupées) et qu'on veut y stocker une nouvelle information provenant de la mémoire source, il faut décider quelle information sacrifier.

En pratique ces mécanismes fonctionnent bien car les accès à la mémoire sont souvent localisés :

- quand on exécute une instruction d'un programme, il y a de fortes chances pour que l'instruction à exécuter ensuite soit l'instruction suivante en mémoire;
- beaucoup de programmes exécutent des boucles. Dans ce cas, les mêmes instructions sont répétées, et la cache améliorera grandement les performances si on arrive à garder les instructions de la boucle en cache tout au long de son exécution;

---

2. Ce principe peut se généraliser à tous les cas où on souhaite consommer des données depuis une source plus lente.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

- un programme manipulant un tableau accédera souvent de manière séquentielle aux éléments de ce tableau. Comme dans le cas de la boucle, la cache peut donc se révéler efficace si le programme accède plusieurs fois de suite au même tableau, qui a pu être maintenu en cache.

L'efficacité du cache est encore accrue par des algorithmes complexes de *prédiction* qui tentent de prédire quelle sera la prochaine instruction ou la prochaine donnée à laquelle on accèdera, de manière à pouvoir la charger dans la mémoire cache avant même que le CPU ne la demande.

Les processeurs modernes comprennent souvent 3 niveaux de caches, appelés L1, L2, L3 :

1. la cache L1 est le plus proche du CPU. Elle est la plus rapide et de très petite taille (quelques dizaines de KO) ;
2. en cas de *cache miss* dans L1, on consulte la cache L2 qui est plus lente mais plus grande (quelques centaines de KO), ce qui augmente donc la probabilité d'un *cache hit*; enfin
3. en cas de *cache miss* dans L2, on consulte la cache L3 qui est encore plus lente mais encore plus grande (quelques MO).



Le processeur i486 possède un seul niveau de cache (L1) de 8 kibioctets, présent directement sur le circuit intégré du processeur. Il peut contenir tant des données que des instructions. On voit cette cache sur la FIGURE 1.4, connectée aux composants servant à communiquer avec la mémoire primaire.

#### 3.1.3. Ordre des octets dans les mots

Quand on stocke en mémoire une information (un mot) qui occupe un espace supérieur à la taille d'une case mémoire, il faut décider dans quel ordre les octets individuels de cette information seront stockés. Par exemple, supposons une machine ayant des cellules mémoire d'un octet, et supposons qu'on manipule des mots sur 16 bits (dans le cas d'un CPU 16 bits, par exemple). Le nombre  $517_{10}$  est représenté en binaire (sur 2 octets) par 00000001 00000101. Si on veut stocker ce mot à partir de l'adresse  $a$ , on peut donc décider :

- soit de stocker les octets constitutifs en partant de l'octet qui contient le bit de poids fort. On stocke donc 00000001 à l'adresse  $a$  et 00000101 à l'adresse  $a + 1$ . Les machines qui procèdent en stocker le byte contenant le bit de poids fort d'abord sont appelées « gros boutistes » ou *big endian*.
- soit de faire l'inverse, et stocker 00000001 à l'adresse  $a + 1$  et 00000101 à l'adresse  $a$ . On parle alors machine « petit boutiste » ou *little endian*.

Il n'y a *a priori* pas de meilleur choix, le choix d'une politique gros ou petit boutiste est un choix arbitraire dépendant du processeur utilisé. Cela peut naturellement poser des problèmes lors de transmissions de données d'une machine à l'autre. On peut imaginer des procédures pour remettre les octets dans l'ordre, mais celles-ci doivent tenir compte du type de données représentée. Par exemple, considérons une machine 32 bits. Elle peut stocker

### 3.1. Mémoire primaire

<b>Petit boutiste</b>			
entier 1 524 852			
01110100	01000100	00010111	00000000
Chaîne ABCD			
A	B	C	D
01000001	01000010	01000011	01000100

<b>Gros boutiste</b>			
entier 1 524 852			
00000000	00010111	01000100	01110100
Chaîne ABCD			
A	B	C	D
01000001	01000010	01000011	01000100

FIGURE 3.2. – Petit et gros boutistes.

des entiers sur 32 bits (4 octets) ou bien des caractères, chacun sur un octet (code ASCII par exemple), ce qui implique qu'un mot de 32 bits contiendra 4 caractères. Dans ce cas, il n'est pas question d'inverser l'ordre des octets qui contiennent des caractères, alors que ce sera nécessaire pour les mots qui contiennent des nombres. La FIGURE 3.2 illustre cela.



Le processeur i486 est un processeur *petit boutiste*. Quand on indique une adresse  $a$  (pour la lecture ou l'écriture d'un mot en mémoire), on indique donc l'adresse de l'octet de poids faible, et les octets de poids plus fort sont stockés en  $a + 1$ ,  $a + 2, \dots$

Notons que les termes « petit boutiste » (*little endian*) et « gros boutiste » (*big endian*) proviennent des célèbres *Voyages de Gulliver* [19] de Jonathan Swift (1667 – † 1745). Ce livre est une satire sociale, dans lequel deux peuples s'affrontent dans des guerres sanglantes, sous prétexte de savoir s'il convient de manger les œufs à la coque en les brisant du petit ou du gros côté [19, Chapitre IV] :

[...] two mighty powers have, as I was going to tell you, been engaged in a most obstinate war for six-and-thirty moons past. It began upon the following occasion : It is allowed on all hands, that the primitive way of breaking eggs, before we eat them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the emperor, his father, published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us, there have been six rebellions raised on that account, wherein one emperor lost his life, and another his crown. [...] It is computed, that eleven thousand persons have, at several times, suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have been published upon this contro-

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

*versy, but the books of the Big-endians have been long forbidden, and the whole party rendered incapable, by law, of holding employments. [...]*

#### 3.1.4. Réalisation de la mémoire primaire

Au cours de l'histoire plusieurs solutions techniques ont été proposées pour réaliser des mémoires primaires. La solution actuelle utilise des transistors intégrés pour réaliser les circuits logiques que nous décrirons dans le Chapitre 4, mais ce n'a pas toujours été le cas. Voici quelques-unes des solutions « historiques » :

**Les tubes de Williams** Ce sont des dispositifs (voir FIGURE 3.3) qui ressemblent aux écrans des anciennes télévision (type CRT) : un couche de phosphore est placée au fond d'un tube cathodique, qu'un canon à électrons vient balayer. Les données sont stockées comme des charges électrostatiques sur la couche de phosphore : en traçant, comme sur un écran de télévision, des points et des traits sur le fond du tube. Ces données restent présentes sur le tube une fraction de seconde, ce qui permet de tracer d'autres points. Les données sont lues à l'aide d'une plaque métallique placée face au fond du tube. Comme les données ne restent en place qu'une fraction de seconde, le dispositif doit continuellement lire les données et les ré-inscrire (éventuellement avec une modification en cas d'écriture).

**Les mémoires à lignes à délai** Dans ces systèmes, les informations étaient représentées sous forme de pulsations électriques qu'on émettait à un bout d'un tube de mercure ou d'un très long câble, et qu'on lisait à l'autre bout. Comme la pulsation électrique mettait un certain temps à travers le medium, on pouvait envoyer plusieurs pulsations différentes (et donc une séquence de bits) l'une à la suite de l'autre. Une fois arrivée au bout du tube, la pulsation était lue, amplifiée, et réinjectée au début. L'information circulait donc en boucle. Pour lire un bit, il suffisait d'attendre que la pulsation correspondante arrive au bout du tube. Pour écrire, il suffisait d'injecter la nouvelle information au lieu de recopier l'ancienne. La FIGURE 3.4 montre deux types de mémoires à lignes à délai.

**Les mémoires à tores magnétiques** consistent en un treillis de fils électriques passant à travers des petits anneaux (tores) de ferrite. Chaque tore peut mémoriser (pour plusieurs années) un bit sous forme d'un champ magnétique. La direction du champ magnétique dans le tore indique l'information : 0 dans un sens, 1 dans l'autre. Chaque tore est traversé par 3 fils : les fils *X* et *Y* correspondent aux coordonnées du tore (un paire de fils *X* et *Y* correspond à un et un seul tore), et le fil *sense* qui est plié de manière à traverser tous les tores. Pour pouvoir changer le sens du champ magnétique dans un tore, il faut faire circuler un courant électrique suffisamment fort dans les fils *X* et *Y* correspondant. La direction des courants dans *X* et *Y* donne la direction du champ magnétique et donc l'information stockée. C'est aussi sur ce principe qu'on effectue la lecture : on force le tore qui doit être lu à passer à 0, en appliquant le courant nécessaire sur ses lignes *X* et *Y*. Si ce tore contenait 1, une pulsation électrique apparaît sur la ligne *sense*. Autrement, le tore stockait 0. La lecture est destructive, et il faut donc re-stocker le 1 si nécessaire.

### *3.1. Mémoire primaire*

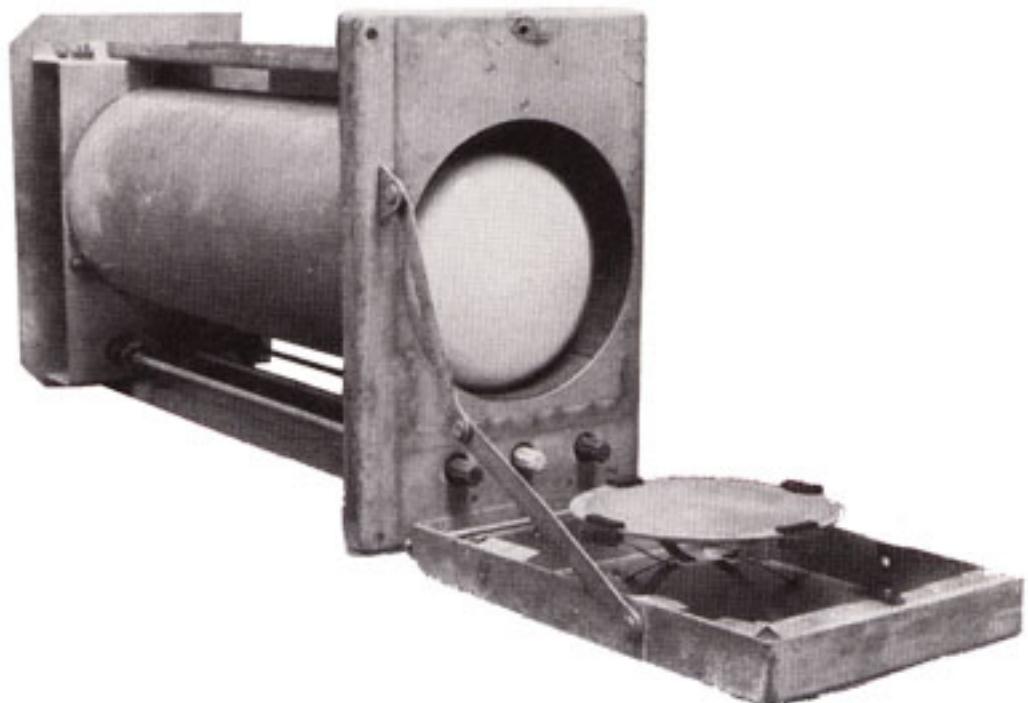


FIGURE 3.3. – Un tube Williams.

Source : Sk2k52 (<https://commons.wikimedia.org/wiki/File:Williams-tube.jpg>, <https://creativecommons.org/licenses/by-sa/4.0/legalcode>).

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

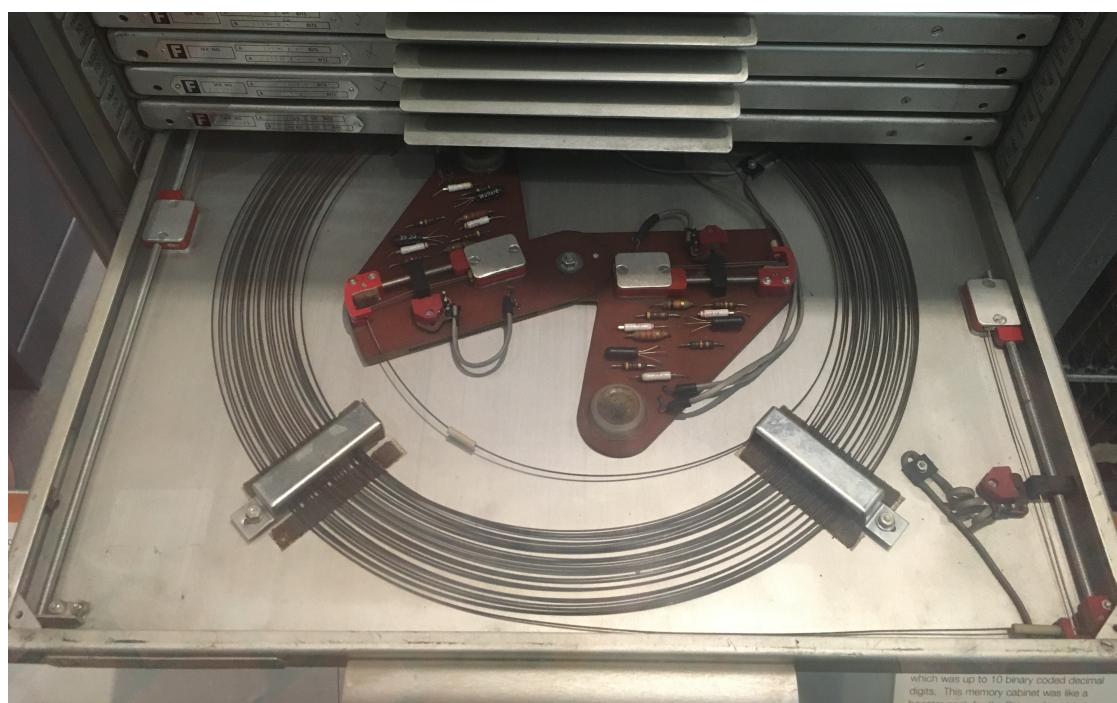
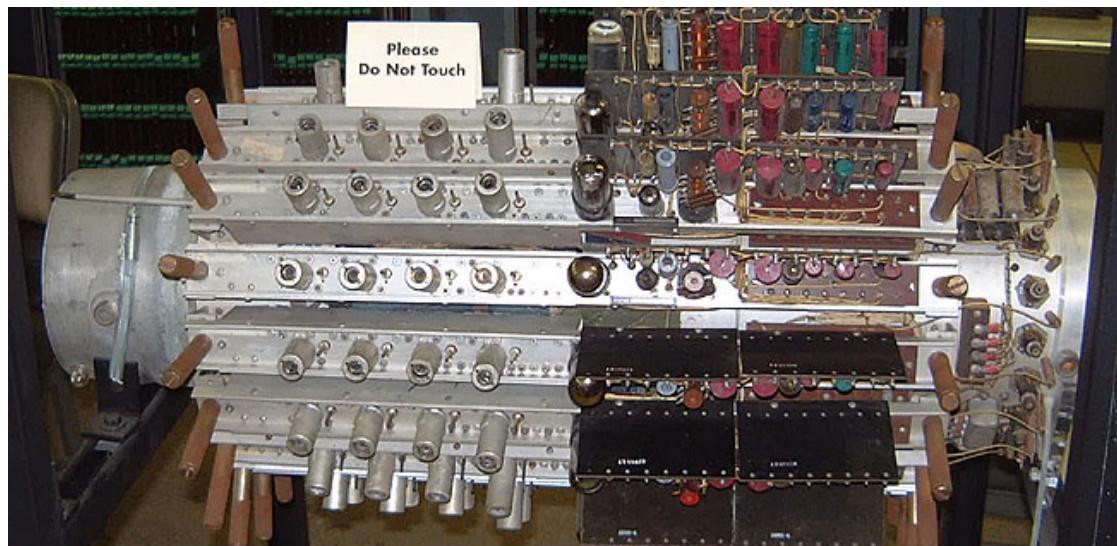


FIGURE 3.4. – Deux types de mémoires à ligne à délai. Au-dessus : ligne de délai à mercure de l'Univac 1, 1951. Dimensions approximatives : 4m × 2m × 2m. Contient 18 canaux de mercure pouvant contenir chacun 10 mots de 12 bits, soit 270 octets. En-dessous : ligne à délai en fil de nickel du Ferranti Sirius au *Computing Museum* de l'Université Monash (Melbourne, Australie). Capacité de stockage : 50 mots, soit environ 125 octets.

Source : Dessus : Ed Thelen ([https://commons.wikimedia.org/wiki/File:Mercury\\_memory.jpg](https://commons.wikimedia.org/wiki/File:Mercury_memory.jpg)). « Mercury memory », <https://creativecommons.org/licenses/by-sa/3.0/legalcode>. Dessous : Photo de l'auteur.

### 3.1. Mémoire primaire

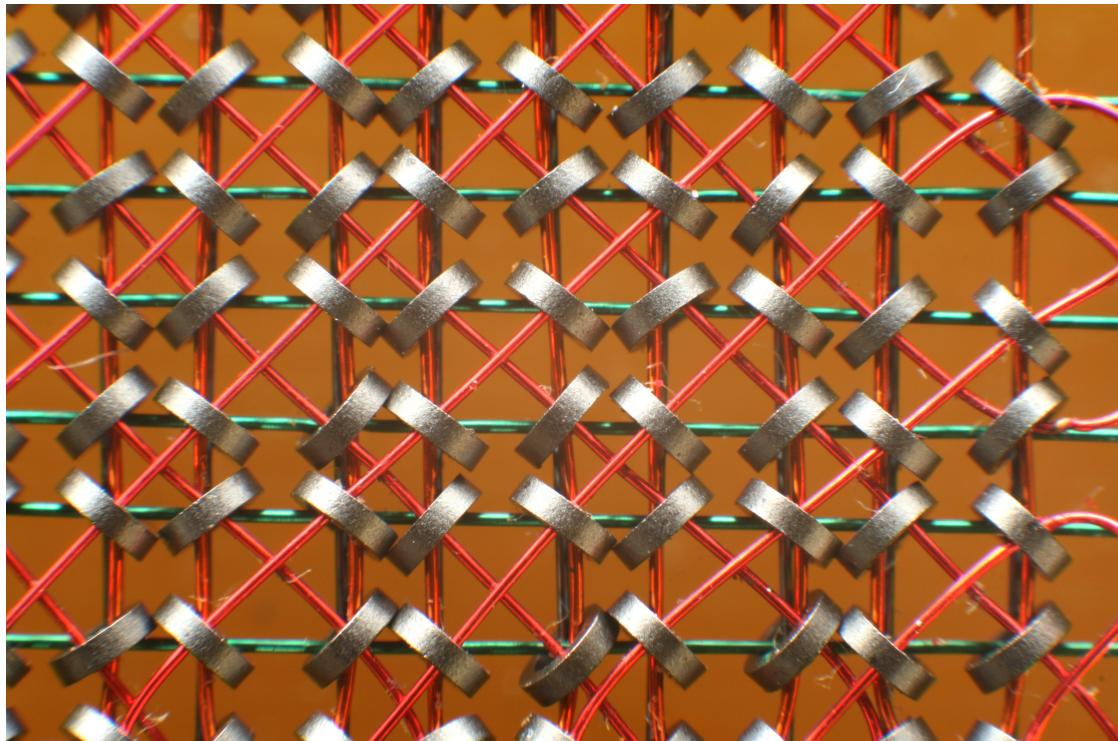


FIGURE 3.5. – Détail d'une mémoire à tores de ferrite.

Source : Konstantin Lanzet ([https://commons.wikimedia.org/wiki/File:KL\\_Kernspeicher\\_Makro\\_1.jpg](https://commons.wikimedia.org/wiki/File:KL_Kernspeicher_Makro_1.jpg)), « KL Kernspeicher Makro 1 », <https://creativecommons.org/licenses/by-sa/3.0/legalcode>.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

Les tores de ferrite ont, à leur époque, révolutionné l'informatique car ils étaient beaucoup plus compacts et pratiques que les autres solutions basées sur des lignes à délai. Par exemple, IBM introduit en 1954 la 737, unité de stockage à base de tore de ferrites pour les machines de la série 700. Ces armoires de plusieurs mètres de large peuvent stocker 4096 mots de 36 bits<sup>3</sup>.

## 3.2. Le processeur

Maintenant que nous avons étudié la mémoire, nous pouvons nous intéresser au composant principal de l'ordinateur : bien évidemment, le processeur.



Le processeur est le composant principal de l'ordinateur. Il exécute (interprète) les programmes en langage machine, qui sont stockés dans la mémoire primaire. Il se compose principalement de l'unité de contrôle, de l'ALU et des registres.

### 3.2.1. Composants du processeur

Rappelons rapidement ce que sont les différents composants du processeur :

- **L'unité de contrôle** se charge d'obtenir les instructions dans la mémoire principale, et commande l'ALU et les registres pour s'assurer que l'instruction est correctement exécutée.
- **L'unité arithmético-logique** (ALU) se charge d'effectuer les opérations arithmétiques et logiques nécessaires à la bonne exécution de l'instruction. Elle obtient les données depuis les registres, et y place également les résultats de ses calculs.
- **Les registres** sont des petites mémoires de travail locales au CPU (c'est-à-dire qu'elles se trouvent physiquement sur le même circuit que le CPU). Le CPU y accède donc plus rapidement qu'à la mémoire primaire. Les instructions machines sont en général exécutées sur les données qui sont dans les registres.

On distingue deux types de registres : les registres *de contrôle*, qui servent à assurer le bon fonctionnement du CPU, et les registres *de travail*, qui contiennent des données.

Parmi les registres de contrôle, on distingue deux registres importants :

- le *pointeur d'instruction*, que nous appellerons ici PC (pour *program counter*), et qui servira à retrouver en mémoire la prochaine instruction à exécuter en donnant son *adresse*; et
- le *registre d'instruction*, que nous appellerons ici IR (pour *instruction register*), et dans lequel nous stockerons la prochaine instruction à exécuter, en vue de son décodage.

Les autres registres de contrôle sont généralement des registres d'état (qui donnent des informations sur l'état de fonctionnement de la machine).

3. Voir : [http://www-03.ibm.com/ibm/history/exhibits/701/701\\_1415bx37.html](http://www-03.ibm.com/ibm/history/exhibits/701/701_1415bx37.html).

Généralement, les registres ont tous la même taille, qui est celle d'un mot. On parlera par exemple de « machine 32 bits » pour désigner un ordinateur dont les registres (de travail) ont 32 bits. Cette taille constitue une caractéristique fondamentale du CPU car :

1. elle permet de savoir la quantité d'information qui peut être traitée en une instruction;
2. le registre d'instruction étant lui aussi de taille fixée, le format des instructions est limité;
3. cette taille s'applique également au registre PC<sup>4</sup>. S'il a une capacité de  $n$  bits, on ne peut différencier que  $2^n$  adresses différentes. La quantité d'information à laquelle on peut accéder via le registre PC est donc de  $2^n \times c$ , où  $c$  est le nombre de bits d'une cellule (on parle de la quantité de mémoire *adressable* car c'est la portion de la mémoire à laquelle on peut assigner une adresse qu'on pourra manipuler dans les registres du processeur). Cela limite donc la taille des programmes que le processeur peut exécuter. En général, cela limite également la taille utile de la mémoire car tous les composants qui communiquent avec la mémoire (registres contenant des adresses et bus) sont limités à cette même taille.

Par exemple, sur les machines 32 bits avec des cellules d'un octet, PC fait 32 bits, et on est donc limité à  $2^{32}$  octets, soit 4 Go. Toute mémoire additionnelle ajoutée à l'ordinateur serait parfaitement inutile, étant donné que le processeur ne pourrait manipuler les adresses de ces cases mémoires supplémentaires.



Comme nous l'avons déjà dit, le processeur i486 est un processeur 32 bits, qui possède 32 registres [9]. Ils se répartissent comme suit :

- les 8 registres de travail de 32 bits `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` et `esp`;
- 1 registre d'état de 32 bits appelé `eip` (pour *instruction pointer*) qui fait office de registre PC;
- 1 *registre d'état* de 32 bits (dont seuls 19 bits sont utilisés), appelé `EFlags`, dont les bits individuels (appelés *flags*) permettent d'avoir de l'information sur l'état du processeur. Par exemple, le huitième bit, appelé `S` indique si le résultat de la dernière opération effectuée par le CPU est une valeur négative;
- 8 registres de 80 bits appelés `st0`, ..., `st7` pour réaliser des calculs sur des nombres en virgule flottante selon la norme IEEE754 (voir Section 2.2.5), à l'aide du FPU. Les mots du FPU de l'i486 sont donc de 80 bits et non pas de 32 bits;
- 4 registres de contrôle (dont seuls 3 sont utilisés), utilisés pour la gestion de la mémoire (voir Chapitre 9); et enfin
- 10 registres supplémentaires qui sont utilisés pour la gestion de la mémoire segmentée (nous n'entrerons pas dans les détails ici).

---

4. Et aux éventuels autres registres contenant des adresses pour la lecture ou l'écriture en mémoire.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

```
1 mov eax, 42 ; place 42 dans eax
2 mov ecx, 7 ; place 7 dans ecx
3 add eax, ecx ; eax reçoit eax + ecx
```

FIGURE 3.6. – Un exemple de programme en langage machine i486.



Regardons maintenant un exemple de petit programme en langage machine, et voyons comment il utilise les registres de travail. Voici un exemple de programme machine pour l'i486, il comporte 3 instructions, toutes les trois données en binaire. Les deux premières tiennent sur 40 bits :

```
1011 1000 0000 0000 0000 0000 0000 0000 0010 1010
1011 1001 0000 0000 0000 0000 0000 0000 0000 0111
0000 0001 1100 0001
```

La première instruction commence par l'*opcode* 1011 qui indique de placer une valeur de 32 bit dans un registre. Le registre est spécifié par les trois bits suivants : c'est **eax** (code binaire 000). Ensuite, viennent les 32 bits qui spécifient la valeur 42. La seconde instruction fait de même en plaçant la valeur 7 dans **ecx** (code binaire 001). Enfin, la troisième indique au processeur de faire la somme (*opcode* 0000 0001 11), du registre **eax** (donné en binaire par 000) et du registre **ecx** (donné en binaire par 001). L'instruction de somme place le résultat dans le premier des deux registres, à savoir **eax**.

Ces instructions sont assez difficiles à lire telles quelles, c'est pourquoi nous adopterons une syntaxe plus lisible<sup>5</sup>, plus proche d'un langage de programmation usuelle : voir FIGURE 3.6.

Notons qu'au lieu de charger des valeurs fixées dans les registres, on aurait tout aussi bien pu charger des valeurs depuis la mémoire, en spécifiant leurs adresses.

Nous allons maintenant nous concentrer sur l'ALU et les registres qui permettent l'exécution des instructions machine.

#### 3.2.2. Le chemin des données – *datapath*

À l'intérieur d'un CPU, l'ALU est connectée aux registres via des *bus* car ce sont eux qui l'alimentent en données et qui recueillent ses résultats. Les données se propagent donc dans le CPU de manière cyclique : elles partent des registres, sont traitées dans l'ALU, et le résultat retourne aux registres.



L'ensemble constitué de l'ALU, des registres et des bus d'interconnection est appelé le *chemin des données* (ou *datapath* en anglais).

5. Qui est en fait celle de l'assembleur.

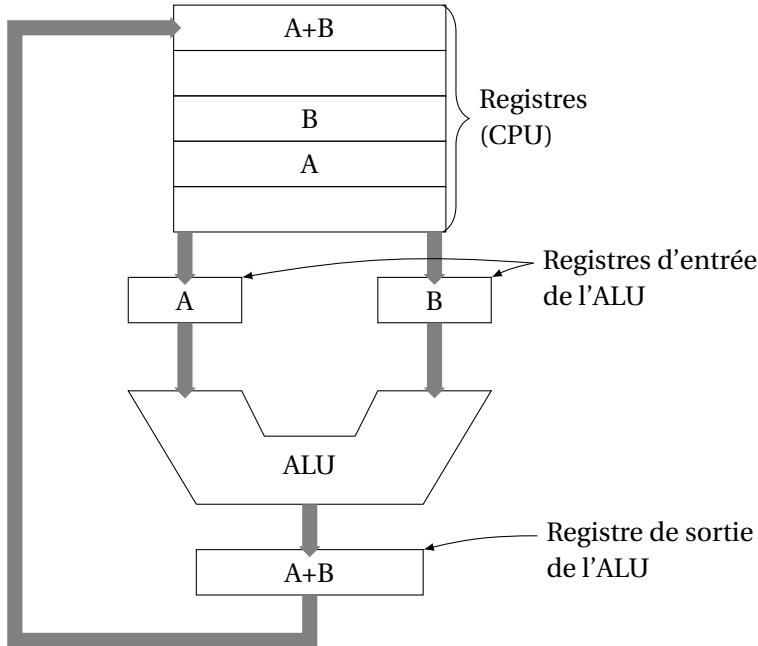


FIGURE 3.7. – Le chemin des données. Les flèches grises symbolisent les bus (internes au CPU).

Le chemin des données est symbolisé à la FIGURE 3.7. On observe en particulier que l’ALU possède ses propres registres d’entrée et de sortie qui sont alimentés et alimentent les registres de travail du CPU.

### 3.2.3. Exécution des instructions

Comme nous l’avons déjà expliqué dans le Chapitre 1, le CPU exécute continuellement la boucle appelée *fetch-decode-execute*, que nous pouvons maintenant donner de façon plus détaillée<sup>6</sup>, en expliquant quels sont les registres qui interviennent dans cette boucle et comment ils sont utilisés.

---

6. On pourra utiliser avec bonheur le simulateur de Dave Reed pour illustrer et manipuler ces notions de manière un peu plus pratique. On y accède à l’adresse : <http://www.dave-reed.com/book/Chapter14/>.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur



La *boucle d'interprétation du processeur* (ou boucle *fetch-decode-execute*) est la boucle exécutée en permanence par le processeur pour interpréter les instructions machine :

1. Charger, dans le registre IR, l'instruction située en mémoire (M) à l'adresse donnée par PC :  $IR \leftarrow M[PC]$
2. Incrémenter PC :  $PC \leftarrow PC + 1$
3. Analyser l'instruction dans IR
4. Exécuter l'instruction (ceci peut modifier PC)
5. Aller en 1.

La ligne 4 peut être vue comme une traduction de l'instruction machine (de niveau 2) en une instruction ou une série d'instructions du niveau inférieur (micro-instructions, voir FIGURE 1.5).

#### 3.2.4. Machine à pile

Jusqu'à présent, nous avons toujours décrit le fonctionnement du processeur comme une machine *à registres* : comme nous l'avons vu dans le cadre du *datapath*, les données à traiter sont lues *dans les registres*, et les résultats sont inscrits également *dans les registres*. Il existe néanmoins un modèle différent, dont nous reparlerons plus largement dans le Chapitre 5 : celui de la *machine à pile*.

Une *machine à pile* ne possède pas de série de registres qui peuvent tous stocker les données pour être traitées par le langage machine<sup>7</sup>. Les données à traiter ainsi que les résultats des opérations proviennent tous d'une *pile*, qui est une structure de données particulière.

**La pile comme structure de données** Commençons par décrire, de manière abstraite, le fonctionnement d'un pile (ou *stack* en anglais), en supposant que nous souhaitons stocker des valeurs entières uniquement sur notre pile. Intuitivement, une pile ressemble à une pile d'assiettes dans la vie réelle : on y empile les données, il y a donc une donnée au sommet, une en bas, et toutes les autres suivent un ordre donné par la pile.

Comme avec une véritable pile d'assiettes, on ne peut accéder qu'à son sommet<sup>8</sup>. Pour notre structure de donnée, cela signifie que seule la donnée au sommet peut être lue, à l'aide d'une opération appelée *Top*. Ensuite, la pile peut être modifiée, mais toujours *via* son sommet : un ajout de données, se fait nécessairement au somme de la pile (pas d'insertion « au milieu »), via l'opération appelée *Push*, et la suppression de données se fait en prélevant la donnée au sommet, via l'opération *Pop*.

7. Par exemple, sur l'i486, les données à additionner lors d'une instruction `add` peuvent être dans n'importe quel registre général : `eax`, `ebx`, ...

8. Nous tous déjà essayé de prendre une assiette au milieu d'une pile, et nous savons tous quels sont les risques, n'est-ce pas ?

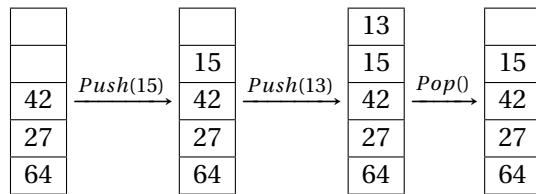


FIGURE 3.8. – Un exemple de pile.



La FIGURE 3.8 montre un exemple de pile avec trois manipulations. Dans son premier état, la pile contient trois données : 64, 27 et 42 (de bas en haut). L'appel à *Top* sur cette pile donnerait donc 42. Ensuite, un *Push* de la donnée 15, ajoute celle-ci au sommet de la pile (et on a donc *Top* = 15 dans ce cas). Un second *Push*, de la valeur 13, étend encore le contenu de la pile. Finalement, le dernier *Pop* supprime la dernière valeur insérée.

Une pile est parfois appelée un LIFO pour *last in, first out*<sup>9</sup>, car c'est toujours la dernière donnée insérée qui est lue en premier.

**Implémentation d'une pile** Concrètement, comment réalise-t-on cette structure abstraite ? Une possibilité consiste à stocker la pile en mémoire, en plaçant l'élément en bas de la pile à une certaine adresse  $a$ , puis celui juste au-dessus à l'adresse  $a + 1$ , etc. Pour accéder à la pile, il suffit alors de retenir l'adresse  $s$  du sommet de la pile. On peut alors : (1) faire une lecture en mémoire à l'adresse  $s$  pour réaliser un *Top*; (2) écrire à l'adresse  $s + 1$  pour faire un *Push*, et incrémenter  $s$ ; (3) décrémerter  $s$  pour réaliser un *Pop*. Il est naturellement possible de faire l'inverse : de stocker le fond de la pile à une adresse  $a$ , l'élément au-dessus à l'adresse  $a + 1$ , etc.

Une autre possibilité est d'utiliser un ensemble de registres, numérotés pour stocker les différentes cases de la pile (par exemple le registre numéro 0 contiendra le bas de la pile, le registre 1 la donnée au-dessus, etc). Comme dans le cas d'un stockage en mémoire, on retiendra la numéro du registre qui contient le sommet de la pile.

**Utilisation dans le langage machine** Comment la pile est-elle utilisée par les instructions machines ? Il suffit de faire des *Push* des opérandes sur la pile, puis d'exécuter l'instruction (en général, sans opérande), qui placera le résultat sur la pile également.

Par exemple, pour une opération d'addition, on pourrait : (1) faire un *Push* des deux valeurs à sommer sur la pile; puis (2) appeler l'instruction d'addition, ce qui aura pour effet de faire un *Pop* des deux opérandes, puis un *Push* de leur somme.

9. À opposer à FIFO, signifiant *first in, first out*, qui définit une file d'attente comme à la caisse des supermarchés.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

```
1 fldpi ; Charge pi au sommet de la pile
2 fldl2t ; Charge log2(10) au sommet de la pile
3 fadd ; Réalise la somme et place le résultat au sommet
4 ; on aurait aussi pu écrire fadd st0, st1:
5 ; réaliser la somme de st0 (le sommet) et
6 ; st1 et placer le résultat dans st0.
```

FIGURE 3.9. – Un exemple de langage machine i486 utilisant le FPU.



Bien que l'i486 soit essentiellement une machine à *registre* comme nous l'avons décrit jusqu'à présent, ce processeur se comporte comme une machine à pile quand on effectue des opérations en virgule flottante, à l'aide du FPU.

L'i486 possède 8 registres de 80 bits utilisés pour contenir la pile sur laquelle agissent les instructions en virgule flottante. Le FPU de l'i486 possède aussi un registre appelé STATUS, de 16 bits, dont les bits 13, 12 et 11 contiennent, en binaire, le numéro du registre qui constitue le sommet du stack. En pratique, on se réfère au registre qui est au sommet de la pile par st0, à celui juste en-dessous par st1, etc.

La FIGURE 3.9 présente un exemple de traitement qu'on peut réaliser à l'aide d'instructions en virgule flottante sur l'i486 et qui illustre l'usage de la pile.

#### 3.2.5. Choix du jeu d'instructions machine

La boucle *fetch-decode-execute* permet une grande flexibilité, et autorise les concepteurs de microprocesseurs à choisir librement les instructions qu'ils souhaitent voir apparaître dans le langage machine de leur processeur. Naturellement, plus le langage machine est riche, plus confortable sera la tâche des programmeurs (qui pourront utiliser directement une instruction machine pour réaliser telle ou telle tâche au lieu de devoir écrire un morceau de programme qui réalise la même chose), mais plus ardue sera la tâche des concepteurs du processeur.

Par exemple, le processeur i486 possède des instructions comme **fadd** ou **fmul** pour effectuer des opérations sur des valeurs en virgule flottante (à l'aide du FPU, ce qui est très efficace). Mais rien n'empêche de réaliser le même traitement en écrivant un petit programme n'utilisant que les instructions du processeur manipulant des entiers (**add**, **mul**, etc).

La question de savoir *que mettre* dans un langage machine est donc un vieux débat. De manière générale, les fabricants de processeurs essaient de faire en sorte que des *familles* de processeurs soient compatibles entre eux, de manière à ce que les programmes écrits pour les anciens processeurs soient toujours exécutables sur les nouveaux. Par exemple, l'i486 est capable (tout comme ses descendants actuels) d'exécuter tous les programmes écrits pour son lointain ancêtre l'Intel 8086 (datant de... 1977!) Cela introduit donc une contrainte forte

### 3.2. Le processeur

sur le jeu d'instruction et sur la conception des processeurs comme le choix des registres<sup>10</sup>. Ces contraintes qui sont communes à une famille de processeurs et qui assurent un certain niveau de comptabilité au cours de temps sont appelés une *architecture*, nous en reparlerons dans le Chapitre 6. Dans le cas des processeurs Intel, par exemple, on parle de l'architecture x86.

L'introduction de la notion de micro-code vers 1951 par Maurice WILKES<sup>11</sup> a grandement simplifié la conception des microprocesseurs, et a eu pour effet un peu inattendu l'explosion des jeux d'instructions des machines. Dans les années '70, on rencontrait des machines avec des centaines d'instructions<sup>12</sup>... Dans les années 1980, plusieurs chercheurs dont David PATERSON<sup>13</sup> [14] ont estimé que ces processeurs devaient trop complexes, et on commencé à plaider pour une simplification des processeurs, permettant des architectures plus simples<sup>14</sup>. Ces architectures sont désignées sous le nom de RISC pour *Reduced Instruction Set Computing*, par apposition aux CISC (*Complex Instruction Set Computing*). Les concepteurs de RISC plaident pour des processeurs sans micro-code, où le nombre d'exécution d'instructions par seconde est maximisé (au lieu de tenter d'optimiser chaque instruction), et avec de nombreux registres.

