

Introduction à la structure des ordinateurs et au stockage des informations

Éléments d'histoire de l'informatique
support de cours, diffusion interdite

D. Roegel

17 août 2023

Table des matières

1 Structure des ordinateurs	11
1.1 Mémoire	11
1.2 Organe de calcul	12
1.3 Architecture de von Neumann	14
1.4 Architecture de type Harvard	14
1.5 Composants	17
1.6 Un peu d'histoire	20
1.6.1 Différentes sortes de calculateurs	20
1.6.2 De la mécanique à l'électronique	29
1.7 Suppléments	39
1.8 Exercices	39
2 Représentation des données	41
2.1 Représentation des caractères	41
2.1.1 Le code ASCII	43
2.1.2 Autres codes 8 bits	43
2.1.3 Pages nationales	44
2.1.4 Unicode	44
2.1.5 UTF-8	47
2.2 Représentation des données multimédia	48
2.2.1 Son	48
2.2.2 Image	48
2.2.3 Vidéos	49
2.3 Suppléments	50
2.4 Un peu d'histoire	51

2.5 Exercices	55
3 Systèmes de numération	59
3.1 Le système décimal	59
3.2 Les systèmes de numération positionnels	60
3.3 Le système binaire	60
3.4 Conversion entre binaire et décimal	62
3.5 Notation hexadécimale	66
3.6 Un peu d'histoire	67
3.7 Exercices	69
4 Logique numérique	71
4.1 Logique formelle	71
4.2 Algèbre booléenne	71
4.3 Portes logiques	74
4.4 Circuits combinatoires	80
4.4.1 Multiplexeurs	84
4.4.2 Décodeurs	88
4.4.3 Démultiplexeur	92
4.4.4 Additionneurs	93
4.5 Un peu d'histoire	98
4.6 Exercices	102
5 Arithmétique de l'ordinateur	105
5.1 L'unité arithmétique et logique	105
5.2 Représentation des entiers	107
5.2.1 Complément à deux	108
5.2.2 Extension de plage	112
5.3 Arithmétique entière	112
5.3.1 Négation	112
5.3.2 Addition et soustraction	113
5.3.3 Multiplication	116
5.3.4 Division	120
5.4 Représentation en virgule flottante	122
5.5 La norme IEEE754	126

5.6	Arithmétique en virgule flottante	128
5.6.1	Addition et soustraction	129
5.6.2	Multiplication et division	131
5.6.3	Arrondis	134
5.6.4	Gestion de l'infini	134
5.6.5	Valeurs anormales	135
5.6.6	Valeurs sous-normales	135
5.7	Histoire des algorithmes	137
5.8	Et encore un peu d'histoire	139
5.9	Exercices	143
6	Fonctionnement d'un ordinateur	149
6.1	Instruction de récupération et d'exécution	149
6.2	Une machine hypothétique	151
6.3	Notion d'interruption	154
6.4	Structures d'interconnexion	157
6.5	Langage machine	161
6.5.1	Adresses mémoires	161
6.5.2	Opérations mémoire	162
6.5.3	Instructions	162
6.5.4	Exemples	164
6.5.5	Un parallèle avec Java	166
6.6	Familles de processeurs	168
6.7	Exercices	172
6.8	Un peu d'histoire	174
7	Mémoire	177
7.1	Mémoire cache	180
7.2	Mémoire principale, interne, ou vive	186
7.2.1	ROM	194
7.2.2	Puces	197
7.2.3	Correction des erreurs	205
7.2.4	Améliorations de la DRAM	205
7.2.5	Mémoires flash	205
7.3	Mémoire externe	207

7.3.1	Disque magnétique	207
7.3.2	Disquette (1967-2011)	207
7.3.3	SSD	215
7.3.4	Mémoire optique	216
7.3.5	Bandes magnétiques	216
7.4	Un peu d'histoire	219
7.5	Suppléments	220
7.6	Exercices	221
8	Gestion de processus	225
8.1	Ordonnancement	225
8.1.1	Swapping	226
8.2	Gestion de la mémoire	230
8.2.1	Partitionnement (ou segmentation)	231
8.2.2	Pagination	234
8.2.3	Pagination à la demande	239
8.2.4	Espace d'adressage	239
8.2.5	Traduction des adresses	242
8.3	Exercices	248
9	Entrées/sorties	251
9.1	Appareils externes	251
9.2	Modules E/S	254
9.3	Opérations E/S	255
9.4	Un peu d'histoire	256
9.5	Exercices	260
10	Petit mémento de C	261
10.1	Édition et compilation	262
10.2	La structure d'un programme C	263
10.3	Types et variables	264
10.4	Entrées/sorties	264
10.5	Structures de contrôle	265
10.6	Appels de fonction	265
10.7	Un exemple un peu plus complexe	266

TABLE DES MATIÈRES

7

10.8 Valeurs signées et non signées	267
10.9 Tableaux	268
10.10 Chaînes de caractères	269
10.11 Manipulation de bits	273
10.12 Accès à la mémoire	274
10.13 Les adresses et les tableaux	277
10.14 Compléments sur la compilation	278
10.15 Autres sujets	279
10.16 Un peu d'histoire	280
10.17 Exercices	283

Introduction

Ce cours propose une introduction à la structure et au fonctionnement des ordinateurs. Il couvre aussi la représentation des données, les systèmes de numération, la logique numérique, comment les calculs sont faits dans l'ordinateur, comment les programmes sont exécutés, la mémoire, la gestion des processus ainsi que les entrées/sorties.

Au cours de cette introduction, on donne de fréquents compléments historiques. De nombreux exercices sont proposés, un certain nombre d'entre eux reposant sur le langage C qui est introduit en annexe.

Ce document est notamment inspiré des ouvrages suivants :

- Carl Hamacher et al. : *Computer organization and embedded systems*, 2012
- William Stallings : *Computer organization and architecture*, 2016
- Gerard O'Regan : *A brief history of computing*, 2021.

mais il reprend aussi beaucoup d'informations d'autres sources, notamment de Wikipédia pour les illustrations. Les exercices reprennent en partie ceux du cours des années précédentes.

Pour certains chapitres, on donne aussi des compléments vidéos.

Chapitre 1

Structure des ordinateurs

Les ordinateurs (numériques) peuvent être bâtis selon diverses architectures, mais les composants essentiels sont un organe de calcul, un module de mémoire et des modules d'entrée-sortie. Ces composants sont interconnectés d'une certaine manière pour réaliser la fonction de base d'un ordinateur qui est d'exécuter des programmes.

En adoptant une vue d'ensemble, on peut caractériser un ordinateur en décrivant d'une part le comportement externe de chaque composant, c'est-à-dire les données et signaux de contrôle qu'il échange avec les autres composants, et d'autre part la structure connectant les composants.

1.1 Mémoire

Une mémoire est un élément permettant de stocker une certaine quantité de données et d'instructions. Ces données peuvent être lues ou écrites. L'unité de base de la mémoire est le bit qui permet de stocker deux états différents, notés habituellement 0 et 1. On parle alors de mémoire binaire. Les multiples binaires du bit seront notés avec les préfixes Ki (2^{10}), Mi (2^{20}), Gi (2^{30}), Pi (2^{40}), plutôt que K, M, G, P, en conformité avec la norme ISO/IEC 80000, afin d'éviter la confusion avec les puissances de 10 qui sont proches de ces valeurs.

Les contenus de la mémoire sont accessibles par leurs adresses,

indépendamment du type des données contenues.

Notons qu'il est possible d'envisager des dispositifs élémentaires qui stockent plus de deux états. Il y a ainsi eu des ordinateurs ternaires¹.

1.2 Organe de calcul

L'organe de calcul est le composant chargé de faire des calculs, des opérations logiques (booléennes) et aussi de déterminer l'ordre dans lequel s'exécutent les instructions d'un programme. C'est le « cerveau » de l'ordinateur.

On peut envisager plusieurs approches pour l'organe de calcul. Dans une approche purement « matérielle » (*hardware*) (figure 1.1), il n'y aurait que des données et des fonctions arithmétiques et logiques câblées « en dur » produiraient les résultats.

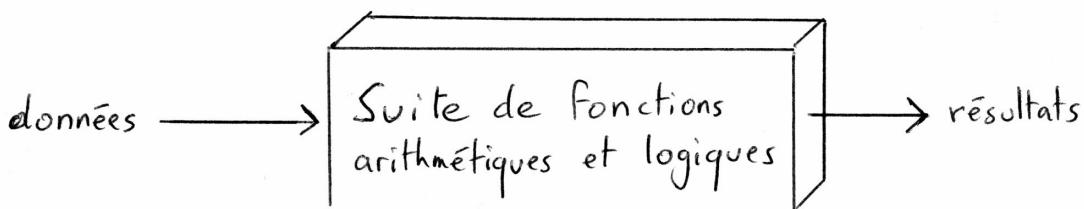


FIGURE 1.1 – Programmation matérielle.

Il pourrait par exemple y avoir un bloc pour faire une certaine opération arithmétique comme une addition ou une multiplication,

1. Pour plus d'informations sur les ordinateurs ternaires, voir les liens suivants :

- https://fr.wikipedia.org/wiki/Ordinateur_ternaire
- <https://louis-dr.github.io/ternalu3.html>
- <https://fr.wikipedia.org/wiki/Setun>
- https://fr.wikipedia.org/wiki/Logique_ternaire

mais de manière câblée.

Une telle approche manque de flexibilité, parce qu'elle ne fait pas ressortir la notion d'instructions.

Dans une approche « logicielle » (*software*) (figure 1.2), il y a à la fois des données et des instructions et les instructions sont d'abord traitées séparément pour produire des signaux qui influencent le traitement des données. Cette approche est beaucoup plus flexible, puisque l'on a alors un organe arithmétique et logique général dont la fonction n'est pas toujours la même.

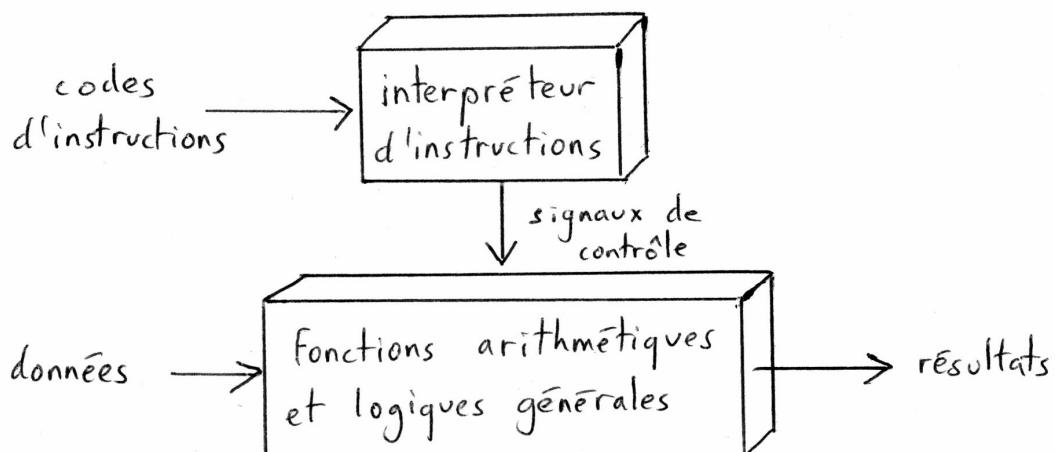


FIGURE 1.2 – Programmation logicielle.

Cette nouvelle approche est plus pratique, parce qu'elle introduit la notion de *programme*. Pour réaliser un autre travail, il n'est pas nécessaire de réaliser un nouveau « câblage », mais simplement de fournir une nouvelle liste d'instructions.

Les deux blocs de la précédente figure font partie du CPU (*Central Processing Unit*). Le CPU, avec les composants de mémoire et d'entrée-sortie constituent les éléments fondamentaux d'un ordinateur.

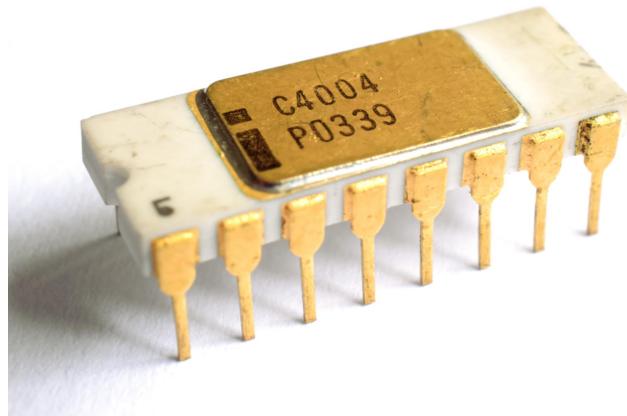


FIGURE 1.3 – L'un des premiers microprocesseurs commercialisés, l'Intel 4004 (1971). (source : Wikipédia)

1.3 Architecture de von Neumann

La conception des ordinateurs s'appuie aujourd'hui presque universellement sur des concepts développés par John von Neumann (1903-1957) (figure 1.4).

Il y a trois concepts fondamentaux :

- les données et instructions sont stockées dans une unique mémoire qui peut être lue ou écrite ;
- les contenus de cette mémoire sont adressables par leur localisation, indépendamment de leur type (données, instructions, etc.) ;
- l'exécution se fait de manière séquentielle d'une instruction à la suivante.

1.4 Architecture de type Harvard

L'architecture de type Harvard est une conception des processeurs qui sépare physiquement la mémoire des données et la mé-



FIGURE 1.4 – John von Neumann (1903-1957). (source : Wikipédia)

moire des programmes. L'accès à chacune des deux mémoires s'effectue via deux bus distincts. (Wikipédia)

Cette architecture est souvent rapportée à l'ordinateur Harvard Mark I, mais en fait la dénomination est beaucoup plus récente.



FIGURE 1.5 – L'ordinateur Harvard Mark I (1943). (source : <https://cdn.britannica.com>)

1.5 Composants

La figure 1.6 montre finalement les composants d'un ordinateur.

Il y a trois modules qui sont reliés par des *bus*. La mémoire contient ici n cases, numérotées de 0 à $n-1$. Ces cases peuvent contenir des instructions ou des données.

Le CPU échange des données avec la mémoire et contient pour cela :

- un registre renfermant l'adresse de la prochaine case mémoire à lire ou écrire (*memory address register*, MAR);
- un registre renfermant la donnée à écrire ou la donnée lue depuis la mémoire (*memory buffer register*, MBR).

Par ailleurs, le CPU contient :

- un registre contenant l'adresse de la prochaine instruction à récupérer de la mémoire (PC, *Program Counter*);
- un registre contenant l'instruction qui a été récupérée (IR, *Instruction Register*);
- un registre qui spécifie un périphérique particulier (*I/O address register*) ;
- un autre registre qui sert à l'échange de données avec un périphérique particulier (*I/O buffer register*).

La figure 1.7 montre la structure de l'un des premiers microprocesseurs, le 6502, qui a été à l'origine du succès des premiers ordinateurs d'Apple.

Un module de périphérique contient aussi des buffers internes pour stocker temporairement des données jusqu'à ce qu'elles puissent être transmises.



FIGURE 1.6 – Composants d'un ordinateur

1.5. COMPOSANTS

19

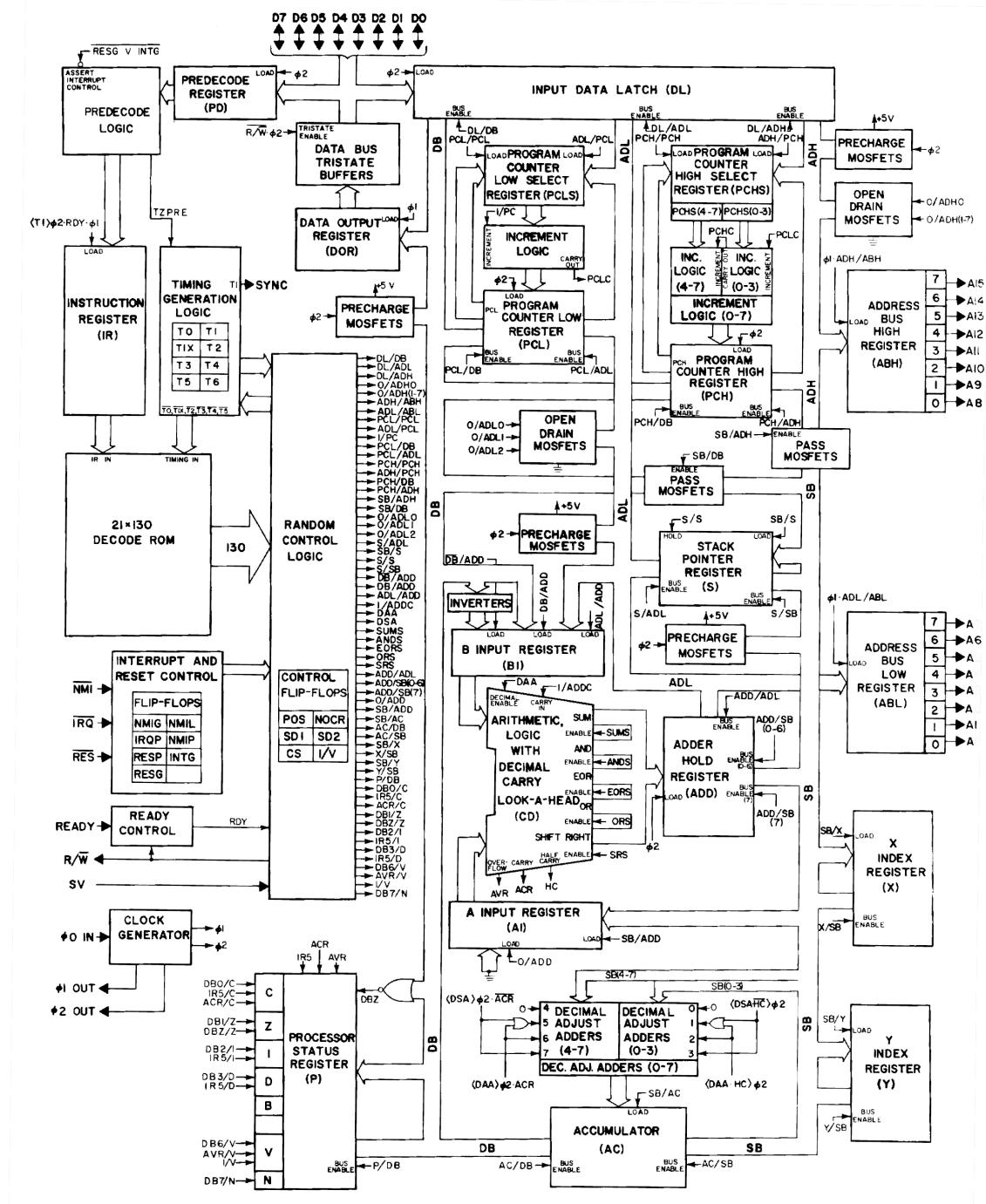


FIGURE 1.7 – Diagramme de bloc du microprocesseur 6502 (1975).
(source : Wikipédia)

1.6 Un peu d'histoire

1.6.1 Différentes sortes de calculateurs

- les premiers ordinateurs : Babbage, etc.
- calcul de différences
- calcul analogique

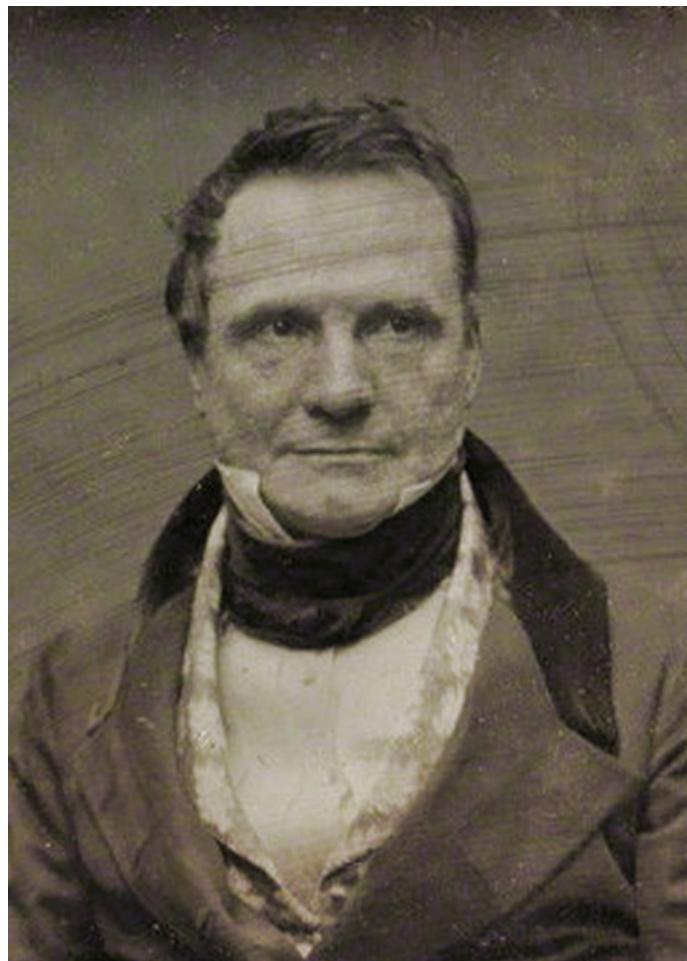


FIGURE 1.8 – Charles Babbage (1791-1871). (source : Wikipédia)

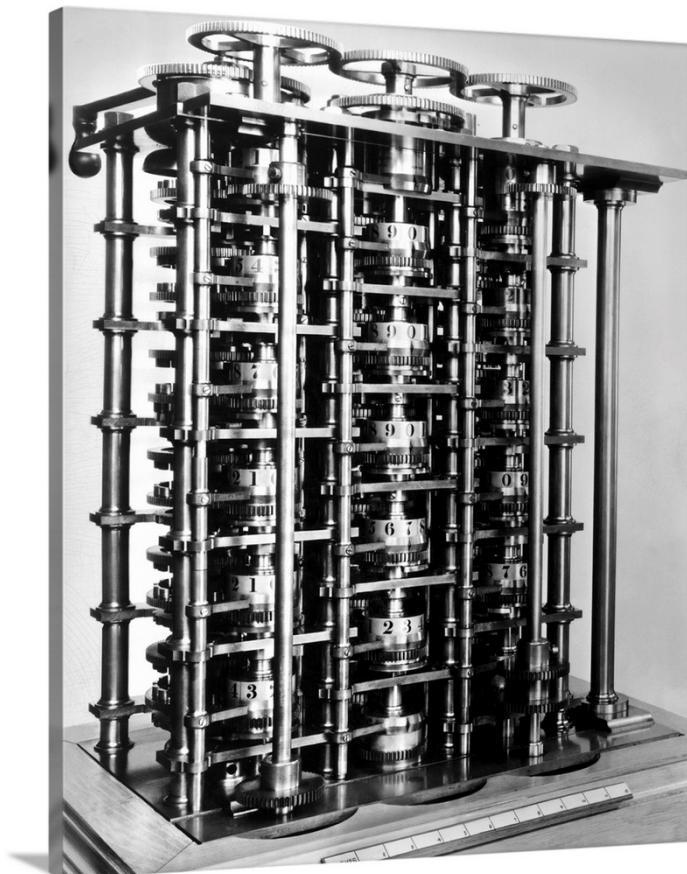


FIGURE 1.9 – Fragment de la première machine à différences de Babbage (1832). (source : <https://static.greatbigcanvas.com>)



FIGURE 1.10 – Fragment de la machine analytique de Babbage (c1871). (source : Wikipédia)

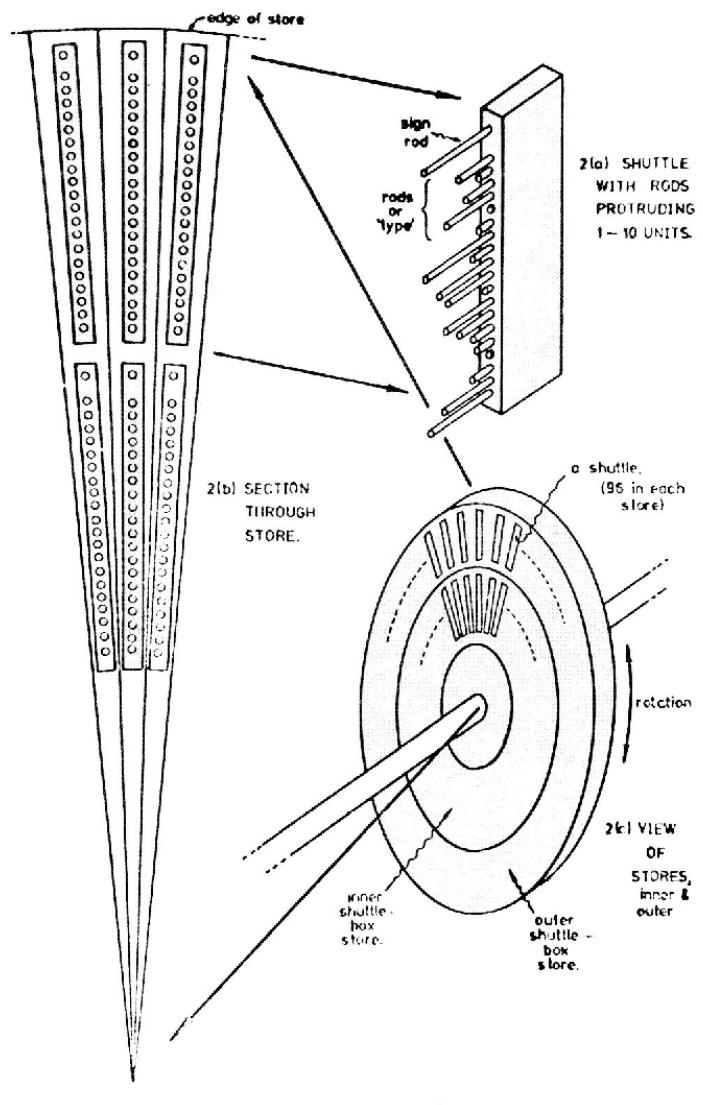


FIGURE 1.11 – Reconstruction hypothétique de la mémoire (mécanique) de la machine analytique de Ludgate. (source : B. Randell, *From Analytical Engine to Electronic Digital Computer: The Contributions of Ludgate, Torres, and Bush*, 1982)

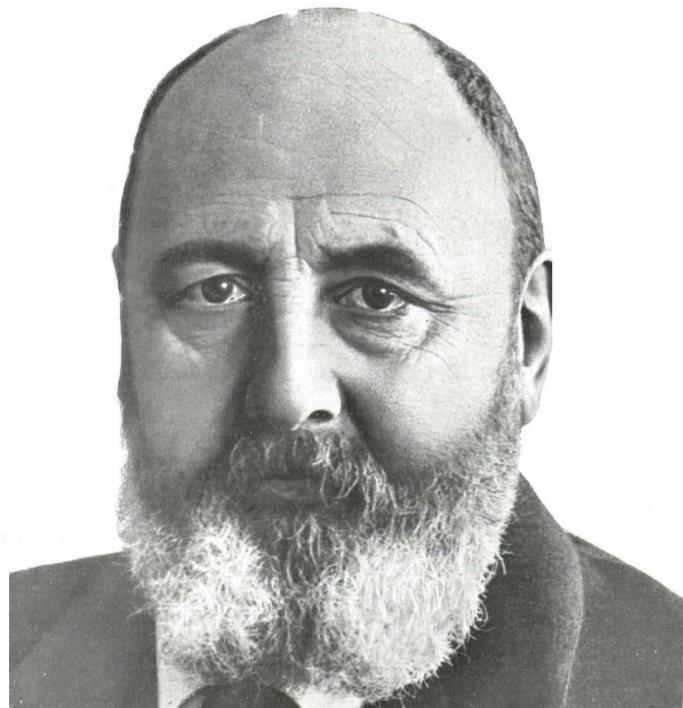


FIGURE 1.12 – Leonardo Torres Quevedo (1852-1936), auteur d'une machine analytique en 1920. (source : Wikipédia)



FIGURE 1.13 – Analyseur différentiel de Vannevar Bush (1931).
(source : Wikipédia)

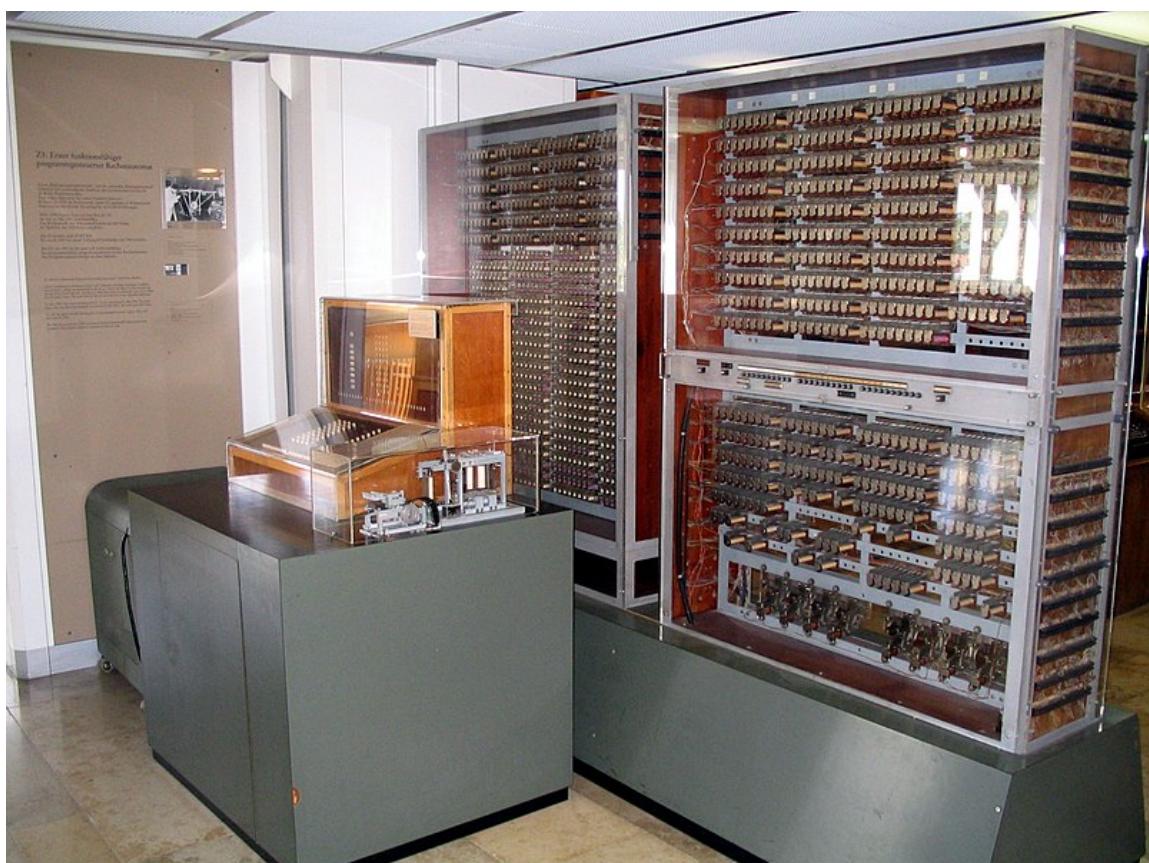


FIGURE 1.14 – Réplique de la machine Z3 à relais électromécaniques de Konrad Zuse (1941). (source : Wikipédia)

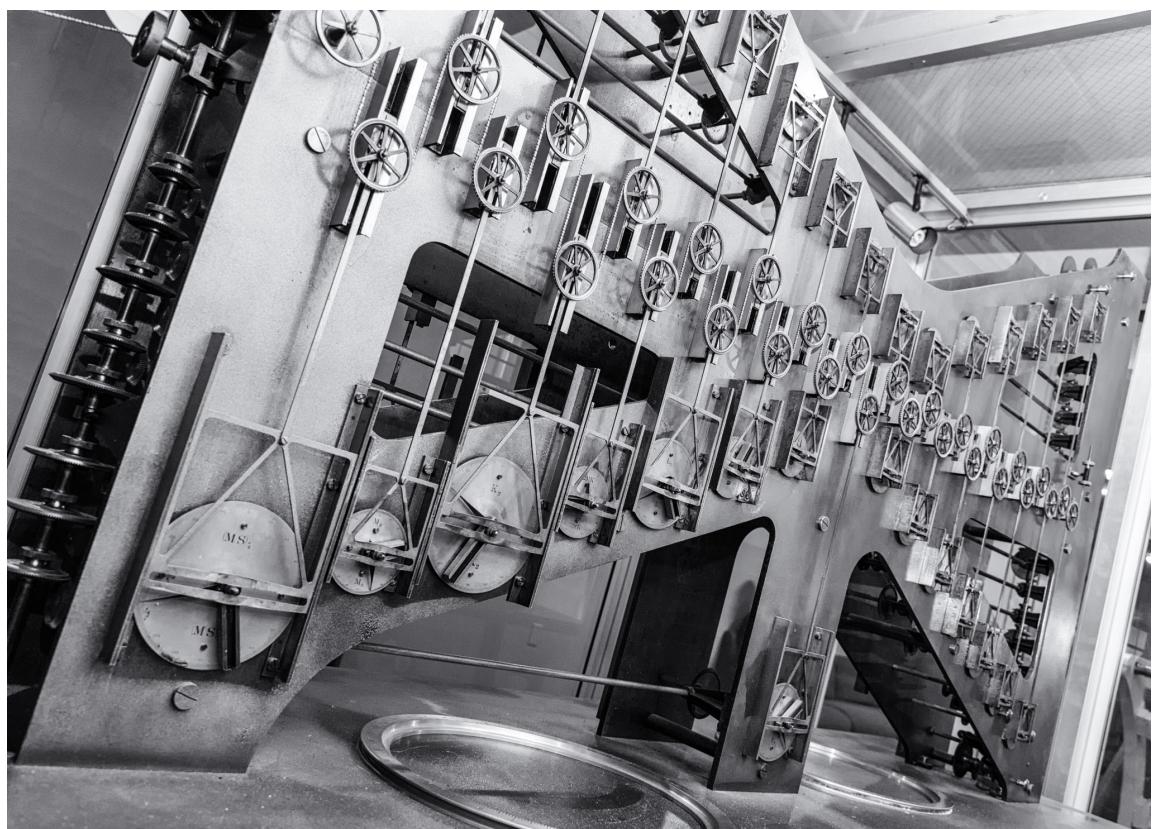


FIGURE 1.15 – Une machine à prédire les marées (*The United States Tide Predicting Machine No. 2, 1910*) (source : Wikipédia)



FIGURE 1.16 – Calculateur analogique de 1949. (source : Wikipédia)

1.6.2 De la mécanique à l'électronique

En même temps que l'automatisation prenait des formes de plus en plus complexes, les éléments de base des machines évoluaient aussi. Au début mécaniques, ils sont devenus électromécaniques puis électroniques. Les grandes étapes des composants ont été les relais électromécaniques, les tubes électroniques (lampes à vide) et les transistors.

Un relais permet de distribuer du courant à partir d'une information logique. Il y a deux circuits isolés mais connectés mécaniquement.

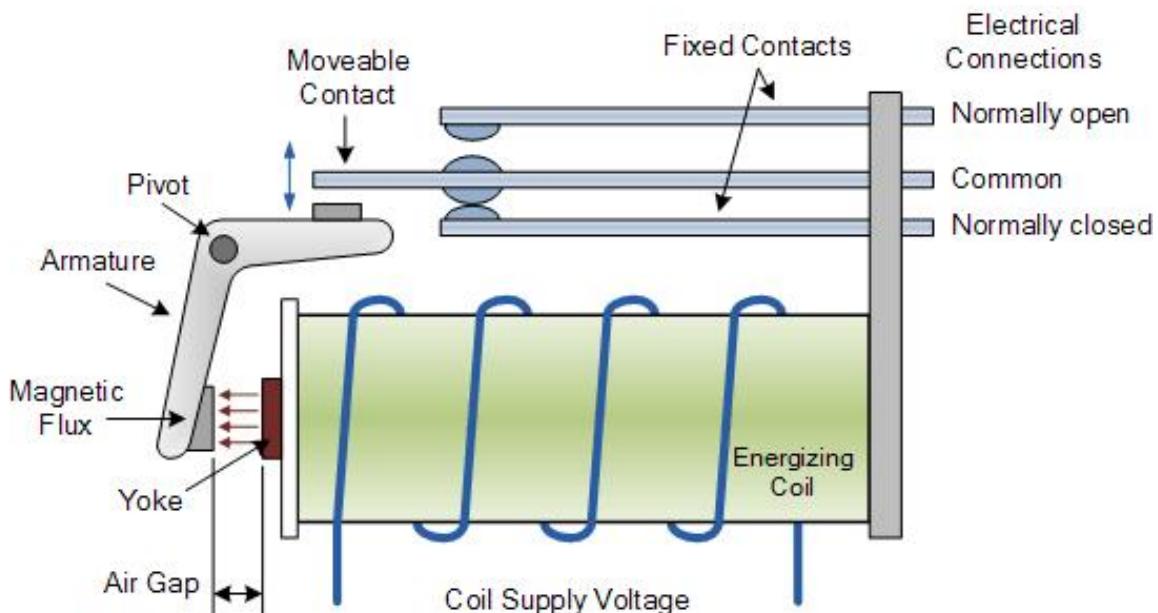


FIGURE 1.17 – Le fonctionnement d'un relais. (source : <https://www.mroelectric.com>)

Un tube électronique est un composant électronique permettant notamment d'amplifier des signaux. Il en existe de nombreuses sortes, notamment la triode inventée en 1907 par Lee De Forest. Les tubes ont joué un rôle essentiel dans le développement de la radio, de la télévision, du radar, etc. Les tubes ont aujourd'hui presque

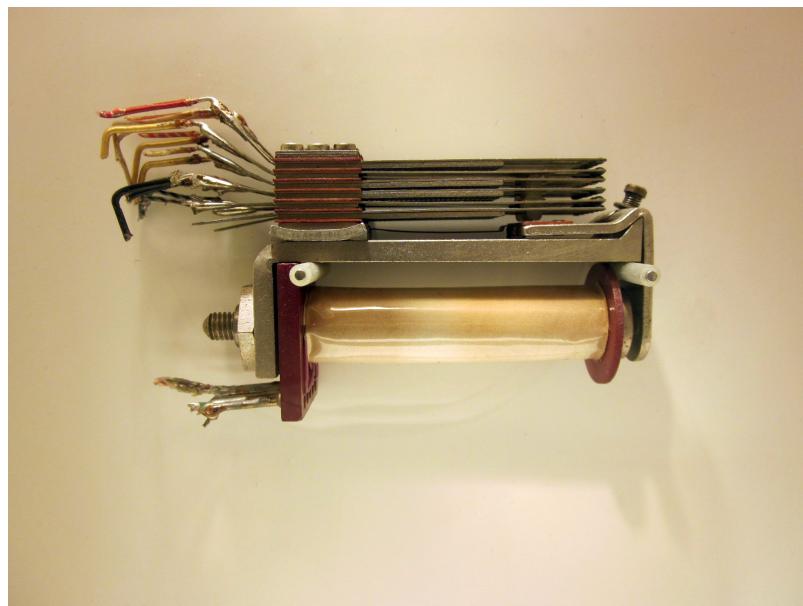


FIGURE 1.18 – Le type de relais électromécanique employé par Zuse (source : Wikipédia).

entièrement disparu, sauf dans certaines applications particulières, comme par exemple les fours à micro-ondes.

Un transistor est un composant permettant de contrôler ou d'amplifier des tensions et des courants électriques. Il a été inventé par Bardeen, Shockley et Brattain en 1947. Ce composant a trois bornes : la base, l'émetteur et le collecteur. Le courant qui passe dans la base permet de contrôler le courant entre le collecteur et l'émetteur.

À partir de 1957, IBM a construit tous ses nouveaux ordinateurs avec des transistors au lieu de tubes à vide. Les transistors interviennent par exemple dans la conception des mémoires SRAM et DRAM introduites dans les années 1960.