Parmi les processeurs RISC, on trouve :

- Les processeurs SPARC, utilisés dans les stations de travail Sun.
- Les processeurs MIPS, utilisés notamment dans les stations de travail Silicon Graphics, dans certains modèles de la PlayStation... Sur ces processeurs, par exemple, il n'y a que trois types d'instructions :
  - Les instructions *Load/Store* qui copient des valeurs entre les registres et la mémoire.
  - Les instructions arithmétiques, qui opèrent sur les registres uniquement (pas d'accès mémoire).



Sir Maurice WILKES en 1980.

Source : Unknown – University of Cambridge Computer Laboratory Archive ([https://commons.wikimedia.org/wiki/File:Maurice\\_Vincent\\_Wilkes\\_1980\\_\(3\).jpg](https://commons.wikimedia.org/wiki/File:Maurice_Vincent_Wilkes_1980_(3).jpg)), « Maurice Vincent Wilkes 1980 (3) », <https://creativecommons.org/licenses/by/2.0/uk/deed.en>

10. Ainsi les 2 octets de poids faible des registres `eax`, `ebx`, `ecx` et `edx` peuvent être vus comme des registres de 16 bits appelés `ax`, ..., `dx`, ce qui correspond aux noms des registres 16 bits originels du 8086. Le `e` dans les noms `eax`, etc est en fait l'abréviation d'*extended*.

11. Né le 26 juin 1913 et mort le 29 novembre 2010, informaticien anglais, professeur à l'Université de Cambridge, et récipiendaire du Prix Turing, il a largement contribué à la conception de l'EDSAC, un des premiers ordinateurs.

12. Le VAX, par exemple. Voir [24] pour un guide de référence de l'architecture.

13. Né le 16 novembre 1947, informaticien américain, professeur à l'Université de Berkeley (Californie, É.-U. d'Amérique) et récipiendaire du Prix Turing en 2017.

14. L'instruction INDEX du VAX, par exemple, était plus lente qu'une implémentation « à la main » du même traitement...

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

- Les instructions de saut/branchement.

Les processeurs Intel de la famille x86, dont l'i486, par contre sont des processeurs CISC.

#### 3.2.6. Techniques pour améliorer l'efficacité des processeurs

L'efficacité des processeurs est évidemment un des facteurs clef pour assurer les performances des ordinateurs modernes, qui doivent traiter de grandes quantités d'information en peu de temps. Une des caractéristiques essentielles d'un processeur est sa *fréquence*, c'est-à-dire le nombre de cycles de traitements qu'il peut effectuer par seconde<sup>15</sup>. Si les premiers microprocesseurs fonctionnaient à environ 1 Mhz, les processeurs modernes atteignent quelques GHz, ce qui les rend, sur ce plan-là, environ 1000 fois plus rapides. Mais l'augmentation de la fréquence entraîne également une augmentation de la chaleur dissipée par les processeurs, ce qui entraîne de gros problèmes d'ingénierie. D'autres techniques ont donc été développées pour augmenter l'efficacité des processeurs.

**Exécution en parallèle des instructions** Il y a différentes façons de découper l'exécution d'une instruction en étapes successives (nous en avons vue une lors de la boucle *fetch-decode-execute*). Une façon de faire est la suivante :

1. lecture en mémoire;
2. décodage;
3. lecture des opérandes;
4. exécution;
5. écriture du résultat.

On peut concevoir un processeur dans lequel chacune de ces étapes est réalisée par un module indépendant, chaque module étant connecté au suivant. Cette architecture, illustrée à la FIGURE 3.10 est appelé un *pipeline* (dans ce cas, à 5 étages), et l'exécution d'une instruction consiste donc à la faire « rentrer » à gauche du *pipeline* et à la faire transiter jusqu'à la sortie à sa droite.

Comme le *pipeline* est purement séquentiel, il n'y a, pour une instruction donnée, qu'un seul de ses éléments qui est actif à chaque instant. On peut donc rendre le processeur plus efficace, en faisant fonctionner les différents étages du *pipeline* en parallèle, sur des instructions différentes : une fois qu'une première instruction a été lue en mémoire, on peut lancer la lecture de l'instruction suivante durant le décodage de la première, et ainsi de suite.

Évidemment, il existe parfois des dépendances entre les instructions, ce qui peut ralentir le *pipeline* en créant des « bulles ». Par exemple, dans le code suivant :

```
1 add eax, ebx ; place dans eax la somme eax+ebx
2 add ecx, eax ; place dans ecx la somme ecx+eax
```

---

15. Attention que cette fréquence ne correspond en général *pas* au nombre de cycles de la boucle *fetch-decode-execute*. Ainsi, un processeur fonctionnant à 1 GHz n'exécute pas forcément 1 milliard d'instructions machine par seconde. Il s'agit en général du nombre de cycles du chemin de données (*cfr*: Chapitre 5), et chaque instruction machine peut demander un nombre de cycles variable pour s'exécuter.

### 3.2. Le processeur

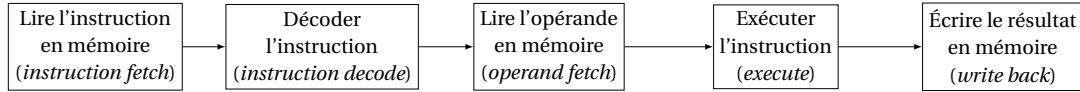


FIGURE 3.10. – Illustration d'un *pipeline* à cinq étages.

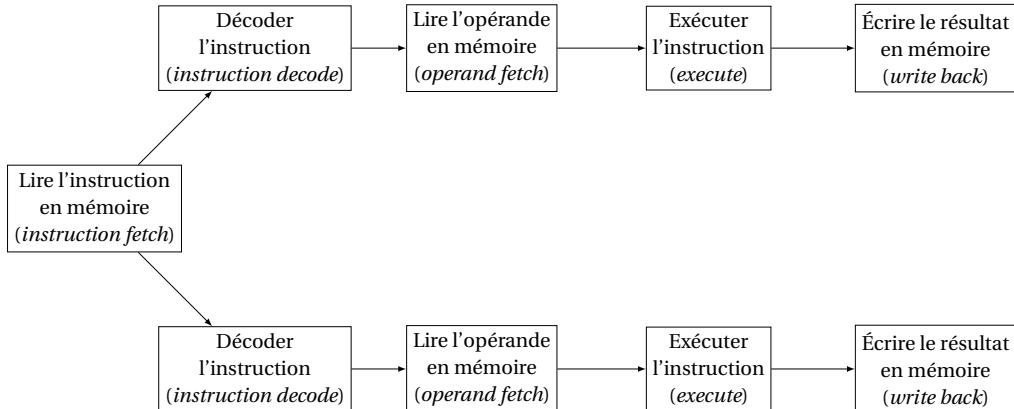


FIGURE 3.11. – Un *pipeline* superscalaire à cinq étages. L'étage « *instruction fetch* », qui est typiquement très rapide, est unique. Les instructions récupérées en mémoire par cet étage sont réparties sur deux *pipelines*, qui peuvent exécuter deux instructions en parallèle.

il est impératif que la première instruction se soit entièrement exécutée pour que la seconde s'exécute correctement, car cette dernière dépend de la valeur calculée lors de la première. Cela peut être résolu par le compilateur, ou bien détecté à l'exécution.

**Architectures superscalaires** Une évolution intéressante du *pipeline* consiste à dédoubler certaines des cinq unités pour traiter en parallèle les étapes correspondantes, comme sur la FIGURE 3.11. C'est particulièrement intéressant si certaines unités sont plus rapides que d'autres (typiquement, le décodage sera plus rapide que l'ALU). On parle alors d'un *pipeline superscalaire*.

Une architecture superscalaire consiste donc à mettre en parallèle plusieurs (morceaux de) *pipelines*. Ces idées étaient déjà présentes dans l'architecture du CDC 6600, un ordinateur conçu par Seymour CRAY<sup>16</sup>, et commercialisé par la *Control Data Corporation* à partir de 1964. Il était, à l'époque, considéré comme l'ordinateur le plus rapide du monde. Son architecture superscalaire était tout à fait révolutionnaire pour l'époque. Ce principe n'a commencé à être utilisées de manière régulière, dans les microprocesseurs, qu'à la fin des années 1980.

16. Né le 28 septembre 1925 et mort le 5 octobre 1996, informaticien américain connu pour être le « père des super-ordinateurs ». Il a fondé plusieurs entreprises ayant donné naissance à Cray Inc. qui conçoit et commercialise encore aujourd'hui des super-ordinateurs.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur



FIGURE 3.12. – Un ordinateur CDC 6600.

Source : Jitze Couperus ([https://commons.wikimedia.org/wiki/File:CDC\\_6600.jc.jpg](https://commons.wikimedia.org/wiki/File:CDC_6600.jc.jpg)), «CDC 6600.jc», <https://creativecommons.org/licenses/by/2.0/legalcode>.

**Processeurs vectoriels ou SIMD** Ces processeurs mettent en œuvre une technique différente, bien que toujours basée sur l'idée de paralléliser les calculs nécessaires à la bonne exécution d'un programme. Dans de nombreuses applications, il est nécessaire d'appliquer la même opération à tous les éléments d'un tableau. Pour ce faire, le programmeur écrit en général une boucle, qui parcourt chaque élément du tableau, et applique l'opération à chaque case séquentiellement. Les *processeurs vectoriels* sont des processeurs spécialisés dans le traitement des vecteurs (un autre nom pour « tableau ») : il appliquent en parallèle la même instruction à une grande quantité de données. En pratique, plusieurs (jusqu'à 256) unités de calcul sont mises en parallèle, mais la partie qui contrôle les instructions est commune. Les ordinateurs ILLIAC IV (voir FIGURE 3.13) et la série des ordinateurs Cray (FIGURE 3.14) sont des exemples historiques.

Aujourd'hui, ces types de processeurs sont appelés SIMD pour *Single Instruction Multiple Data* (application de la même instruction à de multiples données). On retrouve des processeurs vectoriels dans la plupart des cartes graphiques optimisées pour les jeux, car la manipulation des images en 3D se prête bien à ce type de traitements. La plupart des processeurs récents possèdent des instructions de ce type<sup>17</sup>.

**Multiprocesseurs** Une idée très naturelle consiste simplement à multiplier le nombre de processeurs (totalement indépendants) dans l'ordinateur. On parle alors de système *multiprocesseur*, on obtient alors une architecture comme sur la FIGURE 3.15 (avec deux processeurs, mais on peut en avoir plus). Comme on le voit, la mémoire est partagée, ce qui pose certaines difficultés : il ne faut pas que les deux processeurs tentent d'accéder à une même case mémoire au même moment (par exemple, en écriture), sans quoi les données pourraient être corrompues. Le bus est partagé également, ce qui peut ralentir globalement le système en cas de congestion.

Ces architectures sont particulièrement utiles quand on veut exécuter plusieurs tâches indépendantes en même temps, comme avec un système d'exploitation multi-tâche (voir Chapitre 8).

**Systèmes multi-cœurs** Afin d'éviter de surcharger le bus, on peut regrouper plusieurs processeurs sur un même circuit intégré. Chaque processeur est alors appelé *cœur* et le circuit intégré est appelé *processeur multicœur*. Un processeur multicœur ressemble donc très fort à un système multiprocesseur, à la différence que les communications entre cœurs ne doivent pas passer par le bus et sont donc plus rapides. Les différents cœurs partagent par contre une interface d'accès au bus (externe) unique, et souvent la mémoire cache de niveau L2.

La plupart des processeurs présents aujourd'hui dans les ordinateurs personnels sont multicœurs, c'est la solution qui a été privilégiée par les fabricants depuis environ 2005 pour continuer à faire croître les performances des processeurs sans augmenter leur fréquence.

**Grappes et fermes de calcul** Les notions de grappes et de fermes de calcul dépassent la notion de *processeur* en tant que composant d'un ordinateur. Mais ces techniques s'ins-

---

17. Par exemple, l'instruction `mulps` du jeu d'instructions Intel SSE, qui permet d'effectuer plusieurs multiplications en virgule flottante en parallèle.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

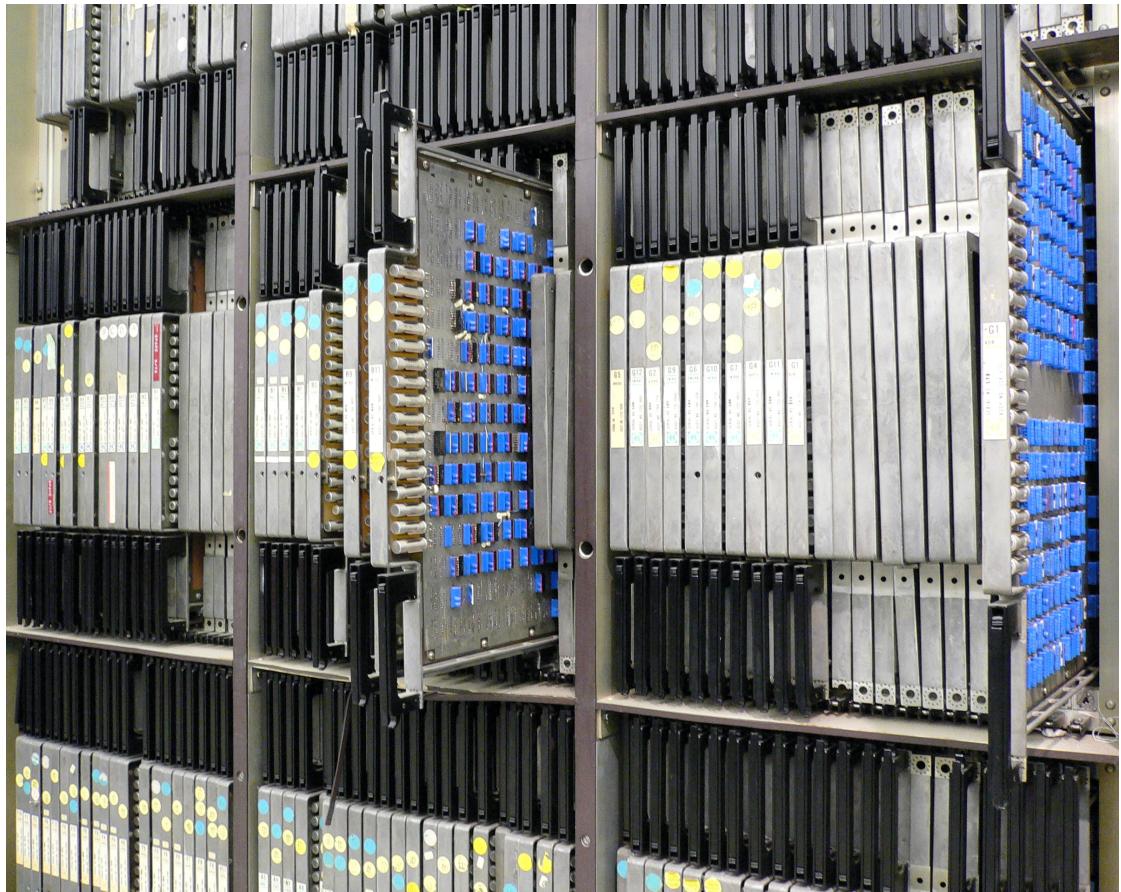


FIGURE 3.13. – Une vue de l'ILLIAC IV, ordinateur développé à l'Université de l'Illinois, et construit par la firme Burroughs, il fut finalement installé en 1972 au centre de recherche Ames de la NASA. Il fut le premier super-ordinateur connecté à ARPANet, l'ancêtre d'Internet.

Source : Steve Jurvetson from Menlo Park, USA ([https://commons.wikimedia.org/wiki/File:ILLIAC\\_4\\_parallel\\_computer.jpg](https://commons.wikimedia.org/wiki/File:ILLIAC_4_parallel_computer.jpg)), « ILLIAC 4 parallel computer », <https://creativecommons.org/licenses/by/2.0/legalcode>.

### *3.2. Le processeur*



FIGURE 3.14. – Une Cray 1 au Deutschen Museum de Munich. Il s'agit du premier modèle de super-ordinateur commercialisé par la firme fondée par Seymour CRAY.

Source : Clemens PFEIFFER (<https://commons.wikimedia.org/wiki/File:Cray-1-deutsches-museum.jpg>), « Cray-1-deutsches-museum », <https://creativecommons.org/licenses/by/2.5/legalcode>.

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

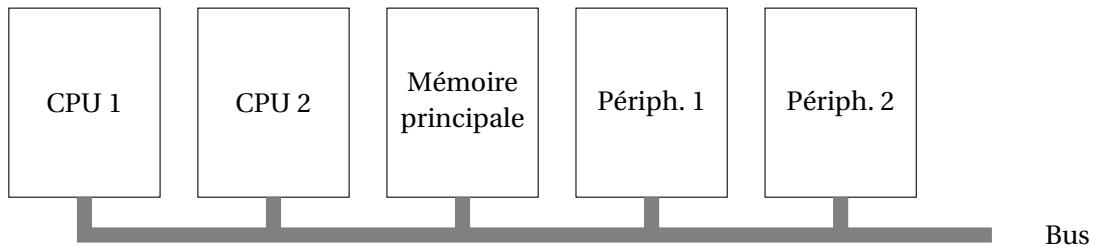


FIGURE 3.15. – L'organisation d'un ordinateur avec 2 CPUs

crivent dans la lignée de celles que nous venons de décrire, car elles visent toutes à augmenter la capacité du système informatique à traiter plusieurs informations en même temps (en parallèle). On peut dès lors connecter plusieurs unités de calcul individuelles, appelées *nœuds*, entre elles, et répartir la charge de calcul sur l'ensemble de ces ordinateurs, en les faisant travailler en parallèle.

Ces nœuds doivent naturellement communiquer et se synchroniser, et doivent donc être connectés entre eux. On distingue :

- les *grappes* de calcul (ou *clusters* en anglais), où tous les nœuds sont réunis en un même endroit, et communiquent entre eux de manière rapide et fiable à travers un réseau local; et
- les *fermes* de calcul (ou *grid* en anglais), où les différents nœuds (qui peuvent être eux-mêmes des *clusters*) sont connectés entre eux par un réseau de grande dimension, comme l'Internet.

Ce qui est important, dans le deux cas, est que l'utilisateur n'a pas à se soucier de la répartition de la charge de calcul sur les différents nœuds : il accède au *cluster* ou au *grid* à travers un point d'entrée unique, et un logiciel spécialisé (parfois intégré dans le système d'exploitation) se charge de répartir le travail sur les nœuds disponibles.



L'ULB et la VUB possèdent un cluster, appelé Hydra, qui se trouve au *Shared ICT Services Center*, et qui était composé en 2018 d'environ 160 nœuds multiprocesseurs multicœur (entre 16 et 24 cœurs par nœud), voir <https://hpc.ulb.be/>. Un exemple de *grid* est composé par le projet SeTI@home (*Search for extraTerrestrial Intelligence at home*, voir : <http://setiathome.ssl.berkeley.edu/>), auquel tout le monde peut participer en y connectant son ordinateur individuel. Le but de ce projet est d'analyser de grandes quantités de données provenant de radio-télescopes pour tenter d'y détecter le signal d'une intelligence extra-terrestre.

## 3.3. Les périphériques

Terminons notre discussion de la FIGURE 3.1 en parlant des *périphériques*. Comme leur nom l'indique, ils ne sont pas, à proprement parler, essentiels à l'exécution des programmes, mais ils permettent d'utiliser les ordinateurs de manière pratique, notamment en l'autorisant à communiquer avec le monde extérieur. Parmi les périphériques, nous commençons par distinguer la *mémoire secondaire*.

### 3.3.1. Mémoire secondaire

La mémoire (primaire) que nous avons décrite jusqu'à présent est la mémoire *de travail de l'ordinateur*. Elle est donc intrinsèquement *temporaire*. Pour des stockages à plus long terme, on fait appel à la *mémoire secondaire*, qui peut être vue comme un cas particulier de périphérique. Le mémoire secondaire est typiquement de plus grande capacité, mais aussi plus lente que la mémoire primaire. Par exemple, on trouve aujourd'hui des ordinateurs personnels avec une mémoire primaire de quelques GO (2 ou 4, voire un peu plus), mais une mémoire secondaire (disque dur ou SSD) de plusieurs centaines de GO (voire même un téraoctet). Le temps d'accès des disques durs est typiquement de l'ordre de la micro-seconde, alors que les temps d'accès des mémoires sont de l'ordre de la dizaine de nano-seconde.

À part ces différences de capacité et de vitesse, la mémoire secondaire se comporte essentiellement comme la mémoire primaire : elle stocke l'information sous forme binaire, dans des cases qui possèdent chacune une adresse. Naturellement, les codes détecteurs et correcteurs d'erreurs peuvent être utilisés.

Au long de l'histoire de l'informatique, la mémoire secondaire a pris plusieurs formes, en fonction de l'évolution de la technologie. En voici quelques-unes.

**Les cartes perforées** L'information est stockée physiquement à l'aide de trous perforés dans une carte en carton (voir FIGURE 3.16). Chaque position sur la carte correspond à un bit, et la présence ou non d'un trou indique la valeur du bit. Les cartes typiques stockaient 80<sup>18</sup>, encodés sur 8 à 10 bits en fonction du type de carte.

Les cartes perforées datent de bien avant les premiers ordinateurs. Elles étaient déjà utilisées au dix-huitième siècle par JACQUARD<sup>19</sup> pour représenter des motifs à broder sur des métiers automatisés. Au début du vingtième siècle, elles étaient en usage pour stocker et traiter de l'information de manière automatique, à l'aide de machines appelées *tabulatrices*<sup>20</sup>

**Les Rubans perforés** Ce *medium* applique le même principe que les cartes perforées, sauf que les perforations sont faites dans ruban de papier (voir FIGURE 3.17).

---

18. Ce qui explique pourquoi certains programmes de courrier électronique formattent encore les messages sur 80 colonnes).

19. Joseph Marie JACQUARD, né le 7 juillet 1752 à Lyon et mort le 7 août 1834 à Oullins. Inventeur français ayant perfectionné des travaux antérieurs pour créer un métier à tisser programmable à l'aide de cartes perforées, fort célèbre et répandu.

20. Ces machines, qui ont fait la fortune d'IBM, appliquaient, à toutes les cartes d'un paquet, un traitement choisi en modifiant des connexions câblées (un peu comme dans les anciennes centrales téléphoniques, du temps des opératrices). On ne peut donc pas vraiment parler d'ordinateur programmable...

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

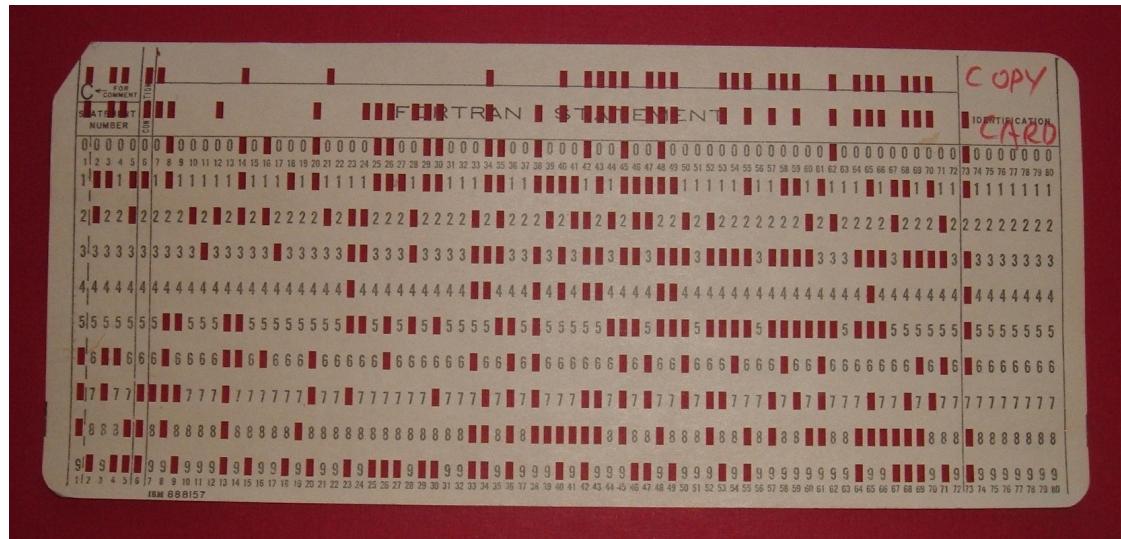


FIGURE 3.16. – Une carte perforée au format standard IBM.

Source : Arnold Reinhold (<https://commons.wikimedia.org/wiki/File:IBM1130CopyCard.agr.jpg>), « IBM1130CopyCard.agr », <https://creativecommons.org/licenses/by-sa/2.5/legalcode>.



FIGURE 3.17. – Du ruban perforé.

**Les bandes magnétiques** Avec ce *medium*, l'information est stockée de manière séquentielle sur une bande magnétisée. Les informations sont encodées par des variations du champ magnétique de la bande. Les bandes sont apparues dans les années 50, et sont encore en usage aujourd'hui, car elles permettent de stocker de grandes quantités d'information à un coût très bas. Les temps d'accès sont par contre assez mauvais, car il faut faire défiler la bande pour atteindre la zone à lire.

**Les disques magnétiques** Les disques magnétiques existent sous plusieurs formes :

- *Les disques durs* : il s'agit d'un ensemble de disques d'aluminium recouverts d'une surface magnétique. Les disques sont divisés logiquement en *secteurs* qui enregistrent chacun, une quantité fixée d'information. Ces disques sont maintenus en rotation en permanence. L'information est inscrite et lue à l'aide de *têtes de lecture* mobiles. L'accès à une zone particulière du disque requiert donc de positionner la tête et d'attendre que la bonne portion du disque passe sous la tête, comme on peut le voir sur la FIGURE 3.19. Un disque dur comporte généralement un circuit qui gère la position des têtes en fonction des demandes qui sont faites, appelé *contrôleur de disque*.

Certains systèmes, comme le système RAID, répartissent l'information sur plusieurs disques physiques, qui apparaissent alors à l'utilisateur comme un seul disque. Cela permet, selon les cas, de paralléliser le traitement de l'information, voire de récupérer l'information en cas de panne d'un des disques, si l'information est stockée de manière redondante sur les différents disques.

- *Les disquettes* : les disquettes fonctionnent sur le même principe que les disques durs, mais ne comprennent qu'un seul disque (qui peut être inscrit en simple ou double face selon les modèles). Les disquettes sont amovibles alors que les disques durs ne le sont généralement pas. Remarque : les premiers ordinateurs utilisaient de gros disques magnétiques, semblables à nos disques durs modernes, mais amovibles (comme les disquettes). La FIGURE 3.18 présente des exemples de disquettes.

**Les disques optiques** tels que les CD-ROMS ou DVD-ROMS, enregistrent les informations à l'aide de trous dans une surface réfléchissante, qui peut ensuite être lue optiquement (par un faisceau laser).

**La mémoire Flash / SSD** Ce type de mémoire est apparu de manière plus récente et a d'abord été disponible en faible capacité (quelques centaines de kilo-octets, quelques mégaoctets), principalement dans des appareils comme les appareils photo numériques. Le développement des technologies permet aujourd'hui de les intégrer dans les ordinateurs personnels, en remplacement des disques durs classiques. Il existe plusieurs technologies différentes, mais toutes stockent l'information de manière électronique, comme la mémoire primaire (à la différence que la mémoire Flash/SSD n'est pas volatile).

### 3. Leçon 5 & 6 – Organisation de l'ordinateur



FIGURE 3.18. – Trois types de disquettes : de 8, 5,25, et 3,5 pouces.

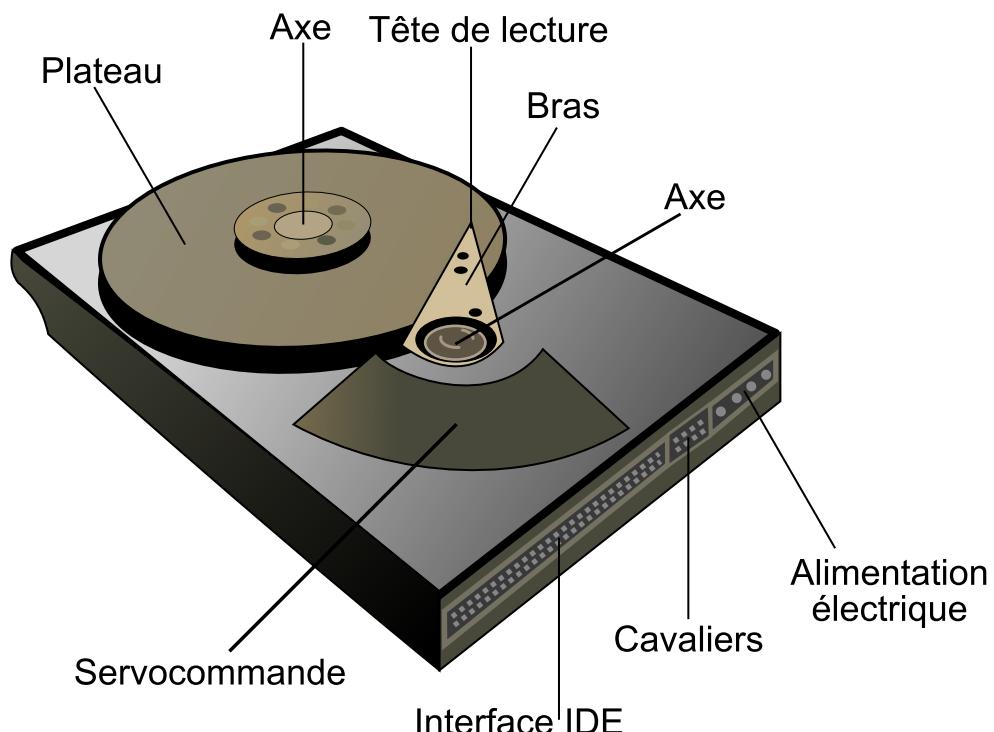


FIGURE 3.19. – Illustration d'un disque dur.

Source : I, Surachit ([https://commons.wikimedia.org/wiki/File:Hard\\_drive-fr.svg](https://commons.wikimedia.org/wiki/File:Hard_drive-fr.svg)), « Hard drive-fr », <https://creativecommons.org/licenses/by-sa/3.0/legalcode>

### 3.3.2. Les périphériques d'entrée/sortie

Pour pouvoir communiquer avec le monde extérieur, l'ordinateur utilise des *périphériques*, qui permettent d'acquérir de l'information ou de la restituer. Les périphériques sont généralement composés de deux parties :

1. le périphérique *physique*, c'est-à-dire le dispositif qui va réaliser l'acquisition ou la restitution de l'information (par exemple : un écran, un clavier) ; et
2. un *contrôleur* : un circuit électronique qui réalise la connexion du périphérique physique au bus de l'ordinateur. Il a pour mission de récupérer les informations qui circulent sur le bus et qui lui sont adressées, et de piloter le périphérique en conséquence (par exemple : la carte graphique, qui va recevoir des instructions du processeur pour afficher des images à l'écran), ou bien de convertir l'information qui vient du périphérique pour l'émettre sur le bus (par exemple : le contrôleur du clavier).

Parmi les différents types de périphériques, on trouve, parmi d'autres exemples :

- les écrans et périphériques d'affichage ;
- les claviers ;
- les souris ou *trackpads* ;
- les modems, qui permettent de transmettre des informations sur des lignes téléphoniques, par exemple ;
- les « cartes son » accompagnées des haut-parleurs, micros, *etc.* ;
- les appareils et caméras numériques.

**Accès direct à la mémoire** Comme nous l'avons dit, la plupart des périphériques servent à échanger de l'information entre l'ordinateur et le monde extérieur. Au sein de l'ordinateur, cette information est stockée dans la mémoire primaire (afin que le processeur puisse la traiter). Les périphériques doivent donc dialoguer en permanence avec la mémoire primaire. Voyons maintenant comment nous pouvons réaliser cela de manière efficace.

Le schéma de base permettant d'acquérir de l'information sur un périphérique, et de la stocker en mémoire fonctionne comme suit :

1. le CPU envoie un ordre au contrôleur du périphérique ;
2. le contrôleur envoie la commande correspondante au périphérique ;
3. le périphérique répond au contrôleur ;
4. le contrôleur émet l'information sur le bus, adressée au CPU ;
5. le CPU récupère l'information et la transmet à la mémoire, *via* le bus.

Si le but final de l'opération est de stocker l'information dans la mémoire, ce schéma est peu efficace. Il n'est en effet pas nécessaire de passer par le CPU, qui pourrait utiliser le temps ainsi économisé à d'autres tâches. Sur les ordinateurs modernes, certains périphériques ont donc la possibilité d'écrire *directement en mémoire* à un endroit bien précis. On appelle cette technique l'*accès direct à la mémoire*, ou *Direct memory access* (DMA). En pratique, chaque périphérique se voit attribuer une plage d'adresses en mémoire primaire où il peut stocker

### 3. Leçon 5 & 6 – Organisation de l'ordinateur

directement de l'information sans la supervision du CPU. Ce mécanisme est chapeauté par un *contrôleur DMA*, sorte de processeur dédié à cette seule tâche, qui peut donc fonctionner en parallèle du CPU. Quand l'information est prête en mémoire, le contrôleur prévient le CPU *via* un mécanisme d'*interruption* (que nous décrirons dans le Chapitre 7).

**Bus dédiés** La FIGURE 3.1 nous présente un modèle de l'ordinateur avec un bus unique, sur lequel transitent tous les échanges d'informations entre le CPU, la mémoire et les périphériques. En pratique, cela peut poser plusieurs problèmes :

- le CPU, la mémoire et les périphériques ne fonctionnent pas à la même vitesse. Un bus lent est amplement suffisant pour des périphériques comme la souris et le clavier, mais pas pour la communication entre le CPU et la mémoire (n'oublions pas que le CPU doit charger chaque instruction exécutée depuis la mémoire!);
- tous les composants connectés au bus doivent communiquer de la même manière (encodage de l'information, etc). Par ailleurs, les périphériques d'un ordinateur sont par nature amovibles, et leur raccordement au bus doit donc se faire via des connecteurs, qui sont naturellement standardisés et dépendent souvent du type de bus. Un bus unique implique donc que tous les périphériques doivent pouvoir être connectés physiquement au bus de la même manière;
- le changement d'un composant essentiel, comme le CPU ou la mémoire, risque de demander de changer également le bus, et donc, potentiellement, tous les périphériques, alors que pour des raisons économiques évidentes, on souhaite pouvoir réutiliser ses anciens périphériques sur son nouvel ordinateur.

Afin de contourner ce problème, l'industrie a développé une série de *bus dédiés*, qui spécifient : un protocole de communication de l'information sur ce bus; et une norme pour les connecteurs, assurant que tous les dispositifs qui respectent cette norme puissent être facilement branchés sur le bus. Ces bus spécialisés accueillent alors une famille de périphériques, et sont connectés au bus principal *via* un contrôleur dédié, souvent appelé *pont* (ou *bridge*).

Par exemple : le bus USB<sup>21</sup> existe en plusieurs versions selon des normes bien définies, spécifiant tant le protocole de communication (vitesse, encodage des données, etc) que les formes des connecteurs USB. Il est bien adapté pour connecter des petits périphériques comme le clavier, la souris, *etc*. Parmi les autres exemples de bus courants on peut citer :

1. le bus ISA, qui était le standard pour les cartes d'extension (carte graphique, carte son) sur les premiers PC;
2. le bus PCI, une évolution d'ISA lui-même récemment remplacé par PCI express;
3. le bus AGP, dédié aux cartes graphiques.



---

21. USB signifie d'ailleurs *Universal Serial Bus*, ou bus série universel

## **Deuxième partie**

# **Les portes logiques**



## 4. Leçons 6 à 9 – Niveau 0 : portes logiques

Dans ce chapitre, nous allons aborder l'explication du fonctionnement d'un ordinateur (alors que les chapitres précédents étaient d'avantage descriptifs). Pour ce faire, nous allons considérer le niveau le plus bas de la FIGURE 1.5, c'est-à-dire le niveau matériel : les portes logiques.

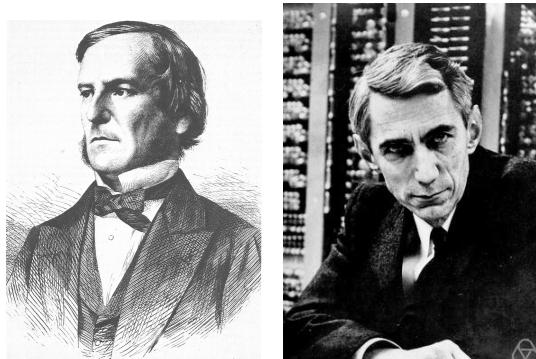
Les portes logiques sont les éléments de calcul les plus simples que nous considérerons dans ce cours. Elles permettent de calculer des opérations très élémentaires, appelées *opérations logiques*. Elles peuvent être réalisées à l'aide de transistors, ce qui, étant donné la technologie actuelle, permet d'en intégrer un nombre considérable (des millions, des milliards) dans des circuits électroniques de petite dimension (environ la surface de l'ongle d'un pouce pour un processeur moderne).

Néanmoins, ces opérations très simples que sont les opérations logiques sont suffisantes pour réaliser des opérations plus complexes. Ainsi, dans ce chapitre, nous verrons comment réaliser un circuit qui effectue l'addition de deux nombres (exprimés en binaire, voir Chapitre 2), comment réaliser une ALU, comment réaliser une mémoire... uniquement à l'aide de portes logiques.

### 4.1. L'algèbre Booléenne

On doit le développement de l'algèbre Booléenne [2] au mathématicien anglais George BOOLE<sup>1</sup>. L'algèbre Booléenne servira de base, en 1937 au mémoire de master de Claude E. SHANNON<sup>2</sup> [17] dans lequel il jette les bases des circuits logiques. Ces travaux sont considérés comme fondateurs pour l'informatique moderne.

Dans l'algèbre Booléenne, on ne peut manipuler que deux valeurs : le 0 et le 1 (parfois remplacées par les valeurs logiques « faux » et « vrai », ce qui fait qu'on parle souvent de *logique* Booléenne au lieu d'algèbre Booléenne). Ainsi, toutes les variables dans les



George BOOLE (g.) et Claude E. SHANNON (d.)

Source : Jacobs, Konrad ([https://commons.wikimedia.org/wiki/File:ClaudeShannon\\_MFO3807.jpg](https://commons.wikimedia.org/wiki/File:ClaudeShannon_MFO3807.jpg)). « ClaudeShannon MFO3807 », <https://creativecommons.org/licenses/by-sa/2.0/de/legalcode>.

1. Né le 2 novembre 1815, mort le 8 décembre 1864.

2. Né le 30 avril 1916, mort le 4 février 2001. Mathématicien et informaticien américain, considéré comme le fondateur de la théorie de l'information.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

expressions de l'algèbre Booléenne ne pourront prendre qu'une de ces deux valeurs (contrairement à l'algèbre à laquelle nous avons été habituées depuis l'école secondaire, où les variables prennent des valeurs entières, réelles... selon le contexte).

Les *opérateurs* qui permettent de combiner ces valeurs sont aussi propres à l'algèbre Booléenne, et ils reflètent les connecteurs logiques que nous utilisons dans les langues naturelles : et, ou, non,...

**Tables de vérité** Afin d'expliquer de manière non-ambiguë quel est le sens de chacun des opérateurs (ainsi que des expressions Booléennes plus complexes), on utilise des *tables de vérité*. Ces tables indiquent, pour chacune des combinaisons des valeurs des variables, quelle est la valeur de l'opérateur ou de l'expression considérée. De ce fait, une table de vérité pour une expression qui a  $n$  variables d'entrée aura nécessairement  $2^n$  lignes (le nombre de combinaisons de valeurs Booléennes pour  $n$  variables).



Voici un exemple de table de vérité, pour deux variables  $a$  et  $b$  d'entrée, et qui nous indique la valeur d'une fonction  $f(a,b)$  de ces deux variables. Cette table a bien  $2^2 = 4$  lignes.

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

Cette table nous indique que, quand  $a = 0$  et  $b = 1$ , par exemple, la sortie doit être 1 ; alors qu'elle doit être 0 si  $a = b = 0$ , etc.

Passons maintenant en revue les opérateurs Booléens de base :

**L'opérateur et** L'opérateur **et** est un opérateur binaire<sup>3</sup>. La notation habituelle de l'opérateur **et** est :  $\wedge$ . Par définition, la valeur de l'opérateur est 1 si et seulement si les deux opérandes valent 1 (la valeur de l'opérateur est donc 0 dans les autres cas). On représente cela par la table de vérité :

$a$	$b$	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

3. C'est-à-dire qu'il s'applique à deux valeurs, comme les +, par exemple. Ces deux valeurs sont appelées les *opérandes*

**L'opérateur ou** L'opérateur **ou** est également un opérateur binaire, qui vaut 1 si et seulement si une des deux opérandes vaut 1. On le représente par  $\vee$  et sa table de vérité est :

$a$	$b$	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

**L'opérateur ou exclusif** Cet opérateur est proche du **ou** : il vaut 1 si et seulement si *exactement* une des opérandes vaut 1 (intuitivement, avoir une des opérandes à 1 *exclut* que l'autre le soit également). On le représente XOR et sa table de vérité est :

$a$	$b$	$a \text{XOR } b$
0	0	0
0	1	1
1	0	1
1	1	0

Observez la différence avec la table de vérité du **ou** (dans la dernière ligne).

**L'opérateur non** Enfin, l'opérateur de négation est un opérateur *unaire*<sup>4</sup> qui inverse la valeur de son opérande. On le note  $\neg$  :

$a$	$\neg a$
0	1
1	0

**Autres opérateurs** Sur base de ces opérateurs, on peut en construire d'autres, notamment le NOR et le NAND que l'on définit ainsi :

$$\begin{aligned} a \text{NOR } b &= \neg(a \vee b) \\ a \text{NAND } b &= \neg(a \wedge b) \end{aligned}$$

Un NOR est donc bien la négation d'un **ou**. Prenons garde à ne pas confondre cela avec la disjonction de la négation des variables, en d'autres termes :

$$a \text{NOR } b \neq (\neg a) \vee (\neg b).$$

---

4. C'est-à-dire qu'il n'a qu'une seule opérande comme le  $-$  dans  $-(x + y)$ , par exemple).

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

On peut s'en convaincre en considérant par exemple le cas où  $a = 1$  et  $b = 0$ . Dans ce cas :

$$\begin{aligned} a \text{NOR } b &= 1 \text{NOR } 0 \\ &= \neg(1 \vee 0) \\ &= \neg 1 \\ &= 0, \end{aligned}$$

alors que :

$$\begin{aligned} \neg a \vee \neg b &= (\neg 1) \vee (\neg 0) \\ &= 0 \vee 1 \\ &= 1. \end{aligned}$$

De même pour le NAND. On peut par contre appliquer les lois de DE MORGAN, dont nous parlerons plus tard aux équations (4.8) and (4.9).

**Priorité des opérateurs** Comme dans l'algèbre « classique », on peut utiliser des parenthèses pour fixer l'ordre dans lequel les opérateurs doivent être évalués, et il existe une priorité fixe des opérateurs, à savoir, du moins au plus prioritaire :

$$\begin{matrix} \vee \\ \text{XOR} \end{matrix}, \wedge, \neg.$$

Le  $\vee$  et le XOR ont la même priorité.



Par exemple, l'expression :

$$x \vee \neg y \wedge z$$

se comprend et s'évalue comme :

$$x \vee ((\neg y) \wedge z)$$

**Identités remarquables** On peut identifier certaines propriétés des opérateurs  $\wedge$  et  $\vee$  qui seront utiles par la suite. En se référant aux tables de vérité données ci-dessus, on peut se convaincre que les équations suivantes sont vraies pour toute valeur (Booléenne) de  $x$  :

$$x \wedge 1 = x \tag{4.1}$$

$$x \wedge 0 = 0 \tag{4.2}$$

$$x \vee 1 = 1 \tag{4.3}$$

$$x \vee 0 = x. \tag{4.4}$$

#### 4.1. L'algèbre Booléenne

En d'autres termes, 1 est neutre pour l'opérateur  $\wedge$ ; 0 est absorbant pour l'opérateur  $\wedge$ ; 1 est absorbant pour l'opérateur  $\vee$ ; et 0 est neutre pour l'opérateur  $\vee$ .

On peut également noter que, pour tout  $x$ :

$$x \wedge \neg x = 0 \quad (4.5)$$

$$x \vee \neg x = 1. \quad (4.6)$$

Intuitivement, la partie gauche de l'équation (4.5) demande que  $x$  soit à la fois vraie (valeur 1) et fausse (valeur 0), ce qui n'est pas possible (le résultat est donc 0). De même, la partie gauche de l'équation (4.6) demande que  $x$  soit *soit* vraie *soit* fausse, ce qui est toujours vrai (le résultat est donc 1).

De plus, l'algèbre Booléenne jouit également d'une loi de distributivité. Ainsi, pour toutes expressions  $\varphi_1$ ,  $\varphi_2$  et  $\varphi_3$ , on a :

$$(\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3) = \varphi_1 \wedge (\varphi_2 \vee \varphi_3) \quad (4.7)$$

On peut s'en convaincre en comparant les tables de vérité.

Finalement, on n'oubliera pas les célèbres lois de DE MORGAN<sup>5</sup>, qui permettent de réécrire la négation d'un **et** ou d'un **ou** : pour toute paire d'expressions  $\varphi$  et  $\psi$  :

$$\neg(\varphi \vee \psi) = \neg\varphi \wedge \neg\psi, \quad (4.8)$$

$$\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi. \quad (4.9)$$

On remarquera que quand la négation « rentre » dans la parenthèse, le **et** se transforme en **ou** et vice-versa.

**Des expressions Booléennes aux tables de vérité** Il est parfois utile, étant donné une expression Booléenne, de donner sa table de vérité *in extenso*. Pour ce faire, on peut construire une table de vérité qui contient une colonne pour chaque étape du calcul de l'expression, en suivant l'ordre de priorité des opérateurs.

---

5. D'après le mathématicien britannique Augustus DE MORGAN (27 juin 1806 à Madurai (Tamil Nadu) – 18 mars 1871).

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques



Par exemple, considérons à nouveau l'expression

$$x \vee \neg y \wedge z.$$

On construit un table de vérité à  $8 = 2^3$  lignes. On commence par remplir une colonne qui correspond à  $\neg y$ ; puis une colonne qui correspond à  $(\neg y) \wedge z$ , en utilisant la colonne construite pour  $\neg y$ ; puis enfin une colonne pour toute l'expression :

$x$	$y$	$z$	$\neg y$	$(\neg y) \wedge z$	$x \vee ((\neg y) \wedge z)$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

**Des tables de vérité aux expressions Booléennes** Dans bien des cas, il est utile de pouvoir traduire une table de vérité en expression Booléenne. C'est ce que nous ferons de manière régulière dans ce chapitre quand nous voudrons concevoir un circuit logique. Nous commencerons par écrire la table de vérité du circuit, qui *spécifiera* ce qu'on attend de lui, puis nous transformerons cette table en expression qu'il sera enfin facile de traduire en circuit logique. Pour ce faire, il existe une technique systématique. Commençons par regarder un exemple :



Considérons la table de vérité :

$x$	$y$	$z$	$s$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



Nous voyons que la sortie  $s$  est *vraie* (elle vaut 1) dans exactement trois cas :

1. soit quand  $x = 0$  et  $y = 0$  et  $z = 1$ ;
2. soit quand  $x = 1$  et  $y = 0$  et  $z = 1$ ;
3. soit quand  $x = 1$  et  $y = 1$  et  $z = 1$ .

Cette formulation suggère une expression Booléenne obtenue comme suit. Tout d'abord, on traite chacun des trois cas ci-dessus séparément, en écrivant, pour chacune des valuations des trois variables, une expression qui n'est vraie *que* pour ces valuations :

1. l'expression  $\neg x \wedge \neg y \wedge z$  n'est vraie que si  $x = 0$ ,  $y = 0$  et  $z = 1$ ;
2. l'expression  $x \wedge \neg y \wedge z$  n'est vraie que si  $x = 1$ ,  $y = 0$  et  $z = 1$ ;
3. l'expression  $x \wedge y \wedge z$  n'est vraie que si  $x = 1$ ,  $y = 1$  et  $z = 1$ .

Ensuite, comme la formule doit être vraie dans un de ces trois cas, et uniquement dans un de ces trois cas, on peut prendre la disjonction des trois formules :

$$s = (\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge z).$$

On peut vérifier que cette formule correspond à la table de vérité : pour chaque valuation d' $x$ ,  $y$  et  $z$ , la formule vaut 0 dans tous les cas où  $s = 0$  selon la table, et la formule vaut 1 dans tous les cas où  $s = 1$  selon la table.

Cet exemple suggère la méthode systématique suivante :



La méthode systématique pour extraire une expression d'une table de vérité (sur les variables  $x_1, x_2, \dots, x_n$ ) peut donc être résumée comme ceci :

1. Pour chacune des lignes  $i$  où le résultat de la table est 1, on construit une formule  $\varphi_i$  comme suit :
  - a) Pour chaque variable  $x$ , on construit l'expression  $\alpha_x^i$ , qui vaut  $x$  si la valeur de  $x$  dans la ligne  $i$  est 1; et qui vaut  $\neg x$  sinon.
  - b) On construit alors  $\varphi_i = \alpha_{x_1}^i \wedge \alpha_{x_2}^i \wedge \dots \wedge \alpha_{x_n}^i$ .
2. On prend la disjonction (**ou**) de toutes les formules  $\varphi_i$  correspondant aux lignes où la sortie vaut 1.



En continuant l'exemple ci-dessus, on a donc :

1. pour la ligne 2 :  $\alpha_x^2 = \neg x$ ,  $\alpha_y^2 = \neg y$  et  $\alpha_z^2 = z$ . Donc,  $\varphi_2 = \neg x \wedge \neg y \wedge z$ ;
2. pour la ligne 6 :  $\alpha_x^6 = x$ ,  $\alpha_y^6 = \neg y$  et  $\alpha_z^6 = z$ . Donc,  $\varphi_6 = x \wedge \neg y \wedge z$ ;
3. pour la ligne 8 :  $\alpha_x^8 = x$ ,  $\alpha_y^8 = y$  et  $\alpha_z^8 = z$ . Donc  $\varphi_8 = x \wedge y \wedge z$ .

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques



FIGURE 4.1. – Quelques portes logiques de base.



La formule finale est donc bien :  $\varphi_2 \vee \varphi_6 \vee \varphi_8 = (\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge z)$ , comme annoncé.

Remarquons que cette formule peut être simplifiée. En appliquant (4.7) sur les deux premières parenthèse, on peut mettre  $(\neg y \wedge z)$  en évidence et on a :

$$\begin{aligned}
 & (\neg x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge z) \\
 = & ((\neg x \vee x) \wedge (\neg y \wedge z)) \vee (x \wedge y \wedge z) \quad \text{Par (4.7),} \\
 = & (1 \wedge (\neg y \wedge z)) \vee (x \wedge y \wedge z) \quad \text{Par (4.6),} \\
 = & (\neg y \wedge z) \vee (x \wedge y \wedge z) \quad \text{Par (4.1).}
 \end{aligned}$$

## 4.2. Les circuits logiques

Une *porte logique* est un dispositif qui calcule un des opérateurs de la logique Booléenne. Une porte logique est représentée graphiquement à l'aide d'un symbole distinctif, qui possède des entrées et une sortie, représentées sous forme de traits. On combine les portes logiques entre elles en connectant la sortie d'une porte aux entrées d'autres portes, ce qui forme un *circuit*, capable de calculer une fonction Booléenne. La FIGURE 4.1 illustre ces portes.

### 4.2.1. Réalisation des portes logiques

Concrètement, nous voulons réaliser ces portes logiques à l'aide de matériel électronique qui possèderont des *connecteurs* (pattes métalliques ou câbles) d'entrée et de sortie pour obtenir les valeurs d'entrée et produire les valeurs de sortie. Des voltages différents sont utilisés pour représenter les valeurs Booléennes. Par exemple, on utilisera 0 volt pour *faux* (valeur 0) et 5 volts pour *vrai* (valeur 1). Une fois cette convention fixée, on peut envisager différentes manières de réaliser ces portes logiques.

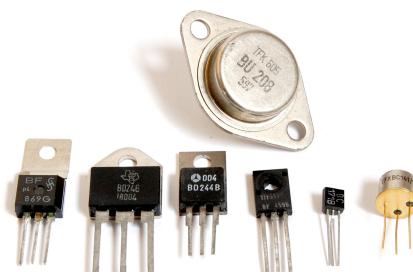


FIGURE 4.2. – Exemples de transistors.

Dans les ordinateurs modernes, on utilise des *transistors*, qui sont des composants électriques à 3 connexions appelées respectivement *base*, *émetteur* et *collecteur*. On en trouve sous différentes formes (voir FIGURE 4.2), selon la fonction qu'ils doivent réaliser, mais on est aujourd'hui capable de les miniaturiser énormément (les processeurs actuels ont de l'ordre de plusieurs milliards de transistor par mm<sup>2</sup>).

En combinant deux transistors, on peut réaliser les portes logiques NAND et NOR. Ces portes peuvent ensuite être utilisées pour réaliser d'autres portes logiques. En effet, on peut observer que :

$$\neg x = x \text{NAND } x,$$

$$\begin{aligned} x \wedge y &= \neg(x \text{NAND } y) \\ &= (x \text{NAND } y) \text{NAND}(x \text{NAND } y), \end{aligned}$$

$$\begin{aligned} x \vee y &= \neg((\neg x) \wedge (\neg y)) && \text{par (4.8),} \\ &= (\neg x) \text{NAND}(\neg y) \\ &= (x \text{NAND } x) \text{NAND}(y \text{NAND } y). \end{aligned}$$

On peut donc exprimer les portes logiques « habituelles » en termes de NAND uniquement. Un raisonnement similaire permet la même conclusion pour le NOR. On peut légitimement se demander s'il est vraiment utile de remplacer une seule porte **ou** par trois portes NAND, comme suggéré ci-dessus. Cette façon de faire est motivée par des considérations pratiques : en ramenant tout à des portes NAND (ou NOR), on simplifie la fabrication des circuits intégrés. Les machines qui les fabriquent ne doivent réaliser qu'un seul type de porte logique, toutes les autres sont obtenues par assemblage.

Notons pour terminer que dans les premiers ordinateurs (avant les années 1950), les transistors étaient remplacés par des tubes à vide, comme sur la FIGURE 4.3. Le principe était *grosso modo* le même, mais les tubes avaient le désavantage de chauffer, de consommer beaucoup de courant, d'être fragiles et d'occuper plus de place. Les transistors ont été inventés en 1947 par John BARDEEN<sup>6</sup>, Walter Houser BRATTAIN<sup>7</sup>, et William SHOCKLEY<sup>8</sup> aux Bell Labs. Ils ont reçu le prix NOBEL de physique pour leurs travaux, en 1972.

### 4.2.2. Réalisation d'un circuit logique

Une fois qu'on possède une formule Booléenne (qu'on a par exemple extraite d'une table de vérité selon la méthode décrite plus haut), il est relativement aisés de construire le circuit logique correspondant. On se contente de suivre l'ordre de calcul des différents opérateurs logiques, selon la priorité ou les parenthèses.

---

6. Physicien américain, né le 23 mai 1908 et mort le 30 janvier 1991.

7. Physicien américain, né le 10 février 1902 et mort le 13 octobre 1987.

8. Physicien américain, né le 13 février 1910 et mort le 12 août 1989.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

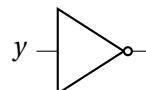


FIGURE 4.3. – Exemple de tubes IBM.

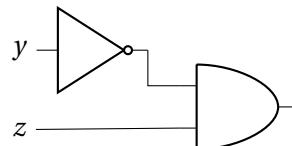
Source : Shieldforyoureyes Dave Fischer / Retro-Computing Society of Rhode Island Native, Established 1994. Website <http://rcsri.org>. Authority control : Q18857750 (<https://commons.wikimedia.org/wiki/File:IBM-tube.jpg>), « IBM-tube », <https://creativecommons.org/licenses/by-sa/3.0/legalcode>.



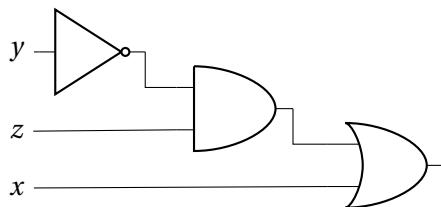
Par exemple, pour l'expression  $x \vee \neg y \wedge z = x \vee ((\neg y) \wedge z)$  déjà considérée ci-dessus, on commence par calculer  $\neg y$  :



On calcule ensuite  $(\neg y) \wedge z$ , en connectant la sortie de la porte **non** (qui calcule donc  $\neg y$ ) à une des deux entrées d'une porte **et**, l'autre entrée étant connectée à l'entrée  $z$  du circuit :



On termine en calculant un **ou** entre la sortie du **et** et l'entrée  $x$  :



Maintenant que nous avons à notre disposition une série d'outils pour manipuler les formules Booléennes et les tables de vérité, nous allons mettre ces connaissances en pratique et concevoir des circuits qui sont nécessaires à la réalisation d'un ordinateur. Nous commencerons par des circuits qui permettent de *manipuler* l'information (en binaire) et d'effectuer des calculs ; ils apparaîtront dans les *processeurs*. Ensuite, nous considérerons des circuits qui permettent de réaliser des *mémoires*.

Durant le cours, nous utiliserons régulièrement l'outil LogiSim, un simulateur de circuits logiques, distribué comme logiciel *libre*, et compatible tant avec les PC sous Windows et Linux qu'avec les Macs. On peut le télécharger à l'adresse : <http://ozark.hendrix.edu/~burch/logisim/>.

## 4.3. Circuits pour réaliser l'arithmétique binaire

Dans cette section, nous allons étudier des circuits logiques qui permettent de réaliser des opérations arithmétiques sur des nombres représentés en binaire. Nous allons donc fixer un nombre  $n$  de bits pour représenter ces nombres, et réaliser des circuits qui ont, au moins  $k \times n$  entrées (s'il y a  $k$  opérandes à l'opération, par exemple  $k = 2$  pour l'addition) et au moins  $\ell \times n$  sorties (si l'opération produit  $\ell$  résultats, par exemple  $\ell = 1$  pour l'addition)

### 4.3.1. Circuit additionneur

On se souviendra qu'on calcule la somme de deux nombres binaires<sup>9</sup>  $a = a_{n-1} \dots a_1 a_0$  et  $b = b_{n-1} \dots b_1 b_0$  comme on le ferait en base 10 : en sommant d'abord les bits de poids faibles  $a_0$  et  $b_0$ , puis les bits  $a_1$  et  $b_1$  avec un éventuel report venant de la somme de  $a_0$  et  $b_0$ , etc. Par exemple, sur  $n = 4$  bits :

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

Cette technique nous permet de décomposer la réalisation de notre circuit additionneur en plusieurs sous-problèmes :

- tout d'abord, il nous faut un circuit capable de réaliser la somme binaire de  $a_0$  et  $b_0$  (les bits de poids faible). Cette opération produit une valeur binaire qui tient sur deux bits<sup>10</sup>. Le bit de poids faible,  $s_0$  constitue le bit de poids faible du résultat ; le bit de poids fort,  $r_0$ , constitue le report de la colonne suivante :

$$\begin{array}{r} r_0 \\ \dots & a_1 & a_0 \\ + & \dots & b_1 & b_0 \\ \hline \dots & & & s_0 \end{array}$$

9. où les  $a_i$  et les  $b_i$  sont les bits individuels des deux nombres, les bits  $a_0$  et  $b_0$  étant les bits de poids faible

10. Le plus grand nombre qu'on peut obtenir est en effet  $3_{10} = 11_2$

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

Un tel circuit a donc *deux entrées et deux sorties*.

- Ensuite, nous avons besoin d'un circuit qui fait la somme de *trois* bits, pour réaliser le calcul qui se présente à nous dans toutes les autres colonnes. Dans la colonne  $i$  ( $i > 1$ ), ces trois bits seront : les deux bits  $a_i$  et  $b_i$  des deux nombres  $a$  et  $b$  à additionner et le report  $r_{i-1}$  provenant de la colonne précédente. Le résultat de l'addition tient à nouveau sur 2 bits : le bit de poids faible,  $s_i$ , apparaît dans le résultat, et le bit de poids fort,  $r_i$ , est le report pour la colonne suivante (si cette colonne existe, autrement,  $i = n$  et  $r_i$  est le bit de poids fort du résultat) :

$$\begin{array}{r} r_i \quad r_{i-1} \\ \cdots \quad a_{i+1} \quad a_i \quad a_{i-1} \quad \cdots \\ + \quad \cdots \quad b_{i+1} \quad b_i \quad b_{i-1} \quad \cdots \\ \hline \cdots \quad s_i \quad \cdots \end{array}$$

Le premier de ces deux circuits est appelé *demi-additionneur*; le second est appelé *additionneur complet* (ou simplement, *additionneur*). Nous allons maintenant étudier ces deux circuits, l'additionneur complet pouvant être compris comme une combinaison de deux demi-additionneurs.

##### 4.3.2. Demi-additionneur

Un demi-additionneur réalise la somme de deux bits  $a$  et  $b$  et produit deux sorties : la somme  $s$  (bit de poids faible) et le report  $r$  (bit de poids fort). La table de vérité est la suivante, elle représente l'effet de l'addition :

$a$	$b$	$r$	$s$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Partant de cette table, on peut facilement reconnaître que les sorties  $s$  et  $r$  correspondent à des portes logiques de base, à savoir :

- $s = a \text{XOR } b$ ; et
- $r = a \wedge b$ .

On en déduit le circuit<sup>11</sup> de la FIGURE 4.4<sup>12</sup>.

**Additionneur** Comme expliqué ci-dessus, nous avons également besoin d'un circuit qui réalise la somme de 3 bits (au lieu de 2 dans le cas du demi-additionneur). Il aura donc trois entrées :  $a$ ,  $b$ ,  $r_{prec}$  (le report provenant de la colonne précédente) et on a toujours deux

11. Voir aussi le fichier LogiSim DemiAdd1bit.circ sur l'UV.

12. Dans nos circuits, nous ajouterons des • quand deux traits sont connectés (l'absence de • indique donc qu'il n'y a pas de connexion logique, il s'agit simplement d'un croisement sur le schéma).

### 4.3. Circuits pour réaliser l'arithmétique binaire

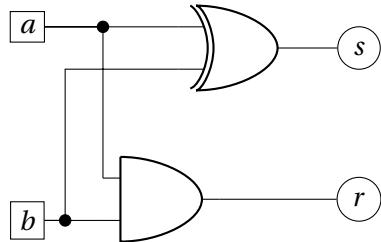


FIGURE 4.4. – Le demi-additionneur.

sorties :  $s$  (bit de poids faible de la somme des trois entrées), et  $r$  (la valeur à reporter pour l'addition suivante, c'est-à-dire le bit de poids fort de la somme des trois entrées).

Ce circuit est donné<sup>13</sup> à la FIGURE 4.5. Pour l'obtenir, on peut observer que faire la somme de trois bits revient à faire la somme de deux d'entre eux (disons,  $a$  et  $b$ ), puis à faire la somme du résultat avec le troisième ( $r_{prec}$ ). Ces deux sommes peuvent être réalisées à l'aide de deux demi-additionneurs : les deux entrées du premier demi-additionneur sont les entrées  $a$  et  $b$ ; et les deux entrées du second demi-additionneur sont : la sortie du premier demi-additionneur et le bit  $r_{prec}$ .

Il reste à expliquer comment combiner les deux reports (notés  $r_1$  et  $r_2$  sur la figure) qui émanent de ces deux sommes en un seul (puisque notre additionneur complet n'a qu'un seul report en sortie). Il est clair que si  $r_1 = r_2 = 0$ , nous aurons également  $r = 0$ . Si  $r_1 = 1$  et  $r_2 = 0$  ou bien si  $r_1 = 0$  et  $r_2 = 1$ , le report  $r$  sera égal à 1 (c'est la somme des deux reports). Le cas  $r_1 = r_2 = 1$  semble plus problématique car la somme de ces deux reports tient maintenant sur 2 bits. Néanmoins, on peut facilement se convaincre que ce cas ne se présentera jamais. En effet, en consultant la table du demi-additionneur ci-dessus, on voit que  $r_1 = 1$  ne peut avoir lieu que si  $a = b = 1$ , auquel cas la sortie du premier XOR est égale à  $a \text{XOR } b = 0$ . De ce fait,  $r_2 = 0$  également, car  $r_2 = (a \text{XOR } b) \wedge r_{prec} = 0 \wedge r_{prec} = 0$ . On conclut qu'il faut combiner  $r_1$  et  $r_2$  à l'aide d'une porte qui renvoie 0 si  $r_1 = r_2 = 0$ , et 1 si un des deux reports vaut 1. Un **ou** correspond à cette spécification (un XOR également).

Ce circuit peut également être obtenu en suivant une méthode plus « classique » : commencer par construire la table de vérité, puis en extraire les formules pour  $s$  et  $r$  qu'on peut simplifier pour obtenir le même résultat. La table est la suivante :

$a$	$b$	$r_{prec}$	$r$	$s$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

13. Voir aussi le fichier Logisim `Add1bit.circ` sur l'UV.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

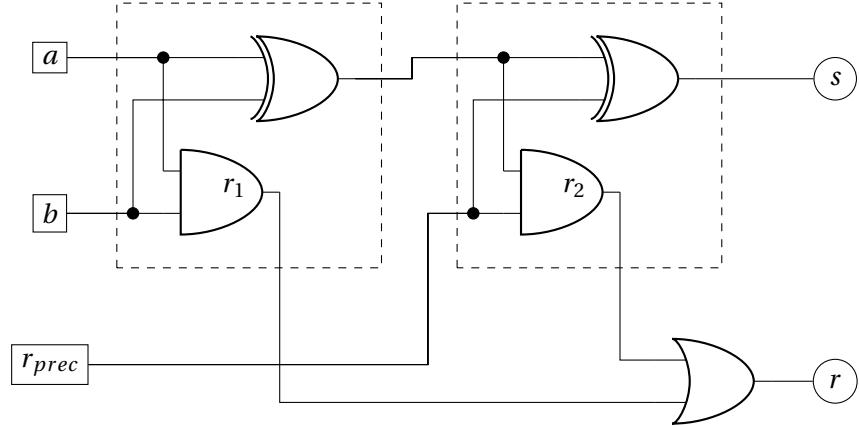


FIGURE 4.5. – Un additionneur complet, sur base de deux demi-additionneurs (rectangles pointillés).

On a donc :

$$\begin{aligned}
 s &= (\neg a \wedge \neg b \wedge r_{prec}) \vee (\neg a \wedge b \wedge \neg r_{prec}) \vee (a \wedge \neg b \wedge \neg r_{prec}) \vee (a \wedge b \wedge r_{prec}) \\
 &= ((\neg a \wedge \neg b) \vee (a \wedge b)) \wedge r_{prec} \vee ((\neg a \wedge b) \vee (a \wedge \neg b)) \wedge \neg r_{prec} \\
 &= ((\neg a \wedge \neg b) \vee (a \wedge b)) \wedge r_{prec} \vee (a \text{ XOR } b \wedge \neg r_{prec}) \\
 &= (\neg(a \text{ XOR } b) \wedge r_{prec}) \vee (a \text{ XOR } b \wedge \neg r_{prec}) \\
 &= a \text{ XOR } b \text{ XOR } r_{prec}.
 \end{aligned}$$

Pour  $r$  on a :

$$\begin{aligned}
 r &= (\neg a \wedge b \wedge r_{prec}) \vee (a \wedge \neg b \wedge r_{prec}) \vee (a \wedge b \wedge \neg r_{prec}) \vee (a \wedge b \wedge r_{prec}) \\
 &= (\neg a \wedge b \wedge r_{prec}) \vee (a \wedge \neg b \wedge r_{prec}) \vee ((a \wedge b \wedge (r_{prec} \vee \neg r_{prec})) \\
 &= (\neg a \wedge b \wedge r_{prec}) \vee (a \wedge \neg b \wedge r_{prec}) \vee (a \wedge b) \\
 &= (r_{prec} \wedge ((\neg a \wedge b) \vee (a \wedge \neg b))) \vee (a \wedge b) \\
 &= (r_{prec} \wedge a \text{ XOR } b) \vee (a \wedge b)
 \end{aligned}$$

**Additionneur  $n$  bits** Finalement, on obtient un additionneur sur  $n$  bits (c'est-à-dire un circuit additionnant deux nombres sur  $n$  bits) en combinant plusieurs additionneurs sur un bit, et en connectant correctement les reports. Un exemple sur 4 bits<sup>14</sup> est donné à la FIGURE 4.6. Chacun des carré en gras représente un additionneur complet avec ses deux entrées  $a$  et  $b$  au-dessus, son entrée  $r_{prec}$  à droite, sa sortie  $r$  à gauche et sa sortie  $s$  en-dessous. L'entrée  $r_{prec}$  du premier additionneur (c'est-à-dire celui qui fait la somme des bits de poids faibles  $a_0$  et  $b_0$ ) est connecté à la constante 0. On aurait donc pu utiliser un demi-additionneur en

14. Voir aussi le fichier Logisim `Add3bits.circ` sur l'UV.

### 4.3. Circuits pour réaliser l'arithmétique binaire

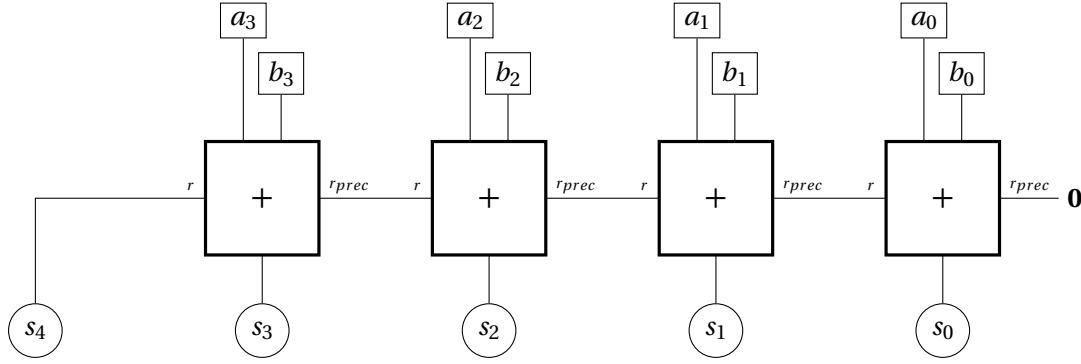


FIGURE 4.6. – Un additionneur 4 bits sur base de 4 additionneurs.

lieu et place de cet additionneur complet. La sortie  $r$  de l'additionneur correspondant aux bits de poids forts constitue le bit de poids fort de la somme.

#### 4.3.3. Décalage

Le circuit de décalage doit réaliser un décalage d'un bit soit vers la droite soit vers la gauche, pour obtenir respectivement une division par 2 ou une multiplication par 2. Pour l'exemple, nous nous contenterons de nombres de 3 bits. Ce circuit aura donc 4 entrées :

- les entrées  $D_2, D_1, D_0$  qui sont les trois bits du nombre à décaler.  $D_2$  est le bit de poids fort et donc « le plus à gauche »; et
- une entrée  $C$  qui indique dans quel sens effectuer le décalage : vers la droite si  $C$  vaut 1, vers la gauche si  $C$  vaut 0.

Ce circuit aura 3 sorties, à savoir les 3 bits  $S_0, S_1, S_2$  du nombre en sortie (à nouveau,  $S_2$  est le bit de poids fort et donc « le plus à gauche ». Si le décalage a lieu vers la droite, on aura  $S_2 = 0$ ; et si le décalage a lieu vers la gauche, on  $S_0 = 0$ .

En d'autres termes, les sorties du circuit de décalage doivent se comporter comme suit :

- si  $C = 0$  (décalage vers la gauche), on veut  $S_2 = D_1, S_1 = D_0$  et  $S_0 = 0$ ; par contre
- si  $C = 1$  (décalage vers la droite), on veut  $S_2 = 0, S_1 = D_2$  et  $S_0 = D_1$ .

On voit donc que les sorties ne dépendent pas toutes des 4 entrées. Plus précisément :

- $S_0$  ne peut prendre que la valeur constante 0 ou la valeur  $D_1$  (en fonction de  $C$ ), et ne dépend donc pas ni de  $D_0$  ni de  $D_2$ . La table de vérité pour  $S_0$  est donc :

$C$	$D_1$	$S_0$
0	0	0
0	1	0
1	0	0
1	1	1

De cette table, on déduit que :  $S_0 = C \wedge D_1$ .

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

- Par un raisonnement similaire, on voit que  $S_2$  ne dépend que de  $C$  et de  $D_1$  également :

$C$	$D_1$	$S_2$
0	0	0
0	1	1
1	0	0
1	1	0

On voit donc que  $S_2 = \neg C \wedge D_1$ .

- Finalement, le cas de  $S_1$  est un peu plus complexe, puisqu'elle doit recopier soit la valeur de  $D_0$ , soit la valeur de  $D_2$ , en fonction de  $C$ . On a donc la table de vérité suivante :

$C$	$D_0$	$D_2$	$S_1$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

En appliquant la méthodologie de la Section 4.1, on trouve une expression simplifiée pour  $S_1$  :

$$\begin{aligned} S_1 &= (\neg C \wedge D_0 \wedge \neg D_2) \vee (\neg C \wedge D_0 \wedge D_2) \vee (C \wedge \neg D_0 \wedge D_2) \vee (C \wedge D_0 \wedge D_2) \\ &= (\neg C \wedge D_0 \wedge (\neg D_2 \vee D_2)) \vee (C \wedge D_2 \wedge (D_0 \vee \neg D_0)) \\ &= (\neg C \wedge D_0) \vee (C \wedge D_2) \end{aligned}$$

On voit bien que  $C$  sert à sélectionner la valeur d'entrée qui détermine la valeur de  $S_1$ . Si  $C = 0$ , la parenthèse  $(C \wedge D_2)$  vaudra 0 (*cfr.* équation (4.2)) et la parenthèse  $(\neg C \wedge D_0)$  vaudra  $(1 \wedge D_0) = D_0$  (*cfr.* équation (4.1)). L'expression vaudra dès lors  $0 \vee D_0 = D_0$  (*cfr.* équation (4.4)). Symétriquement, quand  $C = 1$ , l'expression ci-dessus vaut  $D_2$ .

Le circuit est donné<sup>15</sup> à la FIGURE 4.7.

**Extensions** Sur base de ce circuit, on peut aisément généraliser à des entrées de plus de 3 bits, et à un décalage de plus d'une position. Si on a maintenant  $n$  bits en entrée et en sortie, les équations deviennent :

$$\begin{aligned} S_0 &= C \wedge D_1 \\ S_{n-1} &= \neg C \wedge D_{n-2} \\ S_i &= (\neg C \wedge D_{i-1}) \vee (C \wedge D_{i+1}) \quad \text{pour } 1 \leq i \leq n-2 \end{aligned}$$

---

15. Voir aussi le fichier `Decalage3bits.circ` sur l'UV.

### 4.3. Circuits pour réaliser l'arithmétique binaire

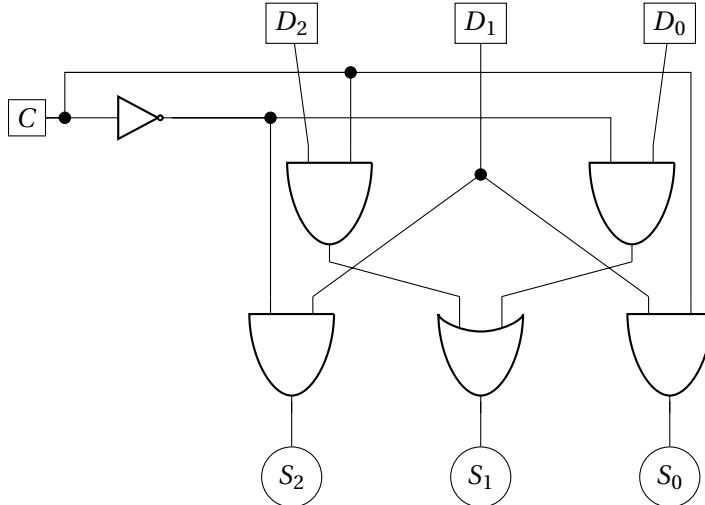


FIGURE 4.7. – Le circuit de décalage 3 bits.

Et si on ajoute un décalage de  $K$  bits (au lieu de 1), on obtient :

$$\begin{aligned} S_i &= C \wedge D_{i+K} && \text{pour } 0 \leq i \leq K-1 \\ S_i &= \neg C \wedge D_{i-K} && \text{pour } n-K \leq i \leq n-1 \\ S_i &= (\neg C \wedge D_{i-K}) \vee (C \wedge D_{i+K}) && \text{pour } K \leq i \leq n-K-1 \end{aligned}$$

#### 4.3.4. Décodeur

Un décodeur  $n$  bits est un circuit qui possède  $n$  entrées  $E_0, \dots, E_{n-1}$  et  $2^n$  sorties  $S_0, \dots, S_{2^n-1}$ , et qui met la sortie  $S_i$  à 1 si et seulement si  $i$  est la valeur  $E_{n-1} \dots E_0$  représentée en binaire sur les entrées. En ce sens, le décodeur « décode » l'information donnée en binaire sur ses entrées, et permet donc de transmettre un choix parmi  $2^n$  en n'utilisant que  $n$  entrées. Un tel circuit a de nombreuses applications, par exemple :

1. pour sélectionner une case dans une mémoire de 4 Go (soit  $2^{32}$  cases d'un octet), on peut se contenter de transmettre à la mémoire l'adresse de cette case, sur 32 bits<sup>16</sup>.
2. Un affichage sept segments, comme on en trouve, par exemple, sur les réveils électroniques (voir FIGURE 4.9) permet d'afficher 10 chiffres différents (de 0 à 9). Ces circuits ont généralement 4 entrées, car 4 bits sont suffisants pour représenter les chiffres de 0 à 9 : la valeur à afficher est transmise au circuit en binaire, la valeur passe à travers un décodeur, et chaque sortie  $S_i$  de celui-ci est connectée aux segments permettant d'afficher la valeur  $i$ .