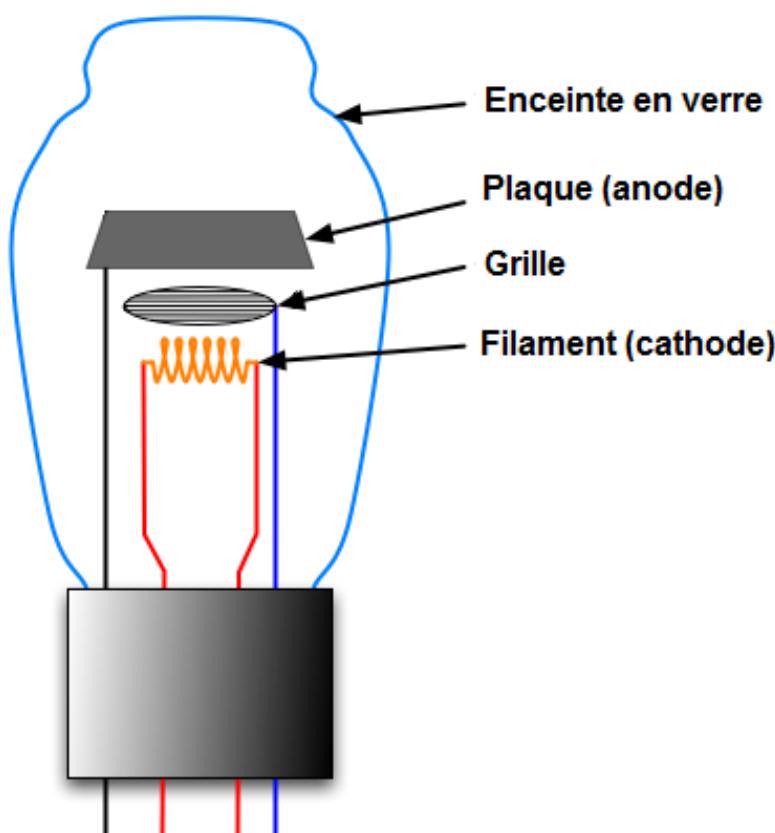


FIGURE 1.19 – Le principe d'une triode, un type très courant de tube.
(source : Wikipédia)



FIGURE 1.20 – Un exemple de tube électronique (ou lampe à vide) (source : Wikipédia).

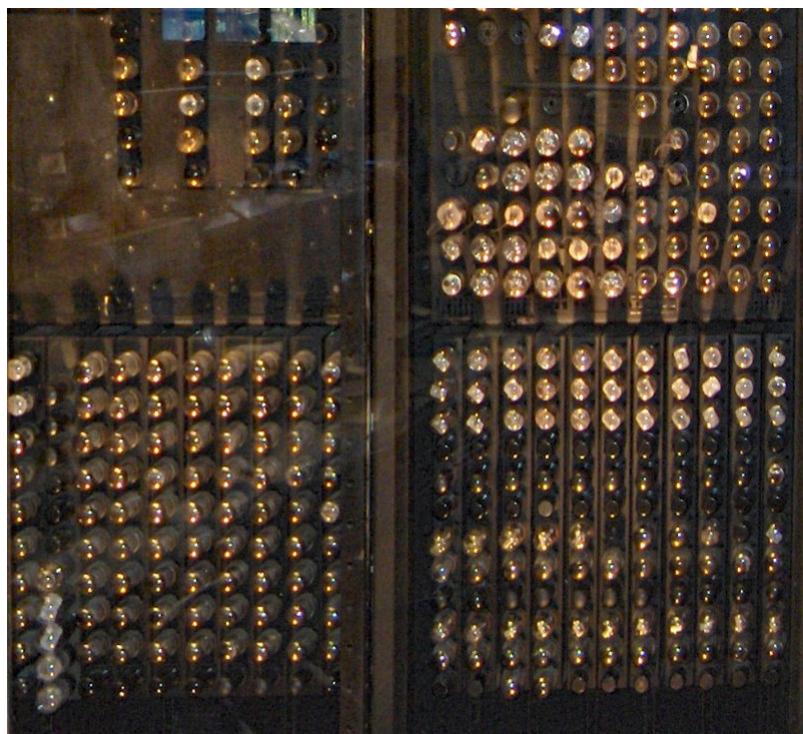


FIGURE 1.21 – Des lampes à vide dans l'ENIAC (source : Wikipédia).

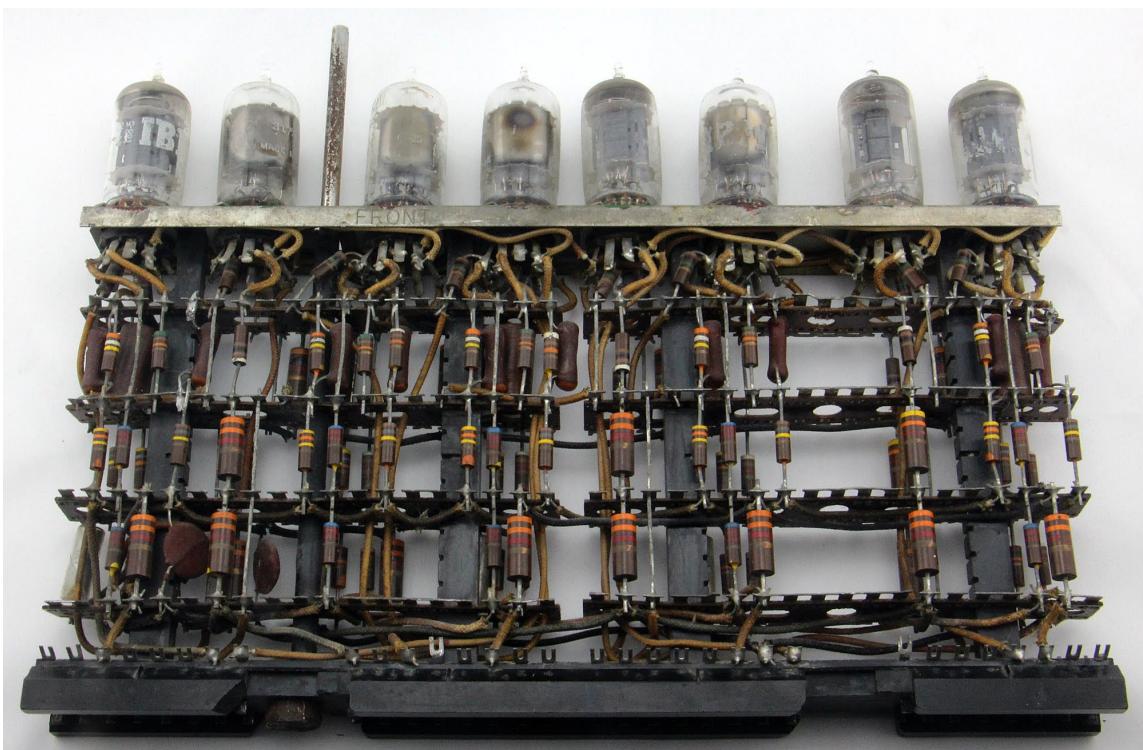
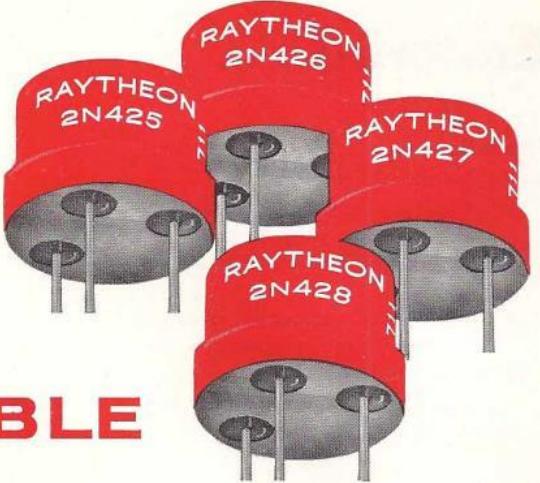


FIGURE 1.22 – Un module avec des lampes à vide pour une ancienne machine IBM 705 de 1954. (source : <http://www.righto.com/2018/01/examining-1954-ibm-mainframes-pluggable.html>)

Designed for Computers
Made for Computers
Tested for Computers
Dependable in Computers



RAYTHEON RELIABLE COMPUTER TRANSISTORS

Reliability must be designed and built into computer transistors. It cannot be obtained by selection.

Raytheon Computer Transistors were developed under Signal Corps contract and are manufactured especially for computer service, on a separate production line. They are backed by five years of experience in the mass production and quality control of Raytheon Fusion-Alloy Transistors.

Maximum stability is guaranteed by rigid test procedures including *strict process control*, 100°C *baking* of every transistor, 100% *steam cycling* to assure positive hermetic sealing.

When you specify Raytheon Computer Transistors you are also assured of:

HIGH VOLTAGE RATINGS • HIGH CURRENT GAIN
FAST SWITCHING SPEED
LOW SATURATION RESISTANCE

FIGURE 1.23 – Une publicité pour des transistors Raytheon. (source : <http://semiconductormuseum.com>)

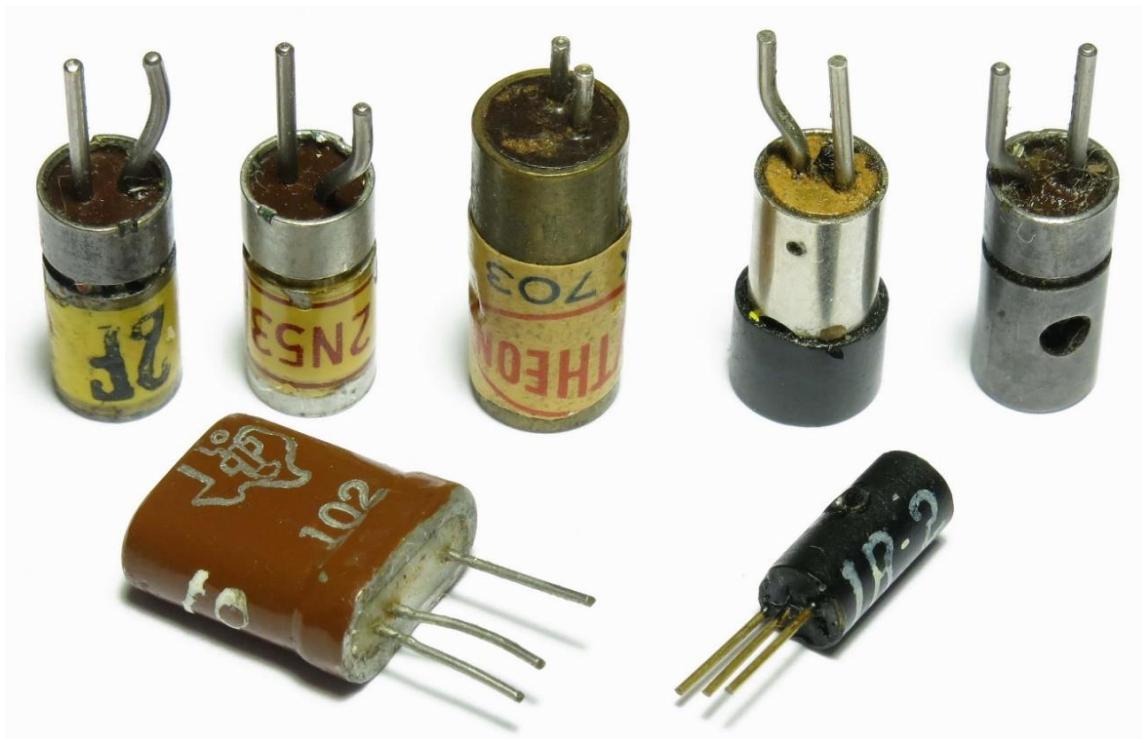


FIGURE 1.24 – Quelques exemples d'anciens transistors, dont le CK703 de Raytheon, le premier transistor mis dans le commerce en 1948. (source : *Mr. Jonathan Hoppe transistor museum donation, March 2014*)

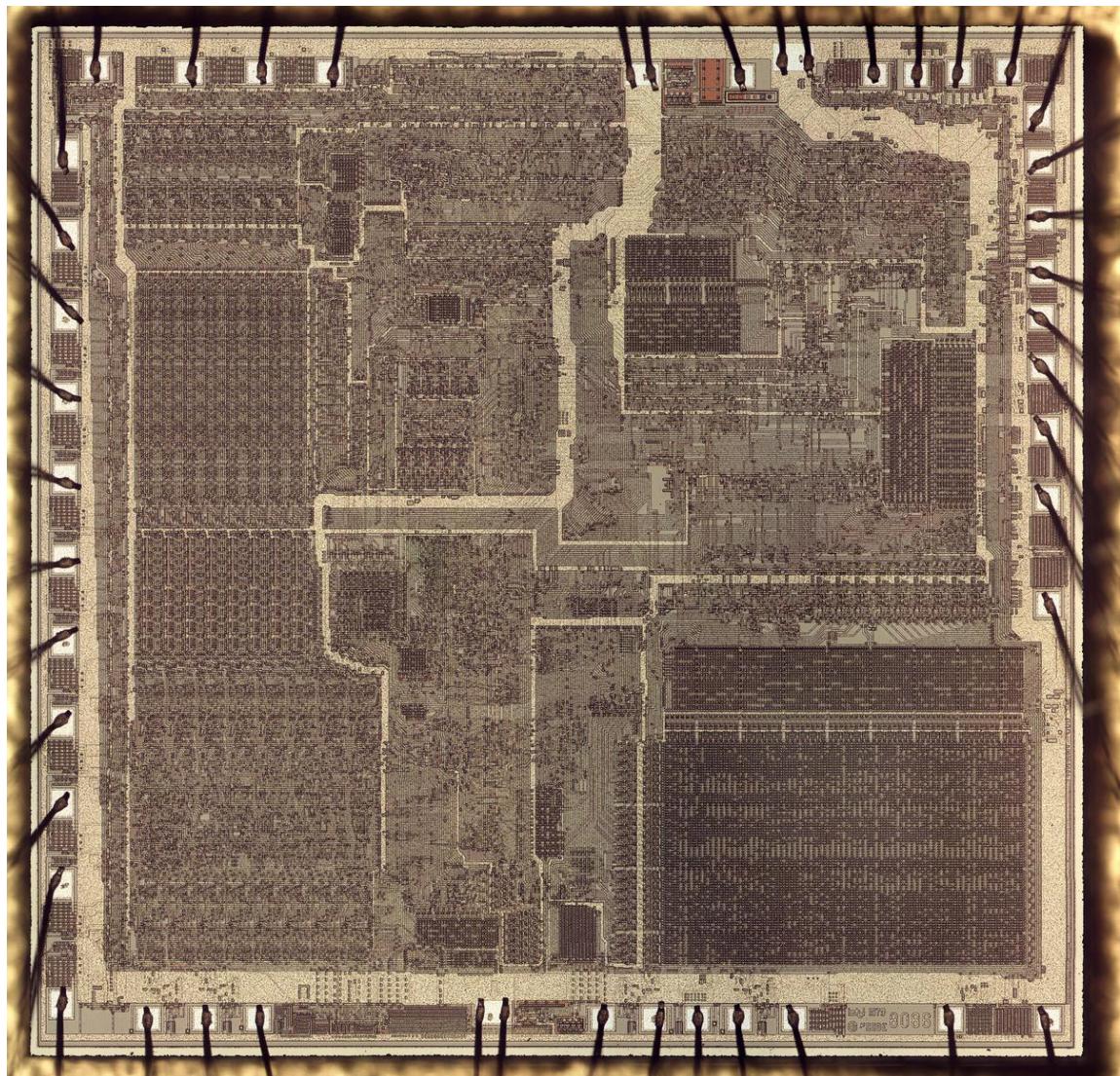


FIGURE 1.25 – Une vue du *die* (puce) du processeur 8086 (1978) utilisant la technologie NMOS pour implémenter 29000 transistors.

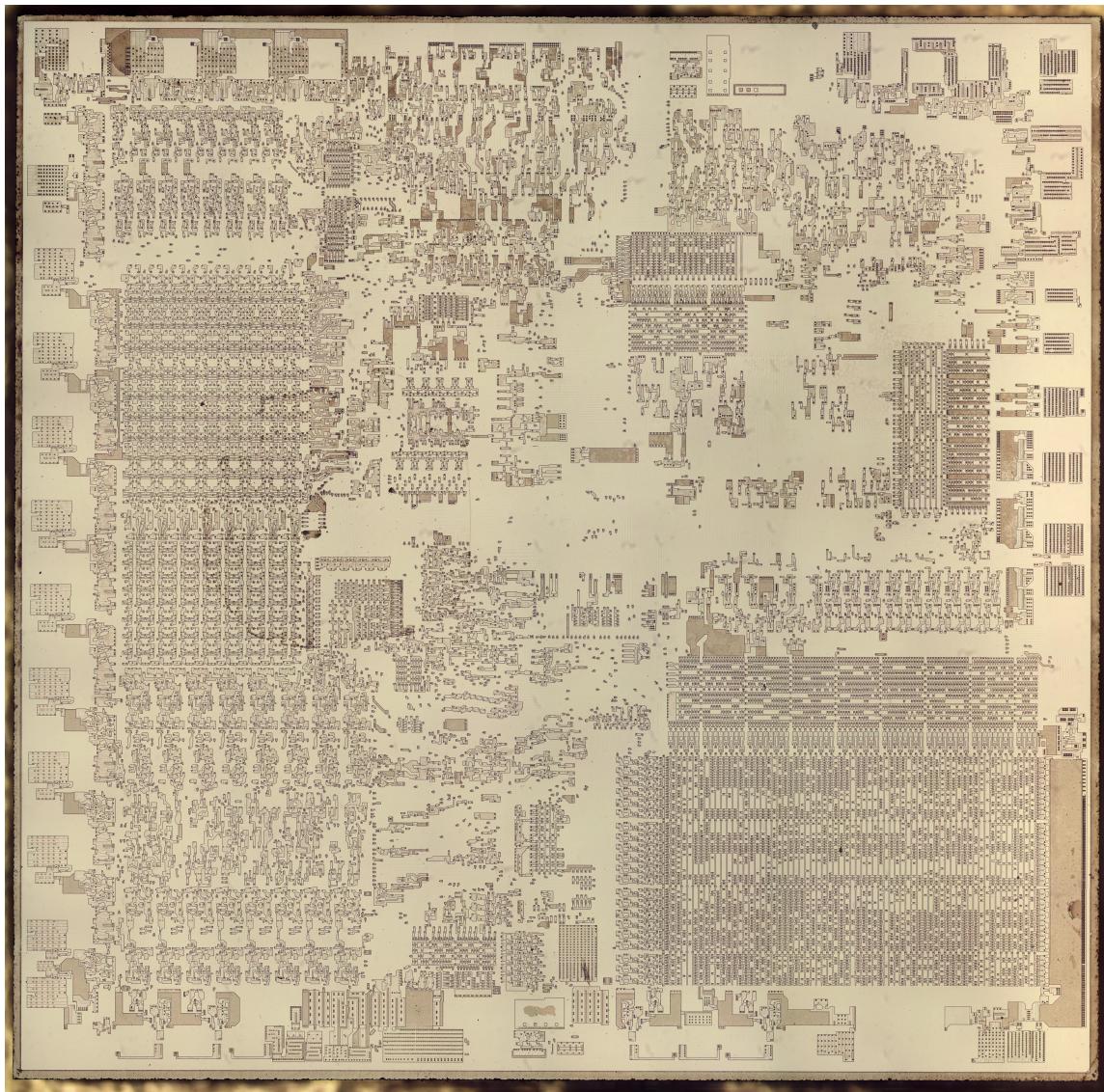


FIGURE 1.26 – Une vue du *die* (puce) du processeur 8086 (1978) où les transistors sont identifiables au microscope.

1.7 Suppléments

Voici quelques vidéos ou documents intéressants, parmi bien d'autres :

- ordinateur Mark I :
<https://www.youtube.com/watch?v=bN7AdQmd8So>
https://www.youtube.com/watch?v=718W96I7_ew
- Ordinateur d'Apollo :
<https://www.youtube.com/watch?v=ndvmF1g1WmE>
- The National Museum of Computing Tour :
https://www.youtube.com/watch?v=_Sw15F2QzMQ
- Simulateur de l'ENIAC :
<https://www.cs.drexel.edu/~bls96/eniac>
- Programmes pour l'EDSAC :
<https://www.dcs.warwick.ac.uk/~edsac/Programs2/EitiPie.zip>
(ouvrir le PDF de l'archive)

1.8 Exercices

1. faites des recherches sur la méthode de calcul de tables mathématiques à l'aide de différences; donnez un exemple de calcul de la table d'une fonction comme $f(x) = 7x^3 - 3x^2 + 8x - 2$; comment cette méthode s'utilise-t-elle pour calculer des valeurs disons de $\sin(x)$ entre $\sin(a)$ et $\sin(b)$? comment une machine mécanique peut-elle être utile?
2. qu'est-ce qu'un analyseur différentiel?
3. cherchez ce qu'est le ternaire équilibré (*balanced ternary*);
4. trouvez le rapport de von Neumann qui a donné le nom à son architecture; quel est le titre de ce rapport? où exactement von Neumann dit-il que les données et les instructions sont stockées au même endroit?

Chapitre 2

Représentation des données

On traite ici de la représentation des caractères et des informations multimédia.

2.1 Représentation des caractères

On suppose que l'on code l'information avec un nombre fixe de bits, chacun valant 0 ou 1.

Si on a un seul bit, on ne peut coder que deux valeurs, 0 et 1. Un tel code peut servir par exemple pour représenter l'alphabet Morse (figure 2.1). Les points pourraient être représentés par des 0 et les traits par 1. Cela dit, dans le code Morse, il y a un caractère supplémentaire, invisible, à savoir un temps mort entre deux séries de symboles. Il faudrait donc plutôt pouvoir coder trois caractères différents. On pourrait le faire en employant un code à 2 bits, qui admet quatre valeurs possibles : 00, 01, 10 et 11.

Si on cherche à coder l'alphabet, disons les lettres majuscules de A à Z, il faut pouvoir coder 26 caractères sur un ensemble fixe de bits. Combien faut-il de bits ? Avec trois bits, on peut coder huit valeurs. Avec quatre bits, on peut en coder 16. Avec cinq bits on peut en coder 32. Il faut donc au moins cinq bits. Bien évidemment,

International Morse Code

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to seven dots.

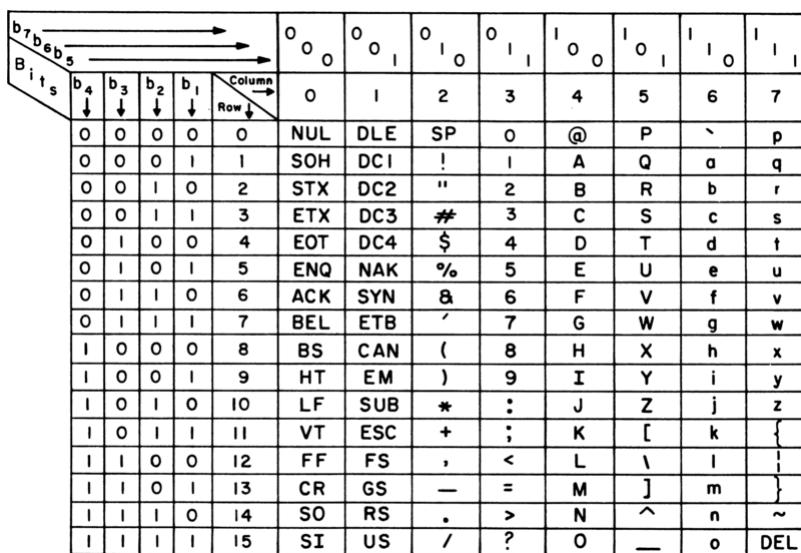
A	• -	U	• • -
B	- - . . .	V	• • - -
C	- - . - .	W	• - - -
D	- - . .	X	- - . - -
E	•	Y	- - . - - -
F	• . - - .	Z	- - - . . .
G	- - - .		
H	•		
I	• •		
J	• - - - -		
K	- . -	1	• - - - - -
L	• - . . .	2	• • - - - -
M	- -	3	• • • - - -
N	- - .	4	• • • • - -
O	- - -	5	• • • • • -
P	• - - - .	6	• - - - • • -
Q	- - - . - -	7	• - - - • • • -
R	• - - . .	8	• - - - - - -
S	• . . .	9	• - - - - - - .
T	- -	0	• - - - - - -

FIGURE 2.1 – Le code Morse. (source : Wikipédia)

avec le code Morse et trois bits, on peut aussi coder l'alphabet, mais chaque lettre de l'alphabet occuperait plusieurs séries de trois bits. Ici, on cherche à coder un caractère complètement.

2.1.1 Le code ASCII

Si on veut coder des ensembles plus grands, il faut davantage de bits. L'un des premiers codes largement répandus pour coder les textes, c'est-à-dire des suites de caractères, a été le code ASCII (*American Standard Code for Information Interchange*). C'est un code sur 7 bits (figure 2.2) qui permet de coder 128 valeurs différentes.



The diagram illustrates the 7-bit ASCII code mapping. On the left, a 3D-style grid shows the 7 bits (b₇ to b₁) mapped to binary values (0 or 1). The first four columns represent the row (b₇, b₆, b₅) and the next three columns represent the column (b₄, b₃, b₂, b₁). The grid contains 128 entries, each consisting of a binary value followed by its corresponding ASCII character. The characters include control codes like NUL, DLE, SP, and various letters and symbols from the standard ASCII set.

b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	Column	Row	0	0	0	1	0	1	0	0	1	0	1	1	0	1	1	1
0	0	0	0	0	0	0	0	0	NUL	DLE	SP	0	@	P	~	p	0	1	2	3	4	5	6	7
0	0	0	0	1	1	1	0	1	SOH	DC1	!	1	A	Q	a	q	0	0	1	0	2	STX	DC2	"
0	0	1	0	0	2	2	1	0	STX	DC2	"	2	B	R	b	r	0	0	1	1	3	ETX	DC3	#
0	0	1	1	0	3	3	0	1	EOT	DC4	\$	3	C	S	c	s	0	1	0	0	4	ENQ	NAK	%
0	1	0	0	1	5	5	1	0	ENQ	NAK	%	5	E	U	e	u	0	1	0	1	6	ACK	SYN	&
0	1	1	0	0	6	6	0	1	BEL	ETB	'	6	F	V	f	v	0	1	1	1	7	BS	CAN	(
1	0	0	0	0	8	8	0	0	BS	CAN	(8	H	X	h	x	1	0	0	1	9	HT	EM)
1	0	0	1	0	9	9	1	0	HT	EM)	9	I	Y	i	y	1	0	1	0	10	LF	SUB	*
1	0	1	0	1	10	10	1	0	LF	SUB	*	10	J	Z	j	z	1	0	1	1	11	VT	ESC	+
1	1	0	0	1	11	11	0	1	VT	ESC	+	11	K	[k	{	1	1	0	0	12	FF	FS	,
1	1	0	1	0	12	12	1	0	FF	FS	,	12	L	\	l	l	1	1	0	1	13	CR	GS	-
1	1	1	0	0	13	13	0	1	CR	GS	-	13	M]	m	}	1	1	1	0	14	SO	RS	.
1	1	1	1	1	14	14	1	1	SO	RS	.	14	N	^	n	~	1	1	1	1	15	SI	US	/
1	1	1	1	1	15	15	1	1	SI	US	?	15	O	—	o	DEL								

FIGURE 2.2 – Le code ASCII. (source : Wikipédia)

2.1.2 Autres codes 8 bits

Si on utilise un octet, soit 8 bits, on peut toujours utiliser le code ASCII, mais se servir du 8^e bit comme bit de parité, c'est-à-dire d'un bit supplémentaire déterminé à partir des sept autres bits pour éviter les erreurs de transmission.

Mais certains constructeurs ont étendu le codage en utilisant ce 8^e bit et donc en construisant des codes 8 bits pouvant coder jusqu'à 256 caractères.

Un exemple de tel code est le code Windows-1252 :

<https://en.wikipedia.org/wiki/Windows-1252>

Un autre exemple est le code EBCDIC :

<https://en.wikipedia.org/wiki/EBCDIC>

2.1.3 Pages nationales

Des pages de code propres à différents pays ont été développées dans les années 1960 à 1990 pour répondre aux besoins d'un certain nombre de langues. Ces pages ont créé une véritable cacophonie et ont compliqué l'échange de données, notamment parce que les documents qui les utilisaient n'indiquaient pas quel code était utilisé.

On pourra trouver une liste de la plupart des pages de code ici :
<https://www.aivosto.com/articles/charsets-codepages.html>

La norme ISO 8859-1 définit une page de code pour l'alphabet latin et elle a été (et est encore) utilisée en France et dans un certain nombre de pays européens.

2.1.4 Unicode

Unicode est un standard de codage de caractères ainsi que de propriétés associées à ces caractères. Le but d'Unicode est de donner à tous les caractères de toutes les langues un nom ainsi qu'un identifiant numérique, afin de faciliter les échanges. Il s'agissait aussi de remplacer les pages de code nationales. La première version d'Unicode date de 1991.

La version 15.0 d'Unicode (13 septembre 2022) comporte 149186 caractères, et ajoute notamment plus de 4000 caractères CJC (chinois-japonais-coréen) à la version précédente.

Les nombres associés aux caractères sont divisés en 17 zones de 65536 points de code. Ces zones sont appelées des plans. Chaque

Windows-1255 ^{[7][8][9][10][11][12][13]}																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	}		~	DEL
8x	€	,	f	"	...	†	‡	^	%	„	„					
9x		,	„	„	•	—	—	~	TM		›					
Ax	NBSP	ı	¢	£	₪	¥	₩	§	„	©	×	«	¬	SHY	®	—
Bx	°	±	²	³	‘	μ	¶	·	„	¹	÷	»	¼	½	¾	€
Cx	„	„	„	„	„	„	„	„	„	„	„	„	„	„	„	„
Dx	।	ং	ঃ	ঁ	ঁ	ঁ	ঁ	ঁ	”							
Ex	א	ב	ג	ד	ה	ו	ת	ח	ט	י	ר	כ	ל	ם	מ	।
Fx	ג	օ	ע	ך	פ	ץ	ך	ך	ך	ך	ך	ך		LRM	RLM	

FIGURE 2.3 – La page de code Windows 1255. (source : <https://en.wikipedia.org/wiki/Windows-1255>)

ISO/CEI 8859-1																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	<i>positions inutilisées</i>															
1x	<i>positions inutilisées</i>															
2x	[SP]	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8x	<i>positions inutilisées</i>															
9x	<i>positions inutilisées</i>															
Ax	[NBSP]	i	¢	£	¤	¥	¦	§	¨	©	™	«	»	□	®	™
Bx	°	±	²	³	‘	µ	¶	·	,	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ā	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Ò	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIGURE 2.4 – Le codage ISO 8859-1. (source : Wikipédia)

point de code est noté « U+ » suivi de quatre à six chiffres en hexadécimal. Par exemple le caractère nommé « Lettre majuscule latine c cédille » (Ç) a le numéro U+00C7. Il appartient au premier plan.

On pourra voir les pages d'Unicode ici : <http://www.unicode.org/charts>

Par exemple, <http://www.unicode.org/charts/PDF/U0000.pdf> est la page correspondant au code ASCII.

<http://www.unicode.org/charts/PDF/U0080.pdf> est la page pour le supplément Latin-1, où se trouvent les caractères accentués français.

<http://www.unicode.org/charts/PDF/U0400.pdf> est la page correspondant au cyrillique de base.

<http://www.unicode.org/charts/PDF/U0840.pdf> est la page correspondant au mandéen.

Et ainsi de suite.

2.1.5 UTF-8

Unicode assigne un nom et un numéro à un caractère, mais ne dit pas comment les caractères sont codés informatiquement. UTF-8 est un codage d'Unicode, c'est-à-dire un moyen de coder un certain point d'Unicode dans une représentation de données.

Avec le codage UTF-8, chaque point d'Unicode est représenté par une suite d'octets, mais pas toujours le même nombre. Les caractères du code ASCII, par exemple, sont représentés par un seul octet. Les lettres accentuées du français sont représentées par deux octets. Par exemple, « é » est le point U+00E9 d'Unicode, mais est codé par les deux octets C3 et A9 en UTF-8. Et certains caractères occupent trois ou quatre octets.

Caractères codés	Représentation binaire UTF-8	Premier octet valide (hexadécimal)	Signification
U+0000 à U+007F	0bbb · bbbb	00 à 7F	1 octet, codant jusqu'à 7 bits
U+0080 à U+07FF	110b · bbbb 10bb · bbbb	C2 à DF	2 octets, codant jusqu'à 11 bits
U+0800 à U+FFFF	1110 · bbbb 10bb · bbbb 10bb · bbbb	E0 à EF	3 octets, codant jusqu'à 16 bits
U+10000 à U+10FFFF	1111 · 00bb 10bb · bbbb 10bb · bbbb 10bb · bbbb	F0 à F3	4 octets, codant jusqu'à 21 bits
	1111 · 0100 1000 · bbbb 10bb · bbbb 10bb · bbbb	F4	

FIGURE 2.5 – Le codage UTF-8. (source : Wikipédia)

2.2 Représentation des données multimédia

2.2.1 Son

Un son peut être représenté par un ensemble de valeurs correspondant à des échantillonnages à intervalles réguliers.

2.2.2 Image

Les deux principaux types de stockage d'images sont le format bitmap et le format vectoriel. Dans le format bitmap, une image est représentée par l'ensemble des points d'une grille ou matrice. Des exemples de tels formats sont JPG et PNG. Les informations de ces points peuvent être compressées.

Dans le format vectoriel, la représentation est plus abstraite. Un segment n'est par exemple pas représenté sous la forme d'un ensemble de points, mais d'une description de plus haut niveau donnant les extrémités du segment, son épaisseur, sa couleur, etc. En général, cette description est alors transformée en bitmap pour l'affichage, c'est ce que l'on appelle la rasterisation.

2.2.3 Vidéos

Une vidéo est essentiellement une suite d'images fixes.

2.3 Suppléments

On donne ici quelques pointeurs complémentaires sur des notions vues plus haut :

- <https://home.unicode.org>
- <http://www.unicode.org/charts>
- <https://fr.wikipedia.org/wiki/Unicode>

2.4 Un peu d'histoire

Une vidéo sur les cartes perforées :

<https://www.ina.fr/ina-eclaire-actu/video/caf97059686/la-carte-perforee>



FIGURE 2.6 – Une carte perforée à 80 colonnes (brevetée par IBM en 1928). (source : Wikipédia)

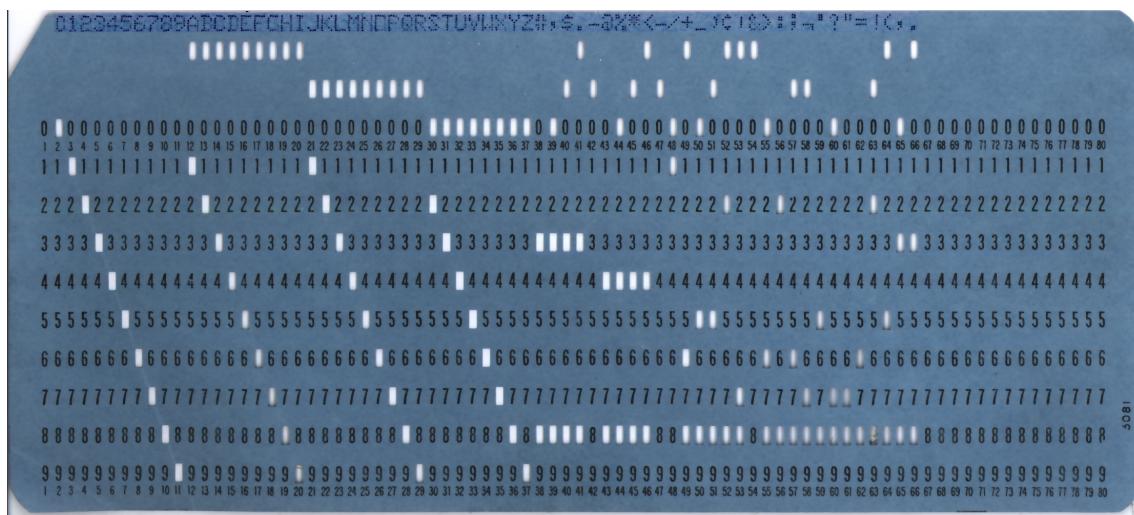


FIGURE 2.7 – Une carte perforée avec le jeu de caractères EBCDIC.
(source : Wikipédia)

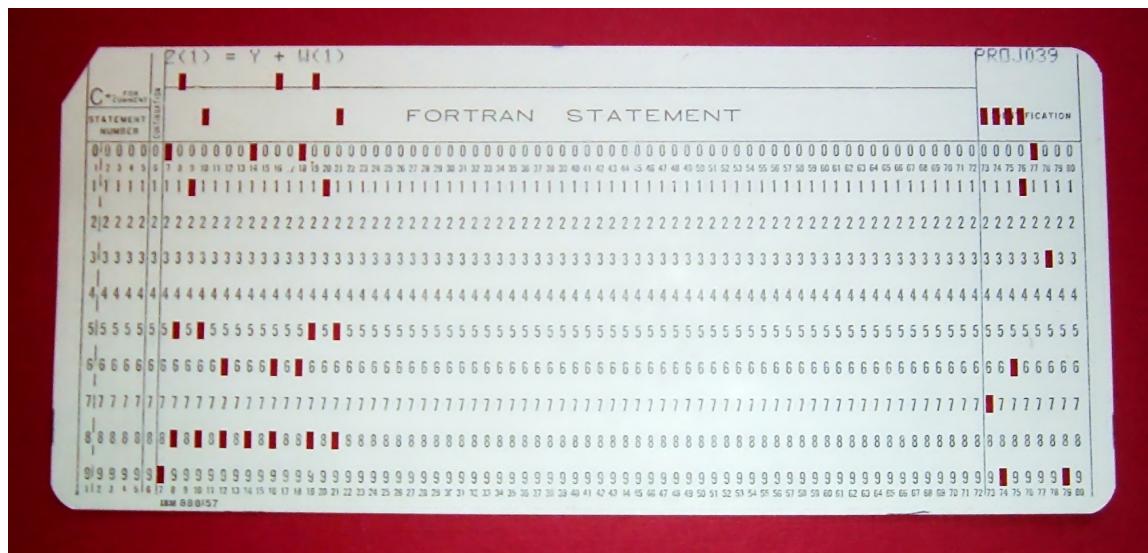


FIGURE 2.8 – Une carte perforée codant une instruction FORTRAN.
(source : Wikipédia)

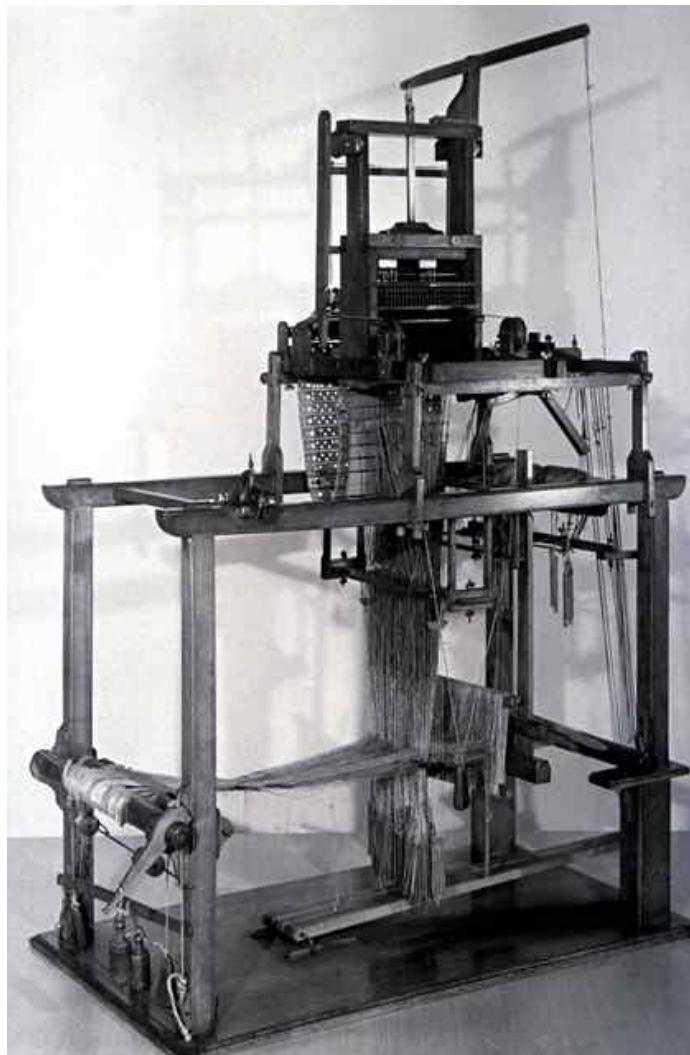


FIGURE 2.9 – Un métier à tisser de Jacquard. (source : Wikipédia)



FIGURE 2.10 – Un extrait du manuscrit Voynich. (source : Wikipédia)

2.5 Exercices

Certains des exercices ci-dessous font appel au langage C. On se reportera au chapitre 10 pour une petite introduction à ce langage. L'intérêt du C ici est de permettre d'accéder précisément à la mémoire.