La table de vérité d'un décodeur 4 bits est la suivante :

---

16. De toute manière, il ne serait pas possible, physiquement, d'avoir plus de 4 milliards d'entrées sur un circuit de mémoire!

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

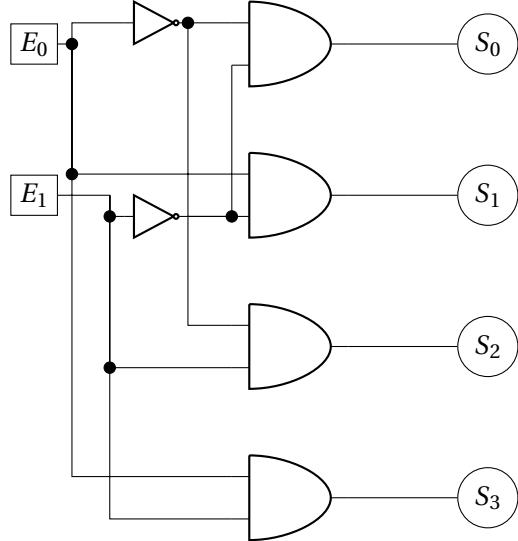


FIGURE 4.8. – Un décodeur 4 bits.

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

On voit donc que :

$$S_0 = \neg E_1 \wedge \neg E_0$$

$$S_1 = \neg E_1 \wedge E_0$$

$$S_2 = E_1 \wedge \neg E_0$$

$$S_3 = E_1 \wedge E_0$$

On obtient dès lors le circuits de la FIGURE 4.8.

De manière générale, pour un décodeur  $n$  bits, la formule correspondant à la sortie  $S_i$  est de la forme :

$$S_i = \bigwedge_{j=1}^n \alpha_j$$

où :

$$\alpha_j = \begin{cases} E_j & \text{Si le } j\text{ème bit de la valeur } i \text{ en binaire vaut 1.} \\ \neg E_j & \text{Sinon.} \end{cases}$$



FIGURE 4.9. – La valeur 102 sur trois afficheurs 7 segments : on peut se contenter d'avoir 4 bits en entrée de chaque chiffre.

#### 4.3.5. Le « sélecteur »

Ce que nous appelons *sélecteur* n'est pas vraiment un circuit en soi, mais plutôt une construction qui sera utile pour la suite. Le problème posé est le suivant : on souhaite un (morceau de) circuit qui possède :

1.  $n$  entrées appelées  $I_{n-1}, \dots, I_0$ ; et
2.  $2^n$  entrées appelées  $E_i$  (pour  $0 \leq i \leq 2^n - 1$ ); ainsi que
3. 1 seule sortie  $S$ .

Ce circuit doit recopier sur sa sortie  $S$  l'entrée  $E_i$  dont le numéro  $i$  est donné en binaire sur  $I_{n-1} \cdots I_0$  (d'où le nom de « sélecteur » : les entrées  $I_i$  servant à sélectionner une entrée  $E_j$ ).



Par exemple, si  $n = 3$ , et que  $I_2 = 0$ ,  $I_1 = 1$  et  $I_0 = 1$ , le numéro de l'entrée sélectionnée est  $011_2 = 3_{10}$ , et le circuit doit donc faire en sorte que  $S = E_3$  (sans que les autres entrées n'aient d'influence sur la sortie).

Le circuit du sélecteur est donné à la FIGURE 4.10. Comme on le voit, on commence par utiliser un décodeur (voir FIGURE 4.8) pour décoder les entrées  $I_0, I_1, \dots, I_{n-1}$ . Il y a donc, à la sortie du décodeur, une sortie que nous appelons  $S_i$  correspondant à chaque entrée  $E_i$  (pour  $0 \leq i \leq 2^n - 1$ ). N'oublions pas qu'une et une seule de ces sorties  $S_i$  sera à mise à 1 ! On combine ensuite, à l'aide d'une porte **et**, chacune des  $S_i$  avec l'entrée  $E_i$  correspondante. Supposons que c'est la valeur  $k$  qui est donnée en binaire sur  $I_{n-1} \cdots I_0$ , on a alors le phénomène suivante :

- $S_k = 1$  (c'est une propriété du décodeur), et la sortie de la porte **et** qui combine  $S_k$  et  $E_k$  vaut donc :

$$\begin{aligned} S_k \wedge E_k &= 1 \wedge E_k \\ &= E_k \quad \text{par (4.1)}; \end{aligned}$$

- pour tout  $j \neq k$ ,  $S_j = 0$ , et la sortie de la porte **et** qui combine  $S_j$  et  $E_j$  vaut donc :

$$\begin{aligned} S_j \wedge E_j &= 0 \wedge E_j \\ &= 0 \quad \text{par (4.2)}. \end{aligned}$$

On voit donc que toutes les portes **et** ont leur sortie à 0, sauf potentiellement celle qui calcule à  $S_k \wedge E_k = E_k$ . La sortie de la porte **ou**, qui donne sa valeur à la sortie du circuit, sera donc :

$$\begin{aligned} (S_0 \wedge E_0) \vee (S_1 \wedge E_1) \vee \cdots \vee (S_k \wedge E_k) \vee \cdots \vee (S_{n-1} \wedge E_{n-1}) &= 0 \vee \cdots \vee E_k \vee \cdots \vee 0 \\ &= E_k \end{aligned}$$

Le circuit présenté à la FIGURE 4.10 réalise donc bien ce qui était attendu.

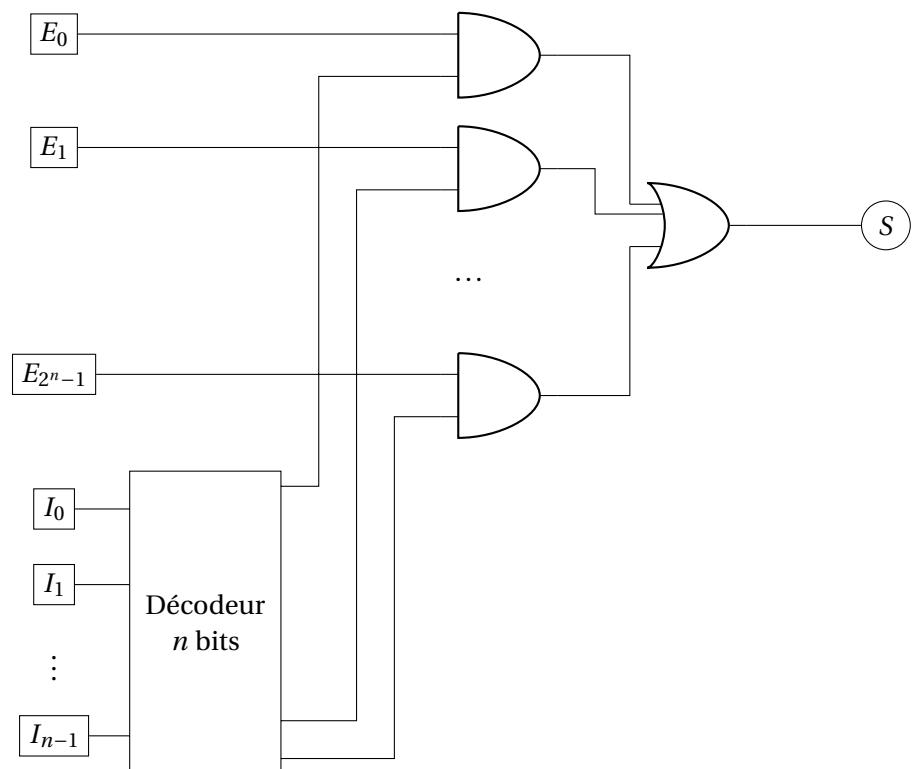


FIGURE 4.10. – Le principe d'un sélecteur  $n$  bits.

TABLE 4.1. – Les 4 opérations de notre exemple d'ALU, ainsi que les entrées qui y correspondent.

$F_1$	$F_0$	opération
0	0	et
0	1	ou
1	0	négation de $B$
1	1	somme de $A$ , $B$ et du report

#### 4.3.6. ALU simplifiée (1 bit)

Sur base des circuits que nous avons vus précédemment, nous pouvons maintenant concevoir une *unité arithméticologique* ou ALU. Ce composant du processeur a été présenté à la section 3.2.

Nous commencerons par considérer une ALU simplifiée qui effectue un opération parmi quatre; et qui effectue l'opération choisie sur deux valeurs d'1 bit. Le principe présenté ici pourra facilement être étendu à plus de 4 opérations, et nous expliquerons plus tard comment combiner plusieurs de ces ALU 1 bit pour obtenir une ALU ayant en entrée des valeurs de taille arbitrairement grande. Les quatre opérations sont données à la TABLE 4.1. Notons que dans le cas où c'est « **non**  $B$  » qui est choisie comme opération, l'entrée  $A$  n'aura pas d'influence sur la sortie, et que l'entrée  $r_{prec}$  n'est utilisée que pour l'addition (avec les autres opérations, elle n'aura, elle non plus, aucune influence sur la sortie). Notre ALU aura donc 5 bits d'entrée :

- 3 bits de données, à savoir : 1 bit pour l'entrée  $A$ , 1 bit pour l'entrée  $B$  et 1 bit de report précédent  $r_{prec}$  (nécessaire pour l'opération d'addition); ainsi que
- les deux bits  $F_0$  et  $F_1$  pour spécifier l'opération à appliquer sur  $A$ ,  $B$  et, le cas échéant,  $r_{prec}$ .

Notre ALU aura également 2 bits de sortie :

- la valeur calculée  $S$ ; et
- le nouveau report  $r_{suiv}$ , dans le cas où l'opération choisie est l'addition (autrement, la valeur sur cette sortie ne sera pas significative).

Pour obtenir l'ALU, nous repartons du circuit du sélecteur (FIGURE 4.10) que nous modifions comme suit :

1. nous l'équipons d'un décodeur 2 bits dont les entrées sont  $F_0$  et  $F_1$ ;
2. nous ajoutons une porte **et**, une porte **ou**, une porte **non** et un additionneur (FIGURE 4.5) pour calculer les 4 opérations demandées; et
3. nous remplaçons les entrées  $E_0, \dots, E_3$  du sélecteur par les sorties respectives de ces portes et de ce circuit (selon la table 4.1).

Le circuit qui en résulte est donné à la FIGURE 4.11.

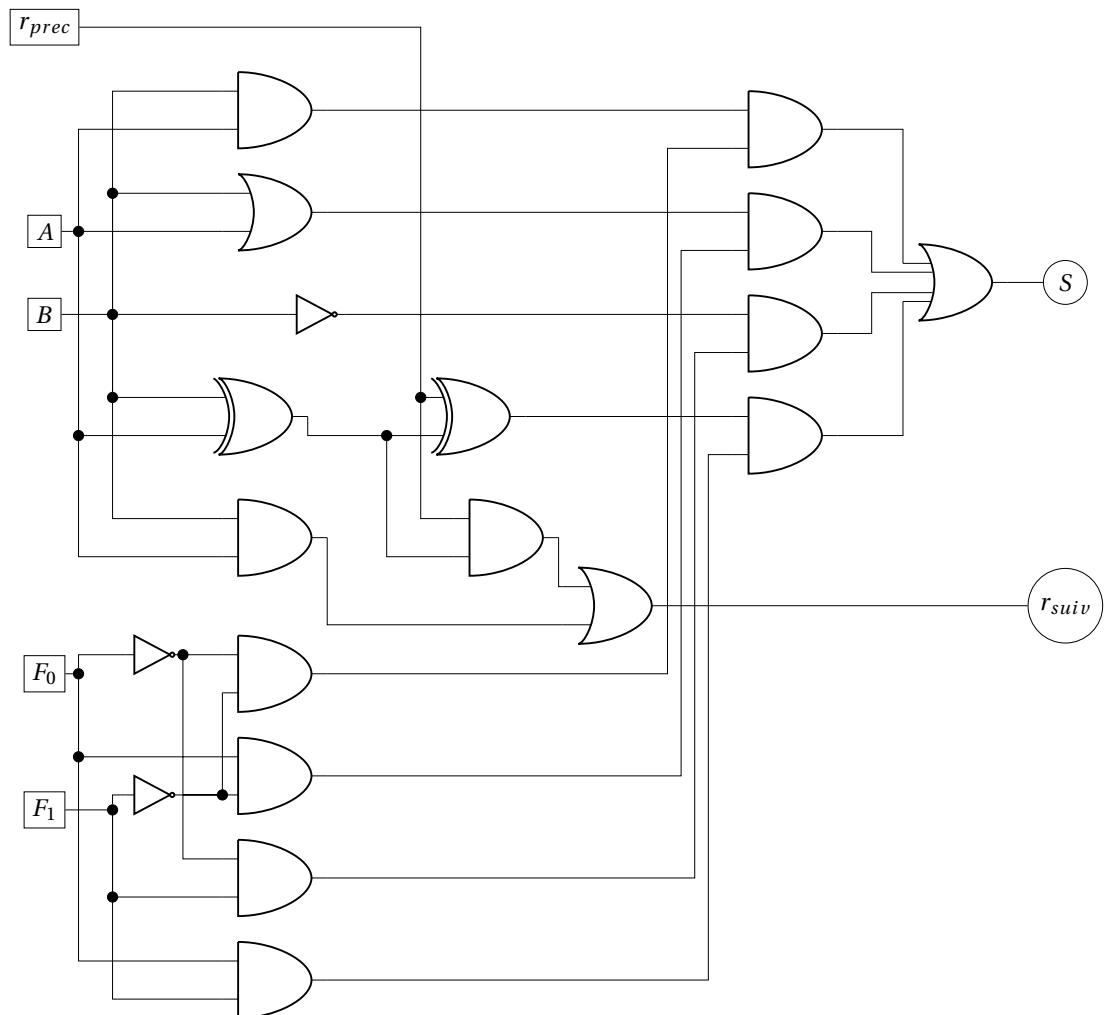


FIGURE 4.11. – Une ALU 1 bit avec 4 opérations.

### 4.3.7. ALU $n$ bits

Expliquons maintenant comment nous pouvons réaliser une ALU  $n$  bits, pour un  $n$  arbitraire (mais fixé). Par exemple, dans un processeur moderne, nous aurons  $n = 32$  ou  $n = 64$  bits. Précisons d'abord ce que nous attendons. Nous souhaitons un circuit avec  $2n+3$  entrées et  $n+1$  sorties. Les  $2n+3$  entrées sont :

- les  $n$  bits  $A_{n-1}, \dots, A_0$  constitutifs de l'entrée  $A$ ;
- les  $n$  bits  $B_{n-1}, \dots, B_0$  constitutifs de l'entrée  $B$ ;
- un bit  $r_{prec}$  constituant le « report précédent » pour l'opération d'addition; et
- les deux bits  $F_0$  et  $F_1$  permettant de sélectionner l'opération à effectuer<sup>17</sup>, selon la TABLE 4.1.

Les  $n+1$  sorties sont :

- les  $n$  bits  $S_{n-1}, \dots, S_0$  représentant la valeur de sortie calculée par l'ALU; et
- le bit  $r_{suis}$  constituant le dernier report dans le cas de l'addition.

La sémantique des opérations est la suivante :

1. Le opérations Booléennes **et**, **ou** et **non** s'appliquent *bit à bit*, c'est à dire que, pour tout  $0 \leq i \leq n-1 : S_i = A_i \wedge B_i$ , ou  $S_i = A_i \vee B_i$  ou  $S_i = \neg B_i$  en fonction de l'opération choisie.
2. L'addition doit réaliser la somme des nombres  $A_{n-1}A_{n-2}\cdots A_0$ ,  $B_{n-1}B_{n-2}\cdots B_0$  et  $r_{prec}$  donnés en binaire (complément à deux). Autrement dit,  $A_0$  et  $B_0$  sont les bits de poids faibles des entrées  $A$  et  $B$ . Le résultat sera représenté par  $r_{suis}S_{n-1}S_{n-2}\cdots S_0$ . Autrement dit,  $S_0$  sera le bit de poids faible de la sortie, et  $r_{suis}$  le bit de poids fort.



Par exemple, pour une ALU 4 bits, si  $A_3 = 1$ ,  $A_2 = 0$ ,  $A_1 = 0$ ,  $A_0 = 0$ ,  $B_3 = 1$ ,  $B_2 = 0$ ,  $B_1 = 1$ ,  $B_0 = 0$  et  $r_{prec} = 1$ , on réalise la somme de  $A = 1000_2 = 8_{10}$ ,  $B = 1010_2 = 12_{10}$  et  $r_{prec} = 1$ . Le résultat  $21_{10} = 10101_2$  tient bien sur 5 bits, on veut donc avoir  $r_{suis} = 1$ , et  $S = 0101$ , soit  $S_3 = 0$ ,  $S_2 = 1$ ,  $S_1 = 0$  et  $S_0 = 1$ .

Nous pouvons maintenant expliquer comment réaliser une ALU  $n$  bits à l'aide de  $n$  ALU 1 bit (telles que décrites à la FIGURE 4.11). L'idée sera similaire à celle que nous avons exploité pour réaliser un additionneur  $n$  bits sur base de  $n$  additionneurs 1 bit (voir FIGURE 4.6). Commençons par fixer une représentation pour une ALU 1 bit : nous utiliserons la FIGURE 4.12, où on retrouve les différentes entrées et sorties de la FIGURE 4.11.

En utilisant cette convention, le schéma de l'ALU 4 bits (qui peut être généralisé à toute taille  $n$  des entrées) est donnée à la FIGURE 4.13. Elle consiste en 4 ALUs 1 bit  $ALU_0$ ,  $ALU_1$ ,  $ALU_2$  et  $ALU_3$  disposées en série. Attention, il faut bien noter que les bits de poids faibles sont cette fois-ci à gauche du schéma et non pas à droite comme d'habitude. Pour tout  $i$ , l'ALU numéro  $i$  se charge de calculer  $S_i$  (et  $r_{suis}$  quand  $i = 3$ ) sur base de  $A_i$  et  $B_i$ , et du report précédent calculé par l'ALU numéro  $i-1$  (ou  $r_{prec}$  si  $i = 0$ ). À noter que toutes les ALUs appliquent la même opération sur les bits d'entrée car les entrées  $F_0$  et  $F_1$  sont communes.

17. Nous continuons à considérer une ALU à 4 opérations. Il est bien entendu possible d'étendre le nombre d'opérations possibles : cela demandera potentiellement plus de bits d'entrée et un décodeur adapté.

4. Leçons 6 à 9 – Niveau 0 : portes logiques

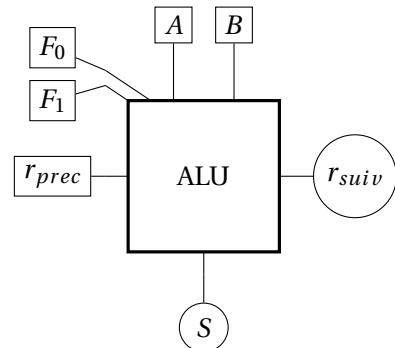


FIGURE 4.12. – La représentation d'une ALU.

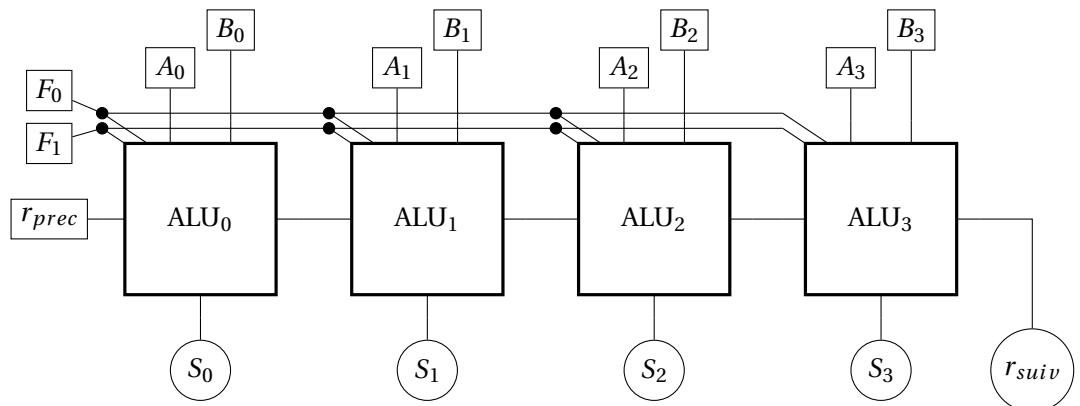


FIGURE 4.13. – Une ALU 4 bits réalisée sur base de 4 ALUs 1 bit. Notez que le bit de poids faible est à gauche et non pas à droite.

**Remarque concernant les reports** On peut se demander quel est l'usage des reports d'entrée et de sortie ( $r_{prec}$  et  $r_{suiv}$ ) dans un ALU, à partir du moment où les registres ont une taille fixe. Par exemple, sur le processeur i486, les registres ont une taille de 32 bits, ce qui sera également la taille des entrées  $A$  et  $B$  de l'ALU. Dans ce cas, si la somme des deux nombres données en entrée requiert 33 bits, nous ne serons de toute manière pas en mesure de la stocker dans un des registres du CPU ; quel sera dès lors le sort du bit  $r_{suiv}$  ?

Une pratique courante (et c'est le cas du processeur i486) consiste à stocker ce bit dans le registre des *flags* du processeur, où il est appelé *carry flag*. Il pourra ainsi être réutilisé pour une addition suivante (comme valeur de  $r_{prec}$ ).

Dans le cas du i486, par exemple, il existe 2 opérations d'addition :

- l'instruction `add` qui ajoute le contenu d'un registre source à un registre destination ; et
- l'instruction `adc` qui ajoute le contenu d'un registre source *et* le contenu du *carry flag* à un registre destination.

Cela permet, par exemple, de réaliser en deux opérations la somme de deux nombres de 64 bits, bien que l'i486 soit un processeur 32 bits. Supposons que les deux nombres à additionner sont dans les registres `eax :ebx` et `ecx :edx`, c'est-à-dire que les 32 bits de poids forts sont respectivement dans `eax` et `ecx`, et les 32 bits de poids faibles dans `ebx` et `edx`. On commence par faire la somme des bits de poids faibles à l'aide de l'instruction `add`, par exemple :

1    `add ebx, edx`

Les 32 bits de poids faible du résultat sont donc placés dans `ebx`, le 33<sup>e</sup> bit dans le *carry flag*. On peut ensuite faire la somme des bits de poids forts et du *carry flag* :

1    `adc eax, ecx`

Le résultat de l'addition sur 64 bits se trouve donc dans `eax :ebx` (avec un éventuel 65<sup>e</sup> bit de poids fort dans le *carry flag*).

## 4.4. Circuits pour réaliser des mémoires

Maintenant que nous avons à notre disposition des circuits pour *traiter* l'information, examinons les circuits permettant de *stocker* l'information, et donc de réaliser des *mémoires*.

### 4.4.1. Mémoire élémentaire

Le plus simple circuit permettant de *stocker une valeur binaire* qu'on puisse imaginer est donné à la FIGURE 4.14. Ce circuit présente plusieurs caractéristiques intéressantes :

1. Tout d'abord, nous constatons que le circuit n'a pas d'entrée. De ce fait, il n'est pas possible de *modifier* la valeur stockée. Ce circuit n'a donc aucune application pratique, mais permet d'illustrer certaines notions intéressantes pour la suite.
2. Ensuite, nous constatons que le circuit est essentiellement une *boucle* : la sortie de chacune des portes est connectée à l'entrée de l'autre. C'est une nouveauté, car les circuits que nous avons considérés jusqu'à présent ne comportaient aucune boucle.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

Dans tous les circuits que nous avons vus précédemment, la ou les sorties dépendaient de manière unique des entrées. Par exemple, pour le demi-additionneur, la sortie  $s$  vaut 0 quand les deux entrées valent 1, et cette sortie ne peut pas prendre d'autre valeur pour ces entrées. Dans les circuits de mémoire, ce phénomène n'est plus vrai (justement en raison des boucles) : pour une même entrée, les sorties peuvent être différentes. Cela peut sembler surprenant, mais c'est en fait une chose heureuse. En effet, une mémoire *retient* l'information qui y a été stockée par le passé, et donc sa sortie (c'est-à-dire la valeur qu'elle contient) ne peut pas, en général dépendre que des entrées à un moment donné, mais également de la valeur qui y a été écrite précédemment. Dans le cas des circuits de mémoire, on s'intéressera donc à la notion d'*état* du circuit, ce que nous allons illustrer à l'aide de l'exemple de la FIGURE 4.14.

Un état du circuit est une fonction qui assigne une valeur à chaque entrée et à chaque sortie. Évidemment, tous les états ne seront pas possibles pour un circuit donné. Par exemple, dans le cas du demi-additionneur, l'état  $a = 1, b = 1, s = 1$  et  $r = 1$  n'est pas possible<sup>18</sup>, car si  $a = b = 1$ , alors  $s = 0$  et  $r = 1$ , puisque  $s = a \text{XOR } b$  et  $r = a \wedge b$  (cf: FIGURE 4.4).

Le circuit de la FIGURE 4.14 n'admet que deux états, à savoir :

$$Q = 1, \bar{Q} = 0$$

$$Q = 0, \bar{Q} = 1.$$

En effet, les équations qui correspondent à ce circuit sont :

$$Q = \neg \bar{Q}$$

$$\bar{Q} = \neg Q,$$

et les deux états ci-dessus sont les deux seules solutions qui satisfont ces équations (on peut facilement vérifier que ni  $Q = \bar{Q} = 0$ , ni  $Q = \bar{Q} = 1$  ne respectent ces égalités). Nous avons donc réussi à créer un circuit qui peut être dans deux états différents, et qui peut donc stocker une valeur parmi deux, soit une valeur binaire (même si nous ne pouvons pas encore modifier cette valeur stockée). On prendra comme convention que la sortie  $Q$  indique la valeur stockée. Autrement dit, quand  $Q = 1$  et  $\bar{Q} = 0$ , le circuit stocke la valeur 1 ; alors que  $Q = 0$  et  $\bar{Q} = 1$  correspond au cas où le circuit stocke la valeur 0.

#### 4.4.2. Bascules

**Bascule simple** Voyons maintenant comment nous pouvons étendre les idées de la section précédente pour obtenir un circuit de mémoire fonctionnel, c'est-à-dire dont on puisse également modifier le contenu. Pour ce faire, nous allons ajouter deux entrées au circuit :

- une entrée  $S$  (pour *set*, en anglais), qui devra forcer l'état du circuit à contenir le bit 1 (autrement dit, qui écrira la valeur 1 dans la mémoire) ; et
- une entrée  $R$  (pour *reset*, en anglais), qui écrira la valeur 0 dans la mémoire.

---

18. À vrai dire, le seul état où  $a = 1$  et  $b = 1$  dans ce circuit est un état où  $s = 0$  et  $r = 1$ . Dans un circuit sans boucle, il n'y aura jamais qu'un seul état qui correspond à des entrées données.

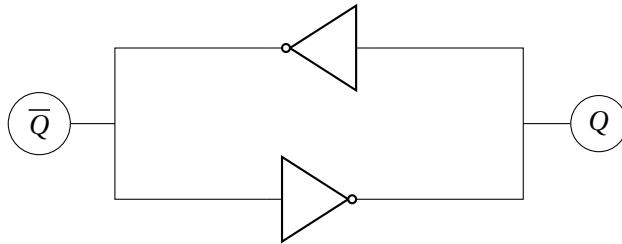


FIGURE 4.14. – Une boucle de portes **non** qui permet de stocker une valeur binaire.

Il est bien entendu qu'il n'y aura pas de sens à mettre ces deux entrées simultanément à 1. On peut donc résumer les 4 possibilités comme suit :

S	R	Effet
0	0	Maintenir la dernière valeur enregistrée
0	1	Écrire 0
1	0	Écrire 1
1	1	Interdit!

Nous garderons les deux sorties  $Q$  et  $\bar{Q}$ , qui seront toujours l'inverse l'une de l'autre. Afin de concevoir ce circuit, nous pouvons nous poser la question suivante :

*Dans quel cas la sortie  $Q$  doit-elle prendre la valeur 1 ?*

Étant donné la discussion ci-dessus, on voit que la sortie  $Q$  doit avoir la valeur 1 si :

1.  $\bar{Q}$  vaut 0 (car cela signifie que la valeur enregistrée par la mémoire était 1); et
2. l'entrée  $R$  vaut 0 (car autrement, il faut forcer  $Q$  à 0).

On remarquera qu'il n'est pas obligatoire d'avoir  $S = 1$  pour avoir  $Q = 1$ . Heureusement ! Autrement, cela signifierait qu'il faut toujours maintenir en entrée la valeur à stocker dans la mémoire, et la mémoire ne « retiendrait » en fait rien, elle se contenterait de recopier sur sa sortie la valeur qu'on lui indique en entrée. Formellement, nous avons donc :

$$\begin{aligned} Q &= \neg \bar{Q} \wedge \neg R \\ &= \neg(\bar{Q} \vee R) \quad \text{Par (4.8)} \\ &= \bar{Q} \text{NOR } R. \end{aligned}$$

Par un raisonnement similaire, on déduit que  $\bar{Q}$  vaut 1 si et seulement si  $Q$  et  $S$  sont faux, autrement dit :  $\bar{Q} = \neg Q \wedge \neg S = Q \text{NOR } S$ . Les deux équations du circuit sont donc :

$$Q = \bar{Q} \text{NOR } R, \tag{4.10}$$

$$\bar{Q} = Q \text{NOR } S. \tag{4.11}$$

Ce circuit est représenté<sup>19</sup> à la FIGURE 4.15. On l'appelle *bascule ou bistable*.

Comme dans le cas du circuit de la FIGURE 4.14, regardons maintenant quels sont les *états* possibles de ce circuit :

19. Voir aussi le fichier `Bistable.circ` sur l'UV.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

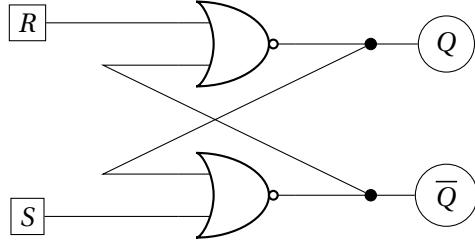


FIGURE 4.15. – Une bascule.

- Si  $S = 1$  et  $R = 0$ , les équations (4.10) et (4.11) donnent respectivement :

$$\begin{aligned}
 Q &= \overline{\overline{Q}} \text{ NOR0} \\
 &= \neg(\overline{Q} \vee 0) \\
 &= \neg\overline{Q} \quad \text{Par (4.4),}
 \end{aligned}$$

et :

$$\begin{aligned}
 \overline{Q} &= Q \text{ NOR1} \\
 &= \neg(Q \vee 1) \\
 &= \neg(1) \quad \text{Par (4.3)} \\
 &= 0.
 \end{aligned}$$

On en déduit donc que  $\overline{Q} = 0$  et que  $Q = \neg\overline{Q} = \neg 0 = 1$ . L'entrée  $S$  a donc bien l'effet attendu<sup>20</sup>, car le seul état possible quand  $S = 1$  et  $R = 0$  est celui où la mémoire stocke la valeur 1 ( $Q = 1$ ).

- Si  $S = 0$  et  $R = 1$ , on déduit, par un raisonnement similaire que  $\overline{Q} = 1$  et  $Q = 0$ . L'entrée « *reset* » a donc également l'effet attendu.
- Finalement, si  $S = 0$  et  $R = 0$ , la bascule est censée conserver la dernière valeur écrite. En remplaçant  $R$  par 0 dans l'équation (4.10), on obtient :

$$Q = \neg\overline{Q} \quad (4.12)$$

comme ci-dessus. En remplaçant  $S$  par 0 dans l'équation (4.11), on a :

$$\begin{aligned}
 \overline{Q} &= Q \text{ NOR0} \\
 &= \neg(Q \vee 0) \\
 &= \neg Q,
 \end{aligned}$$

ce qui est la même équation que (4.12). Il y a deux paires de valeurs pour  $Q$  et  $\overline{Q}$  qui satisfont cette équation, et il y a donc *deux* états<sup>21</sup> qui correspondent au cas  $S = R = 0$  :

20. À condition que  $R = 0$ , bien entendu.

21. Ce qui explique le nom de « *bistable* » : quand  $S = R = 0$ , il y a deux états admissibles, c'est-à-dire stables pour le circuit. Il en va de même avec une bascule (c'est-à-dire cette sorte de balançoire constituée d'une grande planche en bois placée en équilibre en son centre, et sur laquelle on s'assied de part et d'autre).

celui où  $Q = 0$  et  $\bar{Q} = 1$  (la mémoire stocke 0), et celui où  $\bar{Q} = 0$  et  $Q = 1$  (la mémoire stocke 1). C'est bien l'effet attendu : quand  $S = R = 0$ , il y a deux valeurs possibles pour les sorties, valeurs qui dépendent du *passé*. La mémoire stocke donc bien la dernière valeur.

**Utilité d'une horloge** Un des inconvénients du circuit que nous venons d'étudier que est toute modification des entrées  $S$  et  $R$  a immédiatement un effet sur les valeurs stockées dans la mémoire. En pratique, cela peut être problématique, car les valeurs injectées aux entrées  $S$  et  $R$  pourraient être le résultat d'un calcul effectué par un autre circuit (une ALU, par exemple). Or, la propagation des valeurs dans un circuit logique n'est, en réalité, pas instantanée. Il se pourrait donc que, durant une fraction de seconde, la valeur entrée de la bascule ne soit pas correcte, ce qui pourrait mener à l'effacement ou la corruption de la donnée stockée.

Autrement dit, nous devons trouver un mécanisme qui permette d'indiquer à la mémoire *à quel moment* une nouvelle donnée doit être enregistrée. Les trois circuits que nous nous apprêtons à présenter sont essentiellement un raffinement de la bascule, auquel nous ajoutons une entrée appelée *horloge* (indiquée par  $\neg$ ), qui signale, d'une manière ou d'une autre, le ou les moments où la mémoire doit mémoriser une nouvelle donnée.

**Bascule avec horloge** La première idée consiste à faire en sorte que la bascule n'enregistre le changement d'état qu'aux seuls moments où l'horloge est à 1. Ainsi, on peut avoir le schéma de fonctionnement suivant :

- pendant que l'horloge est à l'état bas, les entrées de la bascule peuvent changer (en fonction des calculs qui sont effectués par un circuit en amont), sans que cela n'influe le contenu de la bascule; puis
- les entrées de la bascule finissent par se stabiliser; ensuite
- l'horloge passe à l'état 1, la bascule enregistre l'information; et finalement
- l'horloge repasse à l'état 0 : l'information est « gelée » durant toute cette période.

Ce fonctionnement est obtenu simplement en connectant les deux entrées, *via* deux portes **et**, à l'horloge. Ainsi, quand l'horloge est à l'état bas, les entrées sont annihilées pour la bascule. Ce nouveau circuit est présenté à la FIGURE 4.16<sup>22</sup>.

**Bascule D** Sur le circuit de la FIGURE 4.16, on observe que, si l'horloge est à l'état bas (elle vaut 0), les entrées des deux portes NOR sont à 0, et donc, la bascule mémorise la dernière information enregistrée. Il n'y a donc plus d'utilité à permettre  $S = R = 0$ , et on peut se permettre d'avoir uniquement les deux paires d'entrées suivantes :  $S = 1$  et  $R = 0$  d'une part; et  $S = 0$  et  $R = 1$  d'autre part. Ainsi :

- si on souhaite que la mémoire mémorise la dernière valeur inscrite, on mettra l'horloge à 0 (et les valeurs de  $S$  et  $R$  n'importent pas);
- si on souhaite enregistrer un 1, on mettre  $S = 1$ ,  $R = 0$  et l'horloge à 1;
- si on souhaite enregistrer un 0, on mettre  $S = 0$ ,  $R = 1$  et l'horloge à 1.

---

22. Voir aussi le fichier Logisim `BistableAvecHorloge.circ` sur l'UV.

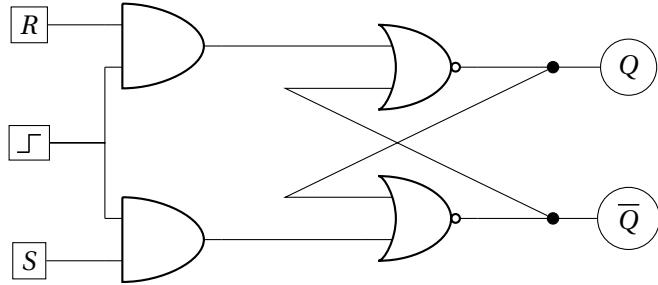


FIGURE 4.16. – Une bascule avec horloge.

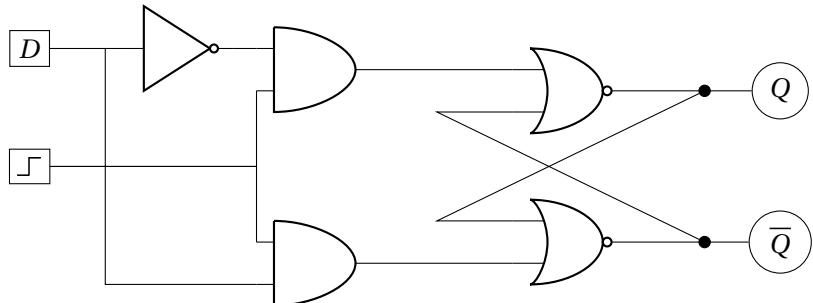


FIGURE 4.17. – Une bascule *D*.

Mais comme maintenant l'entrée *S* est toujours l'inverse de l'entrée *R*, on peut se contenter d'une seule entrée *D* (pour *data*, en anglais), telle que :  $S = D$  et  $R = \neg D$ . Ce circuit est donné<sup>23</sup> à la FIGURE 4.17.

#### 4.4.3. Flip-flops

Bien que la bascule *D* permette d'isoler dans le temps une période où la mémoire va enregistrer la donnée (celle où l'horloge est à 1), il faut encore maintenir une entrée stable pendant tout ce laps de temps. En pratique, cela peut s'avérer difficile, et on souhaiterait avoir un circuit de mémoire dans lequel l'information serait mémorisée à un instant ponctuel (et non plus durant une période relativement longue). C'est l'objectif des *flip-flops*, qui sont des bascules de type *D* à horloge, mais dont l'état change (en fonction de *D*), au moment précis où l'horloge change d'état. Ainsi, tant que l'horloge reste à une valeur stable (que ce soit 0 ou 1), la mémoire maintient son état, quels que soient les changements de *D*. Quand l'horloge change d'état, l'ordre d'écriture indiqué sur *D* est pris en compte.