1. Créer un fichier avec emacs et y mettre l'unique caractère « a ». Quelle est la taille du fichier résultant ?
2. Reprendre le même fichier et y rajouter le caractère « é ». Comment évolue la taille ?
3. En utilisant la commande xxd trouvez le code hexadécimal du retour chariot.
4. Créer un fichier et y mettre une dizaine de lignes de texte en français, avec des caractères accentués. Déterminez le codage utilisé par ce fichier. Ensuite, changez le codage avec le programme iconv.
5. Déterminer en C le code ASCII du caractère « a ».
6. Soit le programme Java suivant :

```
public class Char {  
    public static void main(String args[]) {  
        char[] a = new char[] {83,97,108,117,116,12};  
        for (int i=0;i<5;i++)  
            for (int j=0;j<6;j++)  
                System.out.print(a[j]);  
    }  
}
```

Sans exécuter ce programme, devinez ce qu'il affiche. Vérifiez-le ensuite en le compilant.

7. Écrire la fonction `maJuscule()` qui prend un caractère entre « a » et « z » en paramètre et renvoie le caractère correspondant en majuscule.

8. Écrire une fonction `code()` qui prend un paramètre un caractère entre « A » et « Z » et opère un décalage de trois caractères : « A » devient « D », « B » devient « E », etc., et « Z » devient « C ».
9. Écrire une fonction `typeCarac()` qui prend un caractère de la table ASCII en paramètre et affiche un message pour dire si c'est une lettre, un chiffre ou un symbole.
10. Quelle est la taille (en octets) d'un texte avec n caractères ASCII codé en format
 - (a) UTF-8
 - (b) UTF-16
 - (c) UTF-32
11. Écrire une fonction qui affiche un mot (passé en paramètre) en remplaçant chaque lettre par une étoile.
12. Écrire une fonction qui affiche un texte à l'envers.
13. Écrire une fonction qui détermine si un texte est un palindrome.
14. Écrire une fonction qui n'écrit que la moitié de la chaîne de caractères passée en paramètre et complète par des étoiles.
15. Écrire une fonction `compte` qui compte le nombre de fois où une chaîne de caractères est comprise dans une autre.
16. Décodez la suite d'octets en hexadécimal suivante qui représente un texte en ASCII : 4365 6369 2065 7374 2075 6e0a 6669 6368 6965 720a 656e 2041 5343 4949 2e
17. Décoder le texte suivant :
01000010 01110010 01100001 01110110 01101111 00101100
00100000 01110100 01110101 00100000 01100001 01110011
00100000 01110000 01110010 01100101 01110011 01110001
01110101 01100101 00100000 01110100 01101111 01110101
01110100 00100000 01110100 01110010 01101111 01110101
01110110 11101001 00101110 00101110 00101110
Quel codage utilise-t-il ?

18. Faire les exercices de la page

<https://iut-info.univ-reims.fr/users/nourrit/codages/page10.html>

19. Le codage « base 64 » permet de coder une suite d'octets sous la forme d'un ensemble limité visibles de caractères. En pratique, seuls 64 caractères sont utilisés, d'où le nom. Ces 64 caractères sont les lettres A à Z, les lettres minuscules, les chiffres, le signe + et le signe /. « A » correspond aux six bits 000000, « B » à 000001, etc., jusqu'à « / » qui correspond à 111111. Pour coder une suite d'octets, on aligne tous les bits et on prend à chaque fois 6 bits. À un groupe de 6 bits, on associe l'une des 64 valeurs ci-dessus. Enfin, on remplace les caractères correspondants par leur code ASCII. Si par exemple on a les 6 bits 000001, on va les remplacer par l'octet de valeur décimale 65 qui est le code ASCII de B.

On suppose maintenant que l'on a reçu la suite d'octets suivante (en hexadécimal) et codée en base64 :

596d 467a 5a54 5930 4367 3d3d

Quel était le texte original ?

20. Le pass sanitaire, comme la plupart des QR codes, utilise le codage « base 45 ». Pour coder une suite d'octets A, B, C, \dots , en base 45, on regroupe d'abord les octets deux par deux. À chaque groupe de deux octets, on fait correspondre trois valeurs X, Y, Z , comprises entre 0 et 44 :

$$A, B \rightarrow X, Y, Z$$

Ces valeurs sont telles que

$$A \times 256 + B = X + Y \times 45 + Z \times 45^2$$

On peut facilement calculer X, Y et Z , puisque $X = (A \times 256 + B) \bmod 45$, puis diviser par 45 et réitérer le processus. Par exemple, si les deux octets A et B contiennent les codes ASCII

de « hi », les valeurs sont 104 et 105 et on trouve $X = 44$, $Y = 8$ et $Z = 13$, de telle sorte que

$$104 \times 256 + 105 = 44 + 8 \times 45 + 13 \times 45^2 = 26729$$

À l'aide de la table ci-dessous, on détermine que les valeurs de X , Y et Z correspondent aux caractères « :8D » (un smiley, étonnant, non?).

val	codage	val	codage	val	codage	val	codage
00	0	12	C	24	0	36	Espace
01	1	13	D	25	P	37	\$
02	2	14	E	26	Q	38	%
03	3	15	F	27	R	39	*
04	4	16	G	28	S	40	+
05	5	17	H	29	T	41	-
06	6	18	I	30	U	42	.
07	7	19	J	31	V	43	/
08	8	20	K	32	W	44	:
09	9	21	L	33	X		
10	A	22	M	34	Y		
11	B	23	N	35	Z		

On demande maintenant de faire l'opération inverse et de décoder la chaîne « M+9 ». *On détaillera les opérations et les valeurs décimales de X , Y , Z , A et B .*

21. vérifier que la carte de la figure 2.7 (page 52) utilise bien le codage EBCDIC.
22. quels codages sont utilisés dans les cartes des figures 2.6 et 2.8 ?
23. (difficile) que code le texte de la figure 2.10 ?

Chapitre 3

Systèmes de numération

Ce chapitre donne quelques rappels sur les systèmes de numération, normalement déjà vus au lycée.

3.1 Le système décimal

Le système décimal correspond à la base 10. Un nombre est représenté comme suite des symboles 0 à 9, ainsi qu'un(e) point (ou virgule) décimal(e). Notre notation est une notation positionnelle où chaque chiffre a un poids qui dépend de sa position. On peut écrire

$$442.256 = (4 \times 10^2) + (4 \times 10^1) + (2 \times 10^0) + (2 \times 10^{-1}) + (5 \times 10^{-2}) + (6 \times 10^{-3})$$

mais surtout cela permet de compacter l'écriture :

$$10_{10} = 111111111_1$$

Dans la base 1, chaque chiffre a le même poids. Dans la base 10, le « 1 » de 10 compte pour 10 fois plus que le « 0 ».

Dans l'écriture décimale, le chiffre le plus à gauche est le chiffre le plus significatif et le chiffre le plus à droite est le chiffre le moins significatif.

3.2 Les systèmes de numération positionnels

Plus généralement, dans un système de numération positionnel, un nombre est représenté par une suite de chiffres où chaque chiffre de position i a un poids associé b^i où b est la base :

On peut écrire un tel nombre $(\dots a_3 a_2 a_1 a_0. a_{-1} a_{-2} a_{-3} \dots)_b$, chaque chiffre a_i étant tel que $0 \leq a_i < b$.

La valeur associée est

$$\dots + a_3 b^3 + a_2 b^2 + a_1 b^1 + a_0 b^0 + a_{-1} b^{-1} + a_{-2} b^{-2} + a_{-3} b^{-3} + \dots = \sum_i (a_i \times b^i)$$

Le système décimal est un cas particulier où $b = 10$.

3.3 Le système binaire

Dans le système décimal, 10 chiffres différents sont utilisés pour représenter les nombres en base 10. Dans le système binaire, nous n'avons que deux chiffres, 0 et 1. Les nombres dans le système binaire sont représentés en base 2. Pour éviter les confusions, on écrira quelquefois la base en suffixe. Ainsi

$$10_2 = (1 \times 2^1) + (0 \times 2^0) = 2_{10}$$

$$11_2 = (1 \times 2^1) + (1 \times 2^0) = 3_{10}$$

$$100_2 = (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) = 4_{10}$$

Les valeurs fractionnaires sont représentées avec des puissances négatives de la base :

$$1001.101 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625_{10}$$

En général, pour la représentation binaire de

$$Y = \{\dots b_2 b_1 b_0. b_{-1} b_{-2} b_{-3} \dots\},$$

la valeur de Y est

$$Y = \sum_i (b_i \times 2^i)$$

10 ⁸	Tabulag ita stabilit	2^0
1	1	2^0
10	2	2^1
100	4	2^2
1000	8	2^3
10000	16	2^4
100000	32	2^5
1000000	64	2^6
10000000	128	2^7
100000000	256	2^8
1000000000	512	2^9
10000000000	1024	2^{10}

cm 1 2 3 4 5 6 7 8

FIGURE 3.1 – Un manuscrit de Leibniz vers 1703 sur le système binaire. (source : Wikipédia)

3.4 Conversion entre binaire et décimal

Il est facile de convertir un nombre binaire en décimal, il suffit de multiplier chaque chiffre binaire par la puissance correspondante de 2 et d'additionner les résultats.

Dans le cas où le nombre est fractionnaire, on peut éventuellement multiplier d'abord le nombre binaire par une puissance de 2 pour déplacer la virgule, puis faire la conversion et rediviser par la même puissance de 2.

La même technique pourrait être utilisée pour passer du décimal au binaire, mais en traduisant immédiatement 10^1 , 10^2 , etc., en binaire et en faisant les additions en binaire, ce qui est un peu fastidieux, puisque nous ne connaissons pas bien les puissances de 10 en binaire :

$$\begin{aligned} 10_{10}^1 &= 1010_2 \\ 10_{10}^2 &= 1100100_2 \\ &\dots \end{aligned}$$

et nous ne savons pas rapidement faire les multiplications comme $7_{10} \times 1100100_2$ en binaire, mais ce serait néanmoins possible.

Au lieu de cela, il est plus simple de procéder par divisions successives (figures 3.2 et 3.3).

Pour les valeurs inférieures à 1, on peut faire des multiplications successives par 2 (figures 3.4 et 3.5).

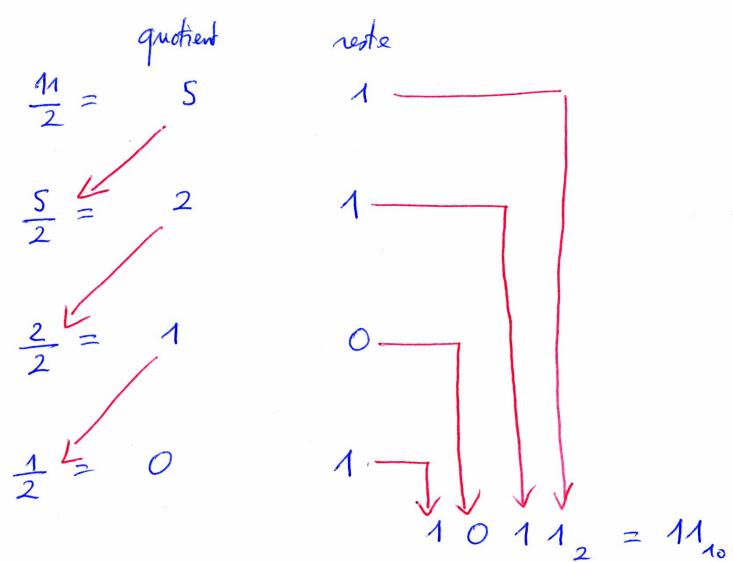


FIGURE 3.2 – Exemple de conversion de décimal vers binaire pour un entier.

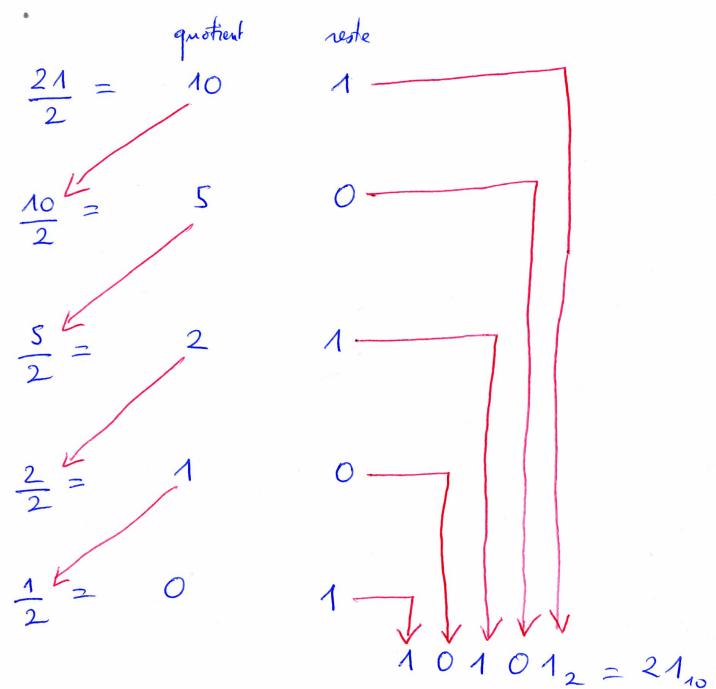


FIGURE 3.3 – Exemple de conversion de décimal vers binaire pour un entier.

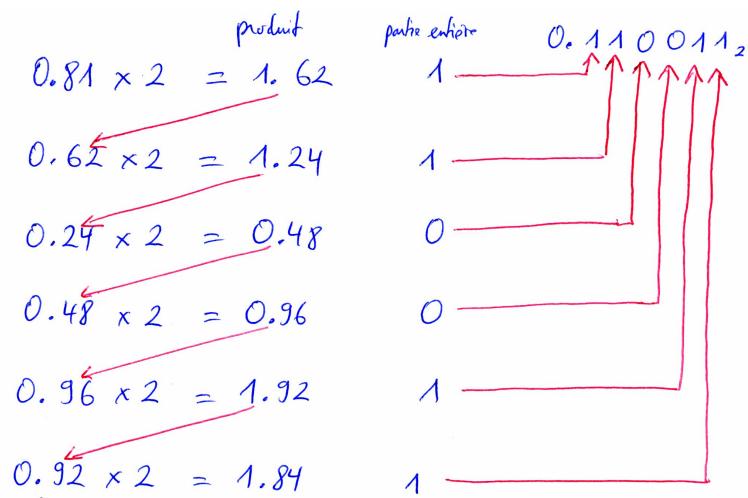


FIGURE 3.4 – Exemple de conversion de décimal vers binaire pour une fraction.

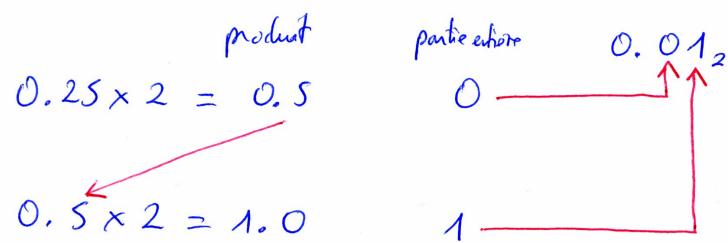


FIGURE 3.5 – Exemple de conversion de décimal vers binaire pour une fraction.

3.5 Notation hexadécimale

La notation hexadécimale est simplement la base 16, avec la correspondance suivante entre les chiffres :

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

La notation hexadécimale n'est pas seulement utilisée pour représenter des entiers, mais aussi comme notation concise pour représenter n'importe quelle suite de chiffres binaires, qu'elle représente du texte, des nombres ou d'autres formes de données. Les raisons pour utiliser la notation hexadécimale sont les suivantes :

- elle est plus compacte que la notation binaire ;
- dans la plupart des ordinateurs, les données binaires occupent un multiple de 4 bits, donc un multiple d'un unique chiffre hexadécimal ;
- il est extrêmement simple de passer de la notation binaire à la notation hexadécimale, ou inversement.

3.6 Un peu d'histoire

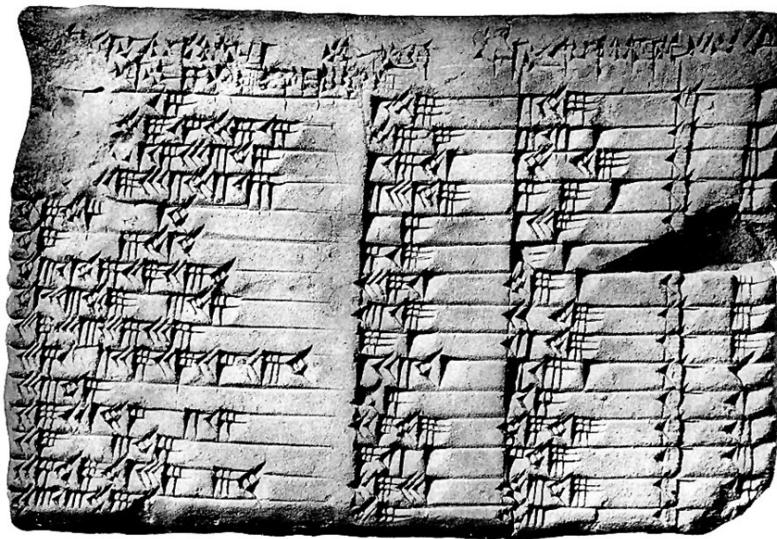


FIGURE 3.6 – Une tablette d'argile babylonienne. Il s'agit de la célèbre tablette Plimpton 322.

Nombres romains									
Milliers, de 1000 à 4000	M	MM	MMM	MMMM	Voir extensions				
Centaines, de 100 à 900	C	CC	CCC	CD	D	DC	DCC	DCCC	CM
Dizaines, de 10 à 90	X	XX	XXX	XL	L	LX	LXX	LXXX	XC
Unités, de 1 à 9	I	II	III	IV	V	VI	VII	VIII	IX

FIGURE 3.7 – Les chiffres romains.

1	α	alpha	10	ι	iota	100	ρ	rho
2	β	beta	20	κ	kappa	200	σ	sigma
3	γ	gamma	30	λ	lambda	300	τ	tau
4	δ	delta	40	μ	mu	400	υ	upsilon
5	ε	epsilon	50	ν	nu	500	φ	phi
6	\digamma	digamma	60	ξ	xi	600	χ	chi
7	ζ	zeta	70	\omicron	omicron	700	ψ	psi
8	η	eta	80	π	pi	800	ω	omega
9	θ	theta	90	φ	koppa	900	ϑ	sampi
10	ι	iota	100	ρ	rho	1000	α	hasta alpha

FIGURE 3.8 – Les chiffres grecs.

3.7 Exercices

1. faire les exercices de la page
<http://www.scientillula.net/NSI/premiere/codage%20des%20valeurs%20numeriques/autoeval.html#conversions>
2. Écrire en décimal le nombre binaire 110011.
3. Écrire en binaire le nombre décimal 1964.
4. Convertir en décimal le nombre 7123 écrit en base 8.
5. Convertir en base 5 le nombre décimal 2048.
6. Un *repunit* binaire est un nombre binaire qui ne comporte que le chiffre 1. Un nombre de Mersenne est un entier naturel qui s'écrit sous la forme $2^n - 1$ avec n entier naturel. Montrer que tout *repunit* binaire est un nombre de Mersenne (en base 10).
7. Convertir les nombres suivants vers la base octale et hexadécimale : $(150)_{10}$, $(210)_{10}$, $(1500)_{10}$, $(2018)_{10}$, $(2230)_{10}$
8. Convertir les nombres suivants vers la base octale et hexadécimale : $(11101010)_2$, $(1100110010)_2$, $(101010011010)_2$
9. Sachant que $(25)_{10} = (100)_b$, déterminer la valeur de b .
10. Trier par ordre croissant les valeurs suivantes données en hexadécimal : F413, 3F3D, F102, CF04, 7F12.
11. Convertir en forme binaire les fractions décimales suivantes.
 - (a) 0,5
 - (b) 0,2
 - (c) 0,9
 - (d) $\frac{1}{3}$
 - (e) 0,25
 - (f) $\frac{3}{8}$
12. Écrivez un programme C qui affiche en hexadécimal la valeur des variables des types `char`, `short`, `int`, `long`, auxquelles on aura donné la valeur `-1`.

13. En Java, écrivez un programme qui initialise des variables des types `byte` (un octet), `short` (deux octets), `int` (4 octets) et `long` (8 octets) à la plus petite valeur (négative), puis retranchez 1 à chacune de ces variables. Que constatez-vous ?

Note : on peut écrire `byte b = (byte)0x01;` pour donner la valeur hexadécimale 01 à `b`.

Est-ce que `long l = 0ABCDEFFFABCDEFFFL;` fonctionne ?

Ou faut-il écrire `long l = 0xABCDEFABCDEFLL;` ?

14. Écrivez un programme Java qui montre si le décalage à droite ou à gauche (`>>` et `<<`) a une influence sur le signe de la variable que l'on décale. On traitera les cas de deux valeurs positives, deux valeurs négatives et deux décalages à droite et deux à gauche.

15. On pourra aussi tester l'opérateur `>>>` de Java. Comment se comporte-t-il avec des entiers positifs et négatifs ? (on pourra utiliser la méthode `Integer.toBinaryString` pour afficher la valeur d'un entier en binaire)

16. Que représentent les inscriptions sur la tablette Plimpton 322 ? Pouvez-vous déchiffrer une ligne ?

17. Écrivez 9184 en utilisant le codage des Grecs ; qu'obtient-on ?

18. Étudiez le rapport de 1703 de Leibniz :

<https://hal.science/ads-00104781/document>

Chapitre 4

Logique numérique

Un ordinateur numérique (par opposition à analogique) est basé sur le stockage et le traitement de données binaires. Ce chapitre examine comment ces traitements peuvent être implémentés.

4.1 Logique formelle

La logique formelle est un vaste domaine qui est très antérieur à l'informatique. Au 4^e siècle avant J.-C., Aristote en a déjà établi les règles. La logique n'a guère évolué jusqu'au 19^e siècle, mais des progrès très importants ont été accomplis au tournant du 20^e siècle, notamment par des mathématiciens comme Gottlob Frege, Giuseppe Peano, Bertrand Russell, et beaucoup d'autres.

4.2 Algèbre booléenne

L'algèbre booléenne est une discipline permettant d'analyser le comportement de systèmes numériques en manipulant deux valeurs particulières, VRAI et FAUX. L'adjectif « booléen » fait référence au mathématicien anglais George Boole (1815-1864) qui a notamment publié en 1854 l'ouvrage *An investigation of the laws of thought on which to found the mathematical theories of logic and probabilities*.

Un autre personnage important dans le développement de cette discipline est Augustus de Morgan (1806-1871) qui a formulé les lois qui portent son nom.

Beaucoup plus tard, en 1938, Claude Shannon (1916-2001) a eu l'idée d'utiliser l'algèbre booléenne pour résoudre des problèmes de conception de circuits.

L'algèbre booléenne s'avère être un outil pratique dans deux secteurs :

- c'est un moyen concis pour décrire la fonction d'un circuit numérique;
- étant donné une fonction à implémenter, l'algèbre booléenne peut être utilisée pour développer une implémentation simplifiée de cette fonction.

Dans cette algèbre, il y a des variables logiques et des opérations logiques. Une variable peut prendre la valeur 1 (VRAI) ou 0 (FAUX). Les opérations logiques de base sont ET, OU et NON, qui sont symboliquement représentées par un point (\cdot) ou une croix (\times), un signe plus (+) et une barre :

$$A \text{ ET } B = A \cdot B$$

$$A \text{ OU } B = A + B$$

$$\text{NON } A = \overline{A}$$

L'opération ET rend vrai (valeur binaire 1) si et seulement si ses deux opérandes sont vraies. L'opération OU rend vrai si l'une ou les deux opérandes sont vraies. Et l'opération unaire NON inverse la valeur de son opérande. Par exemple, dans

$$D = A + (\overline{B} \cdot C)$$

D est égal à 1 si A est égal à 1 ou si à la fois $B = 0$ et $C = 1$. Sinon, D est égal à 0.

En l'absence de parenthèses, l'opérateur ET sera prioritaire par rapport au OU et le point du ET pourra être supprimé s'il n'y a pas d'ambiguïté :

$$A + B \cdot C = A + (B \cdot C) = A + BC$$

La table 4.1 définit les opérations logiques de base sous forme de *table de vérité*. La table liste aussi trois autres opérateurs utiles, XOU, NET et NOU. Souvent, on utilisera les abréviations anglaises XOR, NAND et NOR. Ces opérations sont le OU exclusif (XOR) et les négations de ET et OU.

À l'exception de la négation (NON ou NOT), ces opérations peuvent être généralisées à plus de deux variables.

P	Q	NON P (\bar{P})	P ET Q ($P \cdot Q$)	P OU Q ($P + Q$)	P NET Q ($\bar{P} \cdot \bar{Q}$)	P NOU Q ($\bar{P} + \bar{Q}$)	P XOU Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

TABLE 4.1 – Opérateurs booléens sur une ou deux variables.

L'algèbre booléenne conduit à un certain nombre d'identités :

$A \cdot B = B \cdot A$	$A + B = B + 1$	commutativité
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	distributivité
$1 \cdot A = A$	$0 + A = A$	identité
$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$	inverse
$0 \cdot = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B + C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	associativité
$\bar{A \cdot B} = \bar{A} + \bar{B}$	$\bar{A + B} = \bar{A} \cdot \bar{B}$	th. de De Morgan

4.3 Portes logiques

Le bloc de base de tous les circuits logiques est la porte logique. Les fonctions logiques sont implémentées en interconnectant des portes logiques.

Une porte logique est un circuit électronique qui produit un signal de sortie qui est une simple opération booléenne de ses signaux d'entrée. Les portes de base utilisées sont ET (AND), OU (OR), NON (NOT), NET (NAND), NOU (NOR) et XOU (XOR). Le tableau de la figure 4.1 résume ces portes, en utilisant les symboles du standard IEEE 91.

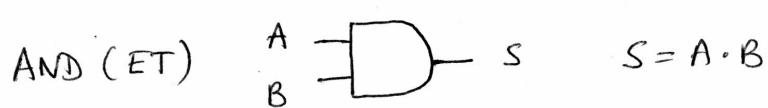
Lorsque une ou plusieurs des valeurs d'entrée sont modifiées, la valeur correcte du signal de sortie apparaît presque instantanément, à part le délai de propagation des signaux dans la porte. Dans certains cas, une porte est implémentée avec deux sorties, l'une étant la négation de l'autre.

L'état VRAI (1) correspond ou bien à un état de tension haut ou bas, suivant le type de circuit.

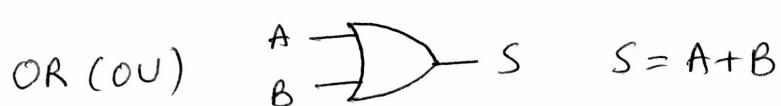
En pratique, on n'utilise pas toutes les portes dans les circuits, mais seulement un ou deux types de portes. Il est donc important d'identifier les ensembles fonctionnellement complets de portes. Des ensembles complets sont :

- ET, OU, NON
- ET, NON
- OU, NON
- NET (NAND)
- NOU (NOR)

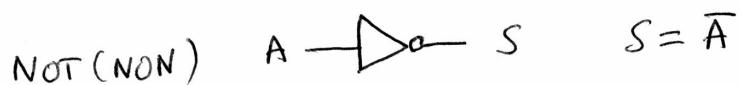
La figure 4.4 montre comment les fonctions ET, OU et NON peuvent être implémentées uniquement à l'aide de portes NET (ou NAND). De même, la figure 4.5 montre comment les fonctions ET, OU et NON peuvent être implémentées uniquement à l'aide de portes NOU (ou NOR).



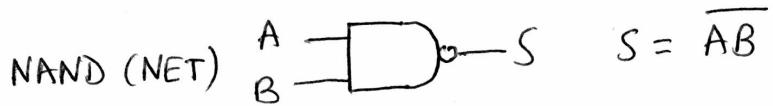
A	B	S
0	0	0
0	1	0
1	0	0
1	1	1



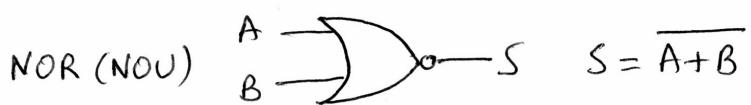
A	B	S
0	0	0
0	1	1
1	0	1
1	1	1



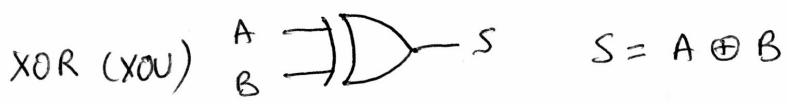
A	S
0	1
1	0



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



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



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

FIGURE 4.1 – Portes logiques de base.

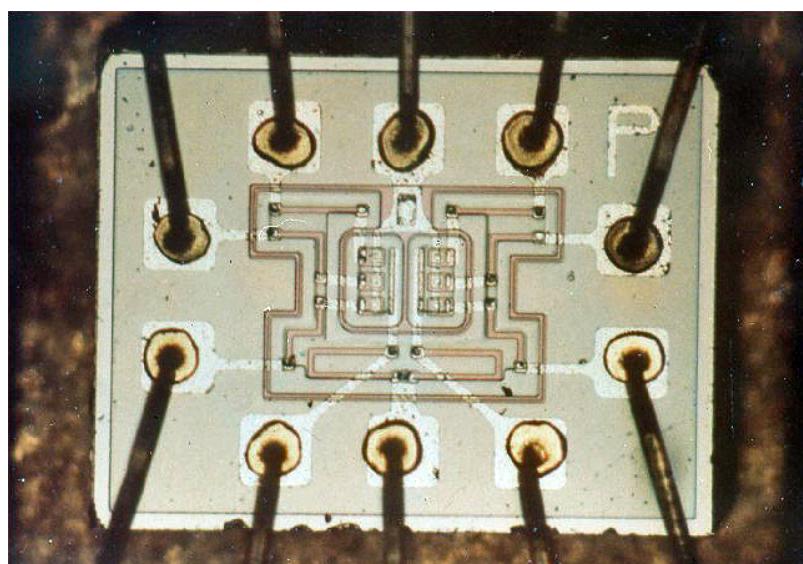


FIGURE 4.2 – Un circuit intégré de NOR (NON-OU) de l'ordinateur qui contrôlait les vaisseaux Apollo, années 1960 (source : Wikipédia).

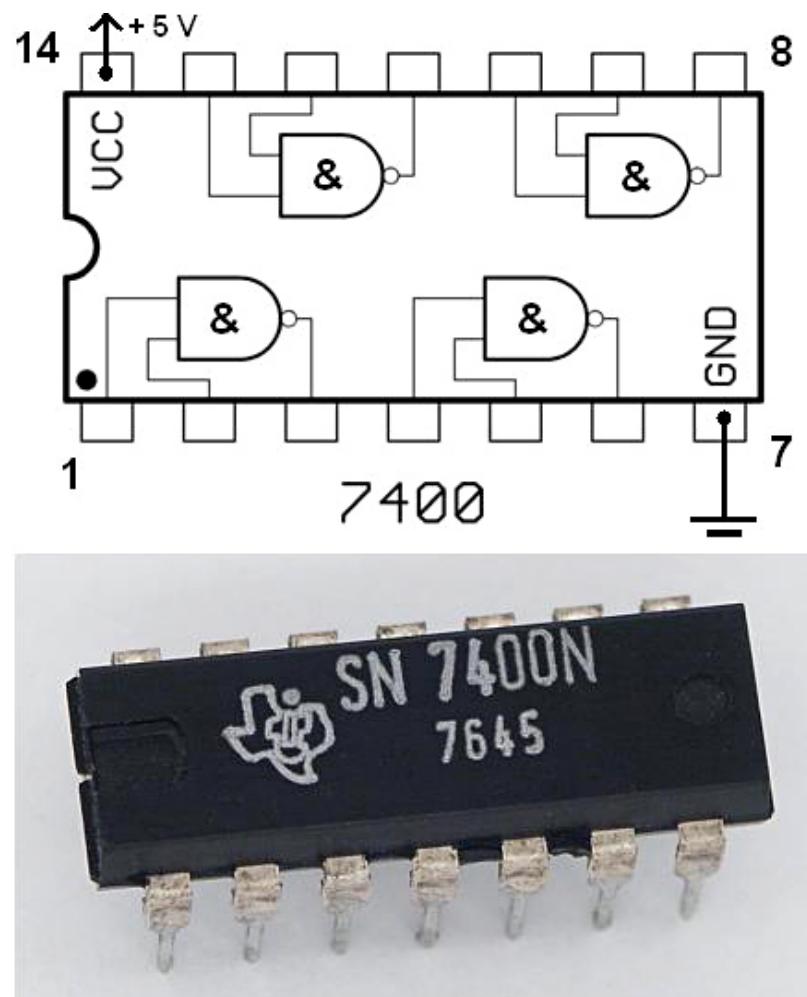


FIGURE 4.3 – Un circuit Texas Instruments SN7400N avec quatre portes NAND.

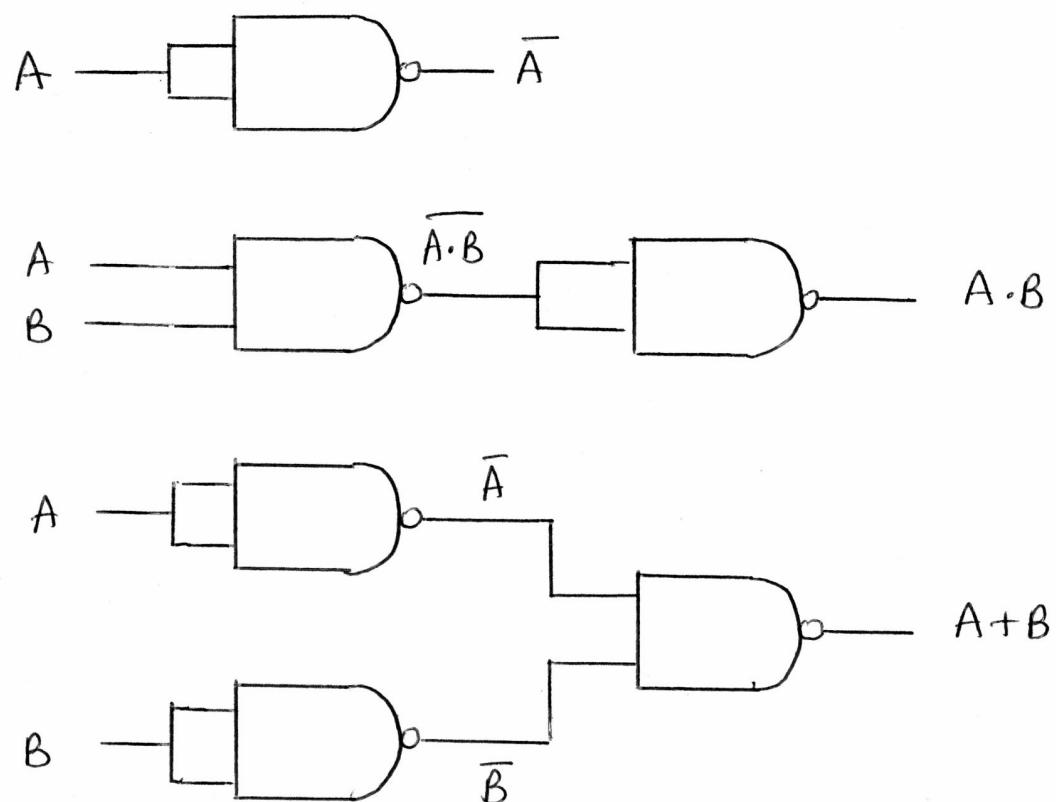


FIGURE 4.4 – Quelques utilisations de portes NAND.

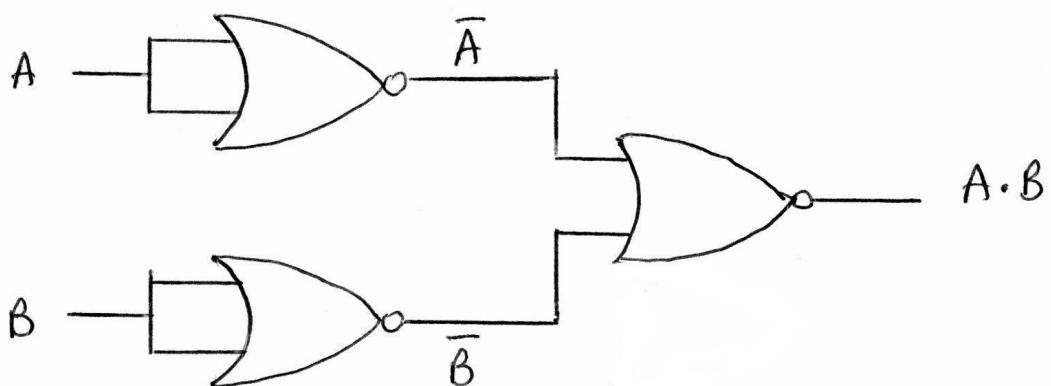
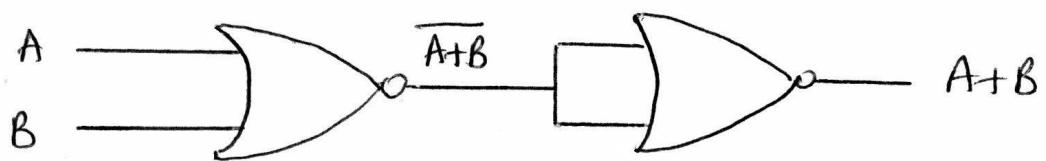
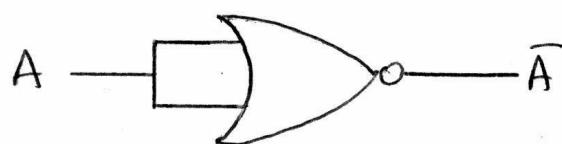


FIGURE 4.5 – Quelques utilisations de portes NOR.

4.4 Circuits combinatoires

Un *circuit combinatoire* est un ensemble interconnecté de portes dont la valeur de sortie n'est fonction que des entrées à ce moment donné. En général, un tel circuit a n entrées et m sorties. On peut définir un tel circuit de trois manières :

- table de vérité : pour chacune des 2^n combinaisons de signaux d'entrée, on liste la valeur binaire de chacun des m signaux de sortie ;
- symboles graphiques : on donne l'organisation des portes interconnectées ;
- équations booléennes : chaque sortie est exprimée sous la forme d'une fonction booléenne des signaux d'entrée.

Pour la fonction de la figure 4.6, on peut écrire

$$F = \overline{ABC} + \overline{ABC} + AB\overline{C}$$

Cette équation conduit à l'implémentation de la figure 4.7.
On peut aussi écrire

$$\begin{aligned} F &= (\overline{\overline{ABC}}) \cdot (\overline{\overline{ABC}}) \cdot (\overline{\overline{ABC}}) \cdot (\overline{\overline{ABC}}) \cdot (\overline{\overline{ABC}}) \\ &\quad (A + B + C) \cdot (A + B + \overline{C}) \cdot (\overline{A} + B + C) \cdot (\overline{A} + B + \overline{C}) \cdot (\overline{A} + \overline{B} + \overline{C}) \end{aligned}$$

Cette équation conduit à l'implémentation de la figure 4.8.
Il existe plusieurs méthodes de simplifications d'expressions logiques, mais elles ne sont pas détaillées ici.

A	B	C	S
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

FIGURE 4.6 – Une fonction booléenne sur trois variables.