**Générateur de pulsation** Pour réaliser une flip-flop, on peut utiliser une bascule et intercaler, entre l'horloge et la bascule, un *générateur de pulsation*. Il s'agit d'un dispositif qui génère une pulsation brève lors de certains changements d'états de l'horloge. La sortie de ce générateur va donc se comporter comme une horloge qui passe très rapidement de 0 à 1 et

---

23. Voir aussi le fichier Logisim Bistable-D-AvecHorloge.circ sur l'UV.

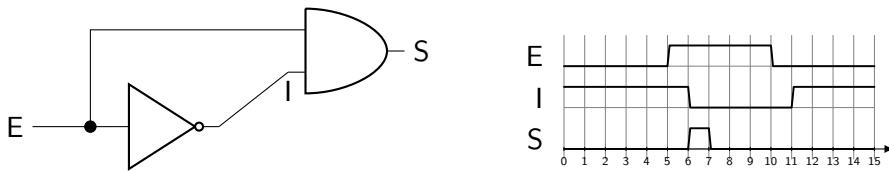


FIGURE 4.18. – Un générateur de pulsation : le délai de propagation dans la porte « et » va générer une sortie à vrai pendant un court instant (on suppose que le délai des portes est d'une unité de temps).

de 1 à 0 lorsque son entrée change.

Dans ce cours, nous allons considérer un générateur de pulsation qui s'active sur les *flancs montants* de l'horloge, c'est-à-dire *quand l'horloge passe de 0 à 1*. Concrètement, cela peut être réalisé comme montré à la FIGURE 4.18. Ce circuit semble avoir peu de sens, car une des deux entrées de la porte **et** aura toujours la valeur 0, et donc la sortie du **et** semble être toujours nulle. Cette conclusion est vraie dans une situation *stable* (c'est-à-dire quand l'entrée du circuit n'a plus changé depuis un certain temps). Par contre, au moment où l'entrée du circuit change, on passe par un régime transitoire qui s'explique par le fait que la porte **non** et la porte **et** (comme toutes les portes logiques) prennent en réalité un certain temps (très court) pour produire leur effet.

Dans notre cas, on supposera que les portes logiques prennent une unité de temps (voir le diagramme sur la droite de la FIGURE 4.18) pour produire leur effet. On voit donc que l'entrée E est à 0 jusqu'au temps 5, moment où elle passe à vrai. En tenant compte du délai de propagation, la sortie de la porte **non** (connectée à l'entrée I de la porte **et**) est à 1 jusqu'au temps 6, moment où elle passe à 0. Les deux entrées du **et** sont donc à 1 entre le temps 5 et le temps 6. La porte **et** (qui souffre du même délai) produit donc une sortie à 1 du temps 6 au temps 7, ce qui constitue la *pulsation* désirée<sup>24</sup>.

**Flip-flop** Le circuit obtenu est donné<sup>25</sup> à la FIGURE 4.19. En pratique, on utilisera la représentation montrée à la FIGURE 4.20 pour désigner une flip-flop ; où on reconnaît les 2 entrées D et horloge, et les deux sorties Q et  $\bar{Q}$ .

#### 4.4.4. Un circuit de mémoire complet

Sur base des *flip-flops*, nous pouvons maintenant présenter un circuit qui réalise toutes les fonctions d'une mémoire classique, à savoir : la lecture et l'écriture à une adresse donnée. Nous nous concentrerons sur un exemple de petite dimension qui peut facilement être étendu à des dimensions plus réalistes : une mémoire capable de stocker 4 valeurs (ou 4 mots, ou 4 cases) de 3 bits chacun. Cette mémoire admettra donc 4 adresse différentes qui seront représentées sur 2 bits. Plus précisément on souhaitera les entrées suivantes :

24. Remarquons que ce phénomène ne peut pas être simulé dans Logisim, mais qu'il existe des flip-flop « tout faits » dans Logisim. Remarquons enfin que dans Logisim, les *flip-flops* sont soit sur *flanc montant* soit sur *flanc descendant* : c'est le passage de 0 à 1 ou de 1 à 0 respectivement qui active la mémorisation.

25. Voir aussi le fichier Logisim *FlipFlop.circ* sur l'UV.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

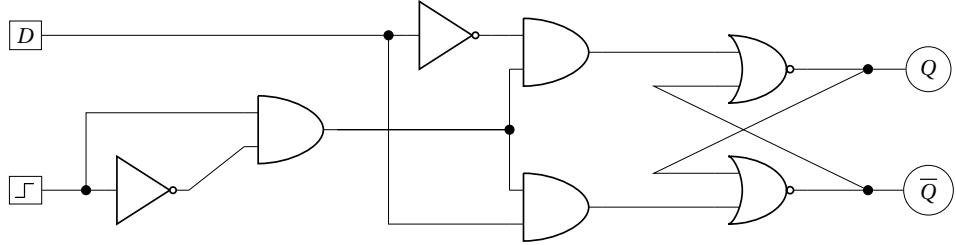


FIGURE 4.19. – Le circuit logique d'une flip-flop.

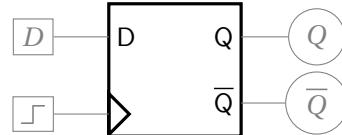


FIGURE 4.20. – Une flip-flop, avec ses deux entrées  $D$  et horloge, et ses deux sorties  $Q$  et  $\bar{Q}$ .

- 3 entrées  $I_0$ ,  $I_1$  et  $I_2$  pour spécifier la valeur (sur 3 bits) à écrire;
- 2 entrées  $A_0$  et  $A_1$  pour spécifier l'adresse (pour lire ou écrire).  $A_1$  est le bit de poids fort de l'adresse, et  $A_0$  le bit de poids faible;
- une entrée  $RD$  que l'on fera passer de 1 à 0 pour déclencher l'écriture de la donnée  $I_2 I_1 I_0$  à l'adresse  $A_1 A_0$ ; et
- une entrée  $CS$  (pour *Chip Select*) qui permet d'inhiber complètement l'écriture quand elle est mise à 0. Cette entrée est utile quand l'ensemble la mémoire est répartie sur plusieurs circuits présents physiquement sur plusieurs circuits intégrés (ou *chips* en anglais). On peut alors activer l'écriture uniquement dans le circuit qui contient l'adresse désirée.

On souhaite également 3 sorties  $O_2$ ,  $O_1$  et  $O_0$  qui donneront, à tout moment le contenu de la case mémoire d'adresse  $A_1 A_0$ , permettant ainsi la lecture.

Pour construire ce circuit, on commence par observer qu'il s'agit d'une mémoire de 12 bits ( $3 \times 4$ ) au total. Chaque bit sera stocké dans une *flip-flop*, nous avons donc au total 12 *flip-flops* dans notre circuit. Pour faciliter la lecture du circuit, nous étiqueterons chaque *flip-flop* par  $(x, a)$ , ce qui signifie que cette *flip-flop* stocke le bit numéro  $x$  de la case à l'adresse  $a$ . Par exemple, la *flip-flop* numéro (0,2) est le bit de poids faible de la case à l'adresse 2.

On peut ensuite concevoir séparément les parties du circuit qui s'occupent de la lecture et l'écriture. Pour la lecture, on a affaire à une nouvelle variation sur le schéma du sélecteur (FIGURE 4.10) : pour chaque bit  $0 \leq i \leq 2$ , la sortie  $O_i$  est obtenue en sélectionnant le contenu d'un des *flip-flops*  $(i, 0)$ ,  $(i, 1)$ ,  $(i, 2)$ , ou  $(i, 3)$  en fonction de l'adresse qui sera préalablement décodée. La FIGURE 4.21 présente le principe général de fonctionnement de la lecture. On reproduira ce principe pour chaque bit  $i$  ( $i \in \{0, 1, 2\}$ ) du contenu d'un case mémoire.

La partie du circuit en charge de l'écriture est très similaire. On commence par réaliser un circuit calculant  $CS \wedge \neg RD$ . Il y aura donc un *flanc montant* à la sortie de ce circuit si et seulement si : (i)  $CS = 1$ ; et (ii) il y a un *flanc descendant* sur  $RD$ . C'est donc la sortie de ce

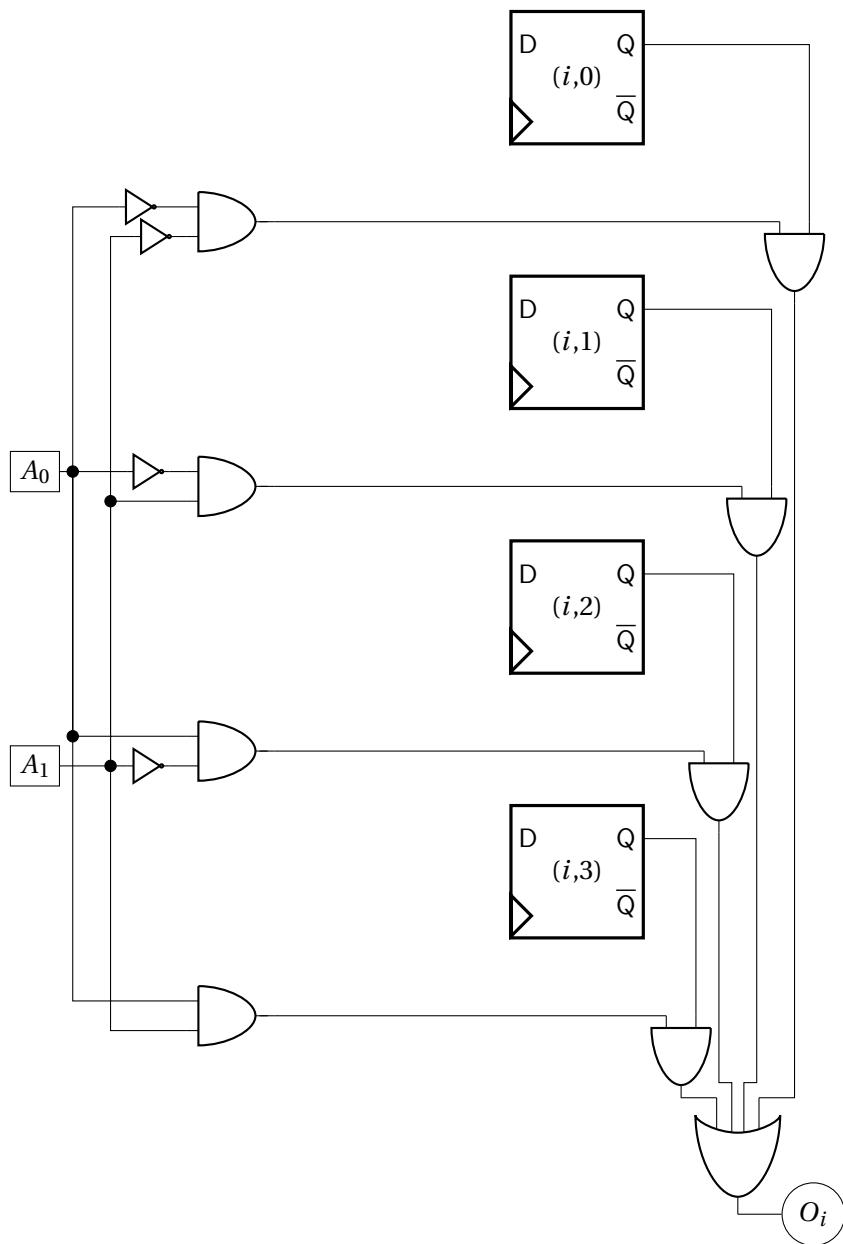


FIGURE 4.21. – Mémoire : lecture du bit  $i$  parmi 4 adresses.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

circuit qui commandera l'écriture dans toutes les *flip-flops* de la case mémoire sélectionnée par l'adresse  $A_1 A_0$ . On opère la sélection de la bonne case mémoire comme dans le cas du sélecteur : en prenant la conjonction de  $CS \wedge \neg RD$  avec chacune des sorties du décodeur. Cela crée dans le circuit quatre « lignes » servant à activer l'écriture, une pour chaque case de la mémoire, que l'on connectera aux entrées *horloge* des *flip-flops* correspondantes. Ainsi, le flanc montant en sortie du circuit  $CS \wedge \neg RD$  ne sera propagé que vers les *flip-flops* de la case d'adresse  $A_1 A_0$ . On n'oubliera pas de connecter l'entrée  $I_i$  aux entrées  $D$  de toutes les *flip-flops*  $(i, j)$  pour  $0 \leq j \leq 3$ . À noter qu'il n'y a pas de problème à connecter la donnée à *toutes* les *flip-flops* puisque c'est l'entrée *horloge* qui commande l'écriture. Le principe général d'écriture du bit  $i$  est illustré à la FIGURE 4.22. Comme dans le cas de l'écriture, on reproduira ce même principe pour chaque bit à écrire.

En combinant ces idées, on obtient le circuit de la mémoire de 4 cases de 3 bits donné à la FIGURE 4.23.



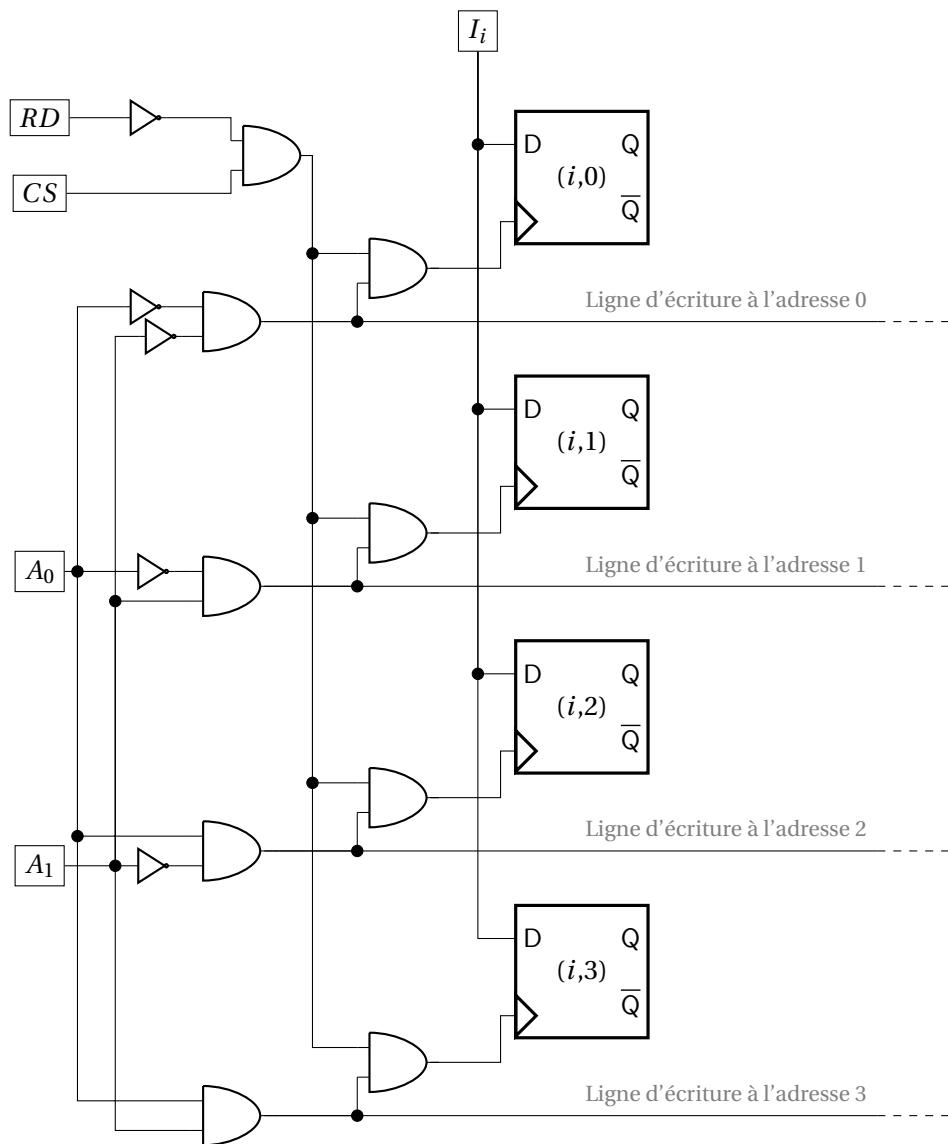


FIGURE 4.22. – Mémoire : écriture d'un bit  $i$  parmi 4 adresses.

#### 4. Leçons 6 à 9 – Niveau 0 : portes logiques

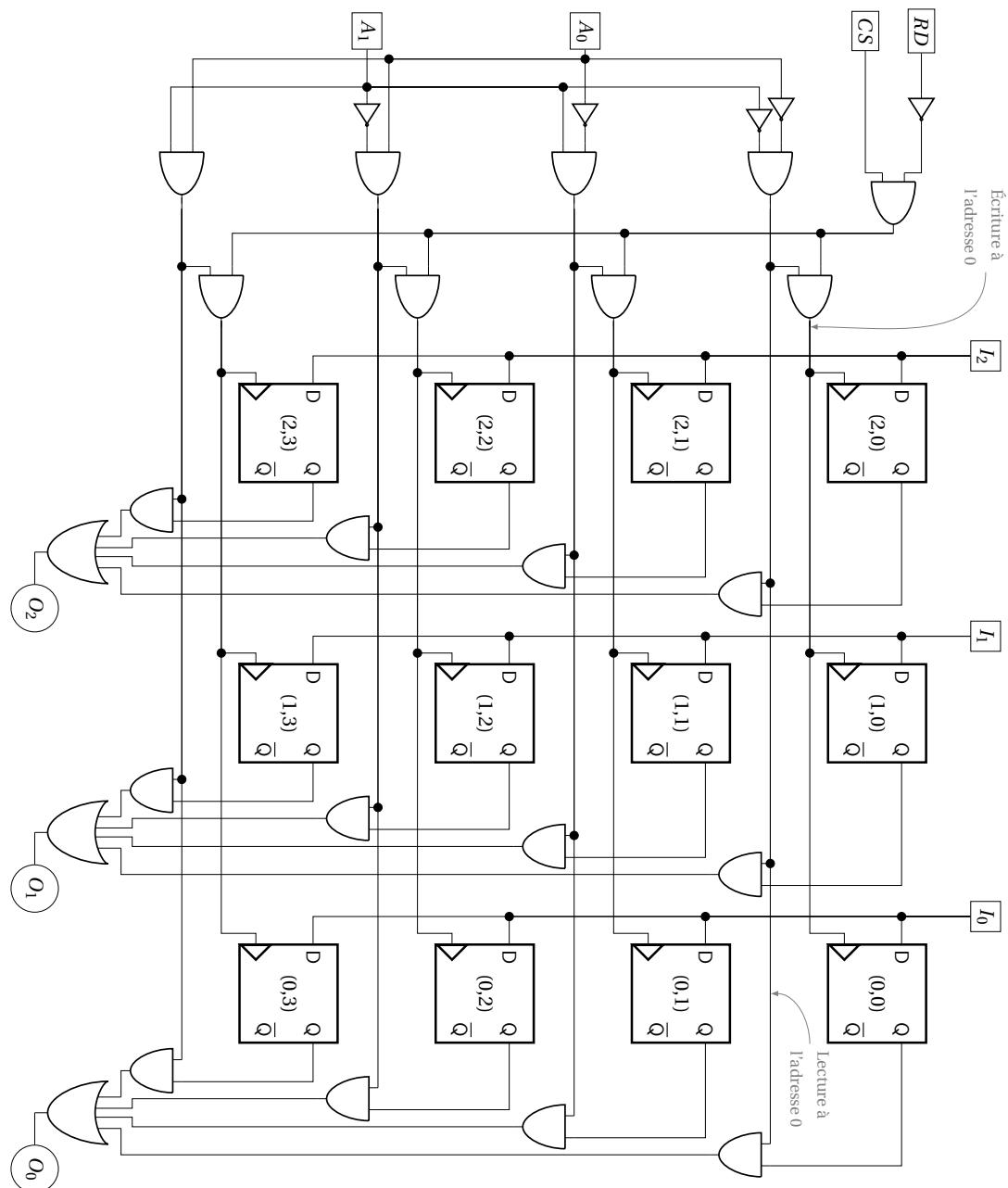


FIGURE 4.23. – Une mémoire de 4 mots de 3 bits.

## **Troisième partie**

### **Le micro-langage**



# 5. Leçons 10 à 12 – La micro-architecture

Le niveau qualifié de « micro-architecture » est le niveau intermédiaire entre les circuits électroniques digitaux et le langage machine, qui est l'ensemble d'instructions que le processeur peut exécuter. Les instructions du langage machine ne seront pas directement traduites chacune en terme de circuits logiques (ce serait trop coûteux, et cela entraînerait une démultiplication des circuits). Chaque instruction du langage machine va être *interprétée* et traduite en une séquence (courte) de *micro-instructions* très simples. Nous allons maintenant étudier un exemple de machine capable d'exécuter un tel jeu de micro-instructions. Il sera facile de se convaincre :

- d'une part que cette machine peut aisément être réalisée à l'aide des circuits digitaux étudiés précédemment.
- d'autre par que ces micro instructions sont suffisantes pour exécuter des instructions machine plus complexes.

L'exemple que nous allons étudier est totalement fictif, mais parfaitement réaliste. Il est inspiré de l'ouvrage d'A. TANENBAUM [22].

## 5.1. Exemple de langage machine : l'IJVM

Il s'agit d'un langage qui est exécuté sur une machine à pile. Cela signifie que le processeur n'a pas de registre *de travail*, mais utilise un *stack* en mémoire centrale, sur laquelle il faut *pusher* les données à traiter.

La mémoire de la machine est organisée en plusieurs zones dont :

- Une zone organisée comme un *stack* pour les opérandes.
- Une zone pour les méthodes (=code en langage machine).

Pour notre exemple, nous allons nous concentrer sur un ensemble restreint d'instructions :

- NOP. Cette instruction ne fait rien. Il est parfois utile de disposer d'une telle instruction, comme nous le verrons par la suite.
- BIPUSH  $v$ , où  $v$  est une valeur sur un octet. *Push* cette valeur au sommet de la pile.
- POP. Cette instruction sans paramètre *pop* le sommet de la pile.
- IADD, ISUB. Ces instructions sans paramètre réalisent respectivement la somme et la différence des deux valeurs au sommet de la pile, *pop* ces deux valeurs, et *push* le résultat à leur place.
- GOTO  $o$ . Cette instruction réalise un saut de  $o$  positions dans le code. Le décalage  $o$  est une valeur sur 2 octets.

## 5. Leçons 10 à 12 – La micro-architecture

- IFEQ  $o$ . Cette instruction *pop* la valeur au sommet de la pile et réalise un saut de  $o$  positions dans le code si et seulement si cette valeur vaut 0. Le décalage  $o$  est une valeur sur 2 octets.

**Remarque** Pour plus de clarté dans ce document, on n'écrira pas explicitement le décalage à effectuer lors d'un GOTO ou d'un IFEQ, mais on utilisera des *étiquettes*. Cela consiste à placer, au début d'une ligne de programme, un nom d'étiquette suivi de « : », par exemple :

```
etiqu: IADD
```

Ce nom doit être unique et identifie donc sans ambiguïté cette ligne.

On pourra alors, si on veut faire un saut vers cette ligne, remplacer le décalage par le nom de l'étiquette :

```
etiqu: IADD
```

```
...
```

```
GOTO etiqu
```

Il est facile d'écrire un petit programme qui collecte les positions de toutes les étiquettes et remplace leurs noms par les bons décalages dans les instructions GOTO et IFEQ.

### 5.1.1. Exemples de programmes en IJVM

Ce premier exemple *push* trois valeurs sur le *stack* et réalise leur somme.

```
BIPUSH 10
BIPSUH 20
BIPUSH 12
IADD
IADD
```

Ce second exemple *pop* les deux valeurs au sommet du *stack*, et *push* la valeur 1 si ces deux valeurs sont égales, 0 sinon :

```
ISUB
IFEQ equ
BIPUSH 0
GOTO fin
equ: BIPUSH 1
fin: NOP
```

### 5.1.2. Représentation d'un programme IJVM en mémoire

Concrètement, les instructions IJVM seront stockées en mémoire sous forme binaire. Afin d'identifier les instructions, nous allons associer un code numérique unique, sur un octet, à chaque instruction. Ce code est appelé *code d'opération*, ou, selon l'acronyme anglais, *opcode*. Les *opcodes* que nous utiliserons dans le cadre du cours sont données à la TABLE 5.1. Par exemple, l'*opcode* de BIPUSH est  $10_{16}$ . Si l'instruction a besoin d'une opérande, celle-ci sera placée en mémoire après l'*opcode*.

## 5.2. Exemple de datapath : le MIC 1

TABLE 5.1. – Les opcodes IJVM que nous utiliserons. La liste complète peut être trouvée dans [22].

Opcode (hexa.)	Instruction	Commentaire
10	BIPUSH $v$	<i>Pushe v sur la pile</i>
A7	GOTO $o$	Saut de $o$ positions : ajoute $o$ à PC
60	IADD	<i>Pope deux valeurs et pushe leur somme</i>
64	ISUB	<i>Pope deux valeurs et pushe leur différence</i>
99	IFEQ $o$	<i>Pope du stack et saute de <math>o</math> positions si la valeur est nulle</i>
57	POP	<i>Pope du stack</i>

**Exemple** Notre premier programme d'exemple sera stocké ainsi en mémoire (toutes les valeurs sont en hexadécimal – les espaces servent à identifier les instructions) :

10 0A 10 14 10 0C 60 60

Le second programme d'exemple est stocké de la façon suivante :

64 99 00 08 10 01 A7 00 05 10 00 00

Remarque : les décalages du GOTO et IFEQ sont à comprendre par rapport à l'adresse de l'*opcode*, comme on le voit sur cet exemple.

Nous allons maintenant étudier la machine qui devra exécuter ces instructions machine.

## 5.2. Exemple de datapath : le MIC 1

Le cœur de la machine qui va exécuter les micro-instructions est le *datapath* qui se compose, comme on le voit sur la FIGURE 5.1 :

- D'un ensemble de registres : MAR, MDR, PC, MBR, SP, TOS, OPC et H. Tous les registres ont 32 bits sauf le registre MBR qui fait 8 bits. L'utilité de ces registres sera expliquée plus loin.
- D'une ALU qui a plusieurs entrées :
  - Les valeurs A et B sur lesquelles effectuer les opérations.
  - Une sortie de la valeur calculée.
  - Deux sorties additionnelles N et Z.
  - 6 bits pour spécifier l'opération à effectuer.

Cette ALU est capable (entre autres) de renvoyer, étant donné deux valeurs d'entrée  $A$  et  $B$ , les résultats suivants :

1.  $A$ ;
2.  $B$ ;
3.  $A + B$ ;

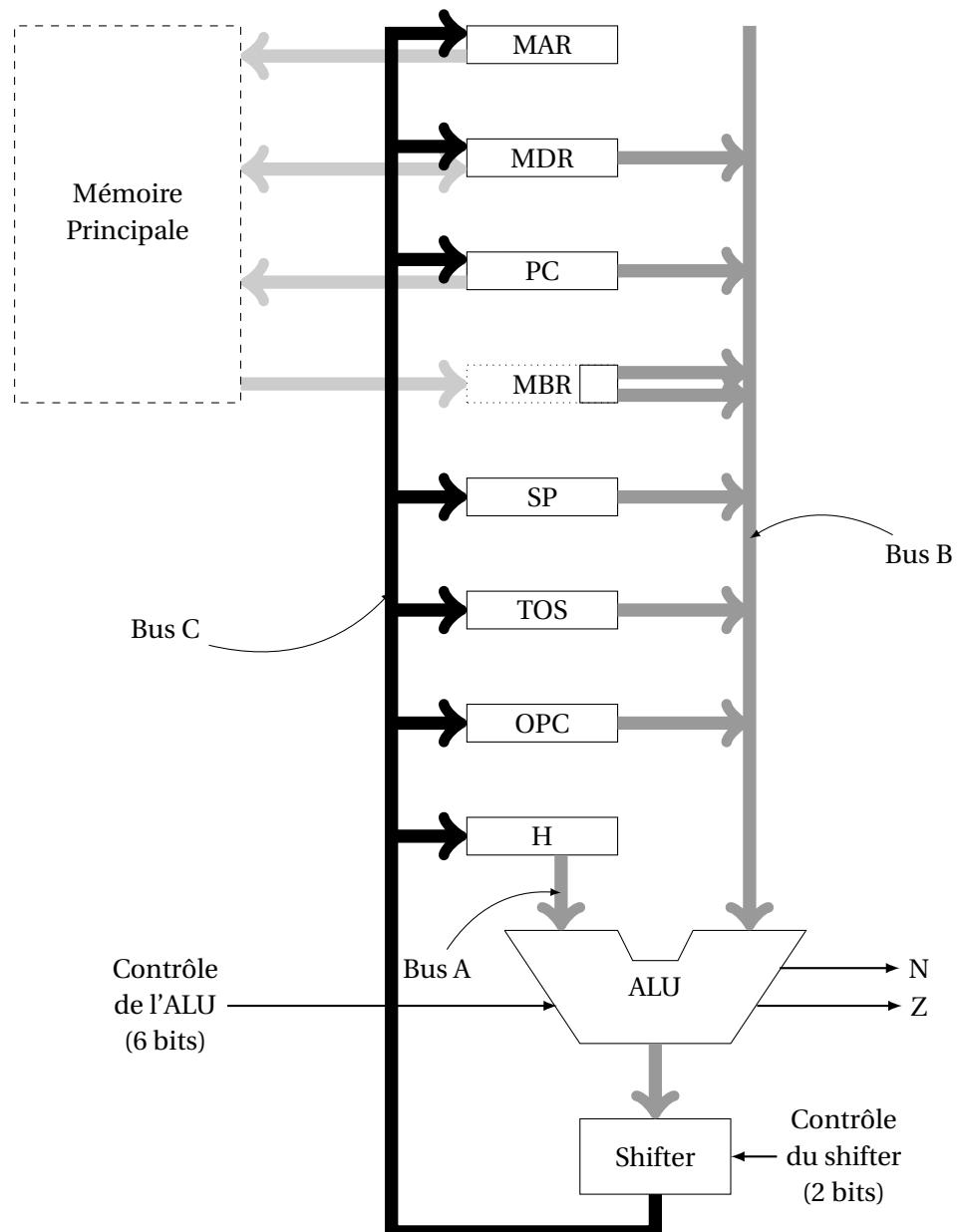


FIGURE 5.1. – Le chemin des données du Mic1. La mémoire principale ne fait pas partie du chemin des données (elle est en-dehors du processeur).

## 5.2. Exemple de datapath : le MIC 1

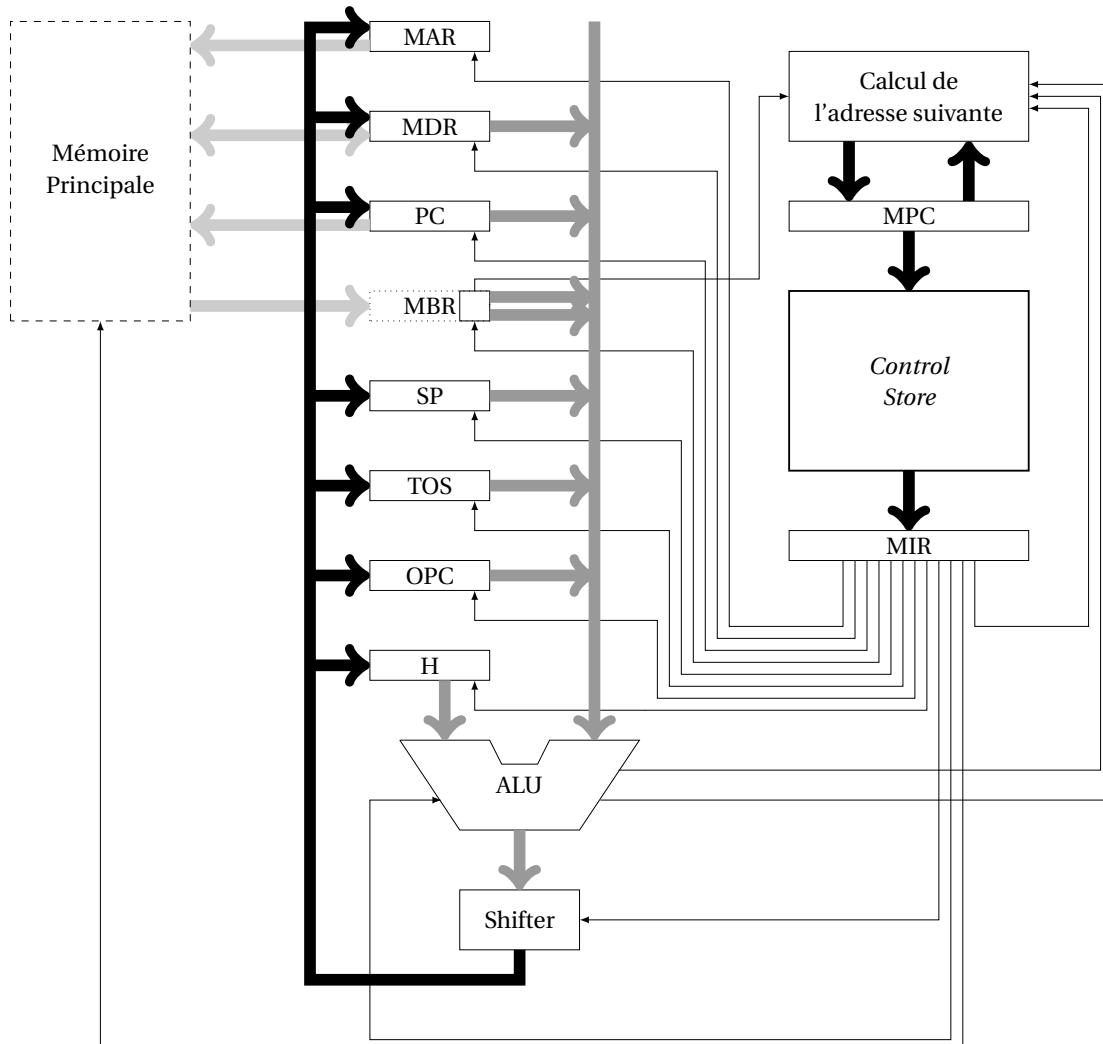


FIGURE 5.2. – Le Mic1 complet, avec le *control store*, le séquenceur (calcul de l'adresse suivante) et les registres MIR (registre de micro-instruction) et MPC (pointeur de micro-instruction).

## 5. Leçons 10 à 12 – La micro-architecture

4.  $A - B$ ;
5.  $A \wedge B$ ;
6. ...

Les 6 bits de contrôle de l'ALU permettent de préciser l'opération désirée.

- D'un circuit de décalage, qui reçoit deux bits de contrôle pour sélectionner l'opération à effectuer. Ce circuit est connecté à la sortie de l'ALU. Il est capable d'effectuer 3 opérations différentes :
  - Ne pas modifier la valeur passée en entrée, et la recopier telle quelle sur sa sortie.
  - Décaler l'entrée d'un octet vers la gauche en remplaçant les bits « perdus » (ceux de poids faible) par des 0 (opération SLL8 : *Shift Left Logical 8 bits*)
  - Décaler l'entrée d'un bit vers la droite en gardant le bit le plus significatif inchangé (opérations SRA1 : *Shift Arithmetic Right 1 bit*)
- 4 bus :
  - Le bus A qui connecte le registre H à l'entrée A de l'ALU. Il permet de lire dans ce registre.
  - Le bus B qui connecte tous les registres sauf H et MAR à l'entrée B de l'ALU. Il permet de lire dans ces registres.
  - Le bus C qui connecte la sortie du circuit de décalage à tous les registres. Il permet d'écrire dans les registres
  - Un petit bus qui connecte la sortie de l'ALU à l'entrée du circuit de décalage.

Par ailleurs, certains registres sont connectés à la mémoire centrale de la machine *via* un bus supplémentaire : ce sont les registres MAR, MDR, PC et MBR.

### 5.2.1. Cycle d'exécution

**Un cycle d'exécution du *data path* correspond à une micro-instruction.**

Un cycle dans ce *data path* peut être décomposé en trois parties :

- Un registre qui peut émettre sur le bus B est activé et émet son contenu sur le bus.
- L'ALU puis le circuit de décalage exécutent les opérations demandées.
- La valeur calculée par l'ALU et le circuit de décalage est émise sur le bus C et est inscrite dans certains registres qui ont été sélectionnés au début du cycle. Il est donc possible d'écrire dans plusieurs registres en un cycle, et même d'écrire dans le registre qui a servi à l'émission sur le bus B.

Concrètement, tout cela est régi par les pulsations de l'horloge. Un cycle commence lors du flanc descendant d'une pulsation et se termine avec l'écriture dans les registres « de sortie » lors du flanc montant suivant.

### 5.2.2. Communication avec la mémoire

Par ailleurs, lors de chaque cycle, il est également possible de commander une lecture ou une écriture dans la mémoire. Pour lire et écrire des données dans la mémoire, la machine dispose de deux possibilités :

1. Soit elle utilise le registre MAR pour spécifier l'adresse d'un *mot de 32 bits*, et utilise le registre MDR pour spécifier la donnée à y écrire (opération *wr*), ou pour recevoir la donnée lue (opération *rd*).
2. Soit elle utilise le registre PC pour spécifier l'adresse d'un *mot de 8 bits*, dont le contenu est écrit dans MBR. Cette opération est appelée *fetch*. Il n'est pas possible d'écrire à travers ce mécanisme.

*Grosso modo* le port 32 bits MAR/MDR servira à manipuler des données, alors que le port 8 bits PC/MBR servira à lire le programme machine (*opcodes + opérandes*).

**Délai mémoire** Malheureusement, les lectures et écritures en mémoire *ne sont pas immédiates* (cette hypothèse est réaliste : sur une vraie machine, un cycle dans le *datapath* est bien plus court que le temps de réaction de la mémoire). Comme le contenu de MAR ou PC n'est inscrit dans les registres qu'*en fin de cycle*, on ne peut pas espérer voir l'effet d'une lecture/écriture mémoire réalisé *au début du cycle suivant*, car il faut laisser à la mémoire le temps de réagir. L'effet n'aura lieu *qu'à la fin du cycle suivant*, et donc les données ne seront exploitabless que *deux cycles plus tard*.

Par exemple, supposons qu'on décide de faire une lecture via MAR/MDR (et qu'on dispose déjà de l'adresse à laquelle lire dans un autre registre) :

1. Cycle 1 : durant ce cycle, le signal *rd* est émis à destination de la mémoire, et le cycle se déroule ainsi :
  - a) On sélectionne un registre qui contient l'adresse à lire
  - b) On passe le contenu de ce registre à l'ALU et au circuit de décalage via le bus B, qui ne le modifient pas.
  - c) On prélève l'information sur le bus C et on la stocke dans MAR.
2. Cycle 2 : deux choses se passent en parallèle :
  - La mémoire effectue la lecture demandée et
  - Une autre opération est effectuée dans le *datapath*. Celle-ci ne peut donc pas utiliser la valeur lue au cycle précédent, car elle n'en dispose pas encore, mais elle peut modifier MAR, car l'adresse a déjà été envoyée à la mémoire.

À la fin du cycle, le contenu de la cellule mémoire MAR apparaît dans MDR, au moment où l'on écrit le contenu du bus C dans les registres.
3. Cycle 3 : au début de ce cycle, la donnée est disponible dans MDR, on peut donc s'en servir durant ce cycle.

Si on demande une lecture en mémoire au cycle  $k$ , on dispose donc de l'information au cycle  $k+2$  (*idem* si on utilise PC/MBR)

## 5. Leçons 10 à 12 – La micro-architecture

### 5.2.3. Micro-instructions

Une *micro-instruction* correspond à l'exécution d'un cycle du *datapath*. Étant donné la discussion qui précède, il est clair que le comportement de la machine à chaque cycle sera régi par les paramètres suivants :

- Quel registre sera sélectionné pour écrire sur le bus B?
- Quels registres recevront la valeur émise sur le bus C?
- Quelle opération l'ALU et le circuit de décalage effectueront ils?
- Y aura-t-il une communication avec la mémoire *via* le port MAR/MDR ou *via* le port PC/MBR?
- Quelle micro-instruction sera exécutée au cycle suivant?

En répondant à ces cinq questions, on spécifie complètement une micro-instruction. En pratique, on utilisera la syntaxe suivante pour les micro-instructions :

opération registres; mémoire; saut

où :

- « opération registres » est une opération sur les registres qui peut être réalisée en un cycle. Par exemple :  $\text{MAR} = \text{SP} = \text{SP} - 1$ . Cette partie à pour effet de calculer  $\text{SP} - 1$  (grâce à l'ALU) et de placer le résultat dans SP et dans MAR. C'est bien possible en un cycle.
- « mémoire » est une opération sur la mémoire :
  - **rd** : lance une lecture en mémoire de la cellule dont l'adresse est dans MAR. Cette valeur sera placée dans MDR au bout d'un cycle.
  - **wr** : lance une écriture en mémoire, de la valeur contenue dans MDR à la cellule dont l'adresse est dans MAR. Cette valeur sera écrite au bout d'un cycle.
  - **fetch** : lance une lecture en mémoire de la cellule dont l'adresse est dans PC. Cette valeur sera placée dans MBR au bout d'un cycle.
- « saut » indique quelle est la prochaine micro-instruction à réaliser. Si rien n'est indiqué, on passe à la suivante. Sinon, on peut utiliser **goto** ou **if() goto...; else goto...**.

Concrètement, les micro-instructions sont représentées de façon binaire et un composant de la machine, appelée *séquenceur* se charge du flot d'exécution des micro-instructions. Nous expliquerons cela plus tard.

### 5.3. Réalisation de l'IJVM à l'aide du MIC1

Maintenant que nous connaissons le langage machine et le micro-langage, écrivons le code qui permet de traduire l'un vers l'autre.

### 5.3.1. Interpréteur

Nous souhaitons disposer d'un *interpréteur* en micro-instructions. Ce programme va continuellement répéter une boucle qui va :

1. Lire la prochaine *instruction machine*
2. La *décoder*
3. Exécuter une séquence de *micro-instructions* qui va *réaliser* l'effet de l'instruction machine.

Pour ce faire, la machine doit pouvoir accéder aux instructions en mémoire et les analyser. Elle utilise deux registres du *datapath* :

- PC, le pointeur de programme, qui pointe vers la prochaine instruction à exécuter.
- MBR, qui est utilisé pour obtenir un octet de données en mémoire (et donc, en particulier, l'*opcode* d'une instruction machine).

On supposera que, initialement (à l'allumage de la machine), le registre PC est initialisé vers la première instruction du programme, et le registre MBR est chargé avec cette première instruction. Ensuite, l'interpréteur en micro-code commence à s'exécuter. La première micro-instruction en est : « PC=PC+1; fetch; goto(MBR) », ce qui a pour effet :

1. d'incrémenter PC pour pointer vers l'instruction machine suivante.
2. de lancer la lecture mémoire de l'instruction suivante (celle pointée par PC). Cette nouvelle instruction ne sera disponible qu'à la fin du cycle suivant.
3. de brancher le micro-code vers la séquences de micro-instructions qui vont exécuter l'instruction machine en cours (qui est encore dans MBR).

À la fin de l'exécution de l'instruction machine, l'interpréteur reviendra vers la première ligne, pour passer à l'instruction machine suivante (qui sera déjà dans MBR étant donné le *fetch*).

Passons maintenant en revue plusieurs instructions machines et traduisons les en micro-instructions. Cette description nous permettra d'introduire les différents registres du *datapath* et leur utilité respective.

### 5.3.2. Manipulation du *stack*

Pour pouvoir accéder au *stack*, le Mic-1 doit disposer de l'adresse de son sommet. Le registre SP (pour *stack pointer*) sert à cela. Par ailleurs, on s'arrangera pour toujours garder dans le registre TOS (*top of stack*) la valeur qui est au sommet du *stack*, et ceci afin de permettre un accès plus rapide au *stack*. Ainsi, si on souhaite simplement consulter ce sommet, il n'est pas nécessaire de faire une lecture en mémoire. Si on souhaite accéder aux deux valeurs au sommet, une seule lecture mémoire est nécessaire, *etc.*

### 5.3.3. Opérations POP et BIPUSH

**POP** Pour effectuer un *pop*, il faut décrémenter le pointeur de *stack* et mettre à jour TOS. Remarquons que l'ancien sommet est toujours présent en mémoire, mais ne fait plus partie du *stack*.

## 5. Leçons 10 à 12 – La micro-architecture

- $\text{MAR} = \text{SP} = \text{SP}-1$ ; rd. Cette micro-instruction décrémente le pointeur de *stack*, place l'adresse du nouveau sommet dans MAR et lance une lecture.
- Un cycle « vide » est nécessaire, pour laisser le temps à la mémoire de répondre la valeur à placer dans TOS.
- $\text{TOS} = \text{MDR}$ ; goto Main1. Cette micro-instruction inscrit dans TOS la valeur réponse par la mémoire (et donc placée dans MDR), puis revient au début de l'interpréteur.

**BIPUSH** Pour effectuer un *push*, il faut récupérer en mémoire la valeur à placer sur le *stack*, incrémenter le pointeur de *stack*, et placer la valeur à cette nouvelle adresse, ainsi que dans TOS.

Pour rappel, une instruction BIPUSH tient sur deux octets en mémoire : le premier contient l'*opcode*  $10_{16}$  et le second la valeur à placer sur le *stack*. Cela signifie qu'au moment où l'on commence à exécuter les micro-instructions qui vont traduire le BIPUSH, la valeur  $10_{16}$  se trouve dans MBR, mais PC pointe déjà vers la case mémoire contenant l'opérande, et la lecture de cette opérande est en cours. Elle ne sera disponible dans MBR qu'au début du second cycle. Il faudra donc penser à incrémenter PC de façon à ce qu'il pointe vers l'*opcode* de instruction machine qui suit le BIPUSH, et à charger cet *opcode* dans MBR.

- $\text{SP} = \text{MAR} = \text{SP}+1$ . Cette micro-instruction calcule la nouvelle adresse de sommet du *stack*, et la place dans SP et dans MAR.
- $\text{PC} = \text{PC}+1$ ; fetch. Lance la lecture de l'instruction machine suivante.
- $\text{MDR} = \text{TOS} = \text{MBR}$ ; wr; goto Main1. La lecture lancée à la micro-instruction précédente n'est pas encore effectuée, et MBR contient donc encore la valeur de l'opérande au début du cycle. C'est donc la valeur de l'opérande qui est copiée dans TOS (ce sera la nouvelle valeur au sommet du *stack*) et dans MDR. L'écriture en mémoire est lancée (l'adresse d'écriture avait été mise dans MAR lors de la première micro-instruction). À la fin du cycle la lecture en mémoire lancée à la micro-instruction précédente est terminée, et l'*opcode* de la prochaine instruction est donc bien dans MBR au moment où l'on revient au début de l'interpréteur.

### 5.3.4. Opérations IADD et ISUB

**IADD** Comme, à tout moment, la valeur au sommet du *stack* est déjà présente dans TOS, on n'a donc besoin de ne faire qu'un seul *read* pour la pénultième valeur du *stack*.

- $\text{MAR} = \text{SP} = \text{SP}-1$ ; rd. Cette instruction décrémente SP et place la valeur calculée dans MAR. Ensuite, un *read* est initié (de la valeur à l'adresse SP-1, donc). Il s'agit donc de lire la pénultième valeur du *stack*, et de la placer dans MDR. Par ailleurs, le pointeur de *stack* est maintenant sur la case où l'on désire stocker le résultat (celle qui contenait – et contient encore – la seconde opérande).
- $\text{H}=\text{TOS}$ . On place le sommet du *stack* dans le registre H en vue de l'addition avec la valeur lue précédemment. Remarquons que cette instruction aurait pu être placée plus tôt, mais elle agit comme un « tampon » pendant laquelle le *read* ordonné au point 5.3.4 est exécuté par la mémoire.

- MDR=TOS=MDR+H; wr; goto Main1. Cette instruction calcule la somme demandée à l'aide de la valeur placée dans MDR, maintenant disponible. Cette somme est placée dans TOS, ce qui garde ce registre cohérent, et également dans MDR. Ensuite, une écriture en mémoire est ordonnée. Le résultat sera inscrit dans la case à l'adresse MAR, qui est toujours, depuis le point 5.3.4, l'adresse de la case contenant la seconde opérande. Le « goto Main1 » permet de passer à l'instruction suivante.

**ISUB** Même principe, seule la commande à l'ALU dans la dernière micro-instruction change.

### 5.3.5. Opérations de saut : GOTO et IFEQ

**GOTO** Contrairement aux opérations précédentes, cette opération ne s'occupe pas du *stack*. Quand on effectue un « GOTO  $o$  », on désire que PC prenne la valeur  $a + o$  où  $a$  est l'adresse mémoire de l'instruction « GOTO  $o$  ». Remarquez qu'au moment où on entre dans la partie de l'interpréteur qui s'occupe du GOTO, PC contient la valeur  $a + 1$ , étant donné que PC est systématiquement incrémenté à la ligne Main1.

Par ailleurs, l'instruction GOTO  $o$  tient en mémoire sur 3 octets (donc aux adresses  $a$ ,  $a + 1$  et  $a + 2$ ), car le décalage  $o$  tient sur 2 octets. Or, la lecture dans la partie « instructions » de la mémoire se fait au travers du registre MBR qui ne peut lire qu'un octet à la fois. Il faudra donc lire les deux octets  $o_1$  et  $o_2$  composant la valeur  $o$  séparément, et recomposer la valeur  $o$ , en faisant un shift gauche de 8 bits de  $o_1$  et en faisant un OR bit à bit entre cette valeur et  $o_2$  :

$$o = \boxed{\begin{array}{c|c} o_1 & o_2 \\ \hline 8 \text{ bits} & 8 \text{ bits} \end{array}}$$

Étant donné l'opération effectuée à la ligne Main1, on voit que, lorsque on entre dans le micro-code qui exécute le GOTO, MBR contient encore l'*opcode* du GOTO, mais la lecture de l'octet  $o_1$  est en cours. Cette valeur sera disponible dans MBR au deuxième cycle.

- OPC = PC-1. Cette micro-instruction sauve dans OPC l'adresse de l'instruction GOTO que l'on s'apprête à exécuter (la valeur  $a$  dans la discussion ci-dessus).
- PC = PC+1; fetch. Au moment d'exécuter cette micro-instruction, la valeur  $o_1$  est dans MBR (en raison du *fetch* de la ligne Main1). On incrémente PC pour qu'il pointe vers la valeur  $o_2$  et on lance une lecture.
- H = MBR << 8. On décale la valeur  $o_1$  (qui est encore dans MBR) de 8 bits vers la gauche et on place le résultat dans H.
- H = MBRU OR H. On combine la valeur  $o_1$  décalé de 8 bits (qui est dans H) avec la valeur  $o_2$  (qui est maintenant dans MBR) à l'aide d'un OR. On utilise MBRU eu lieu de MBR pour compléter correctement la valeur de MBR sur 32 bits (avec des 0). Le décalage du saut est maintenant dans H.
- PC = OPC + H; fetch. On calcule la nouvelle valeur de PC à l'aide de l'adresse du GOTO (stockée dans OPC) et du décalage (calculé dans H). On lance une nouvelle lecture, pour lire l'*opcode* de l'instruction machine suivante.
- goto Main1. On revient au début de l'interpréteur.

## 5. Leçons 10 à 12 – La micro-architecture

**IFEQ** Cette instruction combine les idées vues dans le POP (la valeur au sommet du *stack* doit être supprimée de celui-ci) et dans le GOTO. La comparaison de la valeur au sommet du *stack* et de 0 s'effectue en faisant transiter ce sommet par l'ALU. Quand une valeur est calculée par l'ALU, le bit de sortie Z est mis à 1 si cette valeur vaut 0. On peut alors utiliser un branchement conditionnel dans le micro-code, qui teste la valeur de Z pour sauter vers la partie du micro-code qui va exécuter la bonne branche du IF.



## **Quatrième partie**

# **Le langage Machine**



# 6. Leçon 13 – Langage machine

## 6.1. Caractéristiques

Le langage machine est le langage qui est *directement exécuté par le processeur*. C'est, traditionnellement, le niveau qui sépare le logiciel du matériel (voir FIGURE 1.5).

Pour définir un langage machine, il faut parler de certaines de ses caractéristiques, qui sont également des caractéristiques du processeur qui va l'exécuter :

- Quel est le *modèle mémoire*, c'est-à-dire comment la mémoire est organisée et comment on y accède ?
- Quels sont les *registres* disponibles ? Au niveau du langage machine, on manipule directement les registres, puisque ce sont les mémoires de travail du processeur.
- Quels sont les *types de données* utilisables ? Certaines machines ne peuvent pas faire des calculs sur les nombres en virgule flottante, par exemple.

Par contre, les détails de la réalisation du processeur qui permettent l'exécution du langage machine n'ont pas d'importance ici (micro-architecture, processeur superscalaire, etc).

L'ensemble de ces caractéristiques forme ce qu'on appelle généralement une *architecture*, c'est-à-dire la description de ce qu'un processeur doit être capable de faire : quels registres, de quelles tailles, quels accès à la mémoire, quelles instructions (et ce qu'elles signifient). Le but premier d'une architecture est d'établir un certain niveau de compatibilité entre les machines. Ainsi, deux machines implémentant la même architecture devraient être capables d'exécuter les mêmes programmes en langage machine.

### 6.1.1. Modèle mémoire

Le modèle mémoire spécifie l'organisation de la mémoire et la manière dont on y accède. Sur toutes les machines actuelles, les cases mémoires contiennent 8 bits (un octet). Par contre, suivant la machine, on pourra accéder à des mots de tailles différentes :

- Les premiers Pentium avaient des mots de 4 octets
- alors que les Pentium IV utilisent des mots de 8 octets

Ces machines possèdent des instructions pour manipuler des mots complets.

Il est à noter que :

- La plupart des machines ne peuvent lire ou écrire que des mots qui commencent à une adresse qui est un multiple de la taille du mot. Par exemple, sur des machines qui ont des mots de 4 octets, on peut lire/écrire les mots à l'adresse 0, 4, 8 etc mais pas à l'adresse 2 par exemple. Cela a des conséquences sur la place occupée en mémoire par les informations, car, si celles-ci ont une taille inférieure à un mot, on est obligé :

## 6. Leçon 13 – Langage machine

- Soit de « remplir » les trous avec des 0 (perte de place).
- Soit d'écrire des programmes plus complexes pour découper les mots en plus petites unités et les manipuler individuellement.
- Certaines machines ont des espaces d'adressage différents pour les données et les programmes. Cela signifie que la mémoire est vue *logiquement* comme deux blocs séparés (un pour les données, un pour le code) dont les cases sont numérotées à partir de 0, indépendamment. Un mécanisme de traduction d'adresses (entre les adresses logiques et les adresses physiques) doit alors être mis en place.

### 6.1.2. Registres

Comme nous l'avons vu dans la section « micro-architecture » un microprocesseur peut utiliser de nombreux registres, mais tous ne sont pas accessibles et manipulables par les instructions machines. Dans l'exemple du chapitre précédent, le registre TOS est utilisé pour contrôler le *stack* et ainsi exécuter les opérations arithmétiques, mais ne peut pas être manipulé directement par une instruction machine (cela n'aurait aucun sens).

Il y a typiquement deux types de registres :

- Les registres de travail qui servent à contenir des données pour effectuer des opérations. Ils sont parfois complètement interchangeables, et parfois ont des fonctions dédiées (certains registres utilisés pour la multiplication ou la division sur les Pentium, par exemple).
- Les registres de contrôle, qui contiennent des informations nécessaires au bon déroulement du programme, qui ne sont pas des données (par exemple le *program counter*).

## 6.2. Introduction et exemple du Pentium IV

Nous allons regarder brièvement l'exemple de l'architecture IA-32. Il s'agit de l'architecture 32 bits d'Intel (Intel Architecture 32 bits), utilisée depuis le 80386 (1985) [10].

- La mémoire adressable est composée de 16 384 segments qui contiennent chacun les adresses de 0 à  $2^{32} - 1$ , soit au total  $2^{46}$  octets de mémoire, ou encore 65 TO. En pratique, les OS actuels n'exploitent qu'un seul segment de mémoire limitant la mémoire totale à  $2^{32}$  bits.
- Il y a plusieurs registres : 8 registres de travail : EAX, EBX, ECX et EDX, ESI, EDI, EBP, ESP, qui ont parfois leurs spécificités ; une série de registres de contrôles qui servent uniquement pour la compatibilité avec les anciens programmes (des années '70!) ; un *program counter* appelé EIP (*extended instruction pointer*) et un registre de contrôle, EFLAGS, qui donne des informations sur les opérations effectuées.

## 6.3. Types de données

Il existe principalement deux types de données : les données numériques ou non-numériques.

## Données numériques

- Données entières : typiquement sur 8, 16, 32 ou 64 bits ; toutes les machines modernes stockent les données en complément à 2.
- Données en virgule flottantes : on utilise typiquement la norme IEEE754, et la plupart des machines ont des registres dédiés pour manipuler ces données.

Sur le Pentium IV, les principales données numériques sont [10] :

- Les entiers signés (en complément à 2) sur 8, 16 et 32 bits;
- Les entiers non-signés sur 8, 16 et 32 bits;
- Les entiers représentés en BCD sur 8 bits (un ancien format) ;
- Les nombres en virgule flottante IEEE754 sur 32 et 64 bits.

**Données non-numériques** Certaines machines supportent d'autres types de données, comme les valeurs Booléennes (importantes pour avoir des instructions qui font des tests et décident de la suite de l'exécution du programme en fonction du résultat du test), ou les chaînes de caractères, en ASCII ou en Unicode. Le Pentium IV, par exemple, possède des instructions pour faire de la recherche dans des chaînes ASCII 8 bits (SCASB : *compare byte string* ou SCASW : *compare word string*).

On consultera par exemple <http://ref.x86asm.net/geek64.html>. Pour avoir une idée de la complexité des jeux d'instructions.

## 6.4. Instructions

Les instructions sont caractérisées par :

- Un *opcode* qui est une valeur numérique indiquant de quelle instruction il s'agit.
- Un ensemble d'*opérandes* : il peut y en avoir 0, 1, 2 ou 3 (en général) selon l'instruction.

Naturellement, toutes ces informations doivent tenir dans une place prédéfinie (la taille du registre IR qui contient les instructions lues). Ce problème de place demande parfois de faire des compromis. Par exemple, si le registre IR fait 32 bits, et que la mémoire contient  $2^{32}$  cases, il sera impossible de donner une adresse complète pour opérande d'une instruction. La situation se complique encore quand il y a plusieurs adresses à passer à l'instruction... Nous verrons des solutions dans la suite.

Il est important de noter que le nombre de bits consacrés à l'*opcode* et aux opérandes n'est pas forcément fixe. Sur 16 bits, par exemple, on peut avoir :

- Des instructions qui ont un *opcode* de 4 bits et 3 opérandes de 4 bits.
- Des instructions qui ont un *opcode* de 8 bits et 2 opérandes de 4 bits.
- *etc.*

Pour pouvoir différencier ces différentes situations, il faut adopter des conventions, par exemple :

- Si les 4 bits de poids fort sont différents de 1111, alors l'*opcode* est sur 4 bits, on a donc 15 instructions dont l'*opcode* tient sur 4 bits

## 6. Leçon 13 – Langage machine

- Si par contre, les 4 bits de poids fort sont à 1111 et que les 4 bits suivants sont différents de 1111, on a affaire à une instruction qui a un *opcode* sur 8 bits. Comme les 4 premiers bits sont fixés et que les 4 suivants sont différents de 1111, on a 15 instructions possibles sur 8 bits
- *etc...*

**Exemple** Un exemple simple est celui du jeu d'instructions de l'UltraSparc III. Il existe 4 formats d'instruction différents. Toutes les instructions tiennent sur 32 bits, et donc, il n'est pas possible, en une instruction, d'obtenir une adresse complète. L'instruction SETHI permet d'écrire une constante de 22 bits dans un registre au choix, il faut donc une seconde instruction pour compléter les 10 bits restants si nécessaires. L'instruction CALL permet d'ajouter une valeur sur 30 bits à la valeur courante de PC. Cela permet donc d'exécuter une instruction qui se trouve à une distance de  $2^{30}$  octets de l'instruction courante en mémoire, ce qui devrait être suffisant pour la plupart des cas.

## 6.5. Adressage

Comme nous l'avons vu, une instruction est composée d'un *opcode* et de champs consacrés aux opérandes. Ces champs doivent indiquer *comment obtenir les valeurs des opérandes*. Comme on l'a vu, il n'est *en général* pas possible de donner l'adresse mémoire de chacune de ces valeurs (certaines architectures, comme IA32 par exemple, interdisent d'ailleurs les instructions qui nécessitent deux accès mémoire pour des causes d'efficacité). Les techniques utilisées pour spécifier comment accéder aux valeurs sont connues sous le nom d'*adressage*. En voici quelques-unes :

1. *Adressage immédiat* : la valeur de l'opérande est spécifiée dans l'instruction. Cette valeur est donc forcément une constante et est de taille limitée (par le nombre de bits réservés pour l'opérande). Dans ce cas le terme d'adressage est excessif, puisqu'aucun accès mémoire (en plus de celui qui a permis de lire l'instruction) n'est nécessaire pour obtenir la valeur.
2. *Adressage direct* : l'adresse à accéder est donnée dans l'instruction. Comme on l'a déjà dit, ceci est impossible sur de nombreuses architecture, en raison de la taille des adresses.
3. *Adressage par registre* : l'instruction spécifie dans *quel registre* se trouve *la valeur* de l'opérande. Il faut donc d'abord s'arranger pour stocker la valeur dans le registre, puis on peut appeler l'instruction.
4. *Adressage par registre avec indirection* : l'instruction spécifie dans *quel registre* se trouve *l'adresse* de l'opérande. Il faut donc d'abord s'arranger pour stocker l'adresse dans le registre, puis on peut appeler l'instruction. Un registre peut ordinairement contenir une adresse complète. Dans ce cas, on dit que le registre contient un *pointeur* vers l'emplacement mémoire où se trouve la valeur. Remarquons que l'exécution d'une instruction de ce type est lourde : pour chaque opérande adressée de cette manière il faut : 1) lire le registre, 2) lancer une lecture mémoire à partir de l'adresse lue, et ce, avant d'exécuter l'opération demandée.

5. *Adressage indexé* : dans ce type d'adressage, l'opérande est lue en mémoire et l'adresse est donnée à partir de deux éléments :

- a) Une *adresse mémoire constante* de base, disons  $A$ , qui est spécifiée par l'instruction.
- b) Un registre, qui contient un *décalage* par rapport à  $A$ .

L'opérande est alors lue à l'adresse donnée par  $A +$  la valeur contenue dans le registre.

6. *Adressage indexé avec base* : ce mode est similaire au mode indexé, à la différence que la base est donnée, elle aussi, par un registre. Ainsi, l'opérande est lue en mémoire à l'adresse donnée par la *somme* des valeurs spécifiées dans les deux registres.

Les deux derniers mode d'adressage sont utiles si on souhaite, par exemple, parcourir les valeurs contenues dans un *tableau*. Dans ce cas, toutes les valeurs sont stockées de manière consécutive dans la mémoire : la première à l'adresse  $A$ , la seconde à l'adresse  $A + 1$ , la troisième à l'adresse  $A + 2$ , etc. On utilise alors l'adresse du premier élément comme « base », et le décalage sert à sélectionner la case du tableau qui nous intéresse.

## 6.6. Types d'instructions

Chaque architecture possède son propre jeu d'instructions, mais on peut les classifier *grossièrement* comme suit

**Mouvements de données** Ces instructions permettent de déplacer/copier des données. Combinées avec les différents modes d'adresses, elles permettent de copier des données depuis les registres vers la mémoire, de la mémoire vers les registres, de registre à registre, *etc.*

**Opérations dyadiques et monadiques** Il s'agit des instructions qui permettent de combiner deux valeurs (dyadiques) ou qui modifient une valeur (monadiques). Par exemple : l'addition de deux nombres entiers (opération dyadique) ou un *shift* d'une valeur (opération monadique).

**Instructions de saut** Dans certains cas, on ne souhaite pas exécuter, comme instruction suivante, celle qui suit physiquement en mémoire. Les processeurs possèdent des instructions de *saut* qui permettent de donner l'adresse de l'instruction suivante à exécuter. Ces instructions sont donc des assignations au registre PC.

**Comparaisons et branchements** Les langages de haut niveau comme PYTHON permettent de faire des *branchements conditionnels* :

```

1 if x < 0:
2     print "Nombre négatif"
3 else :
4     print "Nombre positif ou nul"
```

## 6. Leçon 13 – Langage machine

Le processeur doit donc être capable de tester des conditions sur les données qu'il manipule et de choisir, en fonction du résultat du test, la suite des instructions qu'il exécutera. Ces instructions peuvent être de plusieurs formes :

- Une instruction qui teste si un bit est égal à zéro ou non. Si le bit est non nul, l'instruction suivante à exécuter est celle qui suit l'instruction de test, sinon, on *branche* vers une autre instruction dont l'adresse est spécifiée. Schématiquement cela donne :

```
SINUL bit addr
[
    ...
    instructions à exécuter si le bit est non-nul
    ...
]
JUMP addrfin
addr: [
    ...
    instructions à exécuter si le bit est nul
    ...
]
addrfin: [
    ...
    suite des instructions
    ...
]
```

Ce type d'approche est rendu plus simple par certains processeurs qui possèdent des registres de contrôle donnant de l'information sur la dernière opération. On trouve ainsi des processeurs qui mettent systématiquement un bit à 1 dans un registre donné si le résultat de la dernière opération (quelle qu'elle soit) est nul.

- Une instruction plus complexe, qui reçoit plusieurs opérandes, les compare et branche en fonction du résultat de la comparaison. Cela demande donc typiquement 3 accès mémoire.

**Appels de procédures** Les processeurs possèdent des instructions spéciales pour gérer les appels de fonctions, qui ne peuvent pas se résumer à un simple *saut*, car, une fois la fonction terminée, il faut *revenir* au point d'appel. Imaginons par exemple un programme qui contienne 2 fonctions A() et B(), et dans lequel la fonction A() appelle A() à plusieurs reprises :

```
1 def A():
2     ...
3     B()
4     ...
5     B()
6     ...
7     B()
```

```

8     return
9
10    def B():
11        ...
12        return

```

Chaque fois que l'on atteint un appel à `B()`, la situation est simple, puisqu'il suffit de brancher vers la première instruction machine de `B()`. Par contre, quand on atteint l'instruction `return` de `B()`, il faut refaire un saut dans le code de `A()` mais à un endroit différent. Cet endroit de retour dépend de l'endroit où `B()` a été appelée. On pourrait donc imaginer utiliser un *registre* pour stocker l'*adresse de retour*.

Malheureusement, la situation peut être plus compliquée s'il y a plusieurs niveaux d'appel : `A()` peut aussi être appelée par une fonction `C()`, il faut donc aussi retenir où revenir quand se termine, et ainsi de suite. Ce genre de situation apparaît notamment dans le cas de *fonctions récursives*.

On voit donc qu'on peut avoir à stocker un nombre d'adresses de retour qu'il n'est pas possible de borner, et que le nombre de registres disponibles risque de ne pas être suffisant. C'est pourquoi on utilise en général un *stack* et le mécanisme suivant :

- Quand on fait un appel de procédure (instruction dédiée), l'adresse de retour est *pushée* sur le *stack*.
- Quand on arrive à la fin d'une procédure (`return`), on prend l'adresse au sommet du *stack* et on continue l'exécution à partir de cet endroit.

Ainsi, on a, à tout moment, sur le *stack*, l'adresse de retour la plus récente au sommet, et les suivantes au-dessous. En pratique, le *stack* est aussi utilisé pour assurer la communication entre les fonctions : les paramètres passés par la fonction appelante à la fonction appelée, et les valeurs renvoyées par la fonction appelée à la fonction appelante.

**Boucles** Certains processeurs possèdent des instructions dédiées aux boucles, mais on peut réaliser ces dernières à l'aide de sauts et de comparaisons.

**Entrées/sorties** Des instructions spéciales communiquent avec les périphériques.

## 6.7. Modes d'exécution

Ajoutons enfin que les processeurs modernes disposent de *modes d'exécution*, qui permettent d'empêcher d'exécuter certaines instructions considérées comme *privilégiées*. Ces instructions sont celles qui permettent d'accéder à des parties sensibles du système, comme les périphériques. Dans les systèmes les plus simples, le processeur connaît deux modes : (i) le mode *esclave*, qui ne peut pas exécuter les instructions privilégiées; et (ii) le mode *maître*, qui peut tout exécuter. Le mode est stocké dans un registre.

L'utilité de ces deux modes deviendra clair quand on parlera de système d'exploitation. *Grosso modo*, si la machine n'exécute qu'un seul programme, elle peut fonctionner en permanence en mode maître (souvent appelé *mode réel*). Par contre, si plusieurs programmes sont

## 6. Leçon 13 – Langage machine

exécutés « en même temps », comme sur les systèmes modernes, on ne peut plus confier aux programmes la responsabilité d'écrire, par exemple, directement sur le disque, car un programme pourrait ainsi détruire – de manière volontaire ou non – les données d'un autre programme. Dans ces systèmes, il faut un arbitre – ce sera le système d'exploitation – qui gérera le disque et les autres ressources de manière équitable entre les programmes en cours d'exécution. On fera alors en sorte que, quand le processeur exécute un programme utilisateur, il fonctionne en mode *esclave*. Quand le programme a besoin d'accéder au disque, il fera une demande au système d'exploitation, qui, lui s'exécutera en mode maître, et pourra alors accéder au disque à la place du programme; ou encore refuser cet accès s'il est illégitime.

### 6.8. Exemple de programme en langage machine

Regardons un exemple simple de programme écrit à l'aide du langage machine du Pentium IV, où les instructions sont décrites à l'aide de la syntaxe de l'assembleur pour plus de facilité.

```
1      mov eax, 1
2      mov ebx, 10
3      mov ecx, 0
4 our_loop:
5      add ecx, eax
6      inc eax
7      cmp eax, ebx
8      jle our_loop
```

Voici un commentaire de ce programme :

1. `mov eax, 1` stocke la valeur 1 dans le registre `eax`.
2. `mov ebx, 10` stocke la valeur 10 dans le registre `ebx`.
3. `mov ecx, 0` stocke la valeur 10 dans le registre `ecx`.
4. `our_loop:` définit une étiquette utilisée pour les sauts.
5. `add ecx, eax` met dans `ecx` la somme de `ecx` et `eax`.
6. `inc eax` incrémente `eax`.
7. `cmp eax, ebx` compare `eax` à `ebx` et stocke le résultat dans le registre des *flags*.
8. `jle our_loop` si le résultat de la comparaison précédente a révélé que la première valeur était inférieure ou égale (*Lower or Equal*) à la seconde, faire un saut vers `our_loop`. Autrement dit, si `eax ≤ eax`, on fait le saut vers `our_loop`.

On comprend donc que ce programme exécute une *boucle* qui stocke dans `eax` successivement les valeurs 1, 2, ..., 11.

*6.8. Exemple de programme en langage machine*

❀ \* ❀



# 7. Leçon 14 – Interruptions et *traps*

On appelle « flux de contrôle » l'ordre dans lequel les instructions sont exécutées. Normalement, la boucle d'interprétation du processeur fait en sorte que les instructions sont exécutées dans l'ordre où elles apparaissent en mémoire. Néanmoins, l'exécution de certaines instructions, ou certains événements peuvent modifier ce flux :

- En cas de saut.
- En cas d'appel ou de retour de procédure.
- En cas de *trap*.
- En cas d'interruption.

Les sauts et appels/retours de procédure ont déjà été étudiés. Ce sont des modifications « volontaires » du flux d'exécution. Les *traps* et interruption sont par contre des modifications en général « involontaires » de ce flux.

## 7.1. Les *traps*

Un *trap* (piège en anglais) est un appel de procédure que le processeur déclenche pour capturer un événement particulier. Par exemple, si le résultat d'une opération produit un nombre en virgule flottante en *overflow* ou en *underflow*, le processeur réalise un appel à une procédure bien définie, appelée *trap handler*, qui gérera le problème potentiel de la façon la plus adéquate. D'autres exemples d'événements sont :

- un *opcode* non-défini : cette technique était à une certaine époque utilisée pour permettre à des processeurs moins puissants d'exécuter des programmes écrits pour des machins avec un grand jeu d'instructions. Par exemple, certains processeurs n'avaient pas d'instructions pour les nombres en virgule flottante. Quand une telle instruction devait être exécutée, un *trap* avait lieu, et le *trap handler* se chargeait d'effectuer le calcul en virgule flottante à l'aide des autres instructions disponibles (ce qui était bien sûr beaucoup plus lourd que d'avoir une instruction dédiée).
- Une division par zéro.
- Un accès mémoire à une adresse qui n'existe pas

La difficulté est de restaurer le programme qui a été interrompu par le *trap* dans l'état où il se trouvait au moment où le *trap* a eu lieu. *A priori*, le programme ne s'attend pas à ce qu'un *trap* ait lieu, et il ne peut donc pas prévoir de sauvegarder les informations importantes au bon moment.

Pour que le programme puisse continuer à s'exécuter de manière « normale », il faut restaurer *tous* les registres dans l'état où ils se trouvaient au moment du *trap* à l'exception des

## 7. Leçon 14 – Interruptions et traps

registres qui sont censés accueillir le résultat de l'opération. Par exemple, si le *trap* est causé par une opération de somme en virgule flottante (où R1, R2 et R3 sont des registres)

R1 := R2 + R3

il faut qu'une fois le *trap handler* exécuté, R3 et R2 contiennent la valeur qu'ils avaient au moment du *trap*, mais que R1 contienne la valeur calculée par le *trap*. Malheureusement, cette opération de calcul aura vraisemblablement utilisé les registres R2 et R3 et les aura donc modifiées.

Comme dans le cas des appels de procédure, il faut donc utiliser un *stack* (appelé ici *stack système*) pour sauvegarder les valeurs des registres. Ensuite, la restauration des registres doit se faire de manière sélective. Une technique souvent utilisée consiste à écrire les valeurs à renvoyer dans la copie des registres, et à tout restaurer. Dans notre exemple cela donne :

- Sauvegarder tous les registres sur le *stack*
- Calculer la somme de R2 et R3 dans un registre R (pas forcément R1).
- Remplacer la sauvegarde de R1 sur le *stack* par la valeur dans R.
- Restaurer tous les registres à partir du *stack*.

## 7.2. Les interruptions

Quand le processeur veut interroger un périphérique (par exemple lire des données sur un disque sur), il doit :

1. Envoyer l'ordre de lecture au périphérique
2. Attendre que le périphérique ait fini sa lecture
3. Récupérer les données lues par le périphériques (celui-ci les aura écrites sur le bus) et les placer en mémoire.

Le processeur peut alors commencer à traiter les données

Toute cela consomme beaucoup de temps processeur car :

1. Le périphérique est « plus lent » que le processeur : il faut plus de temps au périphérique pour fournir un octet de données qu'il n'en faut au processeur pour le traiter.
2. Le processeur passe du temps à copier les données en mémoire.

Pour résoudre le point 2., on peut utiliser le mécanisme d'accès direct en mémoire, comme nous l'avons vu au début du cours. On a alors la séquence suivante :

1. Le processeur envoie l'ordre de lecture au périphérique
2. Le périphérique lit les données et les écrit en mémoire, pendant ce temps le processeur attend.

Une fois ce travail terminé, les données sont déjà en mémoire et le processeur peut traiter les données.

Néanmoins, le processeur doit encore attendre le périphérique. Il serait plus pratique que le processeur puisse lancer l'ordre de lecture au périphérique, et continuer à exécuter un

autre traitement. Dans ce cas, il faudrait que le périphérique soit capable de *signaler* au processeur qu'il a fini son traitement, pour *interrompre* ce dernier dans son travail en cours. Le processeur devra alors exécuter une *routine de traitement* pour exploiter l'information produite par le périphérique, avant de revenir *au point où il avait été interrompu*. Ce mécanisme est appelé *mécanisme d'interruption*, et est présent sur toutes les machines modernes.

La déroulement d'une interruption peut être schématisé comme suit :

1. Le périphérique (ou son contrôleur) qui désire émettre une interruption met à 1 une ligne du bus système. On parle alors de *demande d'interruption* ou IRQ (pour *interrupt request*).
2. Le CPU détecte ce signal. Cela doit se faire le plus tôt possible, ce qui signifie que, typiquement, le CPU va regarder les lignes d'interruption *à chaque tour de la boucle d'interprétation*. Le CPU détermine qui a émis l'interruption car il peut y avoir plusieurs périphériques qui sont capables de déclencher une IRQ, et le traitement n'est pas forcément le même. Cette étape peut donner lieu à un dialogue entre le CPU et le périphérique.
3. Le CPU sauvegarde PC et certains registres (les registres de contrôle essentiellement – pas les registres de travail) sur le *stack*, car il va arrêter d'exécuter le programme en cours et commencer à exécuter le *gestionnaire d'interruption*. En général, on choisira également de passer la machine en *mode maître* à ce moment-là, puisque les interruptions servent typiquement à communiquer avec les périphériques, ce qui requiert le mode maître.
4. Le CPU modifie le registre PC de manière à ce qu'il commence à exécuter le gestionnaire d'interruption. Pour ce faire, il place dans PC l'adresse de la première instruction de ce gestionnaire, qui aura été chargé en mémoire préalablement. Cette adresse est obtenue à l'aide d'une table que le CPU connaît et qui doit être chargée en mémoire (son adresse de début est en général stockée dans un registre dédié).