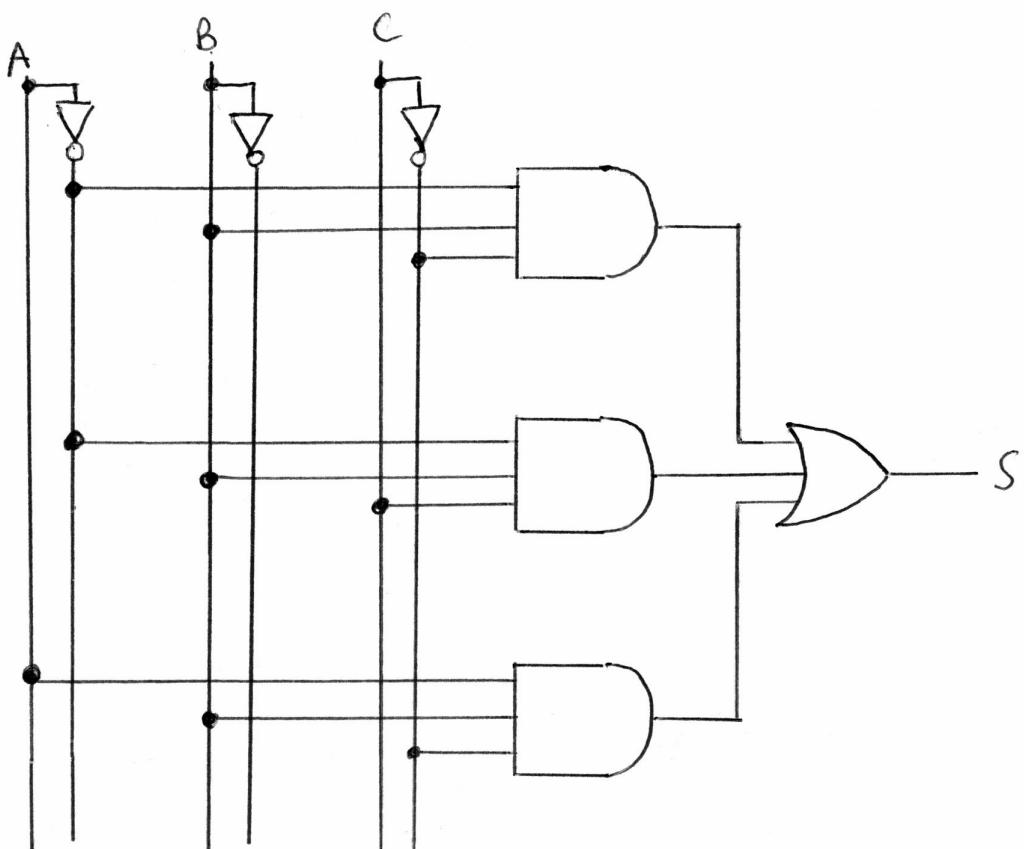


FIGURE 4.7 – Implémentation de la table précédente sous forme de sommes de produits.

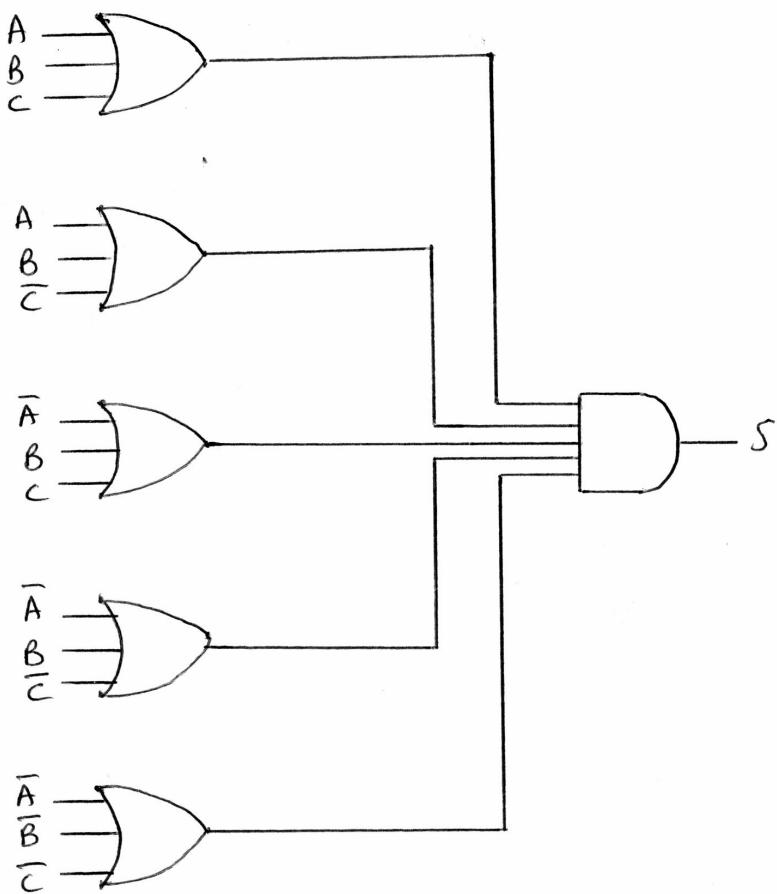


FIGURE 4.8 – Implémentation de la table précédente sous forme de produits de sommes.

4.4.1 Multiplexeurs

Un multiplexeur (MUX) relie plusieurs entrées à une unique sortie. À un moment donné, l'une des entrées est sélectionnée et transmise vers la sortie. La figure 4.9 illustre un diagramme pour un multiplexeur 4-vers-1. Les quatre entrées sont D_0, D_1, D_2 et D_3 . Le choix de l'entrée est déterminé par les deux sélecteurs S_2 et S_1 .

Un multiplexeur peut être définir par une table de vérité telle que

S_2	S_1	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Il s'agit en fait d'une forme simplifiée de table de vérité. Au lieu de montrer toutes les combinaisons des entrées, on montre de quelle entrée la sortie est une copie.

La figure 4.11 montre une implémentation possible de ce multiplexeur avec des portes logiques. Le même principe peut être utilisé pour réaliser des multiplexeurs 8-vers-1, 16-vers-1, etc.

Les multiplexeurs sont utilisés dans les circuits numériques pour contrôler des signaux et le flux des données. Un exemple est le chargement du compteur ordinal (PC). La valeur de ce compteur peut provenir de plusieurs sources différentes :

- un compteur binaire, si le compteur ordinal doit être incrémenté pour la prochaine instruction ;
- le registre d'instruction (IR), si une instruction de branchement utilisant une adresse directe vient juste d'être exécuté ;
- la sortie de l'UAL, si l'instruction de branchement spécifie une adresse calculée.

Ces différentes entrées pourraient être reliées aux entrées d'un multiplexeur et le PC connecté en sortie. Comme le PC contient plu-

sieurs bits, il faudrait plusieurs multiplexeurs. La figure 4.12 illustre ceci pour des adresses sur 16 bits.

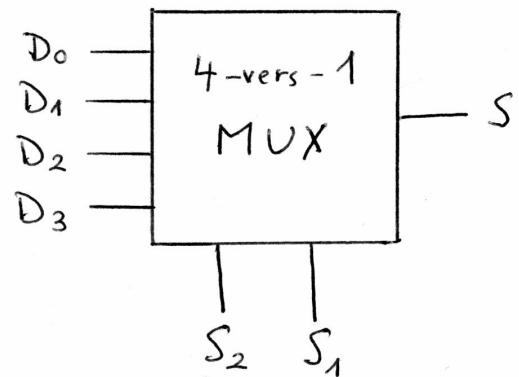


FIGURE 4.9 – Représentation d'un multiplexeur 4-vers-1.

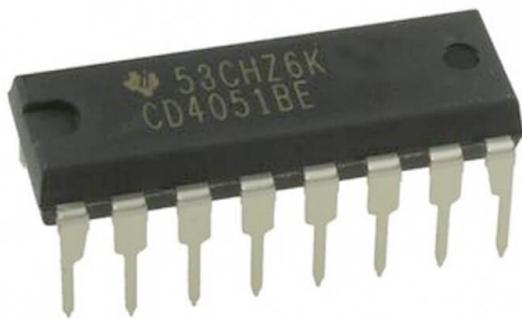


FIGURE 4.10 – Un multiplexeur CD4051.

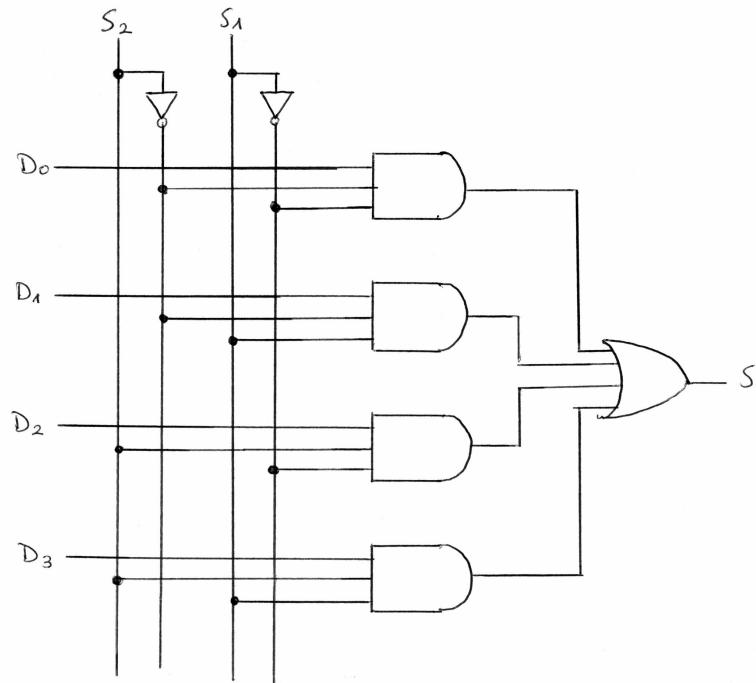


FIGURE 4.11 – Implémentation d'un multiplexeur.

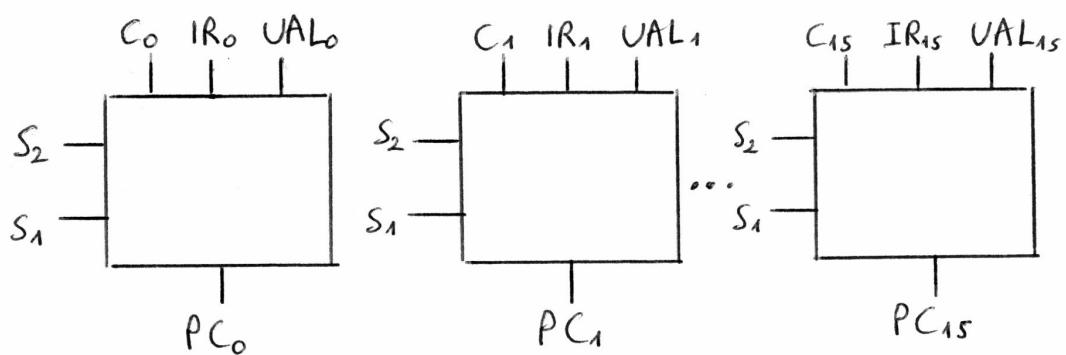


FIGURE 4.12 – Entrée d'un multiplexeur vers un compteur ordinal (Program Counter).

4.4.2 Décodeurs

Un décodeur est un circuit combinatoire avec un certain nombre de sorties, dont une seule est active à un moment donné. La sortie active dépend des valeurs des lignes d'entrées. En général, un décodeur a n entrées et 2^n sorties. La figure 4.13 montre un décodeur avec trois entrées et huit sorties.

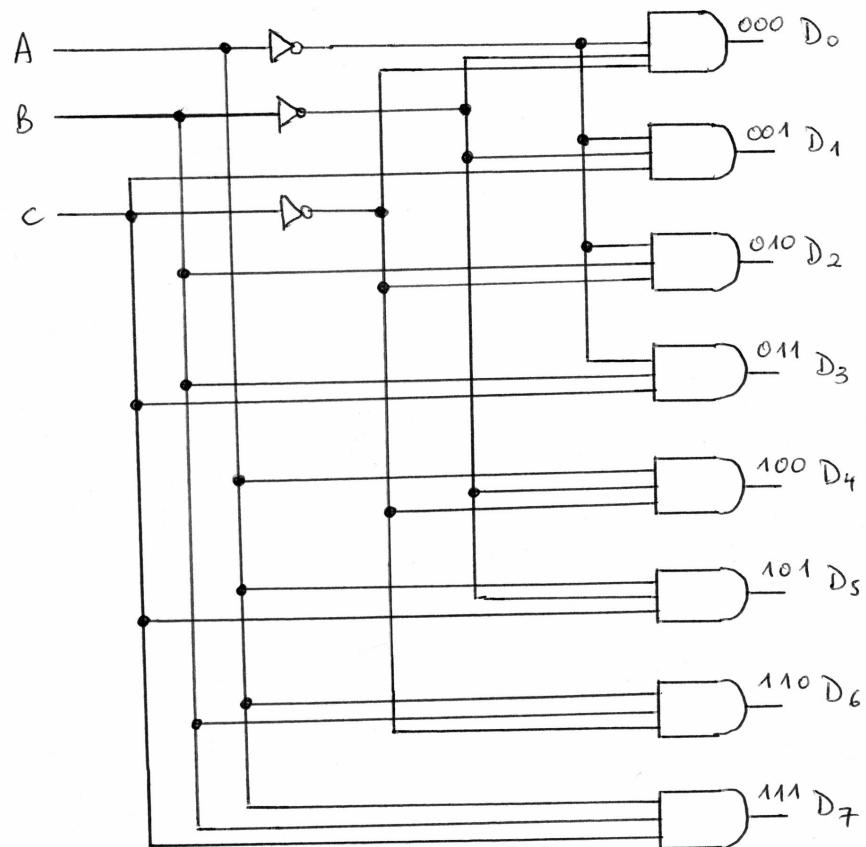


FIGURE 4.13 – Décodeur avec trois entrées et $2^3 = 8$ sorties.

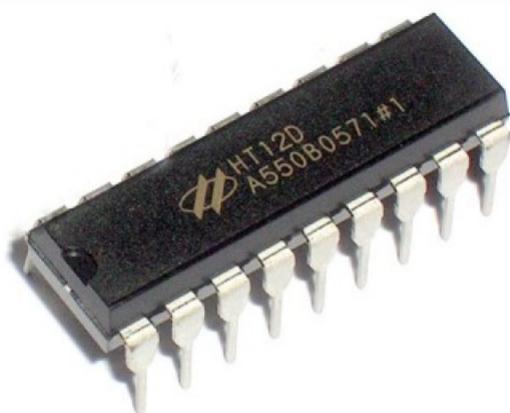


FIGURE 4.14 – Un circuit décodeur.

Les décodeurs ont beaucoup d'applications dans les ordinateurs numériques. Un exemple est le décodage d'adresses. Imaginons que l'on veuille construire une mémoire de 1K-octets en utilisant quatre puces de RAM de 256×8 -bits. On souhaite avoir un espace d'adresse unifié qui peut être découpé comme suit :

adresse	puce
0000-00FF	0
0100-01FF	1
0200-02FF	2
0300-03FF	3

Chaque plage contient 256 valeurs. Chaque puce nécessite huit lignes d'adresse et celles-ci sont fournies par les 8 bits inférieurs de l'adresse. Les deux bits supérieurs de l'adresse de 10 bits sont utilisés pour sélectionner l'une des quatre puces. Pour cela, un décodeur 2-vers-4 est utilisé, dont la sortie active l'une des quatre puces (figure 4.15).

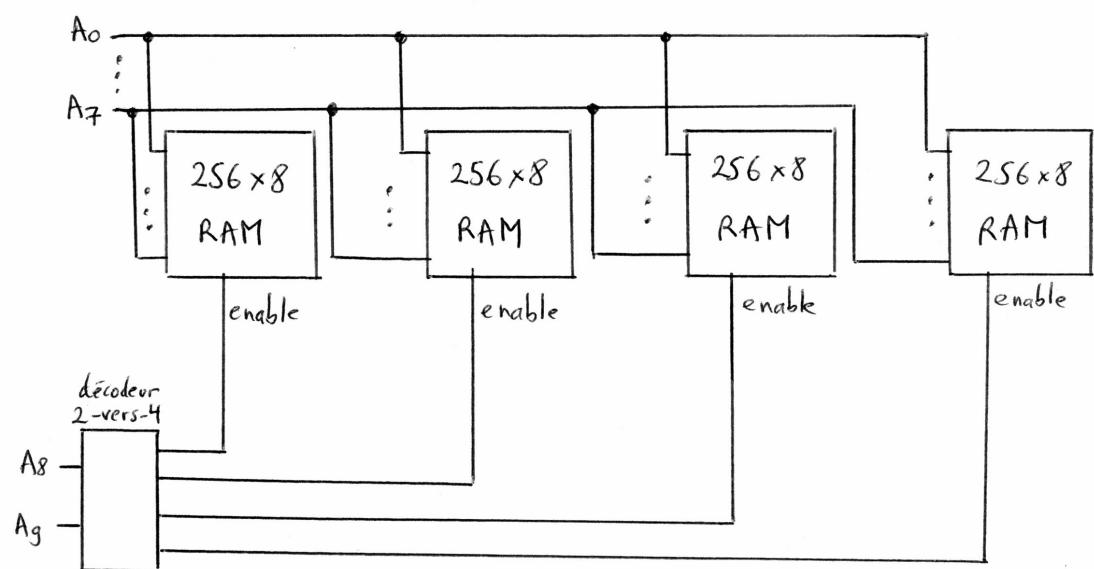


FIGURE 4.15 – Décodage d'adresse.

4.4.3 Démultiplexeur

En utilisant une ligne d'entrée supplémentaire, un décodeur peut être utilisé comme démultiplexeur. Un démultiplexeur fait l'inverse de ce que fait un multiplexeur. Il connecte une seule entrée à une sortie parmi plusieurs. (Un multiplexeur connecte une entrée parmi plusieurs à une seule sortie.) Un exemple est donné dans la figure 4.16.

Dans cet exemple, il y a une ligne d'entrée, à savoir la donnée, et 2^n sorties. L'une de ces sorties reprend l'entrée et les autres sont à zéro. Celle des sorties qui reprend l'entrée est déterminée par l'adresse sur n bits.

Le lien avec les décodeurs est le suivant. Le décodeur prend n entrées et produit une sortie parmi 2^n . On fait ensuite un ET logique entre chacune des 2^n sorties et la donnée en entrée.

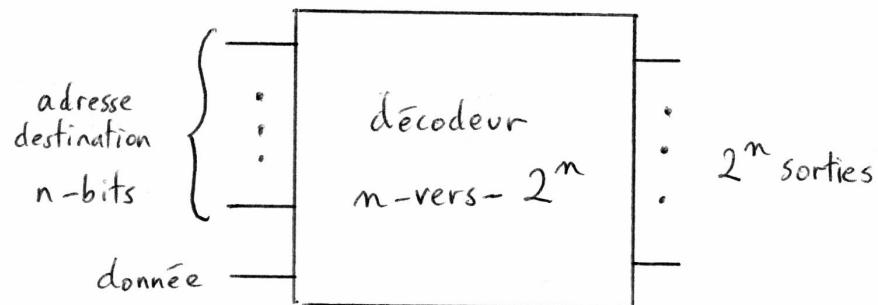


FIGURE 4.16 – Implémentation d'un démultiplexeur en utilisant un décodeur.

4.4.4 Additionneurs

Les circuits combinatoires peuvent être utilisés pour implémenter des fonctions arithmétiques. Nous examinerons ici simplement le cas de l'addition.

L'addition de deux bits peut être représentée par une table de vérité :

A	B	Somme	Retenue
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Chaque somme de deux bits donne un bit d'unité et une retenue (*carry* en anglais). Par exemple $0 + 1 = (0)1$ et $1 + 1 = (1)0$. Le chiffre entre parenthèses est la retenue.

En général, pour faire une addition, même sur un seul bit, il faut aussi récupérer la retenue précédente. Il y donc trois entrées, à savoir la retenue en entrée R_e , A et B , et d'autre part deux sorties, S et la retenue en sortie R_s , d'où la table :

R_e	A	B	S	R_s
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

On peut donc facilement construire un additionneur correspondant à cette table et prenant trois entrées (R_e , A et B) et produisant deux sorties (S et R_s). Quatre de ces blocs peuvent alors être combinés pour réaliser un additionneur sur 4 bits (figure 4.17).

Si les entrées sont appelées A , B et C , une implémentation possible d'un additionneur sur un bit est celle de la figure 4.19.

La figure 4.20 montre comment un additionneur 32 bits peut être construit à l'aide d'additionneurs 8 bits.

L'un des inconvénients de cette approche est qu'il faut attendre la retenue du bloc précédent pour faire le calcul d'un bloc. Il est possible de déterminer la retenue C_i sans faire référence à la retenue C_{i-1} , mais uniquement en utilisant les données. Cette technique s'appelle le *carry lookahead*. Elle devient compliquée pour de grands nombres de bits, mais elle est typiquement appliquée pour 4 à 8 bits à la fois.

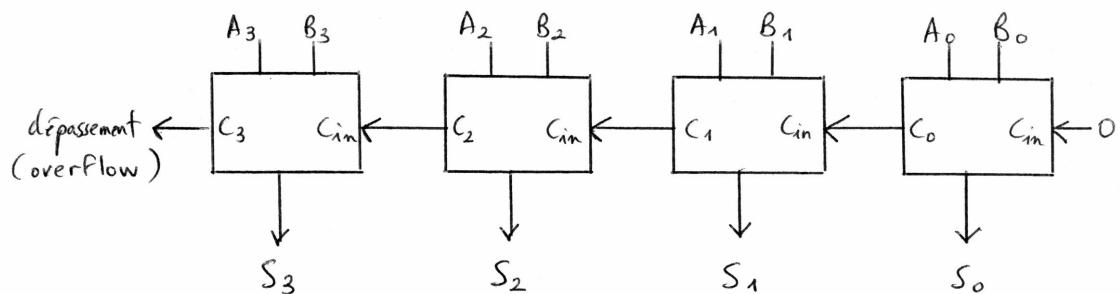


FIGURE 4.17 – Un additionneur 4-bits.



FIGURE 4.18 – Un circuit d'additionneur sur 4 bits.

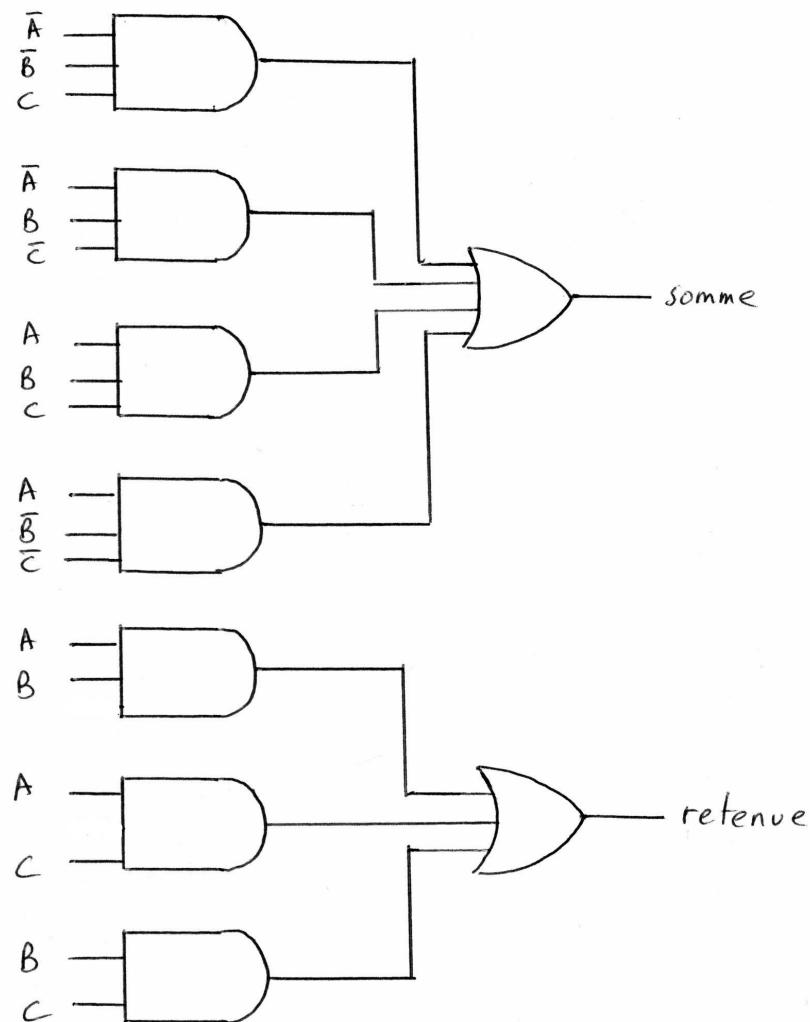


FIGURE 4.19 – Implémentation d'un additionneur.

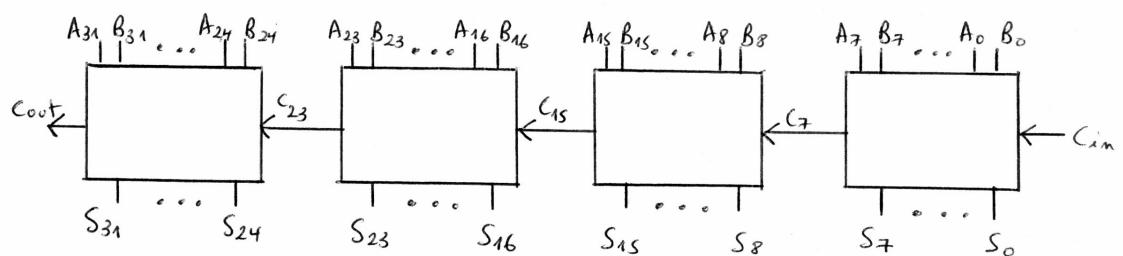


FIGURE 4.20 – Construction d'un additionneur 32-bits en utilisant des additionneurs 8-bits.

4.5 Un peu d'histoire

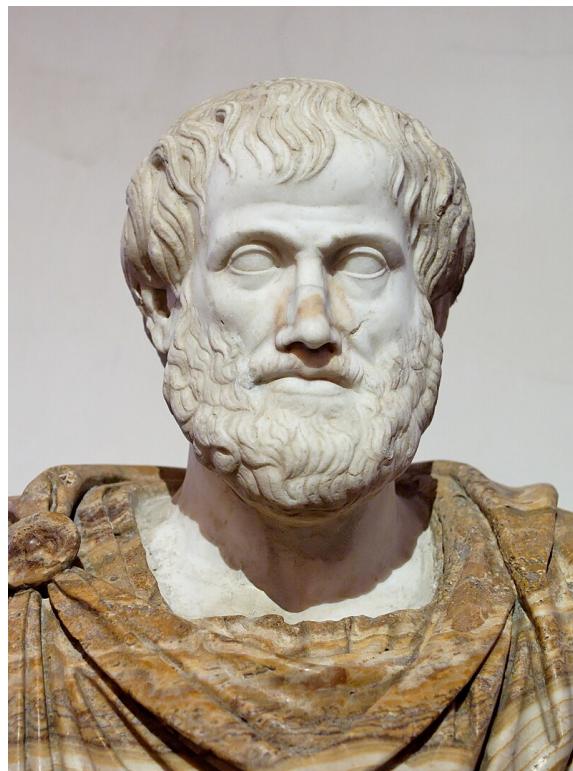


FIGURE 4.21 – Le philosophe grec Aristote (384-322 av. J.-C.), fondateur de la logique aristotélicienne. (source : Wikipédia)

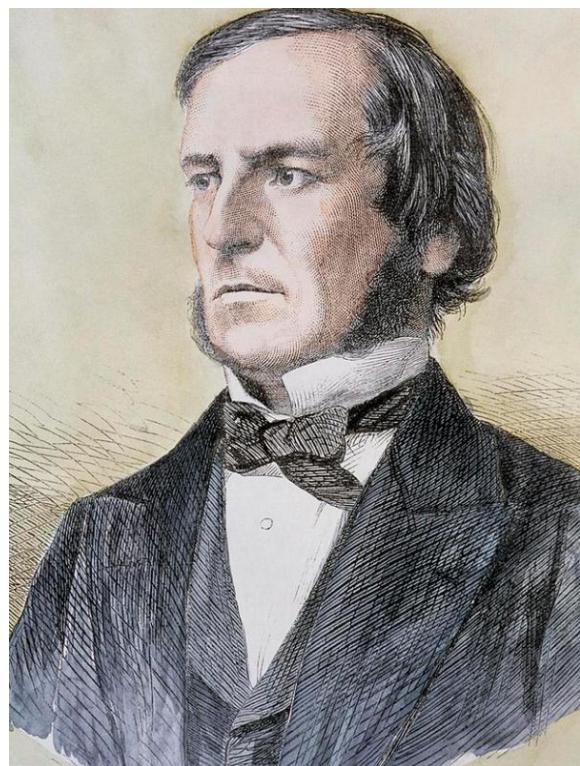


FIGURE 4.22 – George Boole (1815-1864). (source : Wikipédia)



FIGURE 4.23 – Augustus De Morgan (1806-1871). (source : Wikipédia)



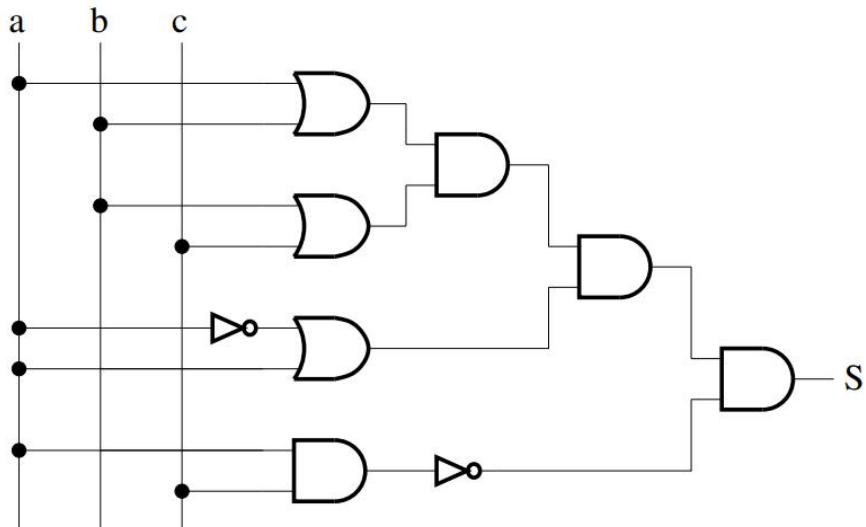
FIGURE 4.24 – Claude Shannon (1916-2001). (source : Wikipédia)

4.6 Exercices

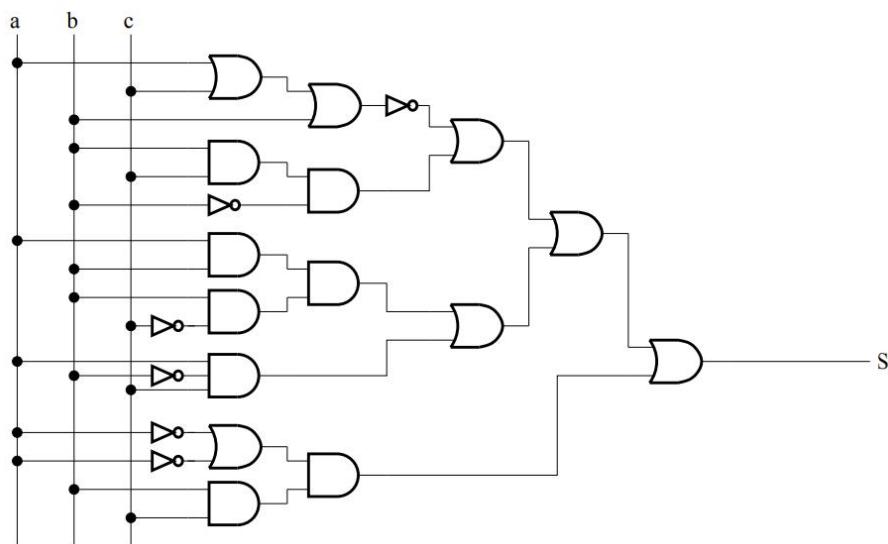
1. qu'est-ce qu'un syllogisme? donnez un exemple du syllogisme Barbara;
2. remplir la table de vérité :

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

3. Quelle est l'expression booléenne de la sortie S pour le circuit suivant?



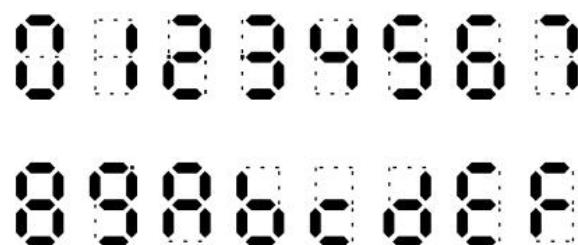
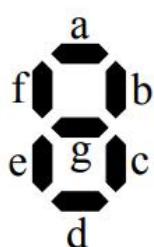
4. trouver le standard IEEE 91 et comparer ses notations à ceux du présent document;
5. Circuit mystère. Donner l'expression booléenne de S dans le circuit ci-dessous en n'utilisant que les opérateurs NON, ET et OU.



Calculer la table de vérité de ce circuit. Que fait-il ?

6. Réalisation d'un afficheur.

On cherche à réaliser un circuit afficheur hexadécimal pour une calculette. L'entrée est un nombre n en binaire sur 4 bits : b_0, b_1, b_2, b_3 . Les 7 sorties sont appelées a, b, c, d, e, f, g . Une sortie est à 1 si le segment correspondant est noir.



(a) Écrire les tables de vérité des sept sorties.

(b) En déduire le circuit correspondant.

7. Réaliser un multiplexeur à quatre entrées.

8. Réaliser le demi-additionneur à l'aide d'un minimum de multiplexeurs à 4 entrées.

9. Réaliser l'additionneur complet à l'aide d'un minimum de multiplexeurs à 8 entrées.

10. Réalisation d'un additionneur complet avec des décodeurs binaires $3 \Rightarrow 8$
11. Soit une information binaire sur 5 bits ($i_4 i_3 i_2 i_1 i_0$). Donner le circuit qui permet de calculer le nombre de 1 dans l'information en entrée en utilisant uniquement des additionneurs complets sur 1 bit ? Exemple : Si on a en entrée l'information $i_4 i_3 i_2 i_1 i_0 = 10110$ alors en sortie on obtient la valeur 3 en binaire (011) puisqu'il existe 3 bits qui sont à 1 dans l'information en entrée.
12. Effectuer à l'aide d'un minimum d'additionneurs de 2 nombres de 4 bits et d'un minimum de portes logiques, la multiplication de deux nombres positifs de 4 bits.
13. Construire un circuit logique capable de comparer deux nombres de 3 bits chacun : $A_0 A_1 A_2$ et $B_0 B_1 B_2$. En sortie, on voudrait avoir : 1 si $A_0 A_1 A_2 = B_0 B_1 B_2$, sinon 0.
14. Concevoir un multiplexeur 8 à 1 avec uniquement des multiplexeurs 2 à 1. Donner le schéma.

Chapitre 5

Arithmétique de l'ordinateur

L'arithmétique d'un ordinateur est habituellement effectuée sur deux types très différents de nombres, les entiers et les nombres réels. Dans chacun de ces cas, il importe de d'abord examiner la représentation choisie et ensuite les opérations possibles sur ces nombres.

5.1 L'unité arithmétique et logique

L'unité arithmétique et logique (UAL, ALU en anglais) est la partie de l'ordinateur qui effectue les opérations arithmétiques et logiques sur les données. Tous les autres éléments du système informatique sont là essentiellement pour apporter des données à l'UAL et ensuite pour en récupérer les résultats.

La figure 5.1 montre comment l'UAL est interconnectée avec le reste du processeur. Des opérandes pour des opérations arithmétiques ou logiques sont présentés à l'UAL dans des registres et les résultats d'une opération sont stockés dans des registres. Ces registres sont des emplacements de stockage temporaires au sein du processeur. L'UAL peut aussi positionner des drapeaux en fonction du résultat d'une opération. Il peut par exemple y avoir un drapeau in-

diquant qu'il y a eu un dépassement (*overflow*) lors d'un calcul. Les valeurs des drapeaux (*flags*) sont aussi stockées dans des registres au sein du processeur. Le processeur fournit des signaux qui contrôlent l'opération de l'UAL et les mouvements des données depuis et vers l'UAL.

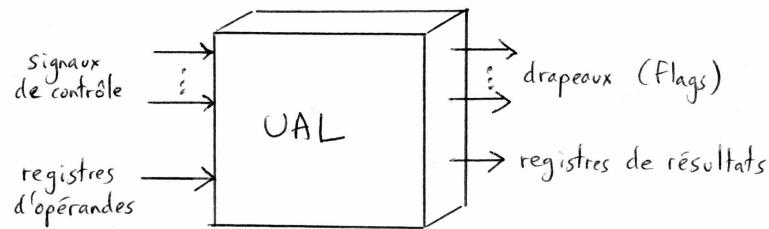


FIGURE 5.1 – Entrées et sorties de l'UAL.

5.2 Représentation des entiers

Dans la représentation arithmétique binaire, on peut représenter des nombres quelconques avec les chiffres 0 et 1, le signe moins et le point décimal (ou virgule). Par exemple

$$-1101.0101_2 = -13.3125_{10}$$

Mais pour des raisons de stockage et de traitement, nous ne disposons pas de symboles spéciaux, uniquement des 0 et des 1. Si on se limite aux entiers positifs ou nuls, la représentation est simple. Ainsi, avec un mot de 8 bits, on peut représenter les nombres de 0 à 255 :

00000000	=	0
00000001	=	1
...		
00101001	=	41
10000000	=	128
...		
11111111	=	255

Pour représenter des entiers négatifs, différentes solutions peuvent être envisagées. On peut utiliser le bit de poids fort pour indiquer le signe. Si le bit est 1, le nombre est négatif et s'il est 0, le nombre est positif ou nul. On pourrait donc avoir cette représentation :

$$\begin{aligned} +18 &= 00010010 \\ -18 &= 10010010 \end{aligned}$$

Cette représentation présente un certain nombre d'inconvénients. Ainsi, on ne peut pas naturellement faire les additions sur les valeurs binaires sans prendre en compte les signes. Il y a aussi le fait qu'il y a deux valeurs de 0 : +0 et -0.

La représentation précédente est de ce fait rarement utilisée. On utilise plutôt la représentation en complément à deux.

```
// En C, les entiers non signés peuvent être déclarés
// avec
unsigned char a; // un octet
unsigned short b; // deux octets (en général)
unsigned int c; // quatre octets (en général)
unsigned long d; // huit octets (en général)
```

```
// En Java, il n'y a pas d'entiers non signés
```

5.2.1 Complément à deux

Cette représentation utilise aussi le bit de poids fort pour indiquer le signe, mais si le signe est négatif, les bits qui suivent ne sont pas représentation binaire de la valeur absolue. Ainsi, -18 ne sera pas représenté par la valeur binaire `10010010`.

Dans la représentation en complément à deux, si on dispose de n bits, on pourra représenter les entiers de -2^{n-1} à $2^{n-1} - 1$. Par exemple, pour $n = 8$, on pourra aller de -128 à 127 . Il y a une seule valeur pour 0 . Et on dispose de règles de calcul simples qui seront détaillées plus loin.

```
// En C, les entiers signés peuvent être déclarés
// avec
char a; // un octet
short b; // deux octets (en général)
int c; // quatre octets (en général)
long d; // huit octets (en général)
```

```
// En Java, les entiers signés peuvent être déclarés
// avec
byte a; // un octet
short b; // deux octets
int c; // quatre octets
long d; // huit octets
```

Soit maintenant un nombre A sur n bits dans cette représentation. Si $A \geq 0$, alors le bit de signe $a_{n-1} = 0$. Les autres bits représentent la valeur absolue du nombre :

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

Il est facile de voir que les entiers positifs ou nul vont de 0 jusqu'à $n - 1$ bits à 1, c'est-à-dire jusqu'à $2^{n-1} - 1$.

Si $A < 0$, le bit de signe $a_{n-1} = 1$. Les $n - 1$ bits restant peuvent prendre une valeur quelconque parmi 2^{n-1} . Par conséquent, s'il n'y a pas de discontinuités (trous), les valeurs négatives représentables vont de -1 à -2^{n-1} .

Par convention, la valeur A représentée lorsque le bit de signe (a_{n-1}) vaut 1 est :

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

En d'autres termes, contrairement à une représentation des entiers uniquement positifs ou nuls, le bit de poids fort n'est pas affecté du poids 2^{n-1} , mais au contraire de son opposé. Les autres bits ont le poids naturel qu'ils auraient pour une représentation d'entiers uniquement positifs.

Comme $a_{n-1} = 0$ pour les entiers positifs ou nuls, l'équation précédente définit la représentation en complément à deux pour tous les nombres entiers, positifs ou négatifs.

Le tableau 5.1 compare différentes représentations d'entiers sur quatre bits. La représentation « biaisée » sera détaillée plus loin lorsqu'il sera question des nombres réels.

Rep. déc.	Rep. sign/abs	Comp. deux	Rep. biaisée
+8	–	–	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
-0	1000	–	–
-1	1001	1111	0110
-2	1010	1110	0101
-3	1011	1101	0100
-4	1100	1100	0011
-5	1101	1011	0010
-6	1110	1010	0001
-7	1111	1001	0000
-8	–	1000	–

TABLE 5.1 – Différentes représentations d'entiers sur quatre bits.

L'équation précédente permet donc de déterminer quelle valeur décimale est représentée par une série de n bits en complément à deux, mais elle permet aussi de faire l'opération inverse, c'est-à-dire de trouver les n bits à partir d'un entier exprimé en base 10. Cet entier doit d'abord se trouver dans l'intervalle de -2^{n-1} à $2^{n-1} - 1$. Si ce n'est pas le cas, il ne peut pas être représenté sur n bits. Par exemple, on ne peut pas représenter 11 sur 4 bits en représentation en complément à deux, car les valeurs représentables vont de -8 à

7.

Si la représentation est possible, on distingue deux cas : si A est positif ou nul, on a une simple représentation binaire et les bits de A sont obtenus par les méthodes décrites dans le chapitre sur la représentation binaire (section 3.3).

Si par contre A est négatif, l'équation précédente nous permet de construire un autre nombre en ajoutant 2^{n-1} . On a :

$$A + 2^{n-1}a_{n-1} = A + 2^{n-1} = \sum_{i=0}^{n-2} 2^i a_i$$

Par exemple, si $n = 4$, et que l'on veut connaître la représentation binaire de -5 , on peut calculer $-5 + 8 = 3$ et il suffit de convertir 3 en binaire. La représentation de -5 est donc 1 suivi de 011 , soit 1011 .

Mais on peut aussi écrire :

$$A + 2^{n-1} = (2^{n-1} - 1) - (-A) + 1$$

$2^{n-1} - 1$ est une suite de $n - 1$ bits à 1 . Par exemple, pour $n = 4$, on a $2^{n-1} - 1 = 7_{10} = 111_2$. Et $-A$ est un entier entre 1 et 2^{n-1} .

Si $-A < 2^{n-1}$, ce qui précède revient à faire le complément à 1 sur $n - 1$ bits, puis à ajouter 1 . Par exemple, pour $A = -5$, on fait $7 - 5 = 2 = 10_2$, puis on ajoute 1 , ce qui donne 11_2 . Par conséquent, la représentation en complément à 2 de -5 est 1011_2 , comme vu précédemment. On peut vérifier que cela fonctionne aussi avec $-A = 2^{n-1}$, par exemple avec $A = -8$ si $n = 4$.

On peut vérifier qu'en faisant le complément à 1 sur n bits (c'est-à-dire en inversant chaque bit), cette procédure fonctionne aussi. Donc, en résumé, on peut facilement obtenir le complément à 2 d'un entier négatif A en déterminant le complément à 1 de $-A$, puis en ajoutant 1 .

Une autre manière de voir les choses est d'observer que si $A < 0$, A est représenté par $2^n + A$. Par exemple, avec $n = 4$, -8 est représenté par $16 - 8 = 8$. -5 est représenté par $16 - 5 = 11$. -1 est représenté par $16 - 1 = 15$.

Or $2^n + A = 2^n - 1 - (-A) + 1$, c'est-à-dire que la représentation en complément à deux est clairement obtenue par le complément à 1 de la valeur absolue ($2^n - 1 - (-A)$) en ajoutant 1, et cela effectivement sur n bits.

5.2.2 Extension de plage

Il arrive que l'on doive transférer un entier sur n bits vers un stockage sur m bits, où $m > n$. Par exemple, un entier pourrait être stocké en complément à deux sur 8 bits et on veut le stocker sur 16 bits. Si la représentation utilise simplement un bit de signe et la valeur absolue, cette extension de plage est très simple.

Par contre, si la représentation utilise le complément à deux, il faut transférer le bit de signe à gauche de la nouvelle valeur et remplir le reste des bits par des copies du bit de signe. Nous ne justifierons pas cette procédure, mais en donnons simplement quelques exemples :

$$\begin{aligned} 18 &= 00010010 \\ &= 00000000\ 00010010 \\ -18 &= 11101110 \\ &= 11111111\ 11101110 \end{aligned}$$

5.3 Arithmétique entière

5.3.1 Négation

Si on dispose d'un entier $A > 0$ en complément à deux (donc inférieur à 2^{n-1}), on a vu plus haut que l'on pouvait trouver la représentation de $-A$ en déterminant le complément à 1, puis en ajoutant 1. On fait donc l'opération

$$A \rightarrow (2^n - 1 - A) + 1$$

Mais on peut voir cette opération comme une opération générale sur une valeur binaire stockée sur n bits. Elle peut aussi être appliquée à une valeur qui représente une valeur négative. Ce que l'on voit simplement, c'est que si l'on applique l'opération précédente une seconde fois, on revient à la valeur initiale :

$$[2^n - 1 - ((2^n - 1 - A) + 1)] + 1 = A$$

On peut donc passer de A à $-A$ et inversement par la même opération, à savoir déterminer le complément à 1 et ajouter 1.

5.3.2 Addition et soustraction

Avec la représentation en complément à deux, les additions et soustractions sont calculées normalement, comme si les valeurs n'étaient pas signées (i.e., positives). On n'a pas besoin de prendre en compte le bit de signe.

Il peut y avoir une retenue qui sort de la valeur mais qui est ignorée. Par exemple, sur 4 bits, $-1 + 1 = 0$, c'est-à-dire $1111 + 1 = (1)0000$.

Par contre, le résultat d'une opération peut conduire à un dépassement (*overflow*). Par exemple, sur 4 bits, on ne peut pas calculer $6 + 3 = 9$, car 9 n'est pas représentable en complément à 2. Le calcul donnerait $0110 + 0011 = 1001$, mais le résultat est négatif (-7). La règle simple pour détecter un tel dépassement est :

Si deux nombres sont ajoutés et qu'ils ont le même signe, il y a un dépassement si le résultat est de signe opposé.

On pourra vérifier qu'un dépassement peut se produire avec ou sans retenue sortante.

Pour faire une soustraction, on ajoute le complément à deux du nombre à soustraire.

La figure 5.2 permet de voir géométriquement l'organisation des nombres en complément à deux sur 4 bits et sur n bits. Dans cette

représentation, le complément à deux correspond aux pointillés horizontaux. 0001 est le complément à deux de 1111, 0101 est le complément à deux de 1011, etc.

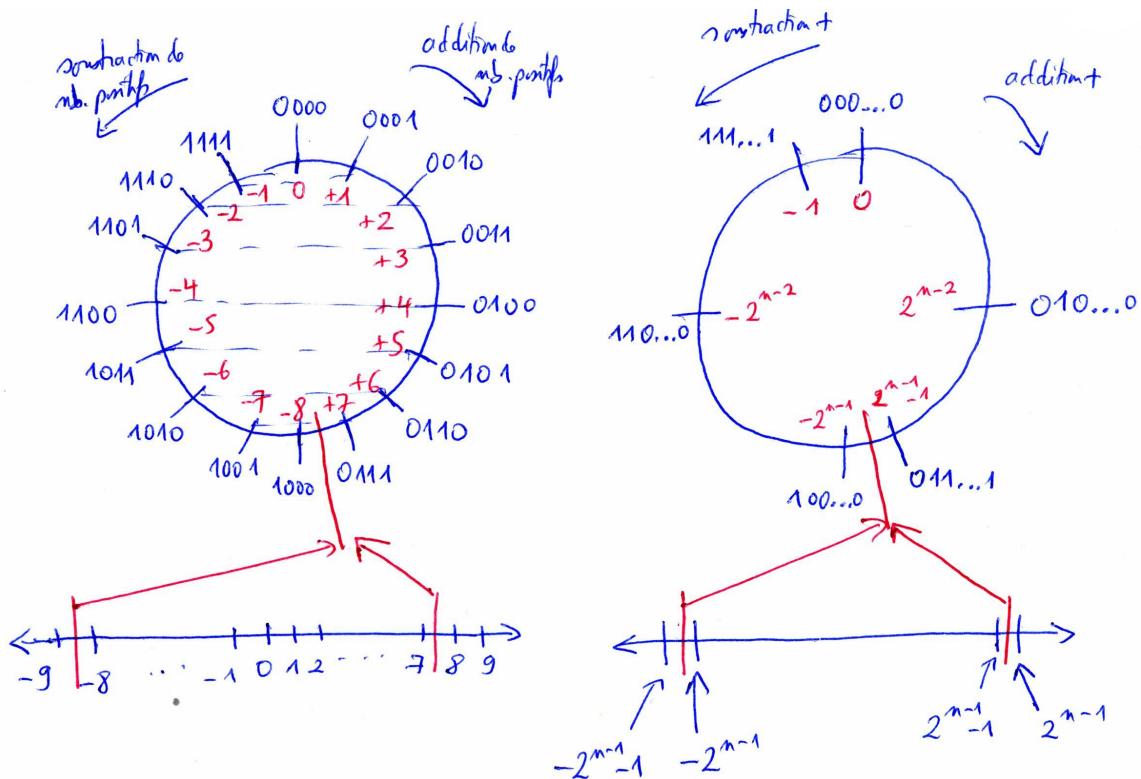


FIGURE 5.2 – Représentation géométrique des entiers en complément à deux.

La figure 5.3 montre la circuiterie qui est nécessaire pour calculer une addition ou une soustraction. L'élément central est l'additionneur qui reçoit deux nombres et calcule leur somme. L'additionneur produit aussi un indicateur de dépassement (bit d'overflow). (Une implémentation d'un additionneur a été donnée dans la section 4.4.4.) Les deux nombres proviennent de deux registres *A* et *B*. Dans le cas de la soustraction, la valeur du registre *B* est passée dans un complémenteur. Un signal de contrôle détermine si le complémenteur est utilisé ou non. En pratique, on va toujours calculer le complément à deux, mais ne l'utiliser que si l'opération est une soustraction.

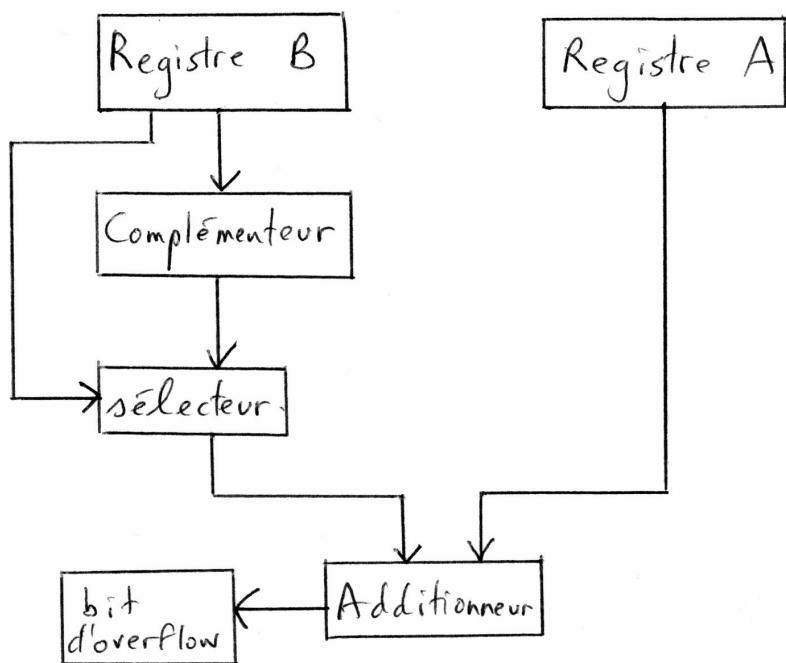


FIGURE 5.3 – Diagramme de bloc pour l'addition et la soustraction hardware.

5.3.3 Multiplication

La multiplication des entiers est plus complexe. Considérons d'abord la multiplication d'entiers non signés. En voici un exemple :

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

Pour faire un tel calcul, il faut donc calculer des produits partiels et les additionner. Il faut aussi les décaler. Les produits partiels sont faciles à calculer. Ils sont ou bien égaux au multiplicande (la première valeur), ou bien à zéro.

Il faut noter qu'en multipliant deux entiers sur n bits, le résultat peut occuper $2n$ bits (mais pas plus).

L'implémentation de cette procédure n'est pas trop difficile. On peut notamment s'affranchir de faire la somme des produits partiels à la fin, et les faire au fur et à mesure, ce qui élimine le besoin de stocker tous les produits partiels. La figure 5.4 montre une implémentation possible. Le multiplicande se trouve en M et le multiplicateur en Q . Le registre A est initialement mis à zéro. On commence par mettre le multiplicande en A (en l'ajoutant à la valeur initiale de A qui est zéro). Ensuite, au lieu de décaler le multiplicande à gauche, on va décaler à droite le résultat partiel. Comme celui-ci va prendre plus de place que le multiplicande, on utilise un second registre qui est Q . Ce registre sert donc à la fois à stocker le multiplicateur et à stocker les bits de droite du résultat. À chaque fois, on va utiliser le bit de droite de Q , à savoir Q_0 pour déterminer si le multiplicande doit être ajouté ou non. Le bit C sert de retenue et doit aussi être décalé à droite en A_{n-1} . Il ne peut pas être ignoré.

On peut résumer ce calcul par le tableau suivant :

<i>C</i>	<i>A</i>	<i>Q</i>	<i>M</i>		
0	0000	1101	1011	valeurs initiales	
0	1011	1101	1011	Add	premier
0	0101	1110	1011	Déc	cycle
0	0010	1111	1011	Déc	2 ^e cycle
0	1101	1111	1011	Add	troisième
0	0110	1111	1011	Déc	cycle
1	0001	1111	1011	Add	quatrième
0	1000	1111	1011	Déc	cycle

On notera que lors du second cycle il n'y a pas d'addition, ce qui correspond au produit partiel 0000.

Le résultat final est dans *A* suivi de *Q*, donc 10001111.

L'organigramme de la figure 5.5 montre le déroulement de cet algorithme.

On peut aussi multiplier des valeurs signées, mais nous ne le détaillerons pas ici. Un algorithme couramment utilisé est l'algorithme de Booth inventé en 1950.

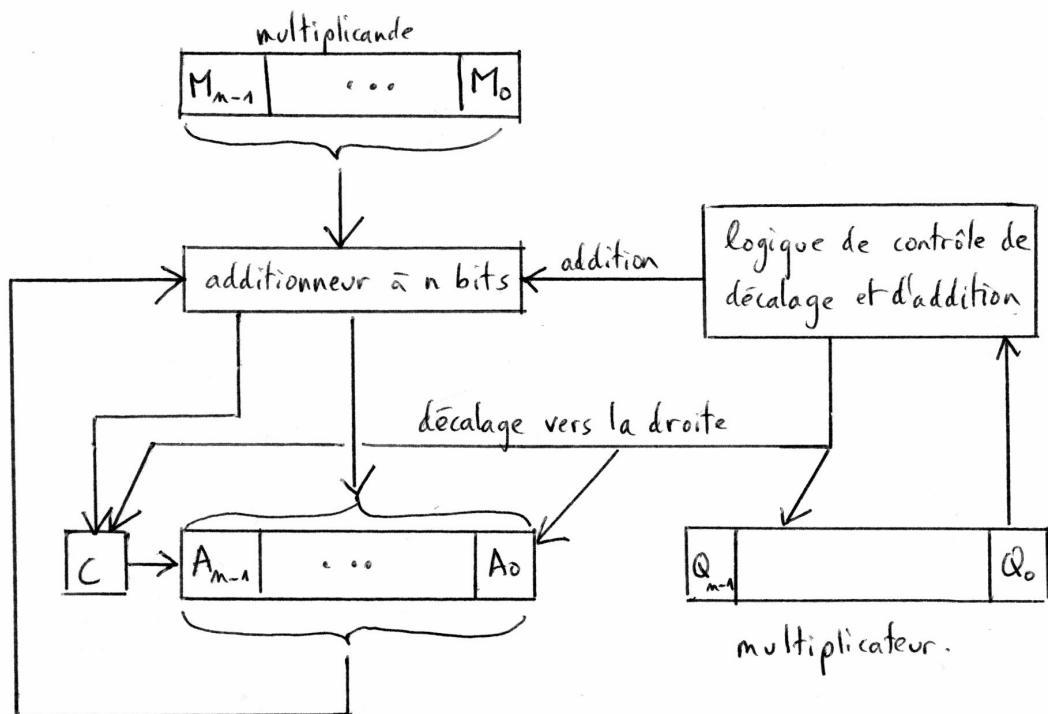


FIGURE 5.4 – Implémentation matérielle pour la multiplication binaire non signée.

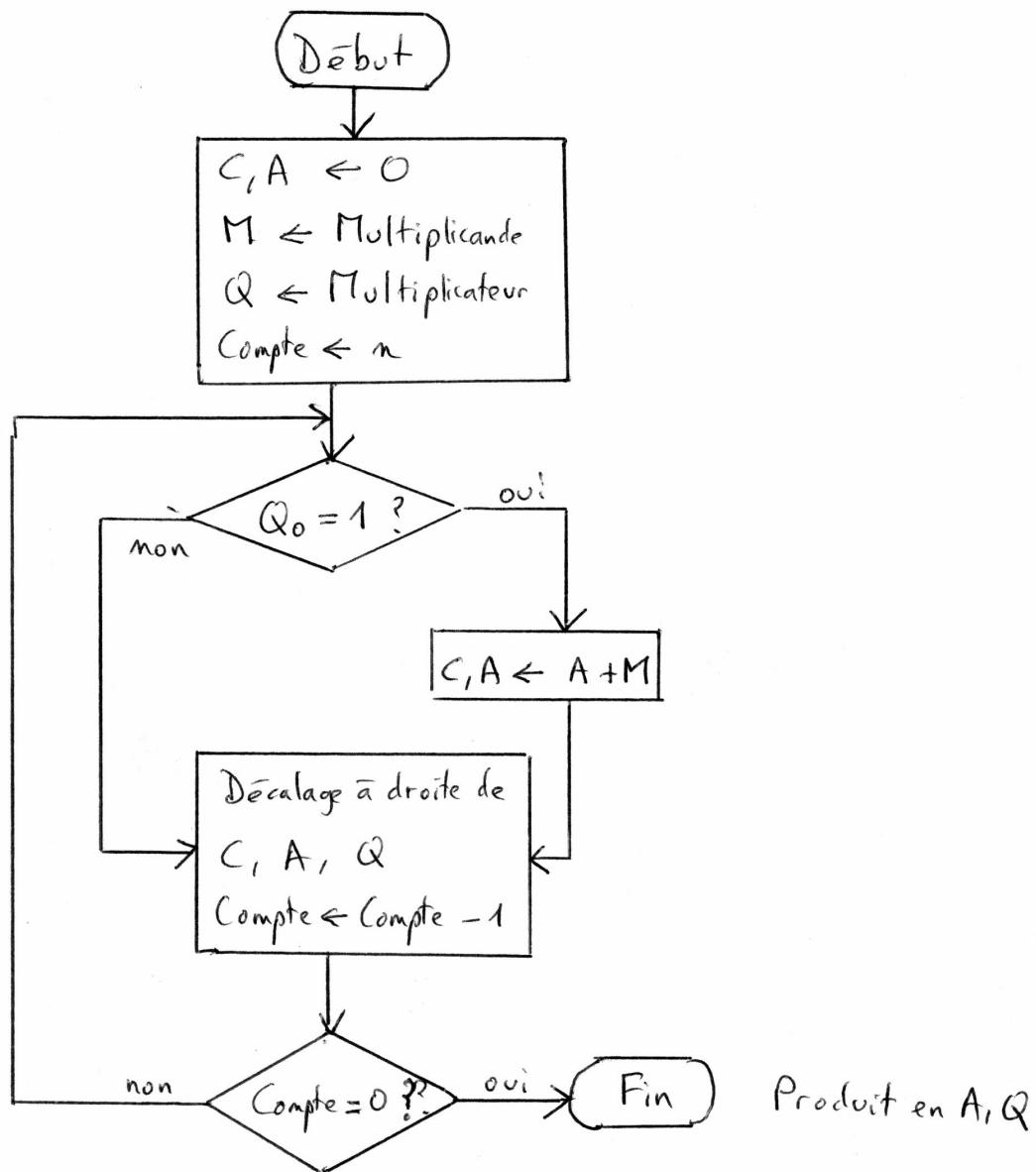


FIGURE 5.5 – Organigramme pour la multiplication binaire non signée.

5.3.4 Division

La division est un peu plus complexe que la multiplication, mais le principe est le même. La figure 5.6 donne un organigramme pour la division binaire non signée. On pourra tester cet algorithme en divisant $Q = 10010011$ par $M = 1011$.

Il y a aussi des algorithmes de division de valeurs signées.

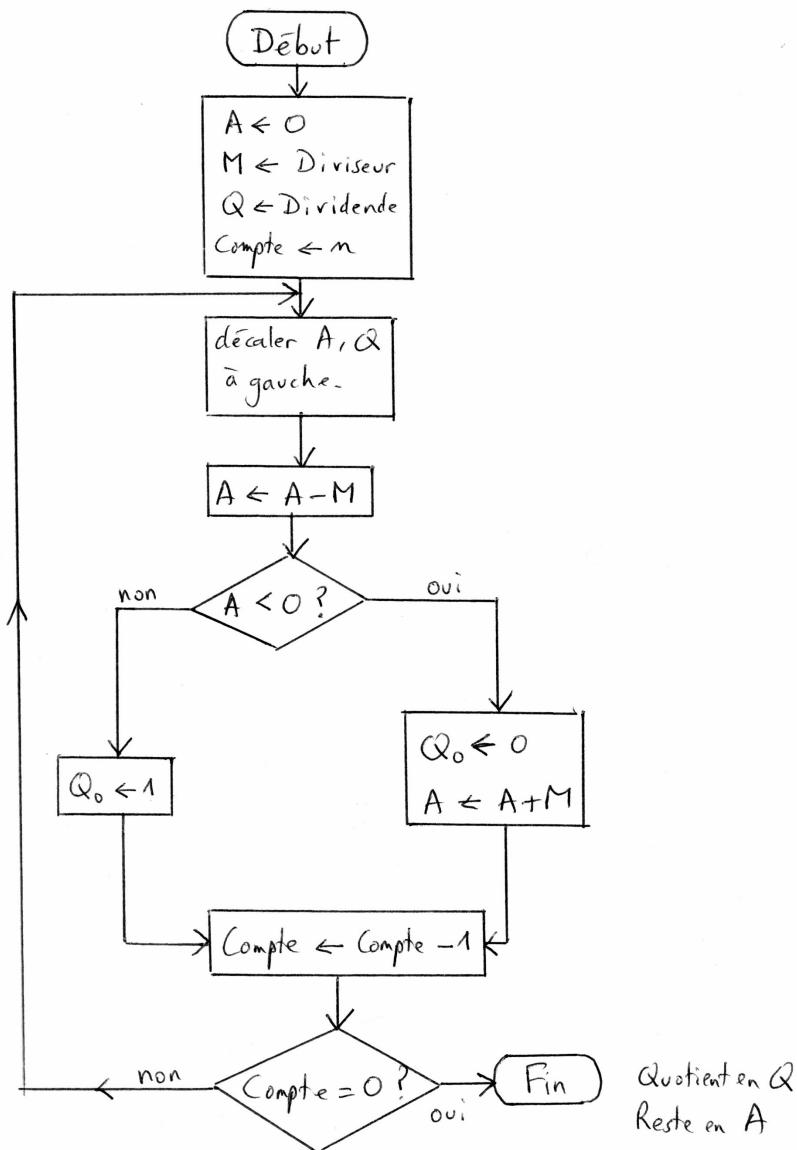


FIGURE 5.6 – Organigramme pour la division binaire non signée.

5.4 Représentation en virgule flottante

Les représentations vues ci-dessus sont aussi appelées à « virgule fixe » (ou *fixed-point*). Dans ces représentations, un bit a toujours le même poids. Le bit de droite compte toujours pour 1, le second pour 2, le troisième pour 4, et ainsi de suite. Cette représentation peut aussi servir à représenter des fractions, on pourrait par exemple avoir une variable pour la partie entière et une autre pour la partie décimale. Ainsi, si l'on voulait représenter la valeur binaire 01100011.10111011, on pourrait placer 01100011 dans une variable 8 bits et 10111011 dans une autre.

Mais cette approche est limitée, car on ne peut pas représenter de très grands nombres, ni de très petites fractions.

Avec les nombres décimaux, on peut contourner ce problème en utilisant la notation dite « scientifique ». On peut représenter 976000000000000 par 976×10^{14} et 0.00000000000976 par 976×10^{-14} . On peut faire la même chose avec les nombres binaires. Un nombre peut être représenté sous la forme

$$\pm S \times B^{\pm E}$$

Ce nombre peut être stocké avec trois champs :

- le signe : plus ou moins
- la mantisse (ou significande) S
- l'exposant E (positif ou négatif)

B est la base et vaut 2 pour le binaire. On n'aura pas besoin de stocker cette base, car elle sera implicite. La figure 5.7 montre un format de nombre réel sur 32 bits. Il y a un bit pour le signe de la mantisse, huit bits pour l'exposant et les 23 bits restants pour la mantisse. Le bit de signe est 0 pour les valeurs positives et 1 pour celles négatives. L'exposant doit pouvoir être positif ou négatif et est stocké selon une représentation appelée « biaisée ». On peut aussi parler de représentation avec excédent. En l'occurrence, une valeur est stockée sur 8 bits, mais pour obtenir la vraie valeur de E , il faut

en retrancher une constante. En général, avec n bits, on retranche $2^{n-1} - 1$. Donc avec 8 bits, on retranche 127. Avec cette représentation, les exposants vont de -127 à 128. Cette représentation diffère de la représentation en complément à deux. Le tableau 5.1 de la page 110 permet de comparer la représentation biaisée sur 4 bits avec la représentation en complément à deux sur 4 bits.

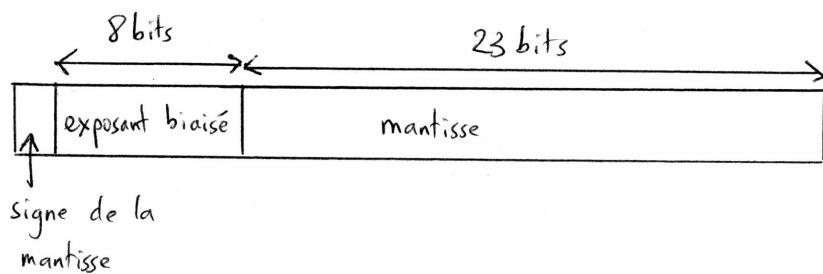


FIGURE 5.7 – Format flottant 32-bits.

Un nombre réel peut être représenté de beaucoup de manières différentes. Les représentations suivantes sont par exemple équivalentes :

$$\begin{aligned} & 0.110 \times 2^5 \\ & 110 \times 2^2 \\ & 0.0110 \times 2^6 \end{aligned}$$

Pour simplifier les opérations, on requiert typiquement que les valeurs soient normalisées. Une valeur normalisée est telle que le bit le plus significatif de la mantisse est non nul. Par ailleurs, on convient qu'il y a exactement un bit à gauche du point décimal. Ceci réduit les trois représentations précédentes à une seule :

$$1.10 \times 2^4$$

La définition précédente ne convient évidemment pas pour la valeur 0, mais celle-ci sera traitée séparément.

Maintenant, comme le bit le plus significatif de la mantisse vaut toujours 1 (sauf si la valeur est nulle), il est inutile de stocker ce bit. Par conséquent, les 23 bits du format de la figure 5.7 servent en fait à représenter la mantisse sur 24 bits. Les valeurs représentables vont de 1 (inclus) à 2 (exclus).

Voici quelques valeurs typiques utilisant ce format :

$$\begin{array}{llll} 1.1010001 \times 2^{10100} & = & 0\ 10010011\ 10100010000000000000000000000000 & = 1.6328125 \times 2^{20} \\ -1.1010001 \times 2^{10100} & = & 1\ 10010011\ 10100010000000000000000000000000 & = -1.6328125 \times 2^{20} \\ 1.1010001 \times 2^{-10100} & = & 0\ 01101011\ 10100010000000000000000000000000 & = 1.6328125 \times 2^{-20} \\ -1.1010001 \times 2^{-10100} & = & 1\ 01101011\ 10100010000000000000000000000000 & = -1.6328125 \times 2^{-20} \end{array}$$

La figure 5.8 montre quelles valeurs entières sont exprimables sur 32 bits en complément à deux. On peut aller de -2^{31} à $2^{31} - 1$. Les valeurs en dehors de cet intervalle ne sont pas exprimables.

La figure 5.9 montre quelles valeurs réelles sont exprimables avec le format vu ci-dessus. Comme le plus petit exposant est -127 , la plus petite valeur représentable non nulle est 2^{-127} . La plus grande est légèrement inférieure à 2^{129} . On a donc un intervalle beaucoup plus grand qu'avec les entiers, mais il y a toujours des zones inaccessibles.

Un dépassement par le haut (*overflow*) correspond à un résultat qui dépasse la plus grande valeur. Un dépassement par le bas (*underflow*) correspond à une valeur non nulle trop petite. Les *underflow* sont en général moins graves que les *overflow*, car on peut souvent remplacer les plus petites valeurs par 0 sans risque.

Il y a par ailleurs normalement une valeur particulière pour représenter 0, par exemple quand tous les bits sont à 0.

Il est important de réaliser que nous ne représentons pas davantage de valeurs dans le format flottant que dans le format entier. Il y a un maximum de 2^{32} valeurs représentables sur 32 bits, que ce soit avec des entiers ou des réels.

D'autre part, la densité des valeurs réelles varie et est de plus en plus faible à mesure que l'on s'éloigne de 0. Et il y a un compromis

entre la précision des valeurs (le nombre de bits de la mantisse) et l'intervalle accessible. Si l'exposant peut devenir plus grand, on va perdre en précision. Si on veut une précision plus grande, on devra accepter une limite plus grande au niveau de l'exposant.

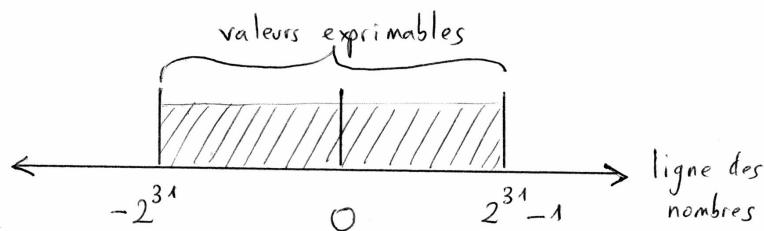


FIGURE 5.8 – Entiers exprimables en 32 bits.

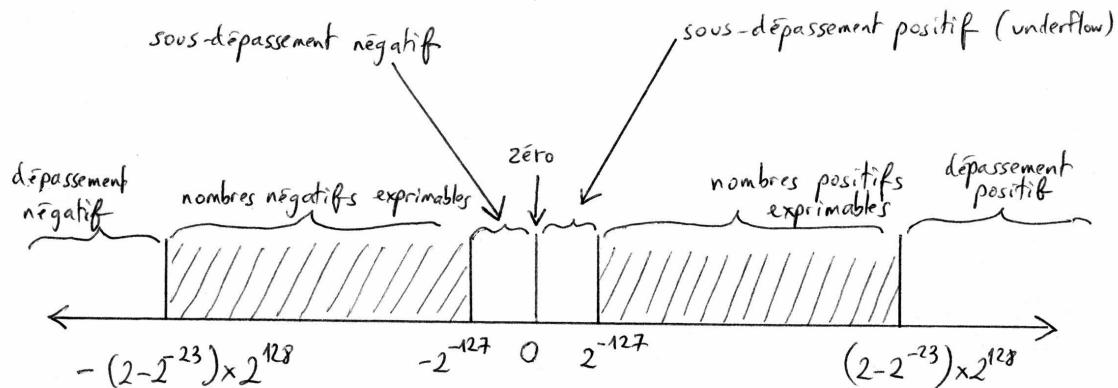


FIGURE 5.9 – Réels exprimables en 32 bits.

5.5 La norme IEEE754

La norme IEEE754 pour les nombres réels a été adoptée en 1985 pour faciliter la portabilité des programmes d'un processeur à un autre. Cette norme est adoptée presque universellement.

Il y a dans cette norme trois formats binaires de base avec 32, 64 et 128 bits appelés binary32, binary64 et binary128 (figure 5.10). Les exposants y sont sur 8, 11 et 15 bits.

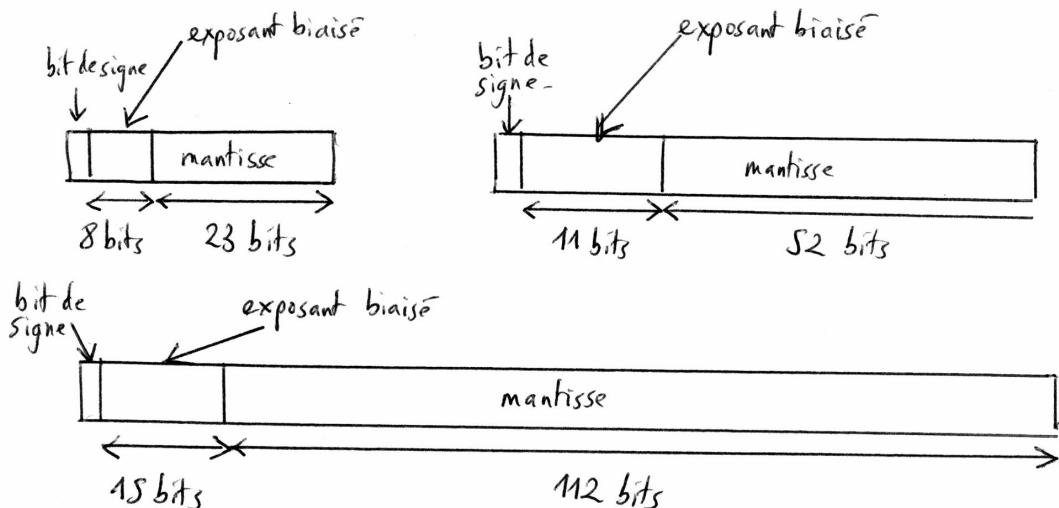


FIGURE 5.10 – Formats IEEE754.

La norme spécifie d'autres formats qui ne sont pas détaillés ici, par exemple des formats décimaux sur 64 et 128 bits.

La norme définit notamment une représentation pour l'infini, positif ou négatif, et des valeurs anormales (NaN, *Not a Number*).

Les intervalles (plus petite valeur positive, plus grande valeur positive) sont approximativement $(10^{-38}, 10^{+38})$ pour 32 bits, $(10^{-308}, 10^{+308})$ pour 64 bits et $(10^{-4932}, 10^{+4932})$ pour 128 bits

Pour les formats binaires, il y a deux valeurs nulles ($+0$ et -0) : tous les bits sont à 0, sauf éventuellement le bit de signe. Une va-

leur où tous les bits de l'exposant sont à 1 et ceux de la mantisse à 0 représente l'infini, positif ou négatif suivant le bit de signe. Une valeur où tous les bits de l'exposant sont à 1 mais où la mantisse n'est pas nulle représente un NaN.

On notera que les bornes ne sont pas exactement les mêmes que celles de la figure 5.9. En effet, dans la norme IEEE 754, l'exposant biaisé ne peut avoir tous ses bits à 1 pour une valeur normale (différente de l'infini et de NaN). La plus grande valeur réelle sur 32 bits est donc un peu inférieure à 2^{128} et non 2^{129} comme dans le cas général.

Par ailleurs, le cas où tous les bits de l'exposant valent 0 et la mantisse est non nulle est aussi un cas particulier. Cette configuration correspond à des nombres « sous-normaux » (*subnormal*). L'idée est d'utiliser l'exposant 0 pour représenter des valeurs beaucoup plus petites que ce qui est normalement défini par les règles précédentes. Si l'exposant est 0 et que la mantisse n'est pas nulle, il n'y a plus de bit implicite 1 à gauche du point décimal, mais un bit à 0. Et dans ce cas, l'exposant réel vaut -126 (32 bits) ou -1022 (64 bits).

5.6 Arithmétique en virgule flottante

Pour l'addition et la soustraction, il faut ramener les deux opérandes à une même valeur d'exposant. La multiplication et la division sont plus simples.

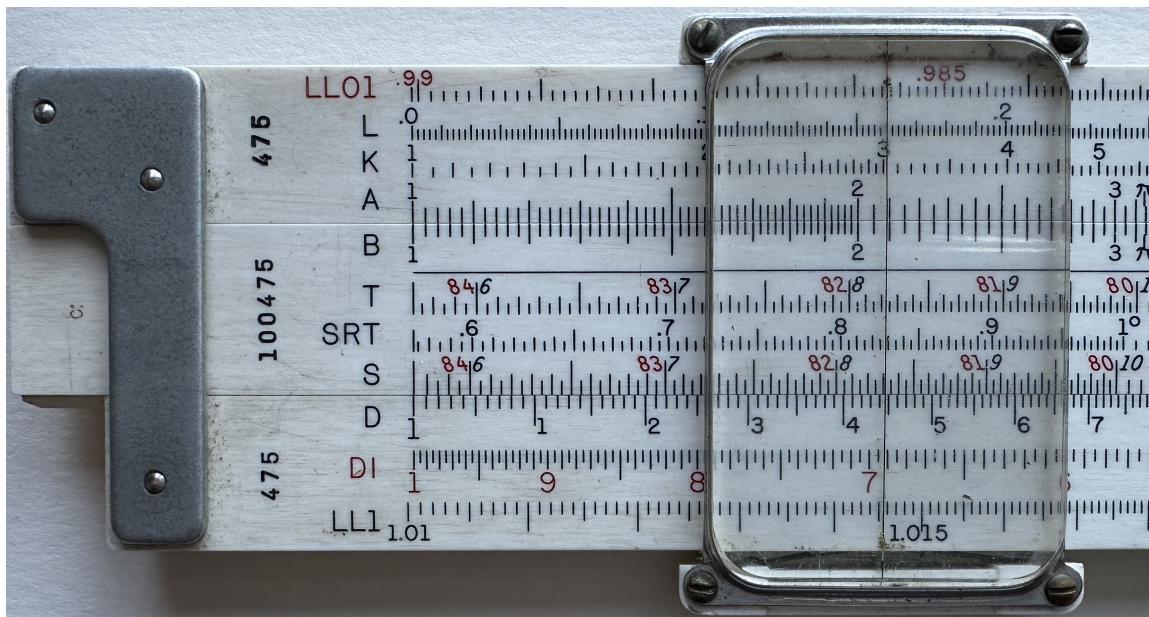


FIGURE 5.11 – Une règle à calcul et le détail du calcul de $\sqrt{2.1}$. L'échelle A indique 2.1 (ou un tout petit peu moins) et l'échelle D donne la racine carrée 1.44.... (source : Wikipédia)

Soit par exemple les deux valeurs suivantes exprimées en base 10 : $X = 0.3 \times 10^2$ et $Y = 0.2 \times 10^3$.

On peut écrire

$$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$$

Une opération en virgule flottante peut produire des dépassements, soit par le haut (*overflow*), soit par le bas (*underflow*). Lors de l'alignement des mantisses, il peut aussi y avoir des dépassements. Enfin, l'addition des mantisses peut produire une retenue qui nécessite un réalignement.