Ensuite, le gestionnaire d'interruption commencera à être exécuté. Il devra :

1. Sauvegarder les autres registres (de travail) si nécessaire.
2. Déterminer, si nécessaire, quel périphérique a causé l'interruption.
3. Traiter l'interruption
4. Restaurer les registres de travail sauvés
5. Exécuter une instruction spéciale du processeur, généralement appelée *Return from interrupt*, qui va restaurer le registre PC et les registres de contrôle, et remettre la machine en mode esclave.

Que se passe-t-il si une demande interruption a lieu durant le traitement d'un gestionnaire d'interruption? Généralement, on fera en sorte que celle-ci soit ignorée jusqu'à la fin du traitement. Il existe pour ce faire un bit dans un registre de contrôle qui indique si la machine est interruptible ou non<sup>1</sup>. Ce bit est mis à 0 par le CPU avant d'entrer dans la routine de gestion d'interruption (voir FIGURE 7.3).

---

1. Cette information est parfois couplée au mode : on décide que la machine n'est jamais interruptible en mode maître.

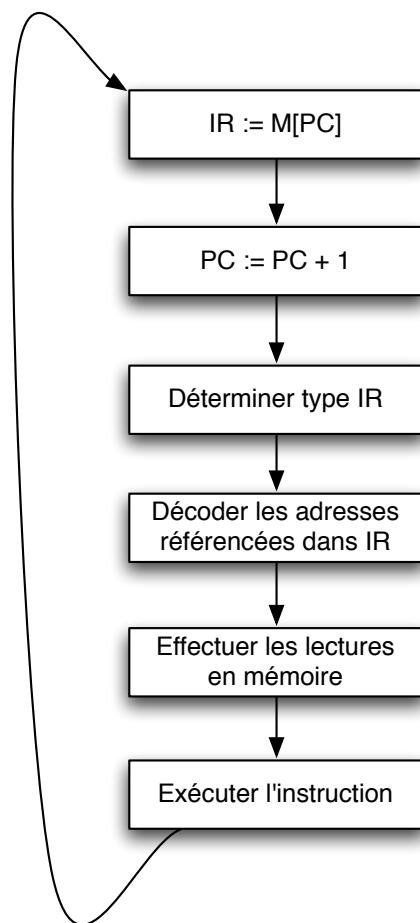


FIGURE 7.1. – La boucle d'interprétation du CPU.

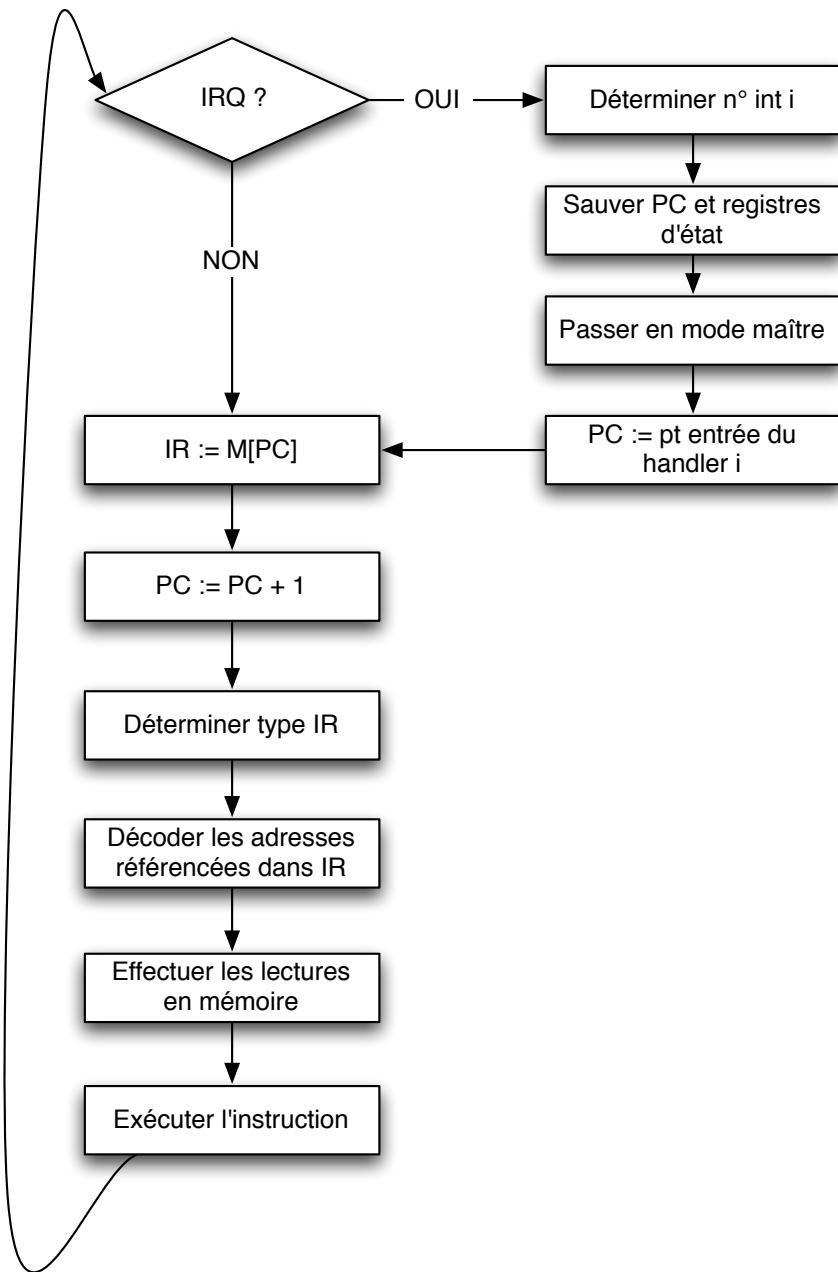


FIGURE 7.2. – La boucle d'interprétation du CPU avec gestion des interruptions.

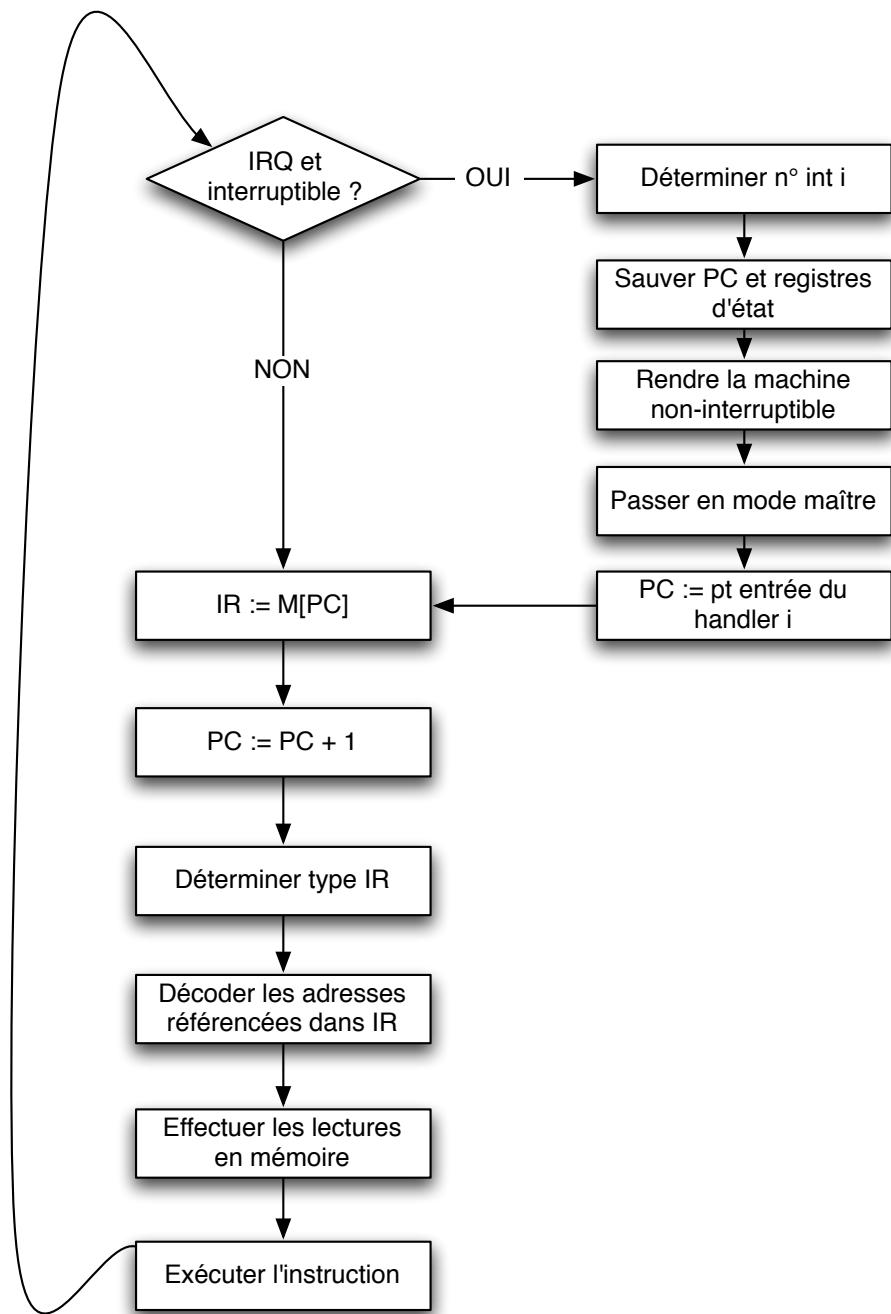


FIGURE 7.3. – La boucle d'interprétation du CPU avec gestion des interruptions et machine non-interruptible en cas d'interruption.

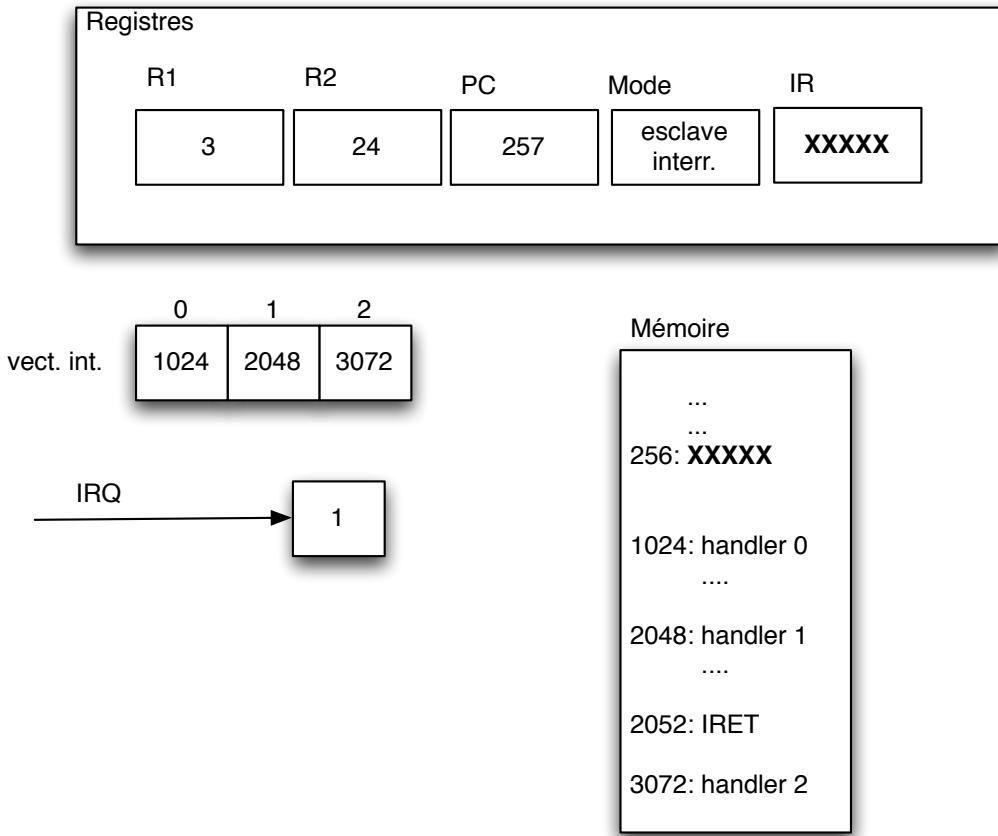


FIGURE 7.4. – Exemple d'interruption (1).

Dans ce cas, l'instruction *Return from interrupt* doit aussi avoir pour effet de rendre la machine à nouveau interruptible. Tout cela (retour de l'interruption, remise en mode interruptible et remise ne mode maître) doit se faire en une instruction, sinon on risque d'être interrompu juste avant le retour!

Les FIGURE 7.4 à 7.8 présentent un exemple d'interruption.

1. À la FIGURE 7.4, on vient d'exécuter l'instruction à l'adresse 256. Avant d'exécuter l'instruction en 257, le CPU détecte une IRQ numéro 1.
2. Le CPU modifie donc PC en fonction des infos dans le vecteur d'interruption. Les registres sont sauvés sur le *stack*. La situation est alors celle de la FIGURE 7.5.
3. Au cours de l'exécution du *handler*, les registres peuvent être modifiés, voir FIGURE 7.6.
4. Après quelques instructions, on s'apprête à exécuter l'instruction à l'adresse 2052, qui est la dernière du *handler* (IRET), comme illustré à la FIGURE 7.7. Les valeurs des registres de travail ont déjà été restaurées, il reste à revenir dans le programme utilisateur.
5. Après exécution de cette instruction, on retrouve la valeur de PC qui avait été sauvegardée, et on continue à exécuter le programme utilisateur là où il avait été interrompu,

7. Leçon 14 – Interruptions et traps

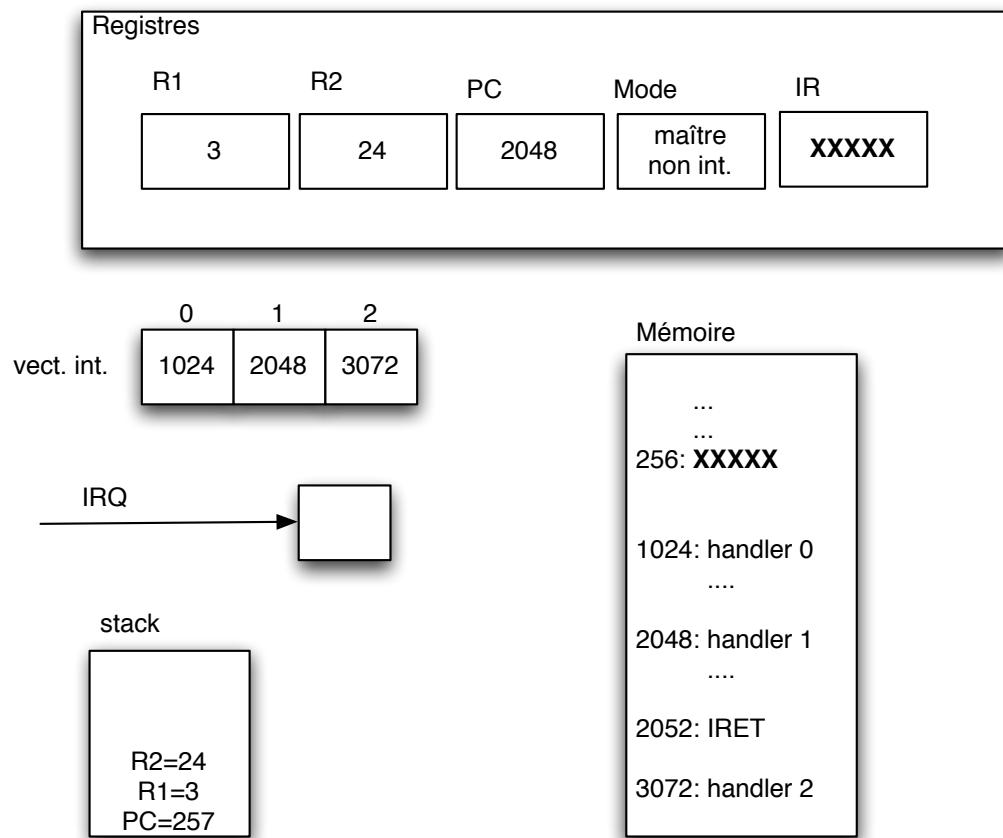


FIGURE 7.5. – Exemple d'interruption (2).

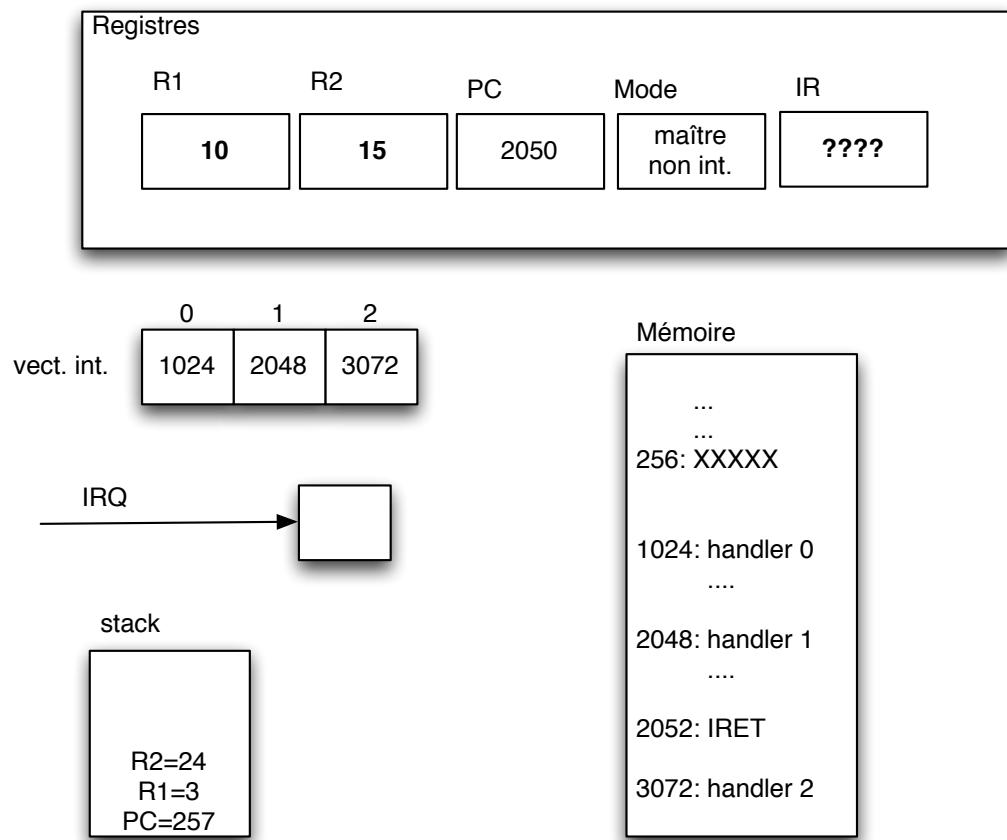


FIGURE 7.6. – Exemple d'interruption (3).

7. Leçon 14 – Interruptions et traps

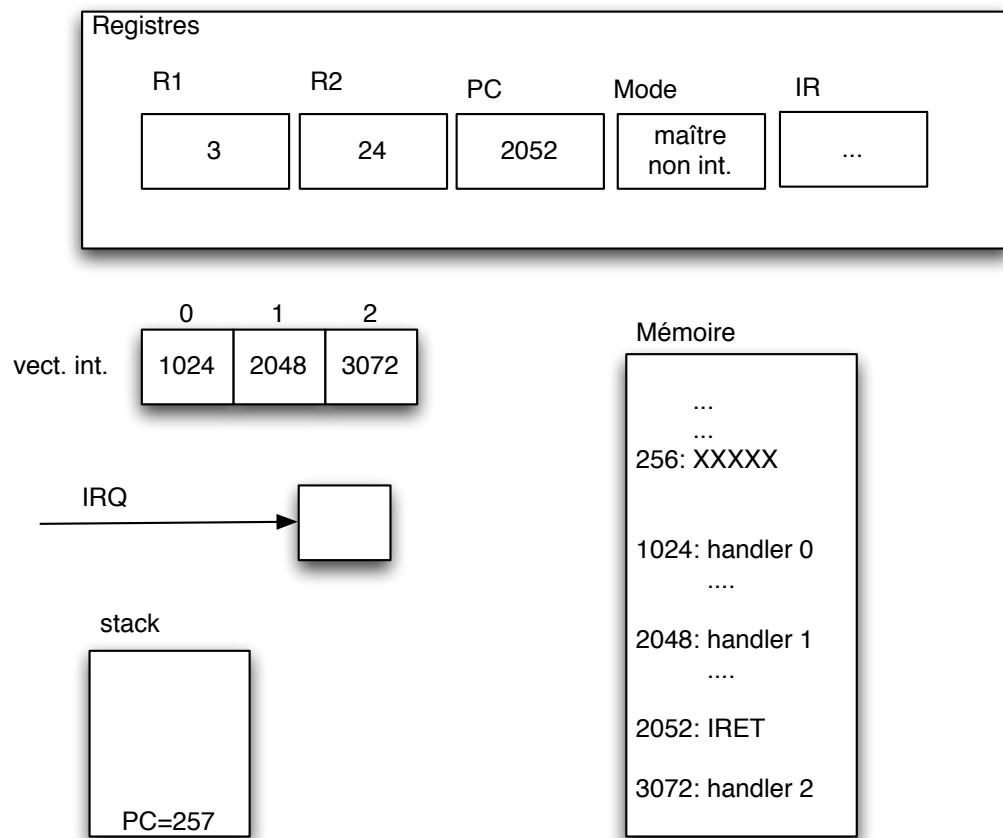


FIGURE 7.7. – Exemple d'interruption (4).

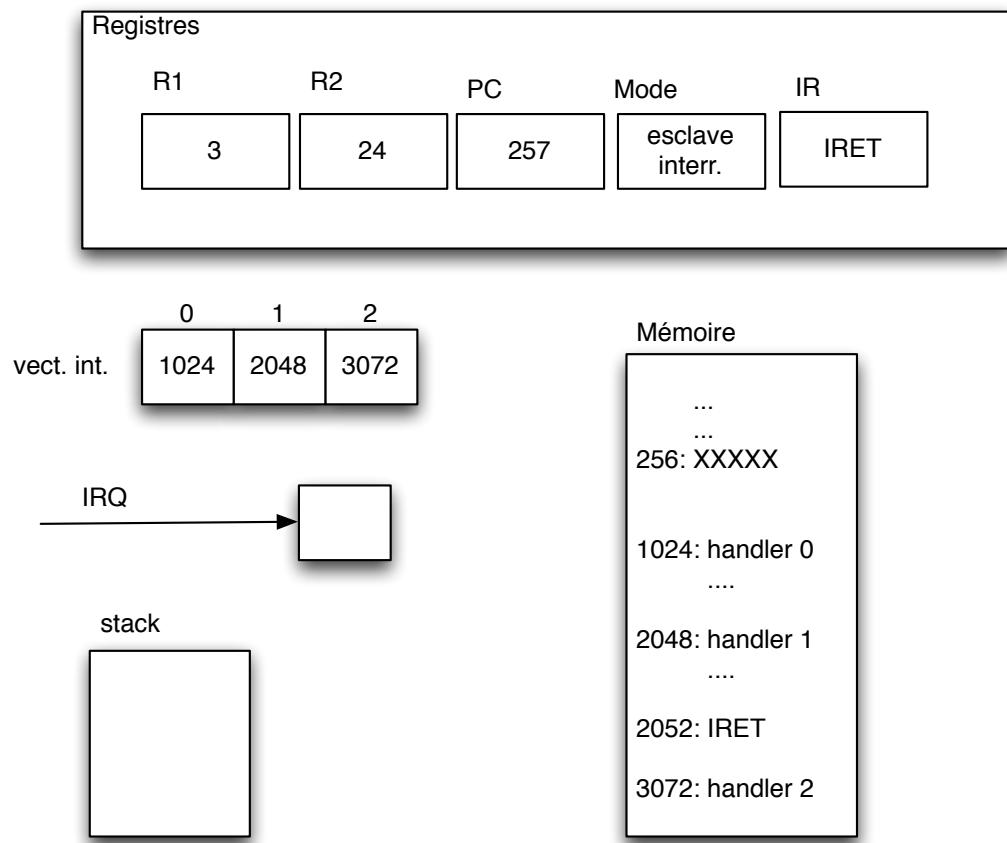


FIGURE 7.8. – Exemple d'interruption (5).

## 7. Leçon 14 – Interruptions et traps

comme illustré à la FIGURE 7.8.

Une autre solution consiste à associer des priorités aux interruptions : dès qu'on entre dans le *handler* d'une interruption, on en désactive toute une série d'autres qui sont considérées comme moins prioritaires. Il faut donc remplacer le bit indiquant si la machine est interruptible par un vecteur qui indique quelles interruptions sont activées.

**Exemple** Intel 8259 : voir documents sur l'UV.



## **Cinquième partie**

# **Le système d'exploitation**



## 8. Leçon 15 – Système d'exploitation

Dans toute la discussion que nous avons eue jusqu'à présent, nous avons supposé que l'ordinateur exécute un et un seul programme provenant d'un et un seul utilisateur, et nous avons volontairement éludé les problèmes qui pourraient survenir si on décidait d'exécuter plusieurs programmes appartenant à plusieurs utilisateurs (qui ont chacun leurs données et leurs caractéristiques propres) sur la même machine, comme on le rencontre sur les machines modernes.

Le principal problème posé par l'exécution de plusieurs programmes sur une même machine est le *partage des ressources*, à savoir : la mémoire, l'espace disque, le processeur, les périphériques,...

Comme exemple, considérons le cas d'un ordinateur installé dans une université, et qui n'est utilisé que par un seul utilisateur, le prof. X. Tous les matins, le prof. X. arrive avec de nouveaux programmes à faire exécuter par l'ordinateur. Il charge le premier programme dans la mémoire, en mettant la première instruction machine à l'adresse 0, demande à l'ordinateur de commencer à exécuter le programme à partir de cette adresse, attend que le programme s'exécute et passe au programme suivant. Durant son exécution, le programme lit et écrit des données sur le disque dur, à des endroits que le prof. X a bien choisi de manière à pouvoir les retrouver ensuite.

Supposons maintenant que l'université engage un collègue, le prof. Y, qui doit lui aussi utiliser l'ordinateur, et, lui aussi doit stocker des données sur le disque. Pour ce qui est de l'utilisation, X et Y se mettent d'accord : chacun accède à l'ordinateur un jour sur deux. Pour ce qui est du disque, l'accord est le suivant : la première moitié va à X qui en dispose comme il le désire, et la seconde moitié à Y.

Grâce à cet *accord préalable*, les deux utilisateurs peuvent sans problème exploiter la machine, mais imaginons maintenant que X et Y se disputent, et que X désire nuire à son collègue. Rien ne l'empêche d'écrire un programme qui va détruire sur le disque toutes les données de Y... Une telle situation peut également se produire par accident... La situation se complique encore si plusieurs autres personnes doivent avoir accès à l'ordinateur...

Cet exemple simple montre que, si un partage coopératif des ressources d'un ordinateur est possible, ce n'est pas une solution raisonnable en pratique. C'est pourquoi, on préfère ajouter un intermédiaire qui va jouer le rôle *d'arbitre*, à savoir, le *système d'exploitation* (OS – *operating system*)

Dans notre exemple, le système exploitera le fait que le processeur possède des *instructions privilégiées*, parmi lesquelles se trouvent, par exemple, celles qui permettent de contrôler les périphériques, et donc, en particulier, de lire ou écrire sur le disque. On mettra le processeur en *mode esclave* quand il doit exécuter les programmes des utilisateurs. Quand un de ces programmes désirera écrire sur le disque, par exemple, il ne pourra pas exécuter directement les instructions (privilégiées) d'écriture sur disque. Il devra faire appel au système d'exploitation,

## 8. Leçon 15 – Système d'exploitation

en lui transmettant une requête d'écriture. Le système d'exploitation, par contre, s'exécutera en *mode maître*. Quand il recevra un requête, il vérifiera d'abord si elle est légitime : le programme tente d'écrire dans une zone du disque qui correspond à un fichier appartenant à l'utilisateur qui a lancé le programme. Si c'est le cas, l'OS exécutera les instructions privilégiées pour le compte de l'utilisateur.

Le mécanisme décrit brièvement ci-dessus permet aux utilisateurs de partager les ressources des disques, mais il peut aussi être appliqué pour :

- Partager la mémoire centrale entre les programmes et éviter qu'un programme n'écrive dans une zone mémoire réservée à un autre programme.
- Partager le processeur entre les programmes pour en exécuter plusieurs de telle manière qu'on ait l'impression qu'ils s'exécutent en parallèle.
- *etc*

**Pour résumer** Le *système d'exploitation* est un *logiciel* qui a pour but de gérer (de manière équitable et sûre) les ressources matérielles de l'ordinateur, en fournissant une *interface* entre le matériel et les applications. Pour ce faire, le système d'exploitation est en général le seul à s'exécuter en *mode maître*, les applications s'exécutent elles en *mode esclave*<sup>1</sup>.

### 8.1. Types d'OS

Il existe différents types de systèmes d'exploitation, en fonction des fonctionnalités qu'ils proposent :

- **OS mono- ou multi- utilisateur(s)** : Sur un OS mono-utilisateur, on considère qu'il n'y a qu'une seule personne qui utilise la machine. Sur un OS multi-utilisateurs, l'OS est « conscient » qu'il existe différents utilisateurs et propose des services en ce sens. Tout d'abord, les utilisateurs doivent s'identifier auprès de l'OS avant de pouvoir commencer à exécuter des programmes. Cela se fait typiquement à travers une procédure de *login* à l'aide d'un mot de passe. Ensuite, l'OS peut retenir à quel utilisateur appartient chaque programme exécuté, et chaque fichier. Il peut empêcher, le cas échéant, un utilisateur d'accéder aux fichiers d'un autre utilisateur (c'est ce qu'on appelle la *gestion des permissions*). Enfin, l'OS peut distinguer différentes catégories d'utilisateurs qui auront des priviléges différents. Typiquement, il y a un *super-utilisateur* qui a tous les droits sur la machine et peut la gérer entièrement (ajout/suppression de programmes ou d'utilisateurs), alors que les utilisateurs normaux ne pourront exécuter que des programmes standards. Certaines catégories d'utilisateurs peuvent également se voir appliquer des restrictions (usage du matériel), des quotas (espace disque consommé), ou des systèmes de facturation (impression).

---

1. Remarquons qu'il y a eu plusieurs systèmes d'exploitation avec lesquels les applications s'exécutaient elles aussi en mode maître, DOS par exemple. Dans ce cas, la protection du système repose sur la bonne volonté des programmeurs des applications, qui ont tout le loisir de contourner les garde-fous mis en place par l'OS.

- **OS mono- ou multi- tâche(s)** : Un OS multi-tâches est capable d'exécuter plusieurs programmes en même temps (éventuellement des programmes de différents utilisateurs si le système est multi-utilisateurs), alors que dans un système mono-tâche, il faut attendre qu'un programme se soit complètement exécuté pour en lancer un autre. De manière générale, il y a moins de processeurs sur la machine que de programmes à exécuter en même temps. Il faut donc trouver une politique pour répartir le(s) processeur(s) entre les programmes en cours d'exécution, et confier la gestion de cette politique à une partie de l'OS, qu'on appelle *l'ordonnanceur* (ou *scheduler* en anglais).
- **OS batch ou interactif** : Dans un OS *batch*, on fournit à l'OS une séquence de commandes à exécuter, et celui-ci s'en charge, sans permettre à l'utilisateur d'intervenir pendant l'exécution des commandes, ni entre les différentes commandes. La partie de l'OS chargée d'exécuter la séquence de commande est appelée *l'interpréteur de commande*. Dans un OS interactif, l'utilisateur peut dialoguer avec les programmes en cours d'exécution, à l'aide de commandes entrées au clavier, ou à l'aide d'autres périphériques, comme la souris.
- **OS temps réel** : Dans certains applications, le délai de réponse des applications est important (exemple : systèmes de freinage dans une voiture,...) Dans ce cas, l'OS doit être capable d'offrir des garanties sur le temps maximum au bout duquel une application donnée sera terminée. Ces OS sont appelés *temps réel*, et sont obtenus en modifiant l'ordonnanceur.

La plupart des OS grand public (Windows, MacOS, Linux) sont multi-tâches, multi-utilisateurs, interactifs mais pas temps réel.

## 8.2. OS et interruptions

Nous avons vu que la gestion d'une interruption se fait en exécutant un *handler* ou gestionnaire d'interruption qui doit avoir été installé en mémoire. En pratique, les gestionnaires d'interruption feront partir de l'OS : au démarrage du système, l'OS est chargé en mémoire et se charge alors de placer les bons gestionnaires d'interruption à l'endroit adéquat (soit une adresse fixe en mémoire, soit une adresse qui est indiquée dans un registre du processeur). Ainsi, lors d'une interruption (par exemple : un périphérique a fini de lire), c'est un morceau de code de l'OS qu'on exécutera, ce qui est bien cohérent avec le rôle de l'OS, et avec le fait qu'on a choisi de mettre la machine en mode maître lors d'une interruption.

## 8.3. Appels système

L'appel système est le mécanisme qui permet à un programme utilisateur en cours d'exécution de faire appel à l'OS pour lui faire exécuter une séquence d'instructions en mode maître. Concrètement, le programme en cours d'exécution doit décider volontairement d'interrompre son exécution pour donner la main à l'OS. Cela peut se faire aisément grâce au mécanisme d'interruption. La différence par rapport aux interruptions vues jusqu'à présent,

## 8. Leçon 15 – Système d'exploitation

est *le programme qui s'exécute* la déclenche *volontairement* : c'est ce qu'on appelle une interruption *interne*, par opposition aux interruptions *externes* qui sont déclenchées par un événement extérieur.

Concrètement le programme qui s'exécute et doit faire appel au système doit :

- Placer dans les registres les informations qu'il désire transmettre à l'OS : nature de la requête (une valeur numérique pour chaque appel système), paramètres. S'il n'y a pas assez de registre, on peut aussi utiliser le *stack*.
- Exécuter une instruction qui permet de déclencher une interruption bien précise (par exemple INT 80 sur les machines Intel).

À ce moment, le gestionnaire d'interruption correspondant aux appels système est exécuté (en mode maître, on est dans l'OS). Sa première tâche est de regarder le contenu des registres pour déterminer la requête qui est faite. Ensuite, il exécute la requête. Si des valeurs doivent être renvoyées (par exemple, dans le cas d'une lecture de périphérique), celles-ci peuvent être placées dans les copies des registres, comme décrit précédemment.

Dans la suite, nous verrons comment l'OS est utilisé pour partager trois types de ressources :

1. La mémoire centrale
2. Le processeur
3. La mémoire secondaire



## 9. Leçon 16 et 17 – Gestion de la mémoire primaire

Dans notre présentation des programmes en langage machine nous avons volontairement passé sous silences certaines complications dues aux adresses mémoires. Prenons comme exemple un programme très simple, en « pseudo-code machine >> :

```
R1 := 5
R2 := 3
R3 := 0
d: R3 := R3 + R1
    R2 := R2 - 1
    if R2 != 0 jump d
```

Ce programme initialise les 3 registres R1, R2 et R3 à 5, 3 et 0, puis commence une boucle dans laquelle il : (i) ajoute R1 dans R3, et (ii) décrémente R2, et ce, tant que R2 est différent de 0. En d'autres termes, ce programme multiplie R1 par R2 et met le résultat dans R3.

Concrètement, l'instruction de saut conditionnel doit contenir une adresse mémoire, c'est-à-dire une valeur numérique, qui sera placée dans le registre PC au moment de son exécution, et non pas d'un symbole d comme nous l'avons représenté pour faciliter la lecture. Au moment où le programme en question est créé, il faut donc disposer de l'adresse à laquelle l'instruction de la quatrième ligne sera stockée en mémoire... Une façon simple d'obtenir cette adresse (et les autres adresse du programme) est de connaître *l'adresse de début* du programme en mémoire (autrement dit, l'adresse de la première instruction). Si on a cette information, connaître l'adresse de chaque instruction est facile puisqu'on connaît la longueur de chaque l'instruction. Malheureusement, on ne dispose pas toujours de cette information au moment où on écrit le programme. Regardons plusieurs cas :

- Sur une machine sans OS et mono-tâche, le problème se résout facilement. Le programme dispose normalement de toute la mémoire. On peut donc supposer qu'il est toujours chargé à partir de la même adresse, par exemple l'adresse 0. C'est la situation la plus confortable, car l'adresse de début est connue à l'avance, c'est-à-dire au moment de la réalisation du programme
- Sur une machine avec OS, l'OS prend de la place en mémoire. Comme l'OS est le premier programme chargé, il se placera typiquement au début de la mémoire. La place occupée n'est pas toujours la même car la taille de l'OS peut varier d'une version à l'autre, et que certains modules de l'OS ne sont pas forcément chargés à chaque fois (en fonction du matériel disponible sur la machine par exemple). La première adresse libre après l'OS peut donc changer, et il n'est pas acceptable d'avoir à ré-écrire/recompiler

## 9. Leçon 16 et 17 – Gestion de la mémoire primaire

tous ses programmes chaque fois qu'on fait, par exemple, une mise à jour de l'OS. On pourrait imaginer réserver une zone (par exemple les 64 premiers Mo) pour l'OS, et charger les programmes utilisateurs toujours à partir de la même adresse, mais c'est limitatif pour l'OS, et on risque d'avoir beaucoup de mémoire inutilisée.

- Dans le cas des systèmes multi-tâches, le problème est encore plus aigu, puisqu'il peut y avoir plusieurs programmes en mémoire, et l'adresse de début d'un programme dépend donc des autres programmes en mémoire... Dans ce cas, l'adresse de début du programme ne sera connue que lors de l'exécution du programme.

Au moment où l'on connaît l'adresse de début du programme en mémoire (dans les pires cas, c'est au moment de l'exécution), il faut alors copier le programme en mémoire (le charger) en modifiant toutes les adresses des *sauts*. Cela signifie qu'on doit être capable de savoir où ces sauts se trouvent, où se trouvent les étiquettes...

Cette solution est assez fastidieuse. La solution la plus simple est clairement celle où le programme est réalisé en tenant compte qu'il sera chargé à partir de l'adresse zéro, mais il faut clairement adopter un mécanisme qui permet de combiner cette facilité avec le fait que plusieurs programmes peuvent être présents en mémoire en même temps...

### 9.1. Pagination

Le mécanisme de *pagination* (à la demande) résout ce problème de manière élégante et efficace (et permet d'augmenter les capacités de l'ordinateur comme nous le verrons plus tard) :

- Du côté du programme : celui-ci est réalisé et s'exécute comme s'il commençait à l'adresse 0. Il est chargé tel quel en mémoire. Évidemment, si plusieurs programmes sont présents en même temps en mémoire, il n'est pas possible de les faire commencer tous à l'adresse 0, ce qui signifie que les adresses données dans les instructions de ces programmes ne sont, en général, pas *correctes* : on parle d'*adresses virtuelles* puisque le programme s'exécute *virtuellement* à partir de l'adresse 0.
- *Concrètement*, le programme est réparti en différents endroits de la mémoire, et il faut donc avoir un mécanisme qui traduise les adresses virtuelles en *adresse réelles* (c'est-à-dire les véritables numéros des cases mémoire). Cette traduction est réalisée à l'aide d'un circuit spécialisé du microprocesseur : le MMU, *Memory Management Unit*. Le processeur manipule donc, lui aussi, des adresses virtuelles (en tous cas dans ses registres), et ce n'est qu'au moment où il doit accéder à la mémoire qu'il traduit les adresses virtuelles en adresses réelles, car la mémoire ne manipule que des adresses réelles.

Le mécanisme de *pagination* fonctionne de la façon suivante. Considérons un programme de longueur  $L$ , à exécuter sur une ordinateur ayant une mémoire de taille  $M$ .

- Le programme est *découpé* en portions qui sont toutes de taille égale  $\ell$  (sauf potentiellement la dernière si  $L$  n'est pas un multiple de  $\ell$ ). Ces portions sont appelées des *pages*. Chaque page correspond donc à une plage d'adresse virtuelles :

- La première page correspond aux adresses virtuelles de 0 à  $\ell - 1$ , puisque le programme a été conçu « comme si » il s'exécutait avec sa première instruction à l'adresse 0.
- La seconde page contient les adresses virtuelles de  $\ell$  à  $2\ell - 1$ .
- ...
- De manière générale, la  $i^{\text{e}}$  page contient les adresses virtuelles allant de  $(i - 1) \times \ell$  à  $i \times \ell - 1$ .

Le programme sera donc composé de  $L/\ell$  pages si  $L \bmod \ell = 0$ , ou de  $\lfloor L/\ell \rfloor + 1$  pages si  $L \bmod \ell \neq 0$ .

- Symétriquement, la mémoire est découpée en portions de taille  $\ell$  également qui peuvent donc chacune accueillir une page du programme. Ces portions sont appelées des *cadres de pages*. Chaque cadre de page correspond lui à une plage d'adresses réelles : la page  $i$  contient les adresses réelles allant de  $(i - 1) \times \ell$  à  $i \times \ell - 1$ . La mémoire contient donc au plus  $\lfloor M/\ell \rfloor$  cadres de page.
- À chaque programme chargé en mémoire est associé une table qui indique *dans quel cadre de page est chargée chacun des pages*. En effet, la souplesse du mécanisme de pagination permet de charger n'importe quelle page dans n'importe quel cadre de page, et ce, de manière non-contiguë, et dans le désordre si nécessaire. Cette table est appelée la *table des pages*, et chacune des ses entrées est appelée un *descripteur de page*. Elle est utilisée par le MMU pour assurer la traduction des adresses virtuelles vers les adresses réelles.

### 9.1.1. Travail du MMU

Vérifions maintenant que la table des pages est suffisante pour permettre au MMU de faire son travail de traduction. Considérons une adresse virtuelle  $\alpha$  (nous avons donc  $0 \leq \alpha \leq L - 1$  :

1. La première chose à faire pour exploiter le descripteur consiste à trouver *dans quelle page se trouve l'adresse* : il s'agit de la page :

$$\alpha \div \ell$$

(où  $\div$  indique une division entière).

2. La table des pages nous renvoie *le numéro du cadre de page contenant la page qui nous intéresse* :

$$\text{desc}(\alpha \div \ell)$$

3. On peut alors en déduire *l'adresse réelle du début du cadre de page* c'est-à-dire *l'adresse réelle du début de la page*, puisque chaque cadre de page est de longueur  $\ell$  :

$$\ell \times \text{desc}(\alpha \div \ell)$$

4. On calcule ensuite le décalage de l'adresse  $\alpha$  par rapport au début de la page qui la contient :  $\alpha \bmod \ell$ .

## 9. Leçon 16 et 17 – Gestion de la mémoire primaire

5. Ce décalage est le même par rapport au début du cadre de page, puisque la page est chargée intégralement et sans modification dans le cadre de page. On connaît donc maintenant le début de la page en mémoire et le décalage de l'adresse qui nous intéresse par rapport au début de la page. Il suffit alors de faire la somme. L'adresse réelle est donc :

$$\ell \times \text{desc}(\alpha \div \ell) + (\alpha \bmod \ell)$$

Toutes ces opérations peuvent sembler lourdes, surtout quand on pense qu'il faut les effectuer à chaque instruction (en effet, le registre *IR* est chargé à partir de l'adresse donnée dans *PC*, qui est une adresse virtuelle). En pratique, elles peuvent être réalisées facilement si  $\ell$  est une puissance de 2 (par exemple  $2^k$ ). En effet, pour des adresses sur 32 bits par exemple :

- On obtient  $\alpha \div \ell$  en isolant les  $32 - k$  bits de poids forts de l'adresse (revient à faire un décalage de  $k$  positions vers la droite).
- On réalise la multiplication de  $\text{desc}(\alpha \div \ell)$  par  $\ell = 2^k$  en décalant  $\text{desc}(\alpha \div \ell)$  de  $k$  positions vers la gauche
- On réalise le modulo en isolant les  $k$  bits de poids faible de l'adresse virtuelle.

On constate alors que la valeur  $\ell \times \text{desc}(\alpha \div \ell)$  est un nombre dont les  $k$  bits de poids faibles sont à zéro. Comme on y additionne le nombre  $\alpha \bmod \ell$  sur  $k$  bits, cela revient à remplacer les  $k$  bits de poids faibles de  $\ell \times \text{desc}(\alpha \div \ell)$  par les  $k$  bits de  $\alpha \bmod \ell$ . Au final, l'adresse virtuelle  $\alpha$  possède les mêmes  $k$  bits de poids faibles que son adresse physique correspondante. Par contre, les  $32 - k$  bits de poids forts de  $\alpha$ , qui constituent le numéro de page, ont été remplacés par les  $32 - k$  bits de  $\text{desc}(\alpha \div \ell)$ , soit le numéro de cadre de page. Pour conclure, si la taille des pages est de  $\ell = 2^k$ , le travaille du MMU *consiste uniquement à remplacer les  $32 - k$  bits de poids forts de l'adresse – qui constituent le numéro de page – par les bits du numéro de cadre de page correspondant*.

## 9.2. Pagination à la demande

Ce système est un perfectionnement du précédent. Il se base sur l'observation qu'à un moment donné, on n'a jamais besoin, pour exécuter un programme, que de la page qui contient la prochaine instruction à exécuter, et des pages qui contiennent les données nécessaires à cette instruction. Il n'est donc pas nécessaire d'avoir en permanence en mémoire toutes les pages du programme qui s'exécute. Cette observation a plusieurs conséquences :

- Cela permet de ne charger que ce qui est nécessaire : économie de temps et de mémoire.
- Cela permet potentiellement d'exécuter des programmes qui sont plus gros que la mémoire disponible (en ne chargeant que ce qu'il faut de pages).

Par contre, il faut (*i*) trouver un endroit où stocker les pages qui ne sont pas chargées, (*ii*) que le MMU puisse déterminer quelles pages sont en mémoire et (*iii*) disposer d'un mécanisme qui permette de rapatrier des pages en mémoire quand on veut y accéder et qu'elles ne sont pas chargées. Voyons ces points en détail :

## 9.2. Pagination à la demande

1. Les pages inutilisées seront stockées sur une mémoire, mais pas la mémoire principale, puisqu'on tente de la libérer. On stockera donc les pages inutilisées sur la mémoire secondaire.
2. Le MMU disposera d'un champ supplémentaire dans le descripteur de chaque page : un bit de présence indiquant si la page est présente en mémoire primaire ou pas.
3. Le travail du MMU sera quelque peu modifié. Considérons une adresse virtuelle  $\alpha$  : le MMU va, comme avant, déterminer le numéro de page  $i = \alpha \div \ell$ . Il va consulter le bit de présence associé : présence( $i$ ). Si la page est présente en mémoire, le MMU continue comme dans le cas de la pagination simple. Sinon, il faut que le MMU fasse en sorte que la page soit chargée en mémoire, et que le numéro de cadre de page choisi soit indiqué dans le descripteur.

Malheureusement, la page est présente en mémoire secondaire, et le MMU n'a pas accès directement à cette mémoire, pour plusieurs raisons : il n'a pas accès en général aux périphériques (seul le CPU, s'il s'exécute en mode maître, peut y accéder), et il ne dispose pas d'informations sur l'organisation du disque (celle-ci est dépendante de l'OS qui s'exécute sur la machine). Par contre, l'OS possède toutes ces informations : c'est lui qui gère les périphériques, et c'est lui qui a lancé les programmes qui s'exécutent. Le MMU doit donc s'arranger, en cas de *défaut de page (page fault)* pour faire en sorte que l'OS prenne la main et charge la page au bon endroit en mémoire. Pour ce faire, le MMU déclenche un *appel système* pour gérer le défaut de page. Le gestionnaire correspondant chargera alors la page manquante et rendra la main au programme.

**Remarques** Il faut encore remarquer plusieurs choses :

1. Pour pouvoir charger une page en mémoire, encore faut-il qu'il y ait un cadre de page libre. Ce n'est pas toujours le cas, car la mémoire peut être bien plus petite que la somme des programmes en cours d'exécution. Il faut donc *choisir une victime*, un cadre de page qu'on libérera, et dont on devra impérativement copier le contenu sur le disque (au cas où le programme aurait écrit dans la page correspondante). Le choix de la victime peut donner lieu à plusieurs stratégies :
  - a) Une technique classique consiste à choisir la page *la moins récemment utilisée*. On peut néanmoins trouver un exemple où cette technique ne fonctionne pas de façon efficace : si on a une boucle qui parcourt itérativement les  $n + 1$  pages alors qu'il n'y a que  $n$  cadres de page.
  - b) Une autre technique est la technique FIFO, qui choisit la page *qui a été chargée depuis le temps le plus long*, et ce, indépendamment de son utilisation. Pour ce faire, on maintient un compteur sur chaque cadre de page, qui vaut initialement 0. À chaque *page fault*, on incrémente ce compteur de 1 pour chaque cadre de page, sauf pour celui qui a déclenché le *page fault*, pour lequel on met le compteur à 0 (cela permet de toujours borner les compteurs). On choisit toujours comme victime la page qui a la valeur maximale. Cet algorithme ne fonctionne néanmoins pas mieux sur l'exemple précédent.

## 9. Leçon 16 et 17 – Gestion de la mémoire primaire

2. Copier une page sur le disque et charger une nouvelle page prend énormément de temps, en raison de la lenteur de la mémoire secondaire. Dans le cas d'un système fortement chargé, ces chargements peuvent avoir lieu trop souvent, et ralentir très fortement la machine jusqu'à la rendre inutilisable. On parle alors de *trashing*.
3. La pagination génère un phénomène de *fragmentation interne* : la dernière page du programme n'est jamais complètement « remplie » (en moyenne, il y a  $\ell/2$  cases mémoires inutilisées, si  $\ell$  est la taille d'une page), et donc, il y a de la mémoire inutilisée dans le cadre de page qui la contient. On pourrait donc penser que diminuer la taille des pages  $\ell$  est une bonne idée, car cela va diminuer la fragmentation. Malheureusement, des pages plus petites impliquent d'avantage de *page faults* et donc d'avantage de lecture sur le disque, ce qui prend toujours du temps. Le problème typique est celui où une matrice est stockée sur plusieurs pages, et où le parcours passe continuellement d'une page à l'autre, pages qui doivent être chargées depuis la mémoire secondaire. Par ailleurs, augmenter le nombre de pages (en diminuant la taille des pages) augmente également la taille de la table des pages qui risque de ne plus tenir en mémoire!
4. Pour pouvoir gérer un *page fault*, il faut exécuter la routine de gestion d'interruption correspondante. Pour cela encore faut-il qu'elle soit en mémoire ! Autrement dit, il ne faut pas que le *page fault* déclenche un nouveau *page fault*. Ces pages doivent donc toujours rester en mémoire.

### 9.3. Exemple : mémoire virtuelle l'Ultra Sparc III

Ce processeur est un processeur 64 bits, mais les adresses mémoire ne peuvent faire que 44 bits. La mémoire est virtuellement découpée en deux blocs : un bloc pour le code et un bloc pour les données, qui commencent tous les deux à 0. Donc une adresse contient :

1. Un bit pour indiquer la zone mémoire
2. 43 bits d'adresse dans cette zone. Un programme peut donc faire au maximum  $2^{43}$  octets.

Par ailleurs, la mémoire physique maximale est limitée à  $2^{41}$  bits. Le mécanisme de traduction des adresses virtuelles en adresses réelles doit donc transformer une adresse sur 64 bits (dont seuls 44 sont significatifs) en une adresse sur 41 bits. La traduction exacte dépend de la taille des pages, qui peut être configurée : soit 8Ko, soit 64Ko, soit 512 Ko, soit 4Mo.

Le problème de ce système est le nombre potentiellement très important de pages. Pour le cas où on a des pages de 8Ko, le décalage dans une page tient sur 13 bits. Cela signifie donc que, des 44 bits significatifs restant, 31 bits servent à coder le numéro de page. Il peut donc y avoir  $2^{31}$  pages, soit plus de 2 milliards de pages. Si un descripteur d'une page tient sur 1 octet (ce qui est très optimiste), une table des pages peut donc occuper plus de 2 Go !

En pratique, l'Ultra Sparc utilise un mécanisme à deux niveaux. Pour chaque partie de la mémoire (code ou données) :

1. Les descripteurs des 64 pages utilisées les plus récemment sont stockés dans un circuit spécial appelé TLB (*Transaction Lookaside Buffer*), qui permet d'obtenir très ra-

### 9.3. Exemple : mémoire virtuelle l'Ultra Sparc III

pidement l'information recherchée (en un seul cycle d'horloge, le numéro de page est comparé à toutes les infos stockées dans la table).

2. Si l'information cherchée est trouvée, elle est transmise au MMU qui travaille normalement (en déclenchant une *page fault* si nécessaire).
3. Sinon, un autre type d'interruption est déclenché : un *TLB miss*, et c'est donc à l'OS de gérer l'absence d'information dans la table des pages en chargeant si nécessaire les informations dans la TLB. Pour ce faire, il peut être aidé par un autre circuit du processeur le TSB *Translation Storage Buffer*, qui est une sorte de TLB plus grande mais plus lente. Au pire cas, ni le TSB ni le TLB ne contiennent l'information demandée, et l'OS doit donc consulter d'autres tables qu'il maintient lui-même (ce qui est le plus lent).

Il faut bien faire la différence entre *TLB miss* et *page fault* :

- Dans un *TLB miss* on recherche l'information sur une page donnée. On ne sait donc pas si la page est présente en mémoire ou pas (elle pourrait l'être). Une fois le *TLB miss* résolu, on peut encore avoir un *page fault*, mais pas toujours.
- Dans un *page fault*, on a l'information sur la page recherchée et on sait qu'elle n'est pas en mémoire. Un *page fault* ne sera jamais suivi d'un *TLB miss* pour la mémoire page.





# 10. Leçon 18 – Gestion des processus et de la mémoire secondaire

## 10.1. Gestion des processus

Comme nous l'avons dit dans l'introduction, un programme est l'ensemble des instructions et de des données qui permettent de réaliser un traitement automatisé. Une fois le *programme* réalisé, il faut l'*exécuter*, c'est-à-dire, faire en sorte que le processeur exécute les instructions du programme. À partir du moment où un programme est exécuté, il devient un *processus*.

Il est important de faire la différence entre *programme* et *processus*. En effet, on peut très bien imaginer, sur un système *multi-tâches*, exécuter plusieurs *copies* du même programme (pour le faire travailler sur des données différentes dans un système interactif, par exemple). Dans ce cas, il faut pouvoir distinguer entre ces différentes versions : ce seront des processus différents. Par ailleurs, un processus est composé du programme qui s'exécute, mais aussi d'un ensemble d'informations qui spécifient l'avancement de l'exécution et qu'on appelle le *contexte*. Ce *contexte* doit contenir au moins les informations suivantes :

- La zone mémoire qui appartient au processus.
- Le compteur de programme, qui indique la prochaine instruction à exécuter. Quand le processus est occupé à s'exécuter, c'est la valeur qui se trouve dans le registre PC.
- Les contenus des différents registres de travail.
- Le *statut* du processus (*cfr. infra*)
- Un pointeur vers un *stack*.

Par ailleurs, si le système possède une mémoire paginée, il faudra également associer au processus une table des pages. Même si le c'est le même programme qui est exécuté dans deux processus différents, on ne peut pas (*a priori*) partager leurs pages en mémoire, car il peut arriver que les processus y écrivent des informations différentes.

Naturellement, la gestion des processus sera une tâche confiée à l'OS. Il reste deux questions à examiner :

1. Quel est le cycle de vie d'un processus ?
2. Comment peut-on exécuter plus de processus qu'il n'y a de CPU disponibles ?

### 10.1.1. Cycle de vie d'un processus

1. **Création** : sur la plupart des OS, un processus ne peut être créé que par l'OS, et uniquement à la demande d'un autre processus. Cette demande se fait à l'aide d'un appel

## 10. Leçon 18 – Gestion des processus et de la mémoire secondaire

système. Après l'appel système, il y a un processus de plus sur le système. Le processus qui a fait la demande de création est appelé *processus père*, et le nouveau processus est appelé *processus fils*.

Lors de la création, l'OS doit essentiellement créer un *nouveau contexte* pour le nouveau processus, et retenir dans ses tables les caractéristiques de ce nouveau processus. Comme un processus ne peut être créé que s'il en existe déjà un autre, il faut qu'un premier processus soit créé au démarrage du système, de manière automatique. Sur les systèmes Unix, par exemple, c'est le processus *init*. Celui-ci, une fois créé, lit un fichier de configuration sur le disque, qui lui indique ce qu'il y a lieu de faire. En général, cela consiste à d'abord exécuter une série d'autres processus en *batch* pour finir d'initialiser le système (configuration de périphériques, lancement de processus d'arrière-plan, *etc*). Ensuite, *init* crée un processus dont le rôle est d'attendre les commandes de l'utilisateur, de les interpréter (à l'aide de l'interpréteur de commande si nécessaire) et de les exécuter. Cela peut être une *invite de commande* où l'utilisateur entre des commandes au clavier de manière textuelle, ou bien un panneau comprenant des boutons sur lesquels on clique pour lancer des applications (barre des tâches, *dock*,...)

2. **Actif** : Lorsque le processus s'exécute sur un des CPU, on dit qu'il est actif.
3. **En attente** : Lorsque le processus n'a pas fini de s'exécuter, qu'il désire disposer du CPU, mais qu'il n'en dispose pas, il est en attente.
4. **Bloqué** : Lorsque le processus n'a pas fini de s'exécuter, mais qu'il n'est pas capable de continuer à s'exécuter (par exemple parce qu'il attend un périphérique), le processus est bloqué.
5. **Terminé** : Lorsque le processus a terminé son exécution, il le signale à l'OS à l'aide d'un appel système. L'OS doit alors libérer les dernières ressources encore utilisées par le processus (mémoire, fichiers encore ouverts,...). Un processus peut aussi être *tué* par l'OS : quand le processus commet une erreur (*trap*, accès à une zone mémoire interdite détectée lors d'un *page fault*), l'OS peut décider de le faire passer à l'état terminé. On risque alors de perdre des données.

**Rem** : sur les systèmes Unix, un processus peut également devenir un *zombie* après être terminé. Cela arrive quand il doit encore renvoyer une valeur à son parent, et que celui-ci ne l'a pas encore lue. Le processus fils est donc bien mort, mais il occupe encore de la place dans les tables de l'OS, comme un processus vivant, pour stocker la valeur de retour. C'est donc bien un mort-vivant...

Un processus peut passer de l'état *actif* à l'état *en attente* de manière volontaire ou non : soit en signalant, à l'aide d'un appel système, à l'OS qu'il ne souhaite plus disposer du CPU temporairement (pour attendre un périphérique, par exemple) ; soit parce qu'il a épuisé son *quantum de temps* (cfr. *time sharing* ci-dessous).

Le passage de l'état *actif* à l'état *bloqué* dépend de la situation du processus et des périphériques. Une fois que le processus est « débloqué » (c'est-à-dire que la condition qui le bloquait est levée), le processus passe en *attente* pour espérer obtenir le CPU.

Par contre, c'est toujours l'OS qui décide quel processus, parmi ceux en attente, va devenir actif. Comme on l'a déjà vu, une partie de l'OS, appelée *ordonnanceur* est en charge de cette

décision. Cette décision sera prise en fonction de plusieurs critères (par exemple si le système est temps réel).

Le changement de processus actif sur un CPU est donc toujours réalisé par l'OS. Pour ce faire, l'OS doit essentiellement réaliser un *changement de contexte* qui consiste à remplacer le contexte de l'ancien processus par le contexte du nouveau afin que celui-ci puisse continuer à s'exécuter correctement. Cela peut naturellement prendre du temps, surtout si le nouveau processus a ses pages présentes en mémoire secondaire et qu'il faut les recharger par exemple.

### 10.1.2. Systèmes en *Time sharing*

La technique du *time sharing* (partage de temps) permet à l'OS de faire exécuter plus de processus qu'il n'y a de CPU disponibles, tout en donnant l'illusion pour l'utilisateur que ces processus s'exécutent véritablement *en parallèle*, c'est-à-dire, comme s'ils s'exécutaient chacun sur un CPU dédié.

Dans ce que nous avons expliqué ci-dessus sur les états des processus, on a vu qu'un processus passait la plus grande partie de sa vie entre les états *actifs*, *bloqué* et *en attente*. Le passage d'*actif* à *bloqué* et de *bloqué* à *en attente* ne dépend pas de l'OS, en général. Par contre, l'OS doit décider quand passer d'*actif* à *en attente* et vice-versa.

Afin de répartir une utilisation *équitable* du (des) CPU(s) entre les processus, le système de *time sharing* consiste à définir un intervalle de temps très court, appelé *quantum de temps*, durant lequel un processus a le droit de s'exécuter. À la fin de cet intervalle, le processus doit libérer le CPU afin qu'un autre processus puisse s'exécuter pour le même *quantum* de temps, et ainsi de suite. L'ordonnanceur du système décide quel est le processus suivant (il peut procéder de manière cyclique, ou bien utiliser une système de priorités, par exemple).

Comment peut-on faire en sorte qu'une fois le quantum de temps épuisé, le processus « rende la main »? Il y a deux techniques :

- *Time sharing coopératif* : le processus consulte régulièrement l'horloge système et rend la main dès qu'il a épuisé son *quantum* de temps. Le bon fonctionnement du système repose donc sur la bonne volonté des programmeurs des programmes actifs, et sur le fait que ceux-ci ne plantent pas en bloquant tout le système... Cette solution a été utilisée dans les anciennes versions de MacOS et de Windows, par exemple. Ces techniques ont l'avantage d'être hautement prévisibles, si on connaît bien les programmes exécutés, et sont donc encore souvent utilisés dans le cadre des systèmes embarqués.
- *Time sharing préemptif* : l'horloge système est programmée pour déclencher, à intervalle réguliers (séparés par le *quantum de temps*), une interruption. Le processus actif sera donc interrompu *volens nolens*, et l'OS effectuera le changement de contexte approprié.

En cas de surcharge du système (trop de processus), les changements de contexte peuvent devenir très fréquents. Si les processus présents consomment beaucoup de mémoire, il y a un risque que chaque changement de contexte entraîne le chargement d'une page mémoire depuis la mémoire secondaire. Les changements de contexte prennent alors énormément de

temps, au pire cas, plus de temps que l'exécution des processus eux-mêmes. On a alors un phénomène de *trashing*.

## 10.2. Entrées/sorties virtuelles et systèmes de fichiers

L'organisation de la mémoire secondaire pose *grosso modo* les mêmes problèmes que celle de la mémoire primaire :

1. Les processus utilisateurs ne peuvent pas avoir accès *directement* à la mémoire secondaire, et ne peuvent donc pas décider de lire ou écrire à une adresse précise.
2. Les données appartenant aux différents utilisateurs doivent être bien séparées et organisées. Un utilisateur ne peut pas avoir la possibilité d'accéder aux données d'un autre utilisateur sans que ce dernier ne lui ait donné la permission. Il faut aussi protéger les données contre les erreurs, effacement involontaires, *etc.*
3. Pour simplifier le travail des programmes utilisateur, les données doivent être organisées de manière linéaire, comme une séquence d'octets. Concrètement, cela n'est pas toujours possible, car les espaces disponibles ne sont pas toujours assez grands.

### 10.2.1. Fichiers

Pour résoudre ces problèmes, l'OS présente les données sous la forme d'une abstraction appelée « fichier ». Un fichier est simplement une séquence d'octets, à laquelle les programmes utilisateurs peuvent accéder en utilisant des appels systèmes. Concrètement, quand un processus veut lire ou écrire en mémoire secondaire :

1. Il doit identifier un fichier dans lequel lire ou écrire. Le cas échéant, il doit demander à l'OS d'en créer un.
2. Il doit demander à l'OS d'*ouvrir* ce fichier, c'est-à-dire obtenir un accès à ce fichier.
3. Il doit faire appel à l'OS pour lire ou écrire les données de manière séquentielle : d'abord la première, puis la seconde, *etc.* Il est aussi parfois possible de revenir en arrière, ou de faire une « avance rapide ».
4. Il doit finalement *fermer* le fichier grâce à un appel système, c'est-à-dire relâcher l'accès qu'il possède à ce fichier.

Comme on le voit, le programme utilisateur voit le fichier comme une séquence d'octets qui ont chacun une adresse, qui est leur décalage par rapport au début du fichier. Ainsi, dans chaque fichier, la première information se trouve au décalage 0. Ces décalages ne sont évidemment pas les adresses sur le disque : l'OS doit se charger d'effectuer une *traduction* d'adresse, un peu comme le MMU dans le cas de la mémoire primaire. On a donc à nouveau affaire à des adresses *virtuelles* qui doivent être traduites en adresse *réelles*.

Concrètement, la mémoire secondaire est divisée en *blocs* dont la taille peut varier en fonction du type de mémoire. Chaque bloc est numéroté, comme les cases mémoire en mémoire secondaire. Afin de pouvoir effectuer la traduction des adresses, l'OS maintient plusieurs tables :

1. Une *table d'allocaiton des fichier* (*File Allocation Table*) : qui reprend la liste de tous les fichiers présents sur le système, et des informations annexes (permissions, possesseur, date de création, etc).
2. Pour chaque fichier (donc chaque entrée de la table d'allocation) : une liste des blocs qui constituent le fichier.

Par ailleurs, pendant l'exécution d'un processus, l'OS lui associe une table qui collecte tous les fichiers ouverts par ce processus. Pour chaque fichier ouvert, on trouve un *descripteur de fichier* qui contient plusieurs informations, notamment :

1. L'identification du fichier, par exemple, son numéro d'entrée dans la table des fichiers.
2. Des informations annexes : on peut avoir la permission de lire mais pas d'écrire par exemple...
3. Un pointeur vers un *cache* (ou *buffer*) en mémoire où copier/lire les données à lire/écrire dans le fichier.
4. Un pointeur indiquant le décalage courant dans le fichier.

En cas de lecture, on a généralement un comportement qui consiste à :

1. Copier le bloc à l'adresse virtuelle donnée par le décalage du descripteur de fichier dans le *cache*.
2. Incrémenter ce pointeur.

En cas d'écriture, la situation est un peu plus délicate si on désire faire une *insertion* (c'est-à-dire ne pas écraser les informations courantes, ou allonger le fichier). Dans ce cas, le fichier va devoir recevoir un ou plusieurs *nouveaux blocs*, que l'OS va lui attribuer. Pour ce faire l'OS doit également maintenir une *liste des blocs libres*. En cas d'allongement d'un fichier l'OS doit :

1. Supprimer un bloc de la liste des blocs libres.
2. L'insérer au bon endroit dans la liste des blocs du fichier.

On procède de manière inverse en cas de libération de bloc.

**Problèmes liés à la politique par blocs : fragmentation** Un des risques liés à ce système est de voir les blocs qui constituent un même fichier se retrouver éparpillés (de manière non-contiguë) sur la mémoire secondaire. Cela peut poser problème pour les lecteurs de disque dur ou de CDROMs par exemple, car cela entraîne un déplacement continu de la tête de lecture, ce qui prend beaucoup de temps. On peut implémenter dans l'OS des politiques d'attribution des blocs qui évitent ces problèmes. Sur certains OS, il existe par contre un programme dédié qui se charge de remplacer correctement les blocs sur le disque pour diminuer la fragmentation, mais cela prend beaucoup de temps...

### 10.2.2. Répertoires

Afin de pouvoir mieux organiser l'information, on peut répartir (virtuellement toujours) les fichiers dans différents *rédertoires*. Un répertoire est une sorte de conteneur pour :

## 10. Leçon 18 – Gestion des processus et de la mémoire secondaire

1. d'autres répertoires
2. des fichiers

Comme les fichiers, les répertoires portent un nom et ont des permissions d'accès. Comme un répertoire peut contenir d'autres répertoires, l'organisation de la mémoire secondaire devient une *hiérarchie* avec un *répertoire principal* (appelé « racine ») et des sous-répertoires. Concrètement :

1. La structure en répertoires doit apparaître dans la table d'allocation, qui a maintenant une structure *arborescente*.
2. Le *nom* d'un fichier doit maintenant être complété par le nom du (des) répertoire(s) qui le contient (contiennent). L'affichage de ce nom à destination de l'utilisateur utilise certaines conventions qui dépendent de l'OS. Par exemple /rep1/rep2/fichier sous Unix, \rep1\rep2\fichier sous Windows ou encore rep1:rep2:fichier sous MacOS.

L'ensemble des répertoires et des fichiers ainsi organisés est appelé un *système de fichier*.

### 10.2.3. Quelques remarques

1. Beaucoup de périphériques se comportent, à un certain niveau d'abstraction, comme des fichiers, c'est-à-dire comme des séquences d'octets. Par exemple, le clavier émet une séquence d'octets représentant les touches enfoncées, qu'on peut lire comme on lit un fichier. De même, pour commander une imprimante, on peut lui envoyer une séquence d'octets comme si on écrivait dans un fichier. C'est également le cas d'autres services de l'OS, par exemple la génération de nombres aléatoires.  
C'est pourquoi, sur plusieurs OS, l'accès à ces périphériques est possible à travers des *fichiers virtuels* : ce sont des fichiers qui ne sont pas physiquement présents sur le disque, mais qui apparaissent quand même dans la table d'allocation des fichiers avec un *flag* spécial qui indique le statut particulier. Quand on y accède l'OS les traite de façon différente. Par exemple :
  - Si on lit un tel fichier qui correspond à un périphérique, l'OS va lire les données fournies par le périphérique pour alimenter le cache. À aucun moment les données lues ne seront stockées sur le disque.
  - Si on lit /dev/random sur un système Unix, l'OS génère des nombres aléatoires pour alimenter le cache mais ne les stocke pas sur le disque non plus. Pour avoir des zéros, on peut utiliser /dev/zero.
  - Certains de ces fichiers spéciaux permettent l'écriture au lieu de la lecture : par exemple /dev/null qui ne fait... rien avec les données qu'on y écrit.
2. Comme le système de fichier est une abstraction de la mémoire secondaire, on peut aisément ajouter des *services* utiles à l'utilisateur :
  - La possibilité d'avoir des *liens symboliques*, c'est-à-dire des fichiers *virtuels* qui sont en fait des pointeurs vers d'autres fichiers. Quand on ouvre le lien virtuel, on ouvre en fait le fichier vers lequel il pointe, qui peut se trouver à un endroit

différent du système de fichier. Cela évite de devoir maintenir plusieurs copies du même fichier à des endroits différents du système. À nouveau, ce type de fichier est identifié par un *flag* spécial dans la table d’allocation.

- La possibilité d’avoir un *indexage* automatique du contenu des fichiers (l’index étant stocké dans les blocs libres du système de fichiers), comme le système *Spotlight* d’Apple.
  - La possibilité d’avoir des systèmes tolérants aux pannes, qui sont capables de remettre le système dans un état cohérent en cas de panne durant une écriture qui n’aurait pas été complètement terminée. C’est le cas sur les systèmes Linux avec le système de fichier Ext3.
3. Le cas où plusieurs périphériques de mémoire secondaire sont présents sur le système est géré de manière différente par différents OS :
- Sur les OS « à la Windows » : chaque périphérique porte un nom différent, en général une lettre. Le nom complet d’un fichier commence donc par cette lettre : C:\Windows\...\fichier. Chaque périphérique possède donc son propre système de fichiers.
  - Sur les OS « à la Unix » : un périphérique principal contient le système de fichier *racine*, c’est-à-dire celui qui contient le répertoire racine. Les autres périphériques apparaissent comme des sous-répertoires (pas nécessairement directs) de ce système (on parle de *monter* un périphérique). Ainsi, tous les périphériques apparaissent dans un seul système de fichier. Ce système permet également d’avoir des répertoires virtuels...





# Bibliographie

- [1] Mike BANAHAN, Declan BRADY et Mark DORAN. *The C Book*. Addison Wesley, 1991. URL : [http://publications.gbdirect.co.uk/c\\_book/](http://publications.gbdirect.co.uk/c_book/).
- [2] George BOOLE. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. Une édition récente (1954) est disponible en ligne à l'adresse <http://www.gutenberg.org/etext/15114>. Walton & Maberly, 1854.
- [3] *Code page 737 — Wikipedia, The Free Encyclopedia*. URL : [https://en.wikipedia.org/w/index.php?title=Code\\_page\\_737&oldid=782268129](https://en.wikipedia.org/w/index.php?title=Code_page_737&oldid=782268129) (visité le 01/09/2017).
- [4] Richard W. HAMMING. “Error detecting and error correcting codes”. In : *Bell System Technical Journal* 29.2 (1950), p. 147-160.
- [5] *IEEE Standard for Floating-Point Arithmetic*. Rapp. tech. 754-2008. IEEE, août 2008, p. 1-70. DOI : 10.1109/IEEEESTD.2008.4610935. URL : <https://ieeexplore.ieee.org/servlet/opac?punumber=4610933>.
- [6] *Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification*. Rapp. tech. ISO/IEC 18004 :2015. International Organization for Standardization, 2015. URL : <https://www.iso.org/standard/62021.html>.
- [7] *Information technology – Computer graphics and image processing – Portable Network Graphics (PNG) : Functional specification*. Rapp. tech. ISO/IEC 15948 :2004. International Organization for Standardization, 2004. URL : <https://www.iso.org/standard/29581.html>.
- [8] *Information technology – Digital compression and coding of continuous-tone still images : Requirements and guidelines*. Rapp. tech. ISO/IEC 10918-1 :1994. International Organization for Standardization, 1994. URL : <https://www.iso.org/standard/18902.html>.
- [9] *Intel 486DX DataSheet*. Rapp. tech. Intel Corporation, 1993. URL : <http://datasheetschipdb.org/Intel/x86/486/datashts/240440-006.pdf>.
- [10] *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Intel Corporation, 2017. URL : <https://software.intel.com/en-us/articles/intel-sdm>.
- [11] John von NEUMANN. *First Draft of a Report on the EDVAC*. Rapp. tech. University of Pennsylvania, 1945. URL : <https://archive.org/download/firstdraftofrepo00vonn/firstdraftofrepo00vonn.pdf>.
- [12] *Ordinateur — Wikipédia, l’encyclopédie libre*. URL : <https://fr.wikipedia.org/w/index.php?title=Ordinateur&oldid=150970460> (visité le 16/08/2018).

## Bibliographie

- [13] Brousentsov N. P. et al. *Development of ternary computers at Moscow State University*. URL : <http://www.computer-museum.ru/english/setun.htm> (visité le 08/08/2018).
- [14] David A. PATTERSON et David R. DITZEL. "The Case for the Reduced Instruction Set Computer". In : *SIGARCH Comput. Archit. News* 8.6 (oct. 1980), p. 25-33. ISSN : 0163-5964. DOI : 10.1145/641914.641917. URL : <http://doi.acm.org/10.1145/641914.641917>.
- [15] *Quantities and units – Part 13 : Information science and technology*. Rapp. tech. IEC 80000-13 :2008. International Organization for Standardization, 2008. URL : <https://www.iso.org/standard/31898.html>.
- [16] Irving S. REED et Gustave SOLOMON. "Polynomial Codes Over Certain Finite Fields". In : *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), p. 300-304.
- [17] Claude E. SHANNON. "A symbolic analysis of relay and switching circuits". Mém. de mast. MIT, 1937. URL : <https://dspace.mit.edu/handle/1721.1/11173>.
- [18] Bjarne STROUSTRUP. *A Tour of C++*. Addison-Wesley, 2018.
- [19] Jonhathan SWIFT. *Gulliver's Travels Into Several Remote Regions of the World*. Version en ligne du projet Guttenberg. Balliet, Thomas M. (Thomas Minard). URL : <http://www.gutenberg.org/ebooks/17157>.
- [20] *Tagbanwa*. Rapp. tech. The Unicode Consortium, 1991. URL : <http://www.unicode.org/charts/PDF/U1760.pdf>.
- [21] *Tagbanwa (Unicode block) — Wikipedia, The Free Encyclopedia*. URL : [https://en.wikipedia.org/w/index.php?title=Tagbanwa\\_\(Unicode\\_block\)&oldid=775143371](https://en.wikipedia.org/w/index.php?title=Tagbanwa_(Unicode_block)&oldid=775143371) (visité le 01/09/2017).
- [22] Andrew TANENBAUM. *Structured Computer Organisation, 5th edition*. Prentice Hall.
- [23] Alan M. TURING. "On Computable Numbers, with an Application to the Entscheidungsproblem". In : *Proceedings of the London Mathematical Society*. T. 42. 1936, p. 230-265. URL : <https://doi.org/10.1112/plms/s2-42.1.230>.
- [24] *VAX-11 Architecture Reference Manual*. Rapp. tech. revision 6.1. Digital Equipment Corporation, 1982. URL : [http://www.bitsavers.org/pdf/dec/vax/archSpec/EK-VAXAR-RM-001\\_Arch\\_May82.pdf](http://www.bitsavers.org/pdf/dec/vax/archSpec/EK-VAXAR-RM-001_Arch_May82.pdf).
- [25] Pierre WOLPER. *Introduction à la calculabilité : cours et exercices corrigés*. Dunod, 2006.