5.6.1 Addition et soustraction

Il y a quatre phases de base dans l'algorithme d'addition et de soustraction :

- voir s'il y a des valeurs nulles;
- aligner les mantisses;
- ajouter ou soustraire les mantisses;
- normaliser le résultat.

La figure 5.12 montre un organigramme typique pour l'addition et la soustraction. Pour les opérations elles-mêmes, les deux opérandes doivent être transférés dans des registres qui seront utilisés par l'UAL. Si le format des flottants inclut un bit implicite, l'opération doit rendre ce bit explicite.

Pour la première phase, s'il s'agit d'une soustraction, on change le signe de Y . Si l'une des valeurs est nulle, le résultat est égal à l'autre valeur.

Dans la seconde phase, on fait un alignement des valeurs. Cet alignement peut conduire à ce que l'une des valeurs devienne nulle, auquel cas le résultat est égal à l'autre valeur.

Une fois les mantisses alignées, elles sont additionnées, puis le résultat est normalisé.

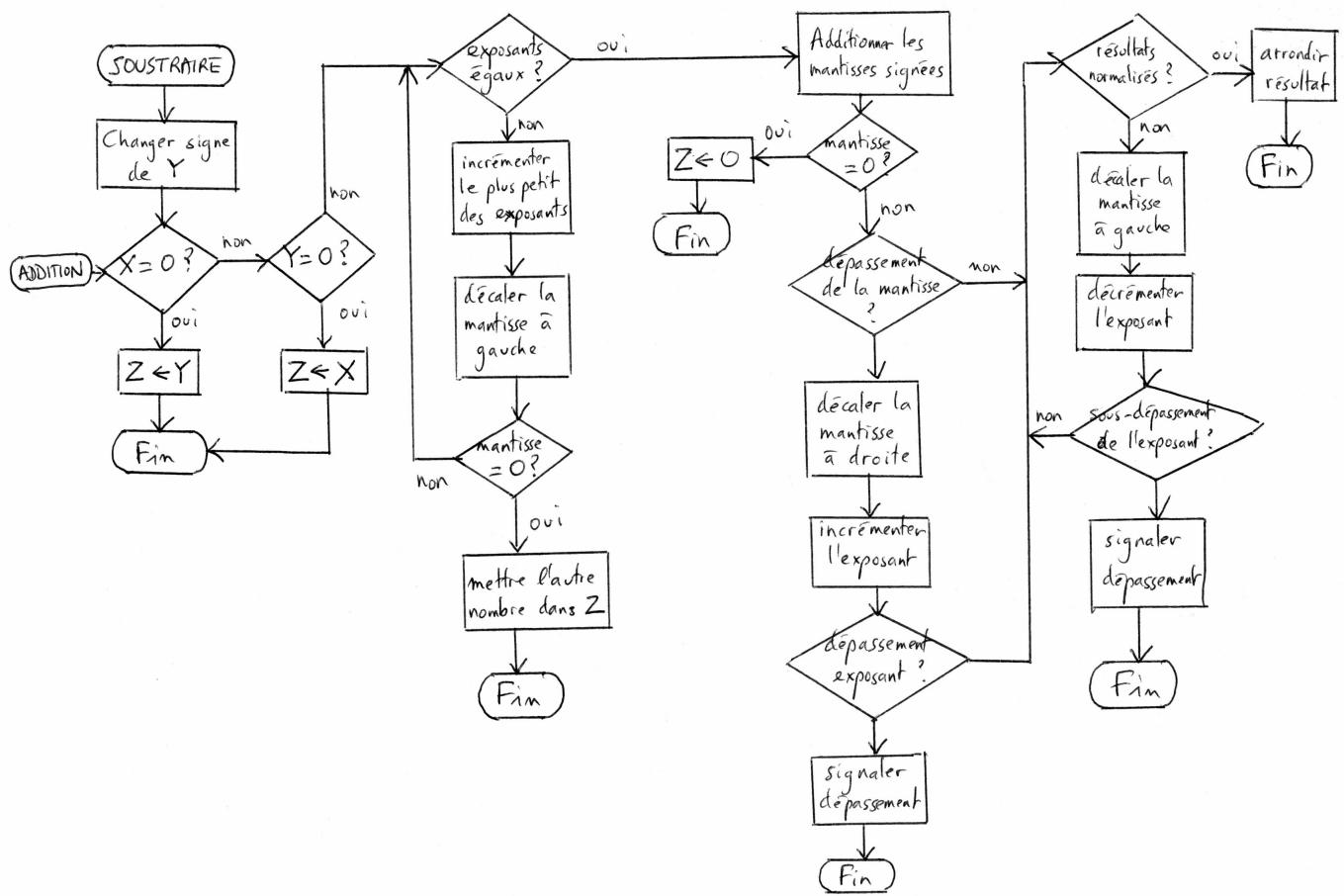


FIGURE 5.12 – Addition et soustraction ($Z \leftarrow X \pm Y$) en virgule flottante.

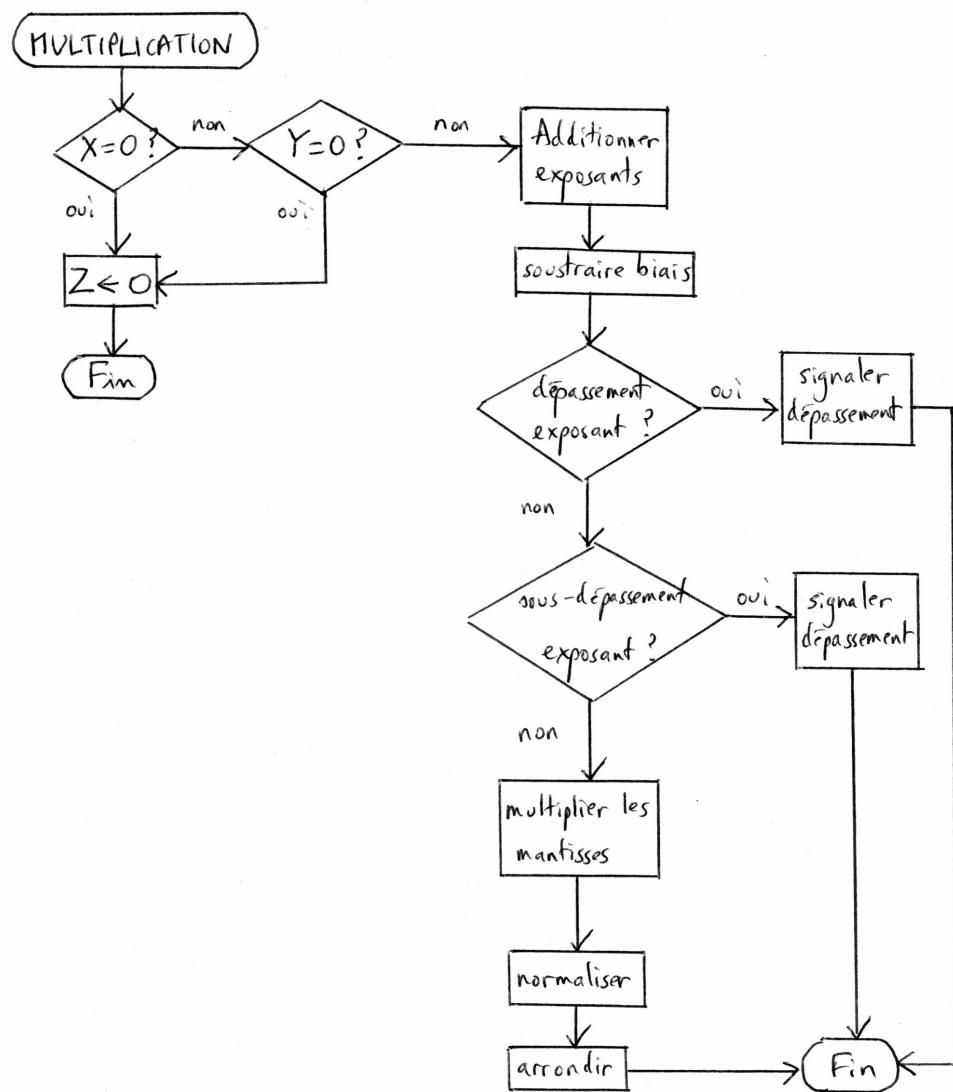
5.6.2 Multiplication et division

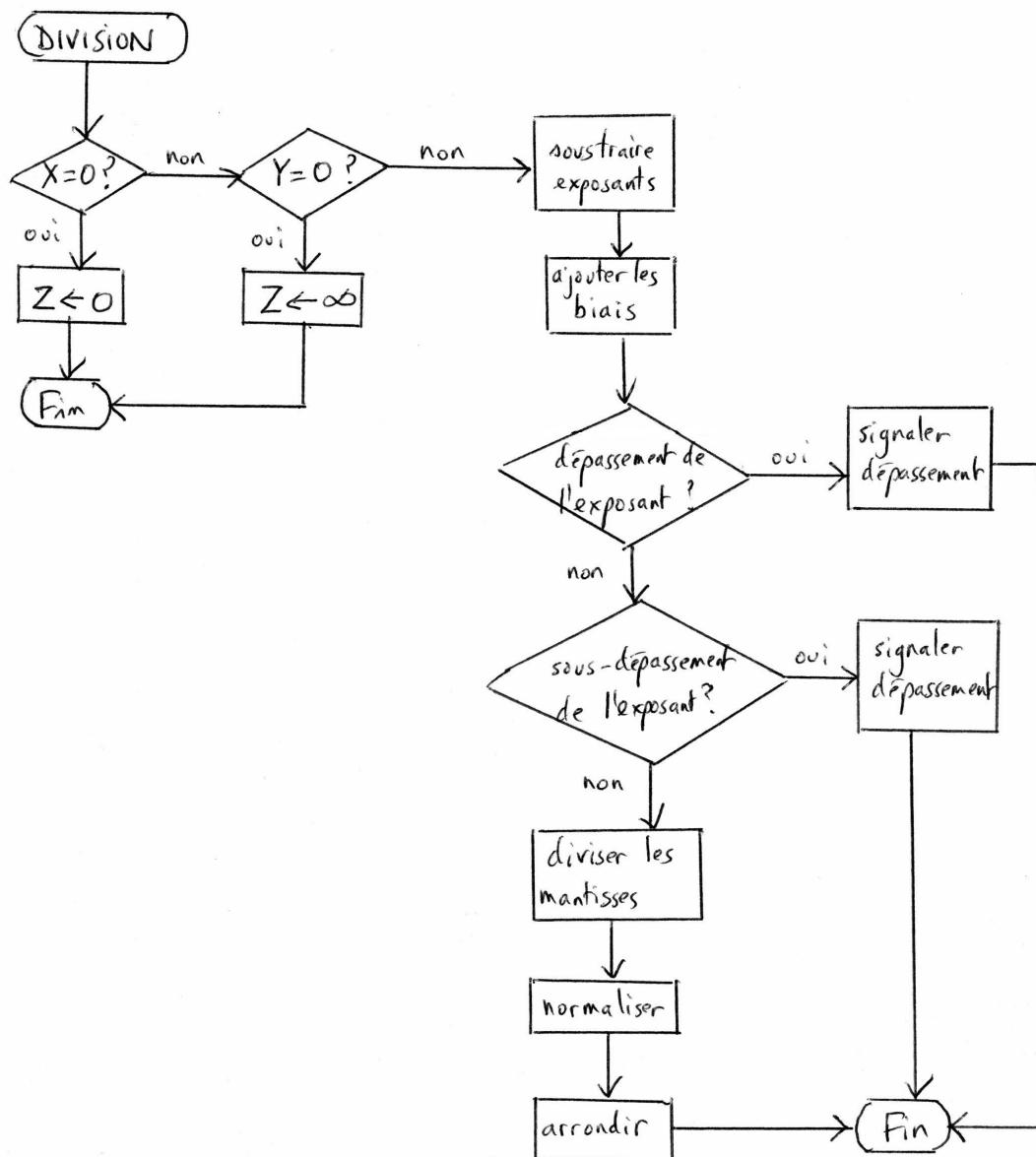
La multiplication et la division sont beaucoup plus simples que l'addition et la soustraction, parce que l'on n'a pas besoin d'aligner les valeurs.

L'organigramme de la multiplication est donné dans la figure 5.13. Tout d'abord, si l'un des opérandes vaut 0, le résultat est 0. Ensuite, on ajoute les exposants. Si les exposants sont stockés avec un biais (excédent), celui-ci se trouve doublé et doit donc être retranché.

Si un sur- (*overflow*) ou sous-dépassement (*underflow*) se produit, celui-ci est signalé et le calcul s'achève. Sinon, on multiplie les mantisses, on normalise et on arrondit. On notera que la normalisation peut conduire à un dépassement d'exposant.

Enfin, l'organigramme de la division est donné dans la figure 5.14. On peut voir qu'il est presque identique à celui de la multiplication.

FIGURE 5.13 – Multiplication ($Z \leftarrow X \times Y$) en virgule flottante.

FIGURE 5.14 – Division ($Z \leftarrow X/Y$) en virgule flottante.

5.6.3 Arrondis

La norme IEEE754 propose plusieurs modes d'arrondis que nous ne décrivons pas ici.

5.6.4 Gestion de l'infini

Une valeur IEEE754 peut être infinie. Cette valeur peut intervenir dans un calcul et certains calculs peuvent se faire, par exemple

$$\begin{array}{ll}
 5 + (+\infty) = +\infty & 5 \div (+\infty) = +0 \\
 5 - (+\infty) = -\infty & (+\infty) + (+\infty) = +\infty \\
 5 + (-\infty) = -\infty & (-\infty) + (-\infty) = -\infty \\
 5 - (-\infty) = +\infty & (-\infty) - (+\infty) = -\infty \\
 5 \times (+\infty) = +\infty & (+\infty) - (-\infty) = +\infty
 \end{array}$$

Le petit programme C suivant illustre quelques manipulations de valeurs infinies.

```
#include <stdio.h>

int main() {
    float a,b,c;
    a=1/0.0;b=-1/0.0;
    printf("a=%f, b=%f\n",a,b);
    c=a-b;
    printf("c=%f\n",c);
    c=b-a;
    printf("c=%f\n",c);
    c=a+b;
    printf("c=%f\n",c);
}
```

5.6.5 Valeurs anormales

La valeur spéciale NaN (*Not a Number*) permet de signaler des erreurs, par exemple si l'on essaie de calculer $-\infty + \infty$. Il faut cependant noter qu'il y a deux sortes de valeurs NaN, celles de « signalement » (*signaling NaN*) et celles « tranquilles » (*quiet NaN*). Nous ne les décrivons pas ici.

5.6.6 Valeurs sous-normales

Les valeurs sous-normales (*subnormal*) sont incluses dans la norme IEEE754 afin de gérer le cas de sous-dépassements d'exposants. Quand un exposant devient trop petit, le résultat est sous-normalisé en décalant la mantisse à droite et en incrémentant l'exposant pour chaque décalage, jusqu'à ce que l'exposant soit dans l'intervalle admissible.

La figure 5.15 illustre l'effet de l'inclusion de nombres sous-normaux. Dans la partie supérieure de la figure, on représente les valeurs des réels ordinaires sur 32 bits, dont la plus petite valeur positive est 2^{-126} (exposant biaisé à 1 et mantisse nulle). Les écarts entre valeurs successives (dont le nombre est simplifié) croissent vers la droite, à chaque changement d'exposant.

Dans la partie inférieure de la figure, on a ajouté les nombres sous-normaux qui occupent l'espace de 0 à 2^{-126} , mais de manière uniforme, le pas d'une valeur à la suivante étant de $2^{-23} \times 2^{-126} = 2^{-149}$. C'est le même pas qu'au début de l'intervalle des nombres normaux.

Le programme C suivant montre que l'on peut stocker dans un flottant 32 bits des valeurs sous-normales et les manipuler.

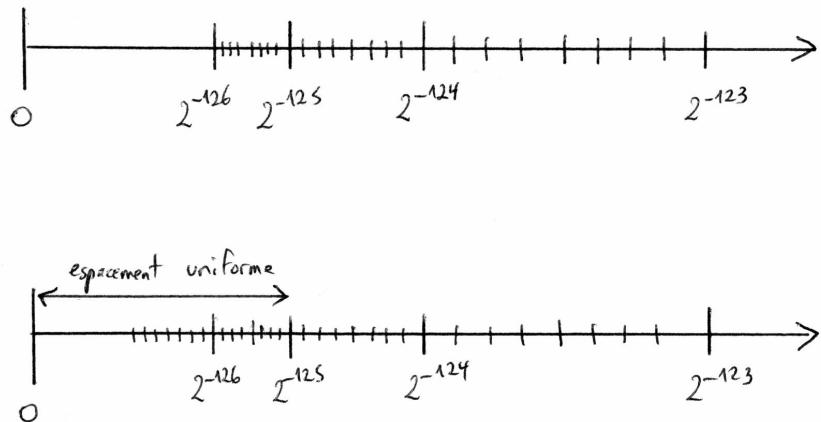


FIGURE 5.15 – Nombres IEEE754 sous-normaux.

```

int main() {
    float a,b,c;
    printf("sizeof(float)=%ld\n", sizeof(float)); // affiche 4
    // 2^-126=.0000000000000000000000000000000011754943508...
    a=.0000000000000000000000000000000000000000000000000000000000000002; // valeur normale
    printf("a=%2.40f\n",a);
    a=.0000000000000000000000000000000000000000000000000000000000000001;
    printf("a=%2.40f (proche de 2^-126)\n",a);
    // si on veut vraiment stocker 2^-126 :
    a=1;a=a/1024;a=a/1024;a=a/1024;a=a/1024;a=a/1024;
    a=a/1024;a=a/1024;a=a/1024;a=a/1024;a=a/1024;a=a/1024;a=a/64;
    printf("a=%2.40f (exactement 2^-126)\n",a);
    a=a*1024;a=a*1024;a=a*1024;a=a*1024;a=a*1024;
    a=a*1024;a=a*1024;a=a*1024;a=a*1024;a=a*1024;a=a*64;
    printf("*a=%2.40f (exactement 1)\n",a);
    // 2^-148=.0000000000000000000000000000000000000000000000000000000000000002802...
    a=.00000000000000000000000000000000000000000000000000000000000000028; // un peu moins q. 2^-148
    printf("a=%2.50f\n",a);
    // 2^-149=.0000000000000000000000000000000000000000000000000000000000000001401...
    a=.00000000000000000000000000000000000000000000000000000000000000014; // un peu moins q. 2^-149
    printf("a=%2.50f\n",a);
    a=.00000000000000000000000000000000000000000000000000000000000000008; // un peu moins q. 2^-149
    // la valeur précédente est arrondie à 2^-149
    printf("a=%2.50f\n",a);
    // on remultiplie jusqu'à 1 :
    a=a*1024;a=a*1024;a=a*1024;a=a*1024;a=a*1024;a=a*1024;
    a=a*1024;a=a*1024;a=a*1024;a=a*1024;a=a*1024;a=a*512;
    printf("a=%2.50f (exactement 1)\n",a);
}

```

5.7 Histoire des algorithmes

Un algorithme est un moyen précis d'obtenir un résultat à partir de certaines données. Les premiers algorithmes étaient bien sûr des recettes diverses et variées.

En mathématiques, on peut citer le travail d'Euclide qui a organisé les mathématiques de son temps (*Éléments d'Euclide*). On doit par exemple à Euclide (vers 300 av. J.-C.) l'algorithme de calcul du PGCD de deux nombres :

```
Entrée = Deux entiers a et b
Sortie = Le PGCD de a et b
fonction euclide(a,b)
    tant que a <> b
        si a > b alors
            a := a - b
        sinon
            b := b - a
    renvoyer a
```

À la même époque, à Babylone, des algorithmes sont utilisés pour prédire la position de la lune et des planètes. Ces algorithmes deviendront de plus en plus sophistiqués et serviront notamment à produire des tables astronomiques.

Le calendrier a aussi joué un grand rôle dans le développement d'algorithmes, puisque le calendrier devait être adapté aux saisons.

Bien plus tard, lorsque les premières machines à calculer ont été construites, il a fallu imaginer des moyens de faire les calculs, que ce soient de simples additions ou des calculs plus sophistiqués. Et lorsque la programmation a été introduite, par Jacquard, Babbage, etc., il a fallu imaginer la codification de ces algorithmes.

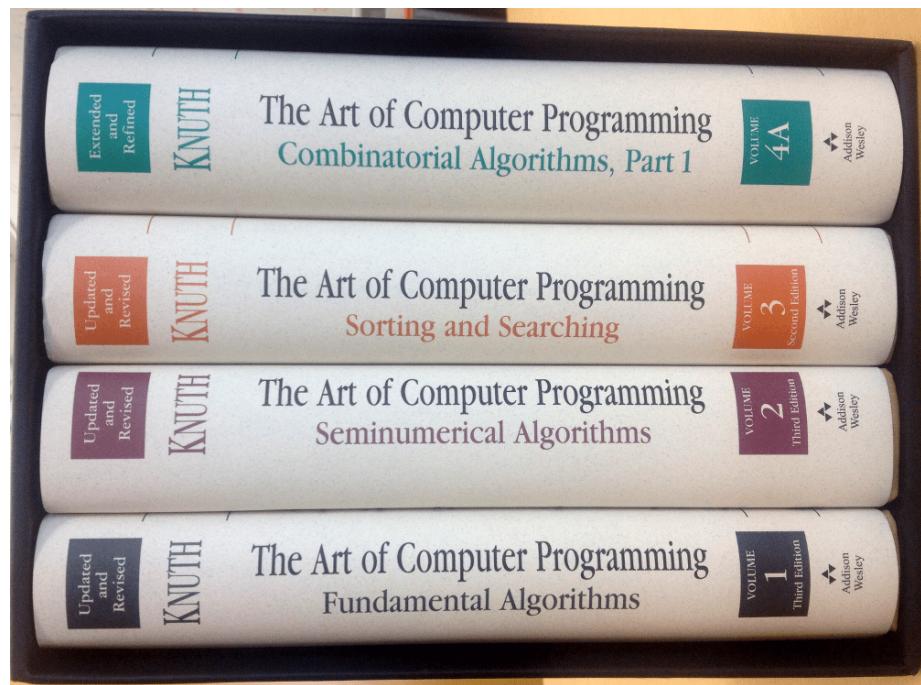


FIGURE 5.16 – Les premiers volumes de *The Art of Computer Programming* de Donald Knuth.

5.8 Et encore un peu d'histoire



FIGURE 5.17 – La machine d'Anticythère. (1^{er} siècle av. J.-C.) (pas celle d'Indiana Jones 2023 !) (source : Wikipédia)



FIGURE 5.18 – La machine à calculer de Blaise Pascal (1623-1662) (imaginée en 1642). (source : Wikipédia)



FIGURE 5.19 – La machine à calculer de Gottfried Wilhelm Leibniz (1646-1716), construite vers 1700.



FIGURE 5.20 – Un arithmomètre Thomas, vers 1875.



FIGURE 5.21 – Une machine Odhner, inventée en 1873.

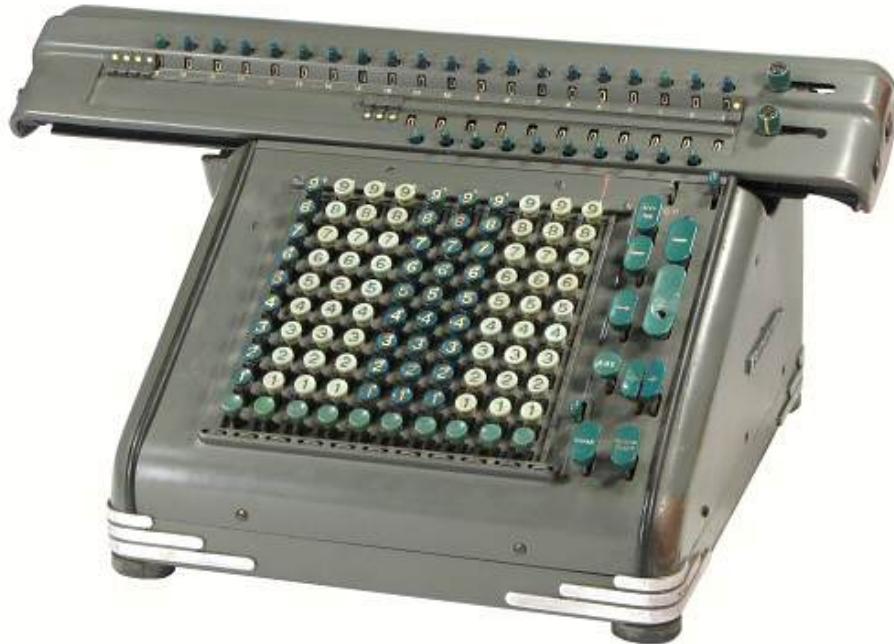


FIGURE 5.22 – Une machine Friden, construite de 1935 à 1949.



FIGURE 5.23 – Une calculatrice de poche Curta.

5.9 Exercices

1. On considère dans cet exercice la notation en complément à 2 sur un octet.
 - (a) Donner la plage de représentation possible des entiers sous la forme d'un intervalle.
 - (b) Dans cette notation, trouver la représentation en binaire des nombres suivants : 115 et -115.
 - (c) Effectuer en binaire, en posant l'opération en ligne l'opération $115 + (-115)$
Et montrer que le résultat en binaire correspond bien au résultat attendu en décimal.
2. Les notions de complément à 1 et à 2 sont des cas particuliers de complément à $b - 1$ et à b pour une base b . Par exemple, si $b = 10$, et sur 4 chiffres, la représentation en complément à 10 de -526_{10} est obtenue en prenant le complément à 9 de 526 et en ajoutant 1, donc en obtenant 9473 puis 9474.
Maintenant, considérez $b = 3$ et exprimez sur 6 chiffres ternaires les valeurs en complément à 3 de $+11011$, -10222 , $+2120$, -1212 , $+10$ et -201 , valeurs ici exprimées en base 3 avec signe. Par exemple, l'avant dernière valeur est en fait égale à 3 (décimal) et sa représentation sur 6 chiffres est 000010. Trouvez les autres.
3. Donnez les tailles (en octets) et valeurs maximales des types C suivants : `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`.
4. Quelles opérations C ci-dessous comportent des erreurs (0x signifie qu'on donne une valeur en hexadécimal)?

```
unsigned char c = 0x1A;  
c = 0xEF;  
unsigned int i = 0x0000FFFF;  
i = 0x2;
```

```
unsigned long l = 0xABCDEF0011121314;
l = 0xFFFFFFFFFFFFFFFFF;
```

5. Quelles opérations ci-dessous vont poser un problème ?

```
unsigned char c, d;
c = 0xF0;
d = 0x0F
c = c + d;
c = 127;
d = 128;
c = c + d;
unsigned long l, g;
l = 0xFFFFFFFF;
g = 2;
l = l + g;
l = 0xF000FFFF;
g = 1;
l = l + g;
```

6. Si une variable de type unsigned short a la valeur décimale 2, que vaudrait sa valeur après un décalage de 3 bits vers la gauche ?

```
unsigned short v = 2;
v = v << 3;
```

7. D'une manière générale, que fait un décalage à gauche de 1 bit ? Et pour un décalage à droite de 1 bit ?
8. Quelles sont les valeurs de la variable c à chaque étape ci-dessous ?
(&, |, ^, ~ sont les opérateurs bit à bit du ET, OU, XOR et du complément)

```
unsigned char c = 0x0F;
c = c & 0x00;
c = 255;
```

```
c = c | 0x7F;
c = 100;
c = c ^ 0x25;
c = c ^ 0x25;
c = 128;
c = ~c;
```

9. même question, mais avec les opérateurs booléens sur la valeur de la variable i ? (`&&`, `||`, `!` pour le ET, OU et NON)

```
unsigned int i = 10;
i = i && 1;
i = 4;
i = !i;
i = 3;
i = i || 3;
i = 4;
i = i && 0;
```

10. expliquez comment on calcule le produit de deux nombres avec une règle à calcul (cf. figure 5.11 page 128); donnez un exemple en l'illustrant;
11. calculez $381 + 214$, $169 + 245$, $130 - 78$, $897 - 358$ en binaire uniquement;
12. déterminez le complément à deux de 79 sur 8 et 10 bits;
13. déterminez le complément à deux de 196 sur 12 bits;
14. déterminez le complément à deux de 658 sur 16 bits;
15. Calculez directement en hexadécimal les soustractions ci-dessous, pour des mots stockés en machine sur 8 bits.

$$\begin{array}{r} 2B \\ - 13 \\ \hline 2B \\ - 3C \\ \hline 23 \\ - 1B \\ \hline \end{array}$$

16. codez les valeurs suivantes avec un excédent de 127 : 10, 28, 40, 64, 156, 244.
17. quel est le plus petit nombre de bits nécessaire pour représenter tous les entiers de 180 à 307 ?
18. écrivez les valeurs négatives suivantes en complément à deux sur 8 et 16 bits, en binaire et en hexadécimal : -12, -68, -128 ;
19. trouvez les nombres décimaux correspondant aux valeurs binaires ou hexadécimales signées en complément à deux : 1000 (4 bits), 1111 (4 bits), 11000110 (8 bits), 10111110 (8 bits), A2, BE, 62AF, CCCC, 3333
20. convertissez les nombres décimaux suivants au format IEEE 754 32 bits : 1.0, -0.1, 2016.0, 0.00390625, -3125.3125, 0.33, -0.67, 3.14
21. peut-on coder la valeur 0.6 de manière exacte en IEEE754 ?
22. quelle est l'équivalent décimal de 0100 0011 0101 0100 0000 0000 0000 0000 en utilisant le format IEEE 754 ?
23. quelle est l'équivalent décimal de 0111 1111 1011 0100 0000 0000 0000 0000 ?
24. convertissez les nombres hexadécimaux en décimal en utilisant le format IEEE 754 : 40000000, bf800000, 3d800000, c1804000, 42c81000, 3f99999a, 42f6e666, c25948b4
25. on pourra comparer les résultats précédents (faits à la main) avec ce que donne un convertisseur en ligne, comme par exemple
<https://www.h-schmidt.net/FloatConverter/IEEE754.html>
26. Soit x un nombre réel dont la représentation IEEE754 est donnée sur 32 bits par 41950000 (en hexadécimal). Donnez en hexadécimal les représentations IEEE754 de $-2x$ et de $x/2$. Donnez aussi l'écriture décimale de x .
27. Recherchez, sans la dépasser, la représentation en simple précision (32 bits) la plus proche de la valeur décimale 0.8. Qu'observez-vous ?

28. Quel est le codage en simple précision d'une valeur approchée du nombre $58 \cdot 10^{-4}$? (on n'utilisera que deux puissances de 2)
29. qu'affiche le programme C suivant et pourquoi?

```
#include <stdio.h>
int main() {
    float f = 0.26;
    if (f == 0.26) printf("OK\n"); else printf("NO\n");
}
```

Que peut-on faire pour remédier au problème?

30. qu'affiche le programme Java suivant et pourquoi?

```
public class Real {
    public static void main(String args[]) {
        double f = 0.3F;
        System.out.println( f == 0.3 );
    }
}
```

Que peut-on faire pour remédier au problème?

31. Faire fonctionner sur un exemple l'algorithme de Booth de multiplication de deux nombres représentés en complément à deux sur les produits 0111×0011 , 0111×1101 , 1001×0011 et 1001×1101 . On trouvera sur youtube de nombreuses vidéos expliquant cet algorithme. Obtenez-vous les bons résultats?
32. Utiliser l'algorithme de la figure 5.6 pour diviser $Q = 10010011$ par $M = 1011$. Votre résultat est-il correct?
33. Qu'affiche le programme C de la section 5.6.4? Pourquoi?
34. écrivez des programmes C qui vérifient les conversions des exercices précédents; on se servira des éléments donnés en annexe; pour mettre une valeur hexadécimale dans un entier a, on peut écrire :
 $a=0xb0c6a563;$

pour stocker une valeur binaire, on peut ou bien la convertir d'abord en hexadécimal, ou bien la mettre dans une chaîne, puis lire cette chaîne :

```
strcpy(s, "10001110");
a=decimal(s);
```

mais la fonction `decimal` est à écrire par vos soins ;

35. faire la même chose en Java ;
36. faire la même chose en Python.
37. Voici un algorithme de calcul (méthode de Héron) de la racine carrée d'un nombre donné en paramètre :

```
b=paramètre
e=1
a=1
while e> 1e-38 do
    f=(a+b/a)/2
    if f> a then
        e=f-a
    else
        e=a-f
    end if
    a=f
end while
print(a)
```

Implémentez cet algorithme en C en utilisant des `float`. Jusqu'à combien de décimales le programme calcule-t-il $\sqrt{2}$? Quelle est la précision du résultat? Que deviennent ces valeurs si on passe aux `double`?

Comparez la précision avec la fonction `sqrt` de la bibliothèque mathématique. (On fera `man sqrt`, on incluera `math.h` et on compilera avec l'option `-lm`.)

Chapitre 6

Fonctionnement d'un ordinateur

Nous avons vu au chapitre 1 quels étaient les principaux composants d'un ordinateur. Le CPU, notamment, est l'organe chargé de l'exécution des instructions. Nous allons regarder ce composant de plus près ici.

6.1 Instruction de récupération et d'exécution

Le traitement d'une seule instruction est appelé un cycle d'instruction et se décompose en deux étapes : cycle de récupération (*fetch*) et cycle d'exécution (figure 6.1).

Dans un processeur typique, un registre appelé PC (*Program Counter*) contient l'adresse de la prochaine instruction à exécuter. Sauf indication contraire, le processeur incrémente toujours PC après chaque récupération d'instruction.

L'instruction chargée en mémoire contient des bits qui spécifient l'action que le processeur doit exécuter. En général, ces actions se scindent en quatre catégories :

- des données peuvent être transférées du processeur vers la

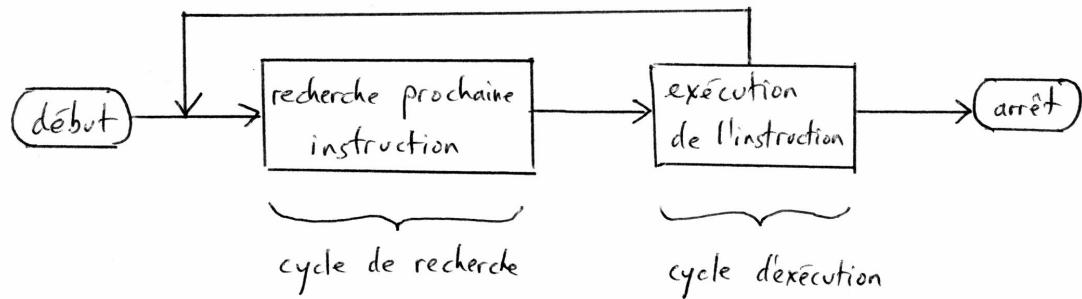


FIGURE 6.1 – Cycle d'instruction de base.

mémoire ou inversement ;

- des données peuvent être transférées entre le processeur et un périphérique ;
- le processeur peut exécuter des opérations arithmétiques ou logiques sur les données ;
- une instruction peut modifier l'ordre d'exécution (sauts).

6.2 Une machine hypothétique

Pour donner une idée de l'exécution des instructions par un processeur, on peut considérer une machine hypothétique. La figure 6.2 montre en haut le format des instructions de cette machine. Elles occupent 16 bits. Le processeur contient un unique registre de données appelé l'accumulateur (AC). Les données (en bas de la figure 6.2) ont aussi 16 bits. Il est donc pratique d'organiser la mémoire en mots de 16 bits.

Dans une instruction, les quatre premiers bits sont le code de l'opération (*opcode* en anglais). Il y a donc $2^4 = 16$ différentes instructions possibles. Les 12 bits restant d'une instruction font référence à une adresse et on peut donc adresser directement 4096 mots de mémoire.

On suppose que l'on a les trois instructions suivantes :

0001 : charger AC depuis la mémoire ;

0010 : stocker AC en mémoire ;

0101 : ajouter à AC depuis la mémoire.

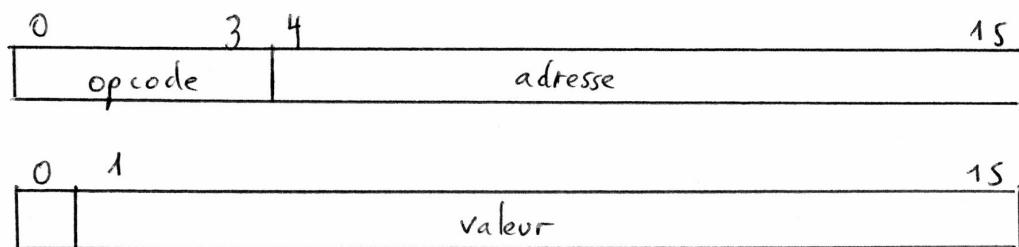


FIGURE 6.2 – En haut, format des instructions d'une machine hypothétique. En bas, format des entiers, aussi sur 16 bits.

La figure 6.3 montre un exemple d'exécution d'un programme avec cette machine hypothétique. IR est le registre contenant l'ins-

truction en cours d'exécution. Il y a six états et les valeurs stockées en mémoire ou dans les registres sont données en hexadécimal.

Dans les processeurs réels, il est possible qu'il y ait plus d'une référence à la mémoire.

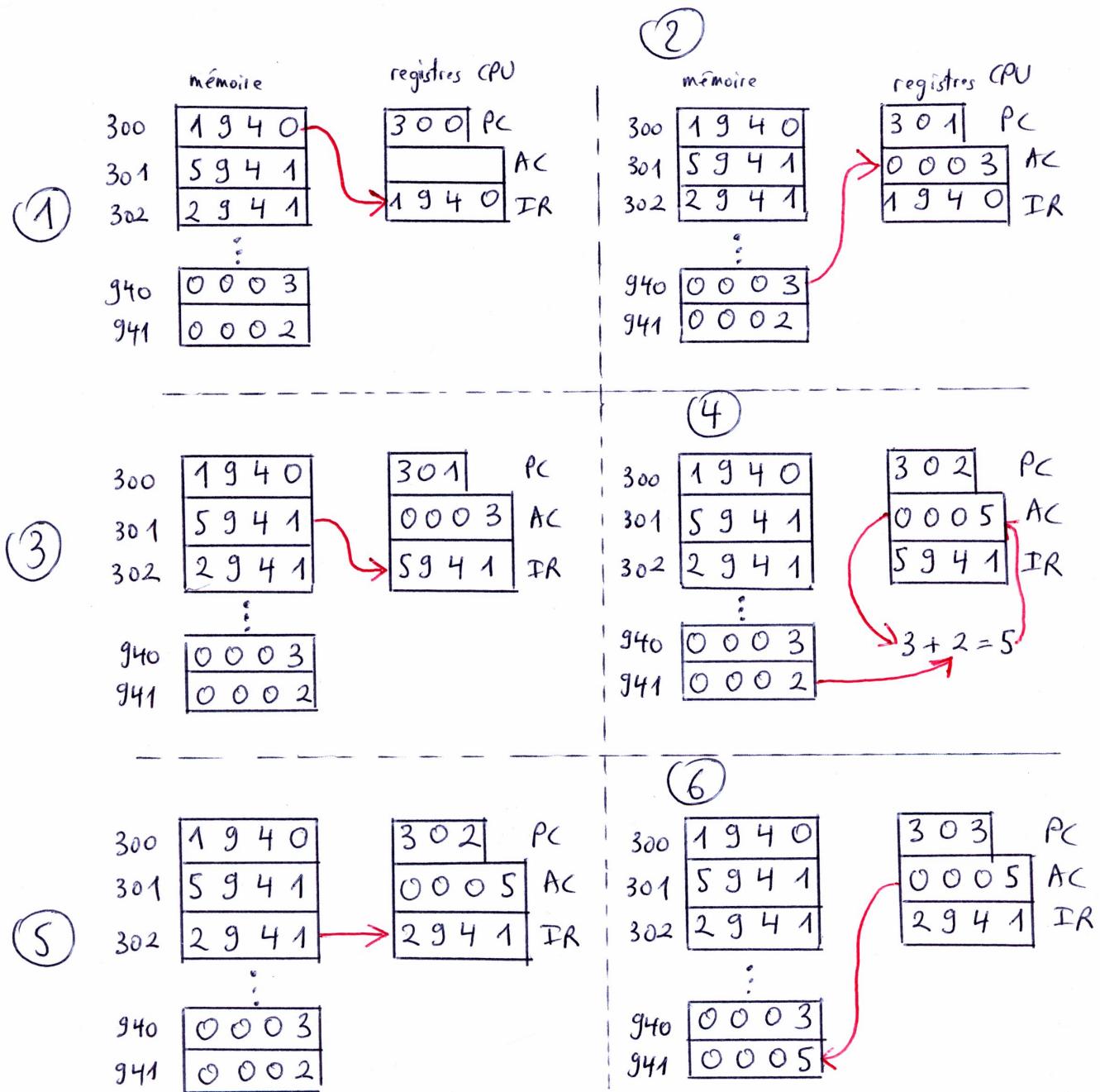


FIGURE 6.3 – Exemple d'exécution d'un programme.

6.3 Notion d'interruption

Pratiquement tous les ordinateurs ont un mécanisme par lequel d'autres modules (entrées/sorties, mémoire) peuvent *interrompre* le déroulement normal d'un processeur. Dans ce cas, le travail en cours est momentanément interrompu et un autre programme (appelé *handler*) est exécuté, puis le programme initial reprend là où il était resté.

Les principales causes d'interruption sont les suivantes :

- un dépassement arithmétique, une division par zéro, la tentative d'exécution d'une instruction illégale, ou la référence en dehors de l'espace mémoire autorisé à un utilisateur ;
- un *timer* : ceci permet au système d'exploitation de réaliser des fonctions de manière régulière ;
- un signal généré par un contrôleur d'entrée/sortie, pour informer de l'achèvement d'une opération, demander l'intervention du processeur, etc.
- un problème matériel, tel qu'une coupure de courant ou un problème de parité mémoire.

Les interruptions permettent par exemple à un programme de continuer de s'exécuter pendant qu'une opération d'entrée/sortie est en cours et gérée par un périphérique. Mais à la fin de l'opération d'entrée/sortie, le processeur est interrompu et doit réaliser un traitement spécial.

Il arrive que plusieurs interruptions se produisent. Par exemple, un programme peut recevoir des données d'une source et imprimer les résultats. L'imprimante va générer une interruption à chaque fois qu'une impression est réalisée. Et la source de donnée va causer une interruption à chaque fois qu'une unité de donnée arrive. Si cela n'était pas fait, le processeur ne pourrait qu'attendre l'arrivée des données, puis les envoyer à l'impression. Grâce aux interruptions, il peut réaliser d'autres opérations, pour un autre processus, un autre utilisateur, etc.

Mais du fait des interruptions multiples, une interruption peut se produire pendant qu'une autre est en cours de traitement. Une possibilité est d'inhiber les interruptions pendant qu'une interruption est en cours de traitement (figure 6.4). Une autre est de permettre un traitement imbriqué (figure 6.5).

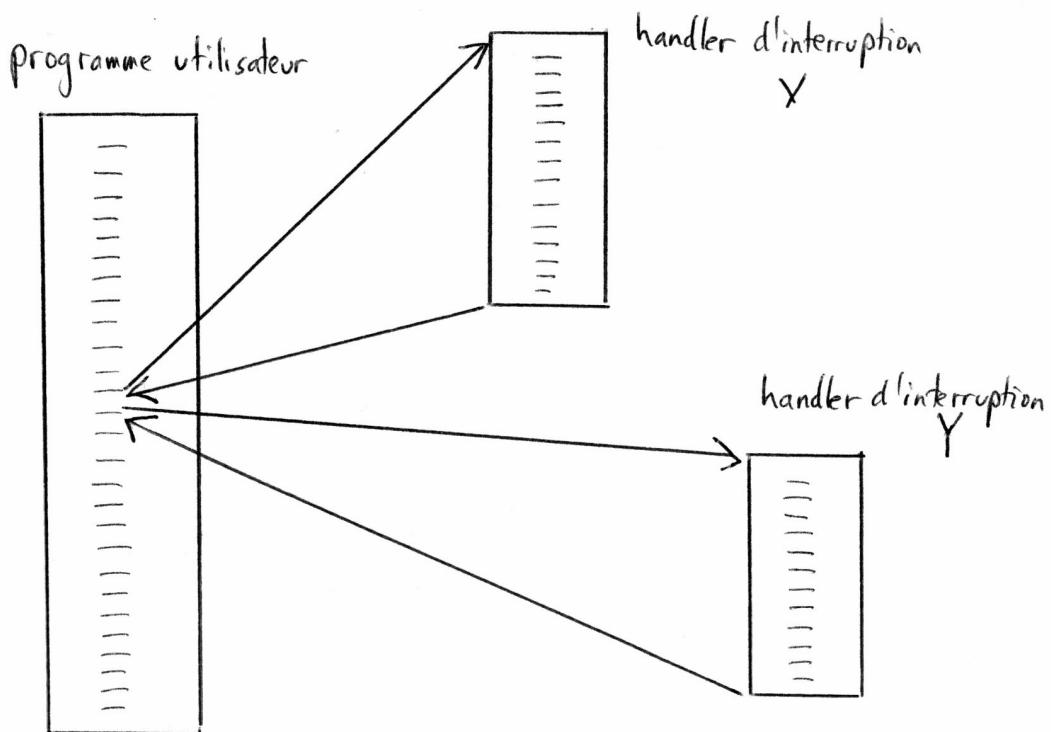


FIGURE 6.4 – Traitement séquentiel des interruptions.

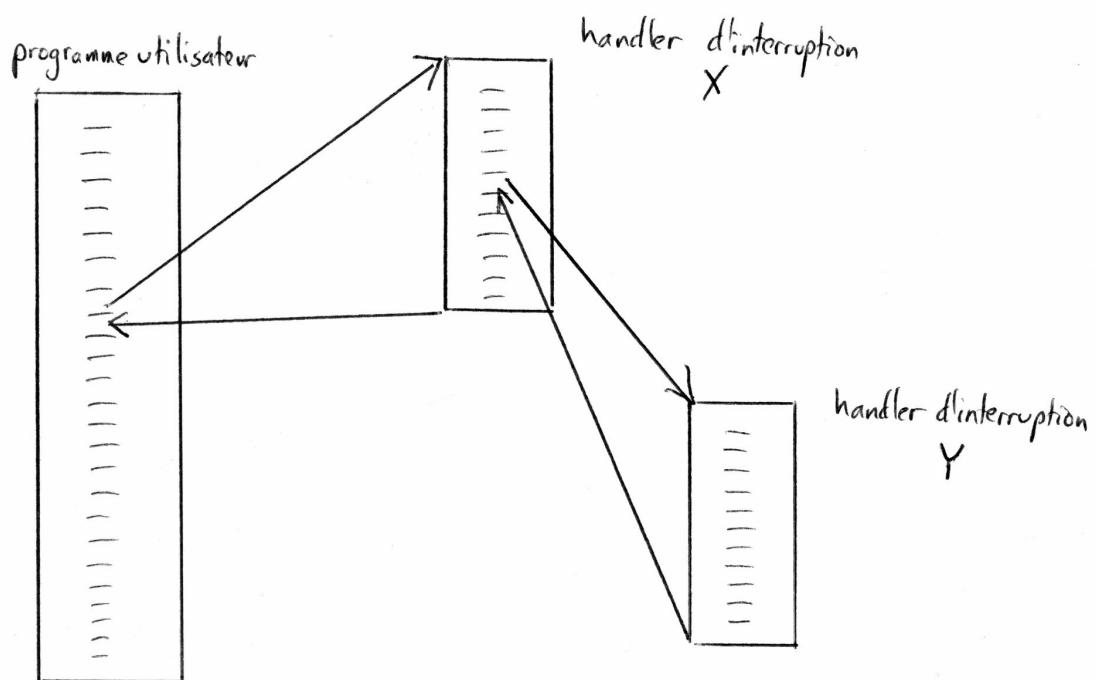


FIGURE 6.5 – Traitement imbriqué des interruptions.

6.4 Structures d'interconnexion

Un ordinateur se compose d'un ensemble de composants ou modules qui communiquent entre eux. L'ensemble des chemins qui connectent les modules est la structure d'interconnexion. La figure 6.6 illustre le type des échanges entre la mémoire, les modules d'entrée/sortie et le processeur.

Pour la mémoire, il peut y avoir des demandes de lecture ou d'écriture, et il faut fournir une adresse. En cas d'écriture, on fournit une donnée. En cas de lecture, on récupère une donnée. La figure montre une voie plus large pour l'adresse et les données, car on fait circuler simultanément plusieurs bits. Les flèches marquées « lecture » et « écriture » sont simplement des sélecteurs.

Un module d'entrée/sortie est fonctionnellement similaire à la mémoire. On peut lire ou écrire vers un tel module. Un module d'entrée/sortie peut contrôler plus d'un périphérique. Chaque interface vers un périphérique peut être appelée un *port* et chaque port peut être doté d'une adresse unique. Des données peuvent transiter vers les périphériques. Enfin, un module d'entrée/sortie peut envoyer des signaux d'interruption au processeur.

Enfin, le processeur lit des instructions et des données, écrit des données, reçoit des signaux d'interruption et utilise des signaux pour contrôler l'ensemble des opérations du système.

La structure d'interconnexion doit permettre les transferts suivants :

- mémoire vers processeur : lecture d'instructions ou de données ;
- processeur vers mémoire : le processeur écrit une unité de donnée en mémoire ;
- entrée/sortie vers processeur : le processeur lit des données d'un périphérique via un module d'entrée/sortie ;
- processeur vers entrée/sortie : le processeur envoie des données vers un périphérique ;
- entrée/sortie de/vers la mémoire : un module d'entrée/sortie

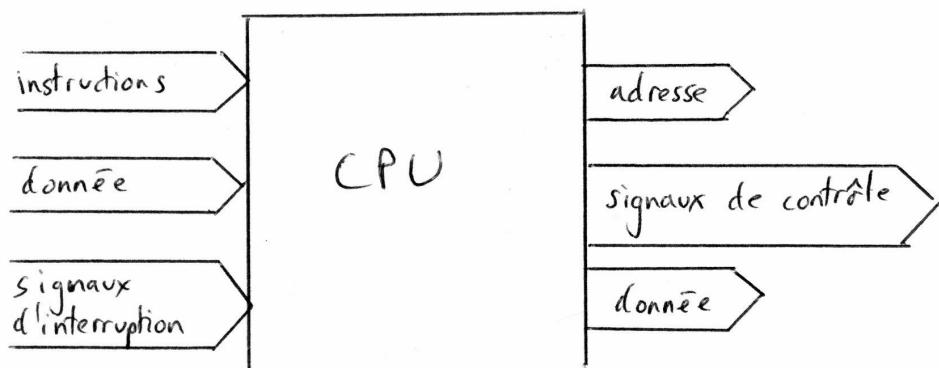
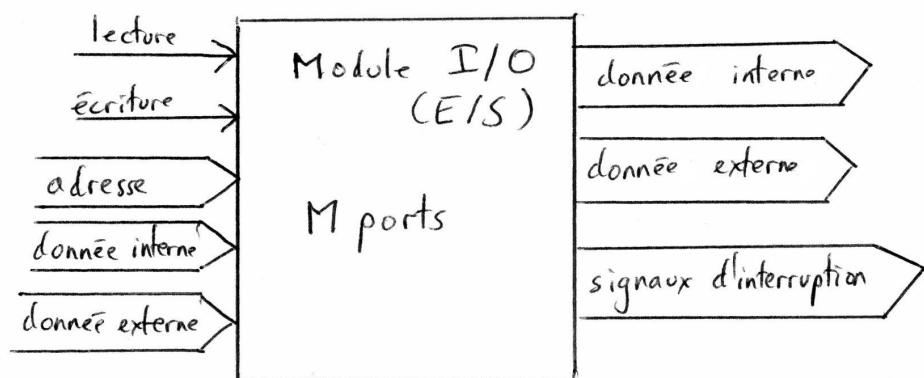
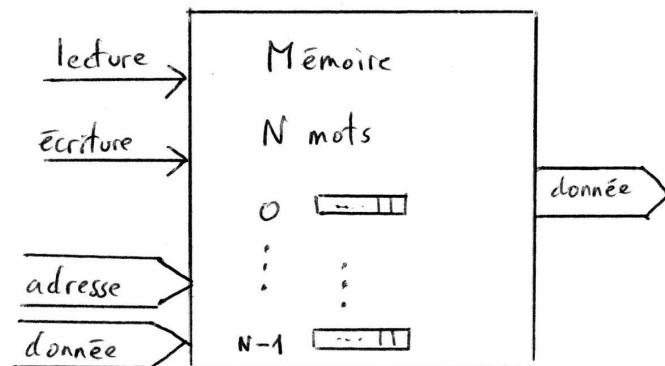


FIGURE 6.6 – Structures d'interconnexion.

peut être autorisé à échanger des données directement avec la mémoire, sans passer par le processeur, en utilisant un accès mémoire direct (*direct memory access*).

Les structures d'interconnexion les plus communes sont le *bus* et les structures point-à-point, dont un exemple est le QPI (*QuickPath Interconnect*) d'Intel. Nous dirons ici simplement quelques mots sur le bus.

Une caractéristique clé d'un bus est que c'est un moyen de transmission partagé. Plusieurs composants se connectent au bus et un signal transmis par un composant quelconque est disponible pour réception par tous les autres composants rattachés au bus. Si deux composants transmettent durant le même intervalle de temps, leurs signaux se superposeront et seront mélangés. Par conséquent, seul un composant peut transmettre à un instant donné.

Typiquement, un bus consiste en plusieurs chemins (ou lignes) de communication. Chaque ligne ne peut transmettre qu'un bit.

Un bus qui connecte les principaux composants d'un ordinateur est appelé un *bus système*. Un tel bus consiste typiquement en 50 à 100 lignes distinctes. On peut en général classer les lignes en trois catégories : lignes de données, lignes d'adresses et lignes de contrôle (figure 6.7).

Les lignes de contrôle sont utilisées pour contrôler les accès aux autres lignes. Il y a par exemple typiquement une ligne de contrôle pour l'écriture en mémoire. Cette ligne permet de faire que la donnée du bus de donnée soit écrite à l'adresse du bus d'adresse.

En général, pour pouvoir utiliser le bus, un module doit en faire la demande.

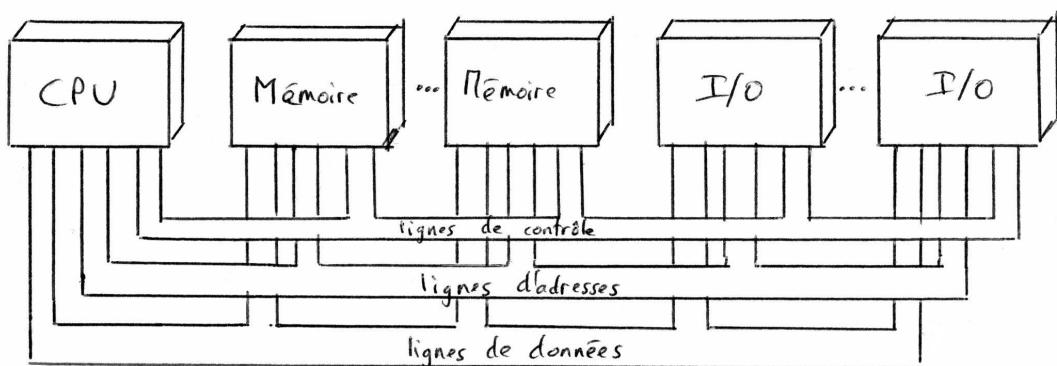


FIGURE 6.7 – Schéma d'interconnexion de bus.

6.5 Langage machine

6.5.1 Adresses mémoires

La mémoire d'un ordinateur peut être représentée schématiquement sous la forme d'une collection de mots. Un mot est un groupe de n bits qui peuvent être stockés ou lus au moyen d'une unique opération de base.

Les ordinateurs modernes ont des longueurs de mots qui vont typiquement de 16 à 64 bits. Une unité de 8 bits est appelée un *octet* (*byte* en anglais).

Pour accéder à la mémoire, il faut avoir des adresses pour tous les emplacements mémoire. On utilise habituellement des nombres de 0 à $2^k - 1$ pour une certaine valeur de k . Les 2^k adresses constituent l'espace d'adressage de l'ordinateur. Par exemple, une adresse de 32 bits crée un espace d'adressage de 2^{32} emplacements, soit 4 giga emplacements.

Il n'est pas pratique de donner des adresses aux bits individuels, et la solution la plus pratique est de donner des adresses aux octets. Les octets successifs de la mémoire ont pour adresses 0, 1, 2, Si les mots font 4 octets, les mots successifs ont pour adresses 0, 4, 8, etc.

6.5.1.1 Petit et gros boutisme

Il faut distinguer le « boutisme » (*endianness* en anglais). Quand certains ordinateurs enregistrent un entier sur 32 bits en mémoire, par exemple 0xA0B70708 en notation hexadécimale, ils l'enregistrent dans des octets dans l'ordre qui suit : A0 B7 07 08, pour une structure de mémoire fondée sur une unité atomique de 1 octet et un incrément d'adresse de 1 octet. C'est le « gros-boutisme ».

Les autres ordinateurs enregistrent 0xA0B70708 dans l'ordre suivant : 08 07 B7 A0 (pour une structure de mémoire fondée sur une unité atomique de 1 octet et d'un incrément d'adresse de 1 octet), c'est-à-dire avec l'octet de poids le plus faible en premier. C'est

le « petit-boutisme ».

Certains processeurs peuvent fonctionner dans les deux modes.

6.5.1.2 Alignement des mots

Pour des raisons d'efficacité, les mots sont généralement « alignés », ce qui signifie que les adresses des mots se trouvent à des multiples de la taille des mots. Si les mots font quatre octets, les adresses sont alors 0, 4, 8, etc.

6.5.1.3 Accès à des nombres et des caractères

Un nombre occupe en général un seul mot. Les caractères occupent en général un octet.

6.5.2 Opérations mémoire

Il y a deux opérations de base, la lecture et l'écriture.

6.5.3 Instructions

Les tâches accomplies par un programme consistent en une suite de petits pas, tels qu'additionner deux nombre, tester une certaine condition, lire un caractère du clavier, etc. Un ordinateur doit avoir des instructions pour les quatre types d'opérations suivantes :

- transfert de données entre la mémoire et les registres du processeur ;
- opérations arithmétiques et logiques sur les données ;
- séquencement et contrôle du programme ;
- transferts entrées/sorties.

Dans la suite, on utilisera une notation simplifiée pour les instructions d'un processeur fictif.

Quand nous écrivons

$$R2 \leftarrow [LOC]$$

cela signifie que le contenu de l'emplacement mémoire LOC est transféré dans le registre R2 du processeur.

$$R4 \leftarrow [R2] + [R3]$$

signifie que les contenus des registres R2 et R3 sont additionnés et que leur somme est placée dans le registre R4.

Les différentes instructions sont représentées par des codes, c'est le langage machine, mais on utilise en général une notation de plus haut niveau, qui est le langage d'assemblage. Ainsi, pour les deux instructions données plus haut, on écrira plutôt :

Load R2, LOC

et

Add R4,R2,R3

Dans les instructions des langages d'assemblage de vrais processeurs, les opérations sont en fait définies en employant des *mnémotechniques*, qui sont des abréviations. Par exemple, au lieu d'écrire Load, on a en général LD

Il existe deux approches fondamentalement différentes pour la conception d'un ensemble d'instructions :

RISC Ici, on utilise un petit nombre d'instructions qui tiennent sur un seul mot;

CISC ici, par contre, on peut avoir des instructions plus complexes.

L'approche RISC a été introduite dans les années 1970.

6.5.4 Exemples

Soit à exécuter l'instruction de haut niveau

$$C \leftarrow [A] + [B]$$

où A, B et C désignent des emplacements mémoire. Cette instruction peut se traduire dans les quatre instructions suivantes :

```
Load  R2,A
Load  R3,B
Add   R4,R2,R3
Store R4,C
```

Il est facile d'examiner le langage machine produit par la compilation d'un programme C. Considérons par exemple le programme suivant :

```
#include <stdio.h>
int main() {
    int n,i;
    n=13;
    for (i=0;i<10;i++)
        n=n+i;
    printf("n=%d\n",n);
}
```

Compilons ce programme avec :

```
gcc -g -c x1.c
```

On obtient un fichier x1.o, dont on peut examiner le contenu avec objdump :

```
objdump -dS x1.o
```

Cela donne (pour x86) :

```
x1.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
#include <stdio.h>
int main() {
    0: f3 0f 1e fa        endbr64
    4: 55                 push   %rbp
    5: 48 89 e5           mov    %rsp,%rbp
    8: 48 83 ec 10        sub    $0x10,%rsp
    int n,i;
    n=13;
    c: c7 45 f8 0d 00 00 00  movl   $0xd,-0x8(%rbp)
    for (i=0;i<10;i++)
    13: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
    1a: eb 0a              jmp    26 <main+0x26>
        n=n+i;
    1c: 8b 45 fc           mov    -0x4(%rbp),%eax
    1f: 01 45 f8           add    %eax,-0x8(%rbp)
    for (i=0;i<10;i++)
    22: 83 45 fc 01         addl   $0x1,-0x4(%rbp)
    26: 83 7d fc 09         cmpl   $0x9,-0x4(%rbp)
    2a: 7e f0              jle    1c <main+0x1c>
    printf("n=%d\n",n);
    2c: 8b 45 f8           mov    -0x8(%rbp),%eax
    2f: 89 c6              mov    %eax,%esi
    31: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax      # 38 <main+0x38>
    38: 48 89 c7           mov    %rax,%rdi
    3b: b8 00 00 00 00       mov    $0x0,%eax
    40: e8 00 00 00 00       call   45 <main+0x45>
    45: b8 00 00 00 00       mov    $0x0,%eax
}
    4a: c9                 leave
    4b: c3                 ret
```

L'affichage précédent mêle le code assembleur (option -d) et le code source (option -S). On peut reconnaître dans le code précédent l'initialisation de n et de n, la boucle (on saute d'abord en 26 au test de fin de boucle, puis on remonte en 1c, l'addition de i à n en 1f, l'incrémentation de i en 22, et enfin l'affichage par un appel à call.

6.5.5 Un parallèle avec Java

Le programme Java équivalent au programme précédent est :

```
class Test {  
    public static void main(String args[]) {  
        int n,i;n=13;  
        for (i=0;i<10;i++) n=n+i;  
        System.out.println("n="+n);  
    }  
}
```

En le compilant, on obtient un fichier `Test.class` qu'on peut désassembler avec `javap -c Test.class` pour afficher le *bytecode* (les commentaires ont été supprimés pour éviter d'encombrer l'affichage) ce qui donne le résultat de la figure 6.8.

Il est possible d'apporter des modifications à un fichier en bytecode, tout comme on peut compiler directement de l'assembleur, mais ces manipulations sortent du cadre de ce cours.

```
Compiled from "Test.java"
class Test {
    Test();
    Code:
        0: aload_0
        1: invokespecial #1
        4: return

    public static void main(java.lang.String[ ]);
    Code:
        0: bipush      13
        2: istore_1
        3: iconst_0
        4: istore_2
        5: iload_2
        6: bipush      10
        8: if_icmpge   21
       11: iload_1
       12: iload_2
       13: iadd
       14: istore_1
       15: iinc      2, 1
       18: goto      5
       21: getstatic   #2
       24: new       #3
       27: dup
       28: invokespecial #4
       31: ldc       #5
       33: invokevirtual #6
       36: iload_1
       37: invokevirtual #7
       40: invokevirtual #8
       43: invokevirtual #9
       46: return
}
```

FIGURE 6.8 – Exemple de bytecode Java.

6.6 Familles de processeurs

La plupart des appareils de calcul actuels sont équipés d'un processeur compatible Intel ou d'un processeur ARM (*Advanced RISC Machine*). AMD est un constructeur fabriquant des processeurs compatibles Intel, comme l'AMD Ryzen. Arm est en fait une forme d'architecture et il n'y a pas un constructeur unique de processeurs ARM. La technologie ARM est utilisée par Apple et Android sur leurs appareils mobiles, tandis qu'Intel ou AMD est utilisé sur les ordinateurs.

L'origine des processeurs modernes Intel/AMD et ARM remonte à des machines mises sur le marché au début des années 1980, à savoir l'Acorn Archimedes (figure 6.10) et l'IBM PC (figure 6.9) qui utilisait un processeur Intel 8088.



FIGURE 6.9 – L'IBM PC (1981). (Source : Wikipédia)



FIGURE 6.10 – L'Acorn Archimedes (1987). (Source : Wikipédia)

Un exemple de code CISC (x86) a été donné plus haut. Un exemple de module ARM est le suivant :

```

AREA      ARMEx, CODE, READONLY
                    ; Name this block of code ARMEx
ENTRY                 ; Mark first instruction to execute
start
    MOV      r0, #10      ; Set up parameters
    MOV      r1, #3
    ADD      r0, r0, r1   ; r0 = r0 + r1
stop
    MOV      r0, #0x18      ; angel_SWIreason_ReportException
    LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
    SWI      0x123456      ; ARM semihosting SWI
    END                  ; Mark end of file

```

Ces deux familles de processeurs ont des avantages et des inconvénients :

Ensemble d'instructions : les processeurs Intel sont des processeurs CISC (*Complex Instruction Set Computer*), tandis que les processeurs ARM sont des processeurs RISC (*Reduced Instruction Set Computer*). L'idée du CISC est d'avoir des instructions diversifiées et donc simples, pour gérer un matériel complexe. À l'opposé, le RISC s'adapte à un matériel simple, mais avec des instructions limitées et du code complexe. Les processeurs ARM sont plus adaptés aux appareils qui doivent limiter leur consommation, comme les appareils mobiles et embarqués.

Compatibilité logicielle : les processeurs Intel ne peuvent comprendre le code ARM et inversement ; le système d'exploitation et les logiciels doivent être adaptés au type de processeur ; il est cependant possible d'exécuter un logiciel conçu pour un type de processeur sur un autre, mais en général en perdant en vitesse et efficacité.

Consommation énergétique : la faible consommation des processeurs ARM est un avantage considérable sur les processeurs Intel ; par ailleurs, les processeurs ARM sont moins

chers que les processeurs Intel ; ceci explique que les marchés des tablettes et téléphones portables est contrôlé par ARM ;

Performance : les processeurs Intel écrasent les processeurs ARM ;

Symétrie des cœurs : un processeur ARM peut contenir plusieurs *cœurs* différents, par exemple quatre cœurs à faible consommation et quatre de haute performance, ce qui donne un avantage à ARM sur l'emploi de cœurs symétriques ;

6.7 Exercices

- la machine hypothétique vue plus haut (section 6.2) a deux instructions d'entrée/sortie (I/O) :
 - 0011 : charger AC depuis I/O
 - 0111 : stocker AC vers I/O
 Dans ce cas, l'adresse sur 12 bits identifie un périphérique d'entrée/sortie particulier. Montrer l'exécution du programme (selon le format de la figure 6.3) pour le programme suivant :
 1. charger AC depuis le périphérique 5 ;
 2. ajouter le contenu de la case mémoire 940 ;
 3. stocker AC dans le périphérique 6.
 On supposera que la prochaine valeur récupérée du périphérique 5 est 3 et que la case 940 contient la valeur 2.
- l'exécution du programme de la figure 6.3 est donnée avec six étapes ; développez cette description en montrant l'utilisation de MAR et MBR.
- considérez un microprocesseur hypothétique de 32 bits avec des instructions sur 32 bits composées de deux champs : le premier octet contient l'opcode et le reste l'opérande immédiate ou une adresse d'opérande ;
- quelle est la mémoire maximale directement adressable (en octets) ?
- quel est l'impact sur la vitesse si le bus du microprocesseur a
 - un bus d'adresse local de 32 bits et un bus de données locales de 16 bits,
 - un bus d'adresse local de 16 bits et un bus de données locales de 16 bits ;
- trouvez des processeurs actuels fonctionnant en « petit-boutisme » et d'autres fonctionnant en « gros-boutisme » ;
- utilisez l'émulateur d'Intel 4004 : <http://e4004.szyc.org/emu> (la fiche technique de ce processeur est disponible en <http://datasheets.chipdb.org/Intel/MCS-4/datashts/intel-4004.pdf>)

- trouvez une liste de toutes les instructions du *bytecode Java* ; combien y a-t-il d'instructions ?

6.8 Un peu d'histoire

When the coding is in final form such that the input teletype tape is to be prepared, one has the instructions reduced to numerical form and has available the true numerics for all of the involved quantities.

Assume, for example, that

a = .075329	e = .83291
b = .12391	f = .69736
c = .017326	x = .32915

The final coding is:

1. EB012DAO10	8. 0.123910000
2. BA011DC014	9. 0.017326000
3. EB007DAO12	10. 0.832910000
4. BA008DE040	11. 0.697360000
5. DA012BA009	12. 0.329150000
6. DD014EC013	13. 0.000000000
7. 0.07532900	14. 0.000000000

FIGURE 6.11 – Un programme pour l'ordinateur MANIAC en 1954. Ce programme calcule $y = \frac{ax^2+bx+c}{ex+f}$ pour des valeurs de x, a, b, c, e et f données. Le programme consiste en l'ensemble des 14 valeurs numériques, les données sont les dernières. (source : manuel de l'ordinateur, p. 28-30)

Programme :

ZERO	00	0000	0000	O in ZERO
ONE	00	0000	0001	1 in ONE
START	LDD	ZERO		Initializing COUNT
	STD	COUNT		
	RAU	1331		
	NZU	I		Go to I if c(1331) ≠ 0
	RAU	COUNT		
	AUP	ONE		
	STU	COUNT I		Count 1 if c(1331) = 0
I	RAU	1332		
	NZU	J		Go to J if c(1332) ≠ 0
	RAU	COUNT		
	AUP	ONE		
	STU	COUNT J		Count 1 if c(1332) = 0
J	RAU	1333		
	NZU	K		Go to K if c(1333) ≠ 0
	RAU	COUNT		
	AUP	ONE		
	STU	COUNT K		Count 1 if c(1333) = 0
K	RAU	1334		
	NZU	L		Go to L if c(1334) ≠ 0
	RAU	COUNT		
	AUP	ONE		
	STU	COUNT L		Count 1 if c(1334) = 0
L	HLT	9999 9999		Stop, showing 9999 9999

This programme demonstrates {i) branching on O
{ii) initializing
{iii) counting
{iv) symbolic addresses
{v) stop instructions.

SOAP :

- Handles constants like ZERO and ONE.
- Gives 2-digit codes to 3-letter OP-codes.
- Gives locations to symbols.
- Inserts locations for consecutive blanks.
- Allocates locations optimally for speed.
- Assembles complete programme.

Hence it is a SOA programme : SOAP

Advantages :

- Easier to write, check and correct.
- Reduces errors in writing.
- Speeds up operation.

Disadvantages :

- A little slower to punch.
- Requires assembling.

SOAP is an example of automatic programming, where the computer is used as part of the programme writing process.

Examples

A few examples of preparing flow charts are now given. Flow charts require analyzing one's problem into its smallest component parts. They are the most important aspect of preparing a problem for the computer. Coding into SOAP, or machine language, is straightforward, once the flow chart is complete.

FIGURE 6.12 – Un programme pour l'ordinateur IBM 650.

```
PROGRAM MAIN
PRINT *,FACT(8)
STOP
END

FUNCTION FACT(N)
INTEGER I,N
FACT=1
DO 10 I=2,N,1
FACT=FACT*I
10 CONTINUE
END
```

FIGURE 6.13 – Un programme FORTRAN calculant 8!.

Chapitre 7

Mémoire

Il s'avère qu'aucune technologie n'est optimale pour satisfaire les besoins en mémoire d'un système informatique. En conséquence, un système informatique est typiquement équipé d'une hiérarchie de sous-systèmes de mémoire.

Les caractéristiques clés des systèmes de mémoire sont :

la localisation interne, externe, etc.

la capacité en octets ou mots

l'unité de transfert est le nombre de lignes électriques qui entrent ou sortent du module de mémoire

un mot est typiquement le nombre de bits utilisés pour représenter un entier;

l'unité d'adressage indique à quel niveau on peut accéder à la mémoire; beaucoup de systèmes permettent d'adresser les octets, mais certains systèmes ne peuvent adresser que les mots.

Il y a aussi différents méthodes d'accès aux unités de données, mais elles ne sont pas détaillées ici (accès séquentiel, direct, aléatoire et associatif).

La performance de la mémoire est une caractéristique importante. Il y a différents paramètres, par exemple le temps d'accès (latence), qui est le temps pour réaliser une opération de lecture ou d'écriture.

La mémoire se distingue aussi par sa nature physique et son caractère volatile ou non.

L'idée est de ne pas employer un unique composant de mémoire ou une unique technologie, mais d'employer une hiérarchie de mémoires (figure 7.1). Plus on descend dans la hiérarchie, plus le coût par bit décroît, plus la capacité augmente, plus le temps d'accès croît, et moins fréquents sont les accès du processeur à la mémoire. Le fait que cette hiérarchie est utilisable repose grandement sur le dernier point, à savoir que les mémoires les plus lentes sont utilisées le moins fréquemment. Ceci est notamment lié au principe de « localité de référence » qui fait que durant l'exécution d'un programme les références à la mémoire, tant pour les instructions que les données, tendent à se regrouper. À la longue, ces regroupements changent, mais sur une courte période de temps, le processeur travaille principalement avec des références de mémoire assez localisées. Cela permet alors de recopier ces références dans des mémoires rapides.

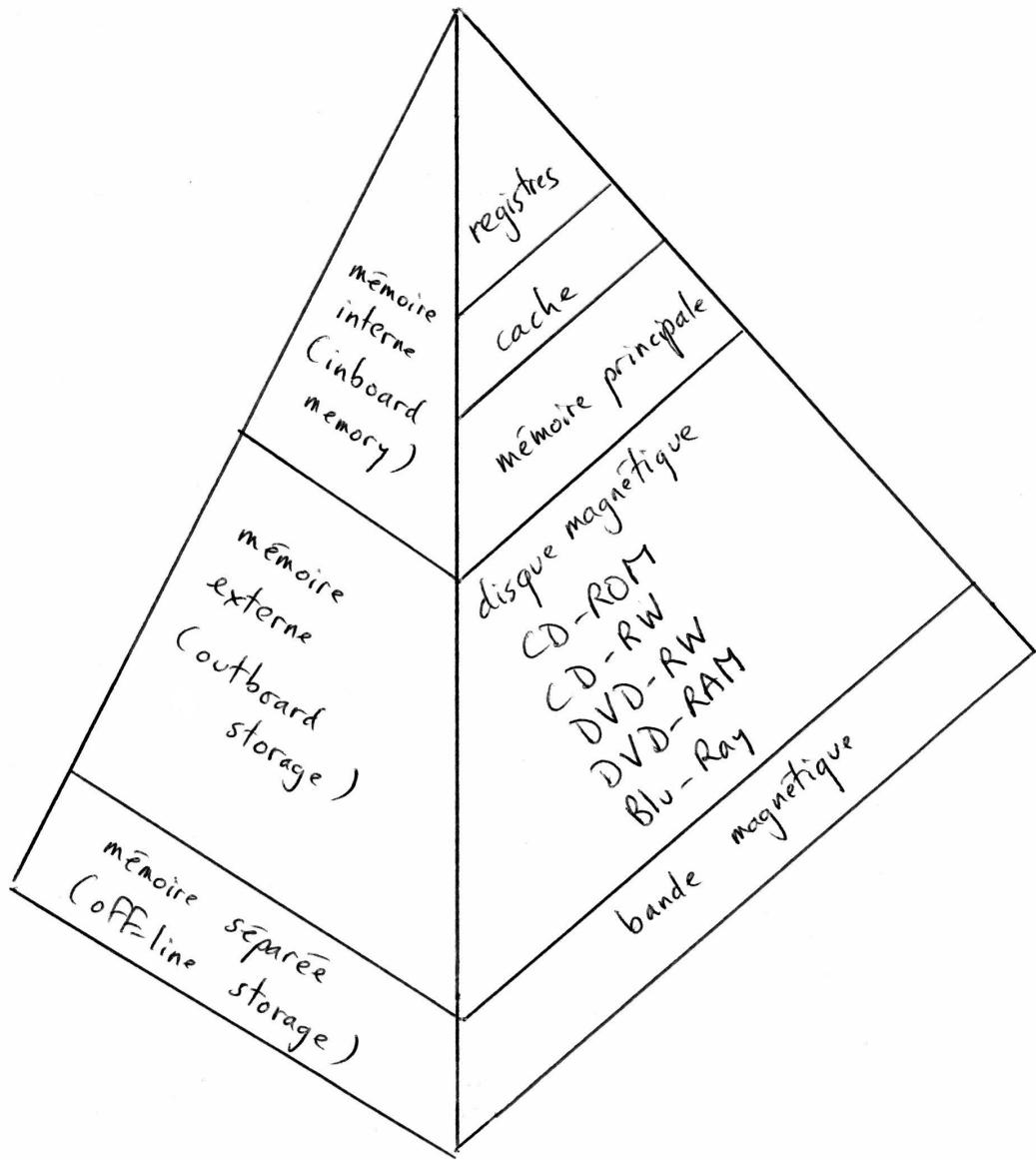


FIGURE 7.1 – La hiérarchie des mémoires.

7.1 Mémoire cache

La mémoire cache est conçue pour combiner l'accès rapide à des mémoires chères avec la grande capacité de mémoires moins chères, mais plus lentes. La figure 7.2 montre un unique cache entre le CPU et la mémoire principale. Le cache contient une copie de portions de la mémoire principale. Quand le processeur tente de lire un mot de la mémoire, une vérification est faite pour voir si le mot se trouve dans le cache. Si c'est le cas, le mot est donné au processeur. Sinon, un bloc de mémoire principale est lu dans le cache, puis le mot est donné au processeur. En raison du phénomène de localité des références, il est probable qu'il y aura de futures références à d'autres mots du même bloc.

La figure 7.3 montre l'emploi de plusieurs niveaux de caches. Les processeurs modernes comme i7 et i9 ont trois niveaux de caches (cf. figure 7.6), tous localisés dans le processeur. Dans des systèmes plus anciens, certains caches pouvaient être localisés sur la carte mère, donc en dehors du processeur.

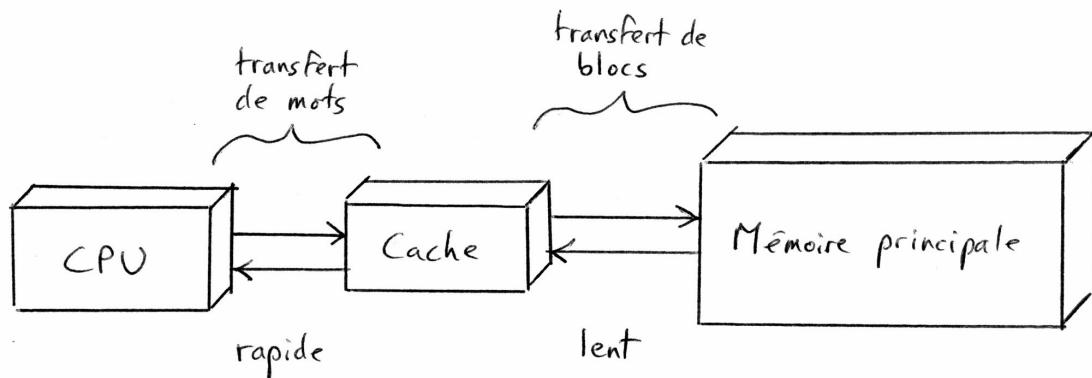


FIGURE 7.2 – Utilisation d'un seul cache.

La figure 7.4 illustre la structure d'un système de cache/mémoire principale. La mémoire principale consiste en 2^n mots adressables,

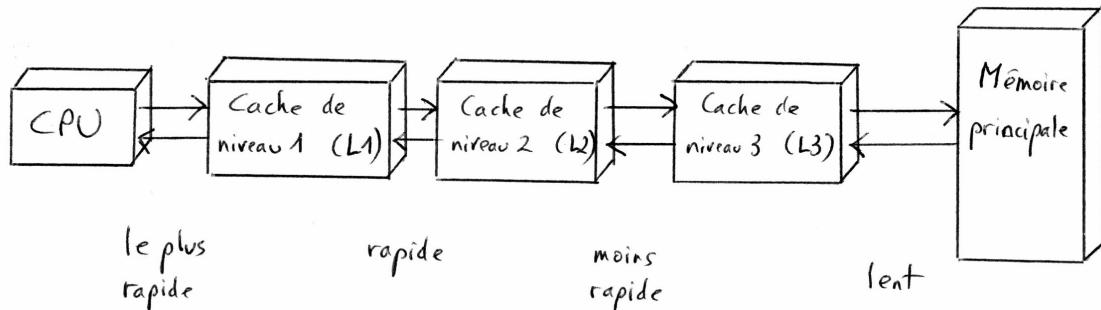


FIGURE 7.3 – Utilisation d'un cache à trois niveaux.

chaque mot ayant une unique adresse sur n bits. La mémoire principale est supposée formée de blocs de longueur fixe de K mots. Il y a donc $M = 2^n/K$ mots en mémoire principale.

Le cache comporte m blocs appelés *lignes*. En général, m est une puissance de 2. Chaque ligne contient donc K mots ainsi qu'un *tag* de quelques bits. Ce *tag* permet de savoir à quel bloc de la mémoire principale correspond la ligne du cache.

Une ligne contient aussi des bits de contrôle (non illustrés), notamment un bit pour indiquer si la ligne a été modifiée depuis qu'elle a été chargée en mémoire.

À tout moment, un sous-ensemble des blocs de la mémoire principale se trouve dans le cache.

La figure 7.5 illustre l'opération de lecture. Le processeur génère une adresse de lecture AL d'un mot à lire. Si le mot est contenu dans le cache, il est donné au processeur. Sinon, le bloc contenant le mot est chargé dans le cache et le mot est envoyé au processeur. Ces deux dernières opérations peuvent être faites en parallèle.

Il y a des algorithmes de mapping entre la mémoire principale et le cache. L'algorithme le plus simple est le mapping direct (*direct mapping*), où un bloc de la mémoire principale est toujours associé à la même ligne de la mémoire cache. Nous ne détaillerons pas da-

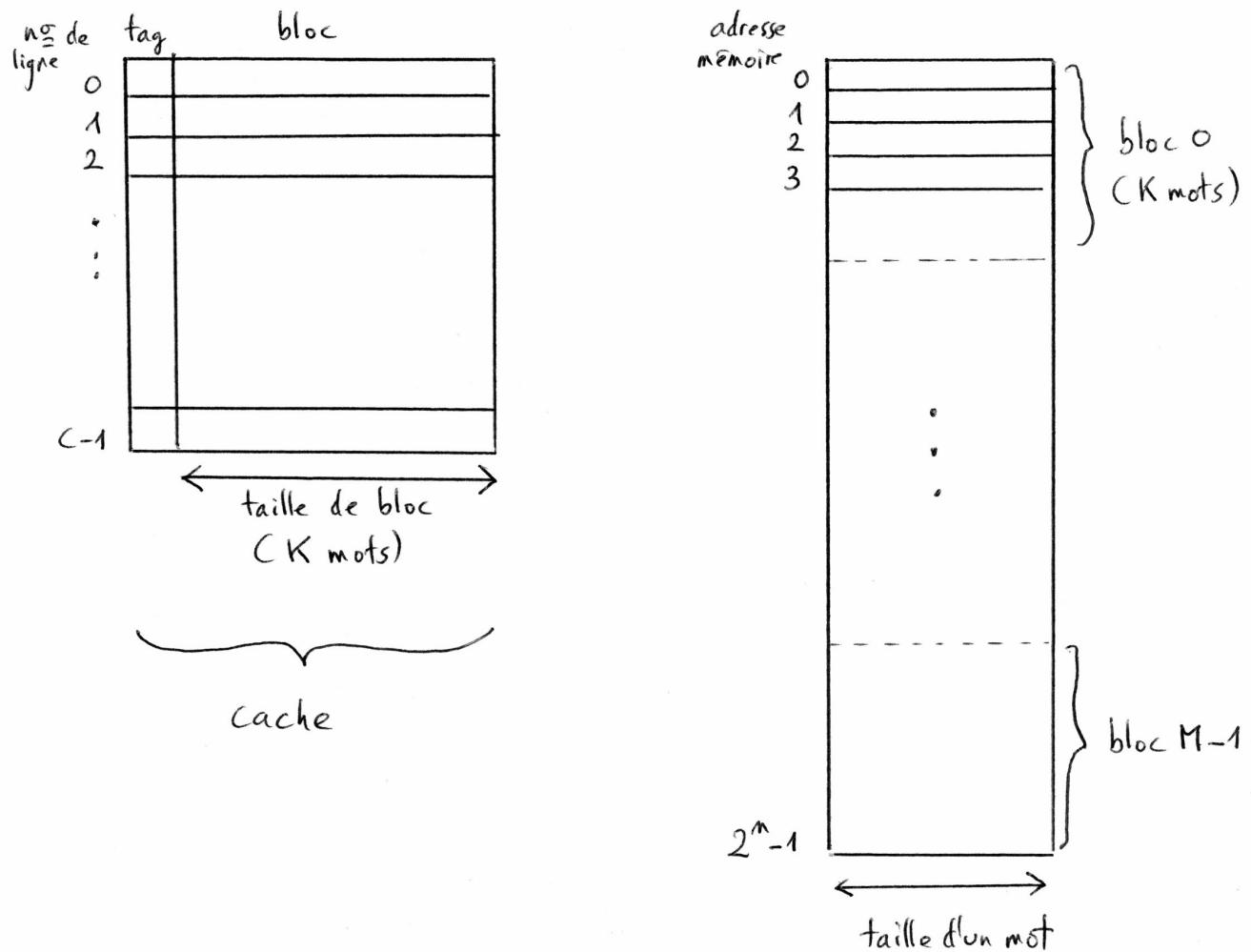


FIGURE 7.4 – Structure d'un cache et de la mémoire principale.

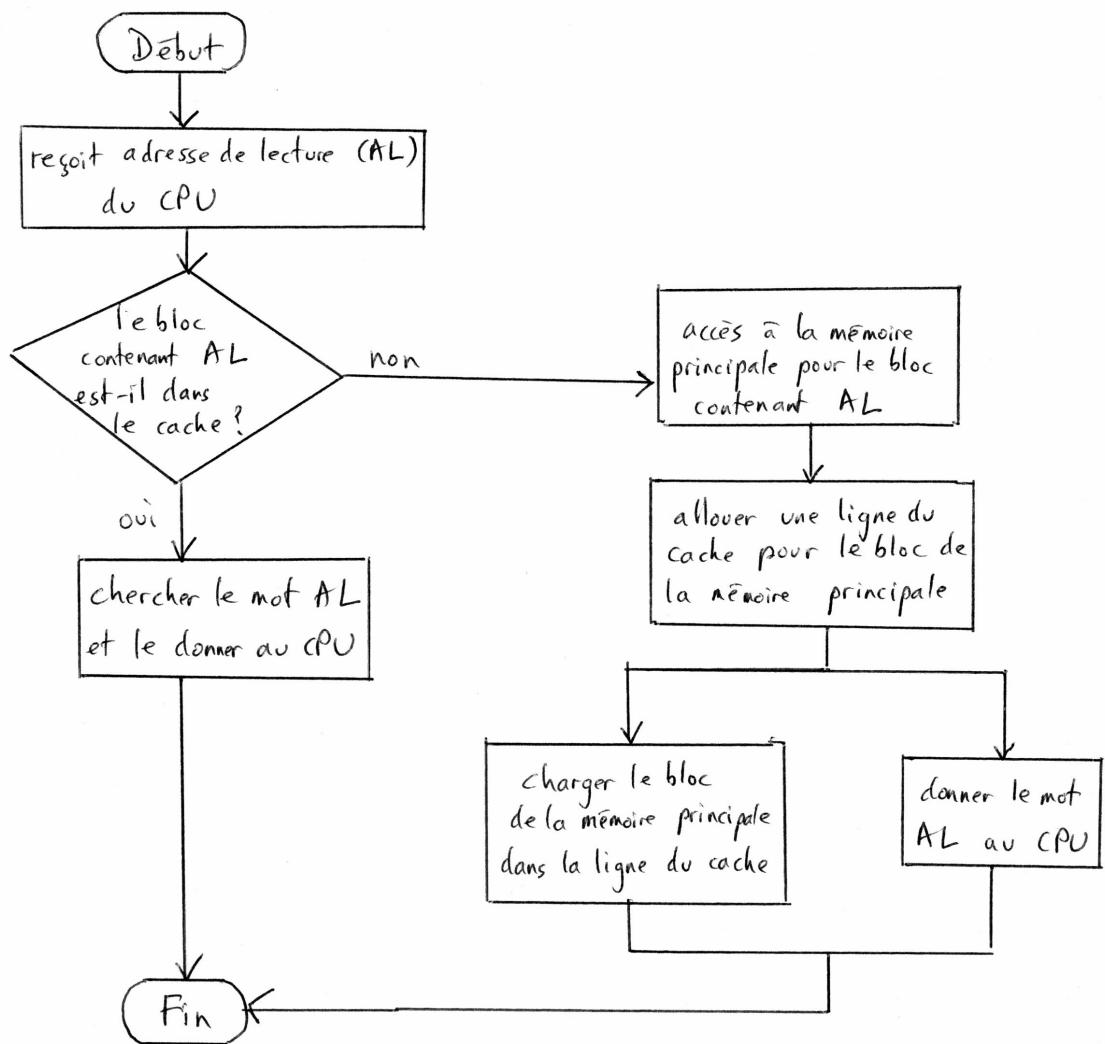


FIGURE 7.5 – Opération de lecture d'un cache.

vantage les autres algorithmes ici.

On notera enfin que la mémoire cache se place entre le processeur et la mémoire principale, mais que les mécanismes qui interviennent ici sont similaires à ceux de la mémoire virtuelle qui concerne les rapports entre la mémoire principale et le stockage externe, par exemple sur disque magnétique.

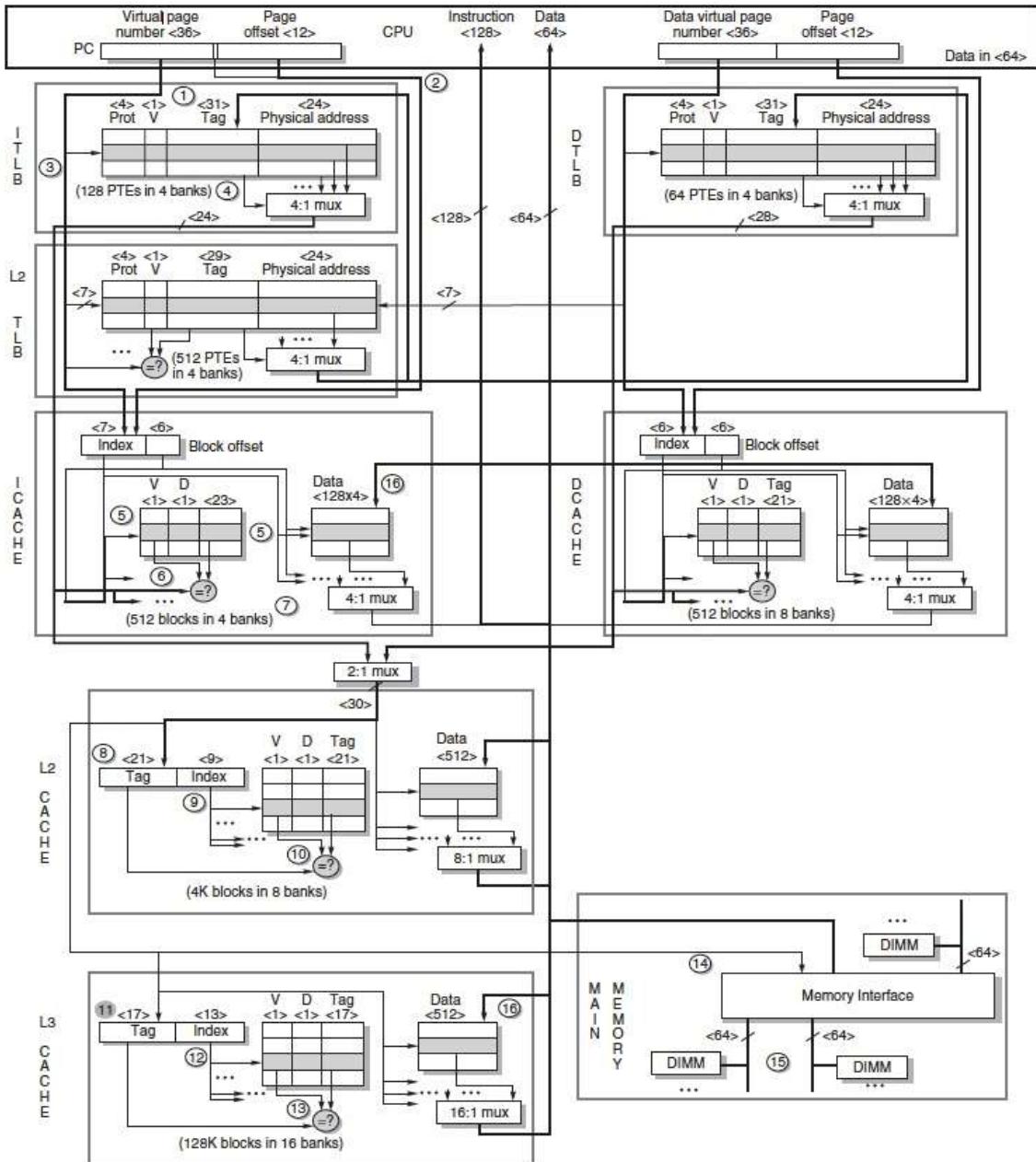


FIGURE 7.6 – La hiérarchie mémoire de l'Intel i7. (source : Hennessy, Patterson, *Computer architecture, A quantitative approach*, 2012, p. 119)

7.2 Mémoire principale, interne, ou vive

L'accès à la mémoire principale (ou vive) (*random access memory*, RAM) se fait par un accès direct (aussi appelé aléatoire) et non de manière séquentielle.

Les premiers stockages étaient mécaniques (Babbage, Ludgate), ou électromécaniques (Zuse). Dans ce dernier cas, on faisait appel à des relais électromécaniques.

Par la suite, des lampes à vide (*vacuum tubes*) ont été utilisées. L'ENIAC utilisait par exemple 36 tubes pour stocker un chiffre. Dans les années 1950, les lampes à vide ont été remplacées par des transistors.

Des années 1930 aux années 1950, une forme courante de mémoire était le tambour (*drum memory*) (figure 7.7). Il s'agissait d'une mémoire vive tournante, un peu comme les disques durs.

Dans les années 1940, on a aussi tenté d'utiliser des tubes électroniques pour servir de mémoire, par exemple avec le tube Selectron qui pouvait stocker 4096 bits.

Entre 1955 et 1975, la forme de stockage directe la plus fréquente employait de petites boucles (tores) ferromagnétiques (ferrite) appelées *core* (figure 7.8).

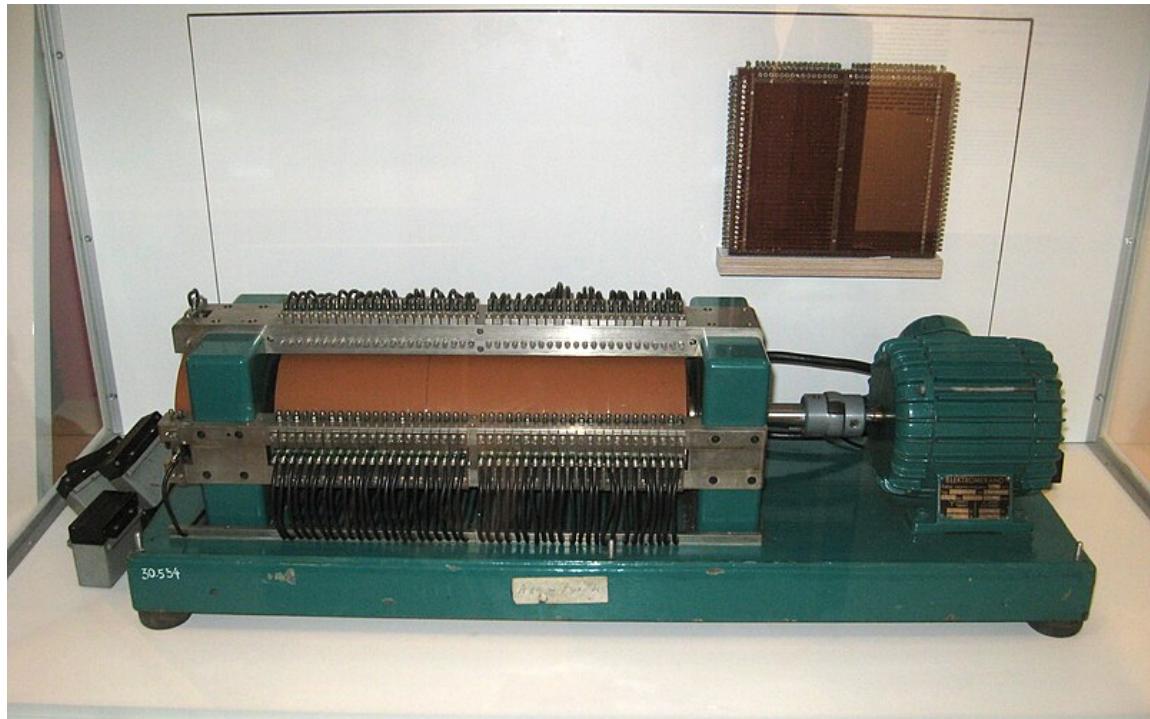


FIGURE 7.7 – Le tambour de mémoire vive de l'ordinateur BESK, le premier ordinateur binaire suédois (1953-1966). (source : Wikipédia)

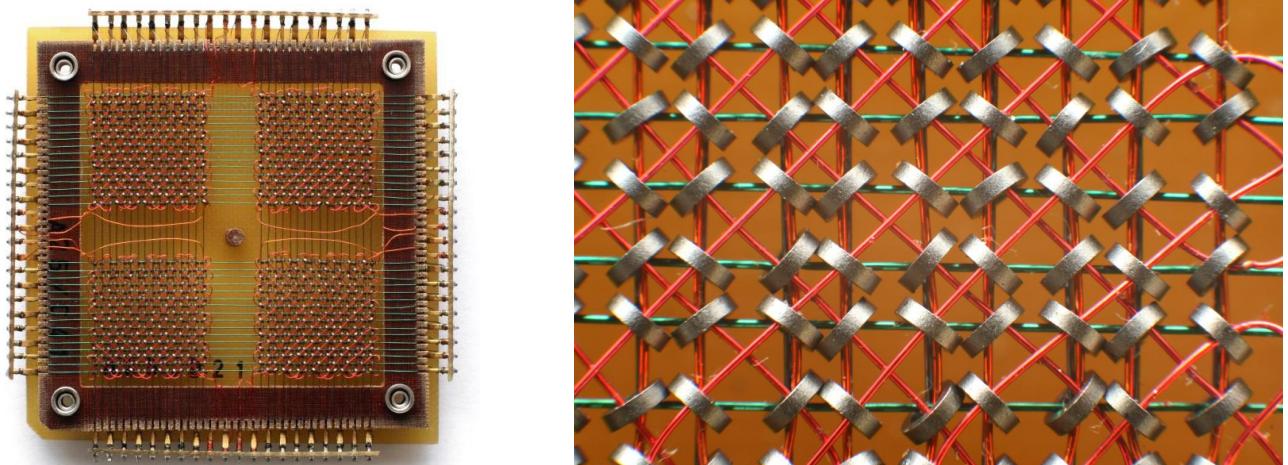


FIGURE 7.8 – Un module de 1024 *core* (donc 1024 bits) et détail.
(source : Wikipédia)

Aujourd’hui, la presque totalité des mémoires vives est à base de semi-conducteurs (silicium, germanium, etc.). L’élément de base est la cellule :

- il y a deux états stables (ou semistables) qui peuvent être utilisés pour représenter les valeurs 0 et 1 ;
- on peut écrire au moins une fois pour initialiser l’état ;
- la cellule peut être lue.

La figure 7.9 illustre les opérations d’une cellule de mémoire. La plupart du temps, une cellule a trois connecteurs permettant de transporter un signal électrique. Le connecteur de sélection permet de sélectionner une cellule pour une opération de lecture ou d’écriture. Le connecteur de contrôle indique s’il s’agit d’une lecture ou d’une écriture. Pour une opération d’écriture, le troisième connecteur fournit un signal électrique qui met l’état de la cellule à 0 ou 1. Pour une opération de lecture, le troisième connecteur permet de récupérer l’état de la cellule.

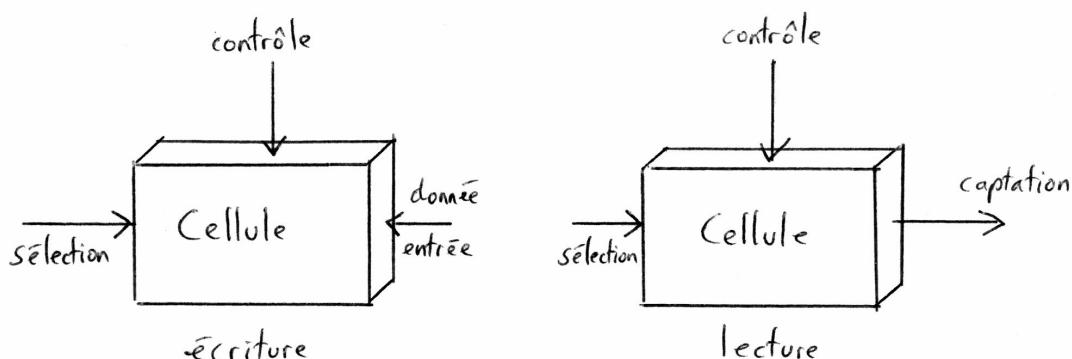


FIGURE 7.9 – Opérations de base sur une cellule de mémoire.

Tous les types de mémoire considérés ici sont à accès direct, c'est-à-dire que les mots de la mémoire sont directement adressés par une logique d'adressage câblée.

La RAM (*Random Access Memory*) désigne une mémoire à accès direct, où l'on peut lire et écrire rapidement via des signaux électriques. Cette dénomination est un peu malheureuse, car les autres mémoires décrites plus loin sont aussi à accès direct. Ce qui distingue cette RAM des mémoires suivantes, c'est le fait qu'elle puisse être écrite. Une autre caractéristique importante de la RAM est sa volatilité. Il doit y avoir une source de courant permanente. Si le courant est coupé, les données sont perdues. La RAM ne peut donc servir que pour un stockage temporaire. Les deux formes traditionnelles de RAM utilisées dans les ordinateurs sont la DRAM et la SRAM.

Des formes plus récentes et non volatiles de RAM existent (STT-RAM, PCRAM, ReRAM), mais ne sont pas décrites ici.

DRAM (*Dynamic RAM*, ou mémoire vive dynamique) : les cellules stockent les données dans des condensateurs; comme les condensateurs ont une tendance naturelle à se décharger, les DRAM doivent périodiquement être rafraîchies pour maintenir le stockage (par exemple toutes les 64ms pour une puce de 256Mbits); l'adjectif « dynamique » fait référence au fait que les condensateurs se déchargent;

SRAM (*Static RAM*, ou mémoire vive statique) : les valeurs binaires sont stockées en utilisant des bascules logiques traditionnelles.

La SRAM est plus rapide que la DRAM, mais aussi plus chère.

La figure 7.10 représente une structure typique pour une cellule de DRAM stockant un bit. La ligne d'adresse est activée lorsque le bit de cette cellule doit être lu ou écrit. Le transistor laisse passer le courant si la ligne d'adresse est activée. Lors de l'écriture, le bit à écrire est placé sur la ligne de bit *B* et est transféré au condensateur. Pour la lecture, la charge du condensateur est envoyée vers la ligne de bit *B*, elle est amplifiée, puis comparée à une valeur de référence pour déterminer s'il s'agit de 0 ou 1. Cette opération de lecture décharge le condensateur qui doit immédiatement être rechargé.

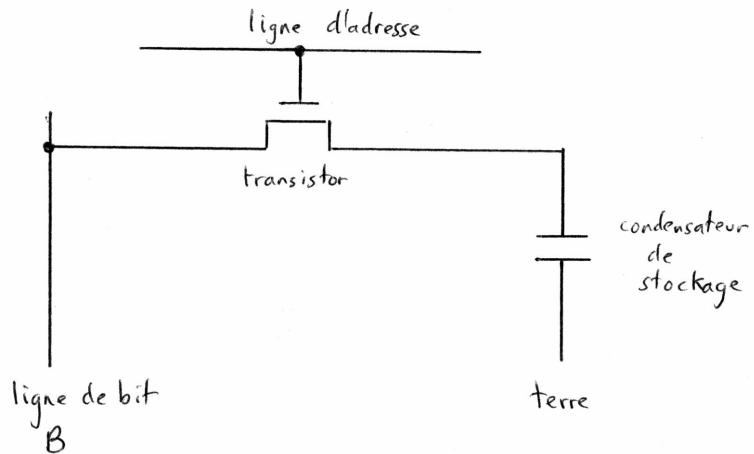


FIGURE 7.10 – Cellule de RAM dynamique (DRAM).

La figure 7.11 représente une structure pour une cellule de SRAM. Il s'agit d'un dispositif numérique utilisant les mêmes éléments logiques que dans le processeur. Les valeurs binaires sont stockées en utilisant des bascules (*flip-flop*) logiques. Une cellule SRAM conserve sa donnée tant que la tension est maintenue. Il n'est pas nécessaire de faire des opérations de rafraîchissement.

Dans l'exemple de la figure, quatre transistors (T_1 , T_2 , T_3 et T_4) sont interconnectés d'une manière telle qu'il en résulte un état logique stable. Dans l'état logique 1, le point C_1 est à un niveau haut et le point C_2 est à un niveau bas. Dans cet état, les transistors T_1 et T_4 ne sont pas passants, tandis que T_2 et T_3 le sont. Dans l'état logique 0, toutes les configurations sont inversées. Comme pour la DRAM, la ligne d'adresse est utilisée pour accéder à une cellule. Lors de l'écriture, la valeur du bit à écrire est placée sur la ligne B et son complément sur la ligne \bar{B} . Lors de la lecture, la valeur du bit est lue depuis la ligne B .

À la fois la DRAM et la SRAM sont volatiles, mais les DRAM sont plus simples et donc plus denses. Par contre, la DRAM nécessite

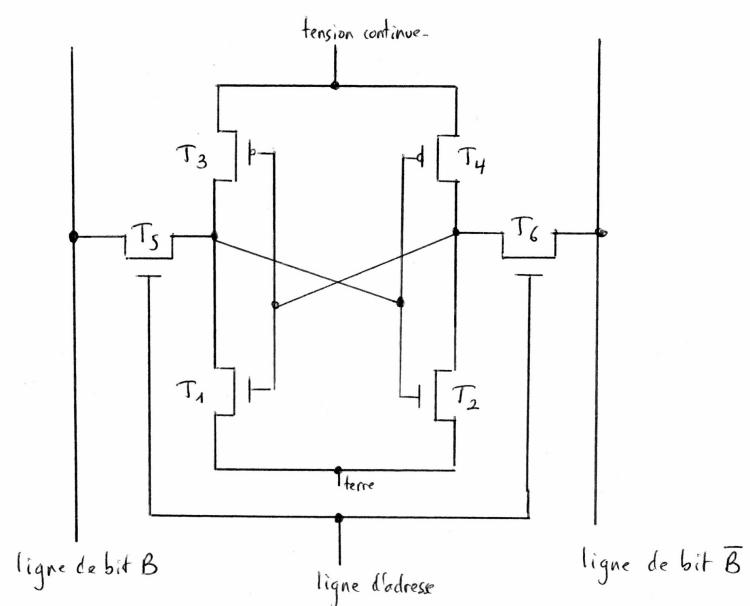


FIGURE 7.11 – Cellule de RAM statique (SRAM).

un circuit de rafraîchissement. La SRAM est un peu plus rapide que la DRAM. Pour toutes ces raisons, la SRAM est utilisée pour la mémoire cache et la DRAM pour la mémoire principale.

7.2.1 ROM

La ROM (*Read Only Memory*) contient des informations qui ne peuvent pas être changées. On ne peut pas écrire dans la ROM. De plus, la ROM n'est pas volatile.

Les applications de la ROM sont la microprogrammation, les routines de bibliothèques, les programmes systèmes, les tables de fonctions, etc.

Une ROM est créée comme n'importe quel autre circuit intégré, avec les données câblées dans la puce lors de la fabrication. On n'a pas droit à l'erreur lors de la fabrication de la ROM. Si un seul bit est erroné, toute la série de ROMs doit être mise au rebut.

On peut implémenter une ROM avec des circuits combinatoires. Considérons la table de vérité de la figure 7.12.

Entrée				Sortie			
X_1	X_2	X_3	X_4	Z_1	Z_2	Z_3	Z_4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

FIGURE 7.12 – La table de vérité d'une ROM.

On peut voir cette table comme ayant quatre entrées et quatre sorties. Mais on peut aussi la voir comme définissant le contenu d'une ROM de 64 bits consistant en 16 mots de 4 bits chacun. Les quatre entrées spécifient une adresse et les quatre sorties spécifient les contenus de l'emplacement spécifié par l'adresse.

La figure 7.13 montre comment cette ROM pourrait être implémentée avec un décodeur 4-vers-16.

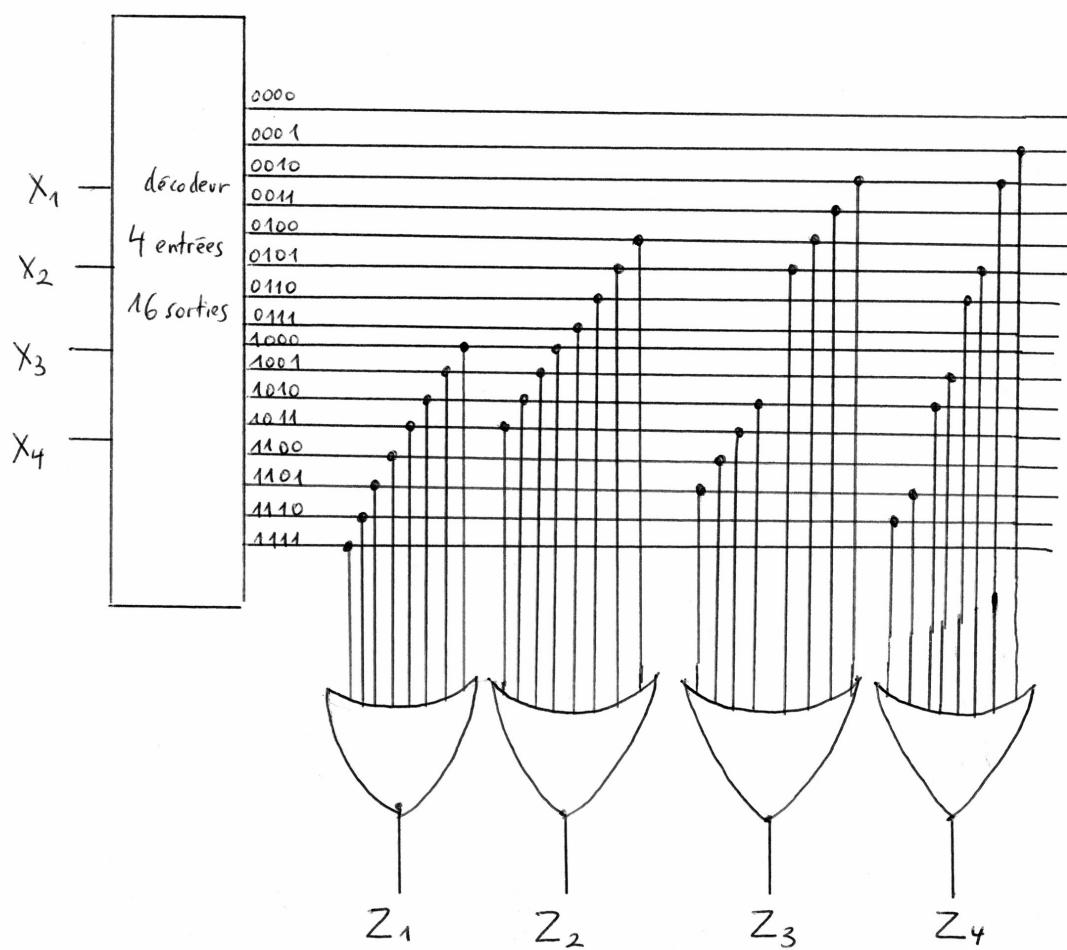


FIGURE 7.13 – Une ROM 64-bits.

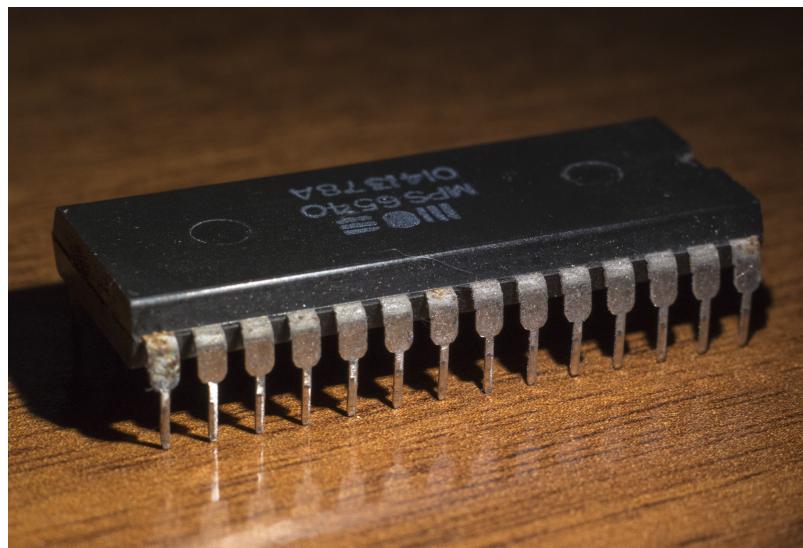


FIGURE 7.14 – Un circuit de ROM. (source : Wikipédia)

Il y a d'autres variantes plus flexibles, comme la PROM, l'EPROM, l'EEPROM ou la mémoire flash, qui est intermédiaire entre l'EPROM et l'EEPROM.

7.2.2 Puces

La mémoire est disponible sous forme de puces et chaque puce contient un tableau de cellules mémoires. Ces puces peuvent avoir différentes organisations et un paramètre clé est le nombre de bits de données pouvant être lus/écrits à un instant donné. Par exemple, une puce de 16 Mibits (16×2^{20}) pourrait être organisée en 2^{20} (1048576) mots de 16 bits.

La figure 7.15 montre l'organisation typique d'une DRAM 16Mi-bits. Dans ce cas, quatre bits (D_1, D_2, D_3 et D_4) sont lus ou écrits à la fois. De manière logique, la mémoire est organisée en quatre tableaux de 2048×2048 éléments. Une adresse de ligne sur 11 bits ($2^{11} = 2048$) est fournie (A_0 à A_{10}) et est utilisée pour sélectionner une ligne donnée dans chacun des quatre tableaux. D'autre part, une autre adresse est utilisée pour sélectionner une colonne. Il y a donc quatre bits (un par tableau) qui sont sélectionnés. Ces quatre bits peuvent être lus ou écrits.

Comme seulement quatre bits sont lus ou écrits avec cette DRAM, il doit y avoir plusieurs DRAM connectés au contrôleur de mémoire pour écrire un mot sur le bus.

On remarquera qu'il n'y a que onze lignes d'adresses et non vingt-deux, les lignes et colonnes étant sélectionnées les unes après les autres.

Lors du rafraîchissement, les données sont lues et réécrites au même endroit et la DRAM est inactivée pendant ce temps.

Un circuit intégré est monté dans une boîte qui contient des connecteurs vers le monde extérieur. La figure 7.17 montre un exemple de boîtier d'EPROM pour stocker 1Mi mots de 8 bits. Les adresses de ces mots sont données sur les connecteurs A_0 à A_{19} ($2^{20} = 1\text{Mi}$). Les données à lire sont D_0 à D_7 . Le connecteur CE (*Chip Enable*) permet de sélectionner ou inhiber cette puce au cas où plusieurs puces sont reliées au même bus.

La figure 7.19 illustre une configuration typique de connecteurs DRAM pour une puce de 16 Mbit organisée en $4\text{Mi} \times 4$. Parmi les différences avec une ROM, il y a le fait que les connecteurs de don-

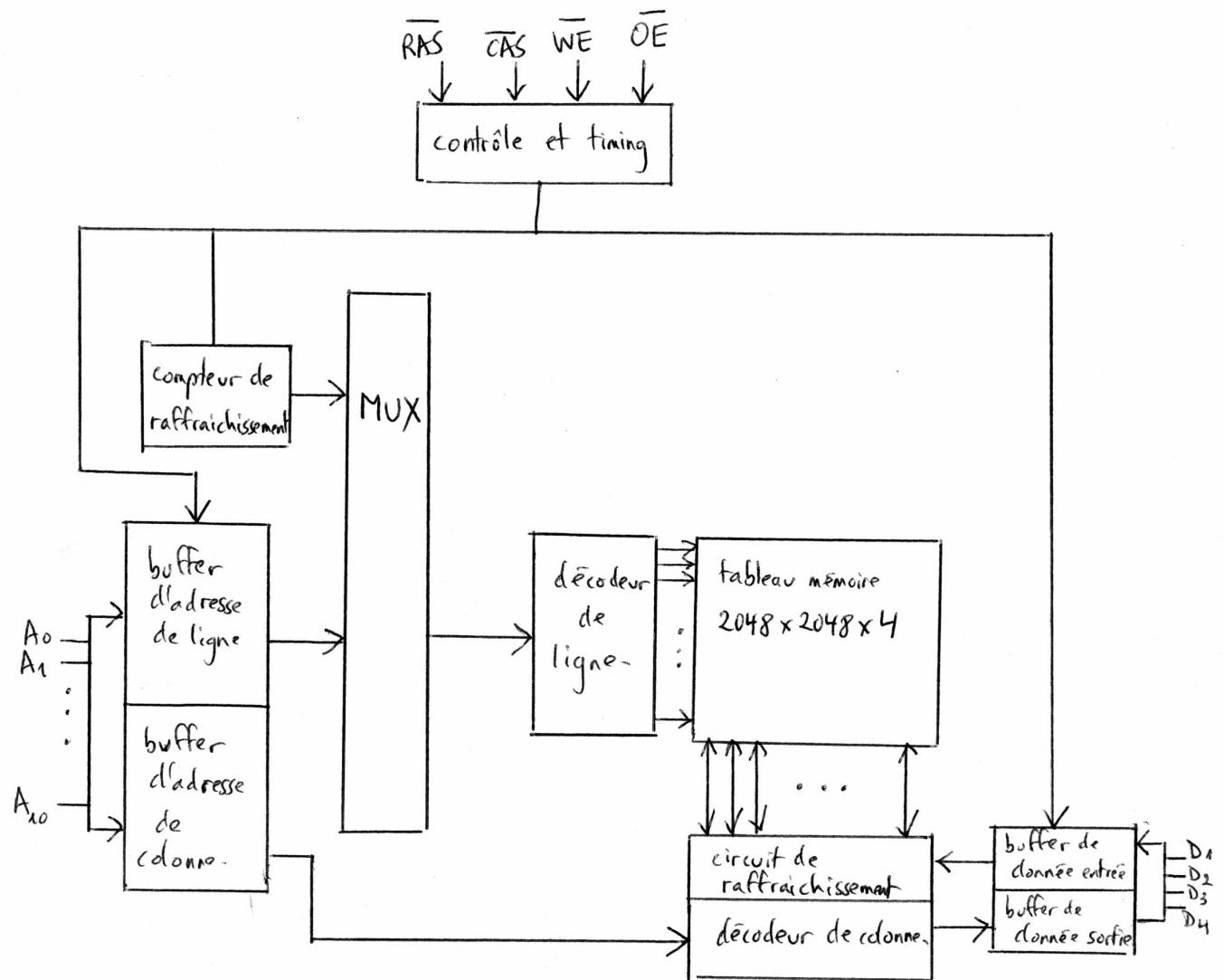


FIGURE 7.15 – DRAM 16Mbit typique (4Mx4)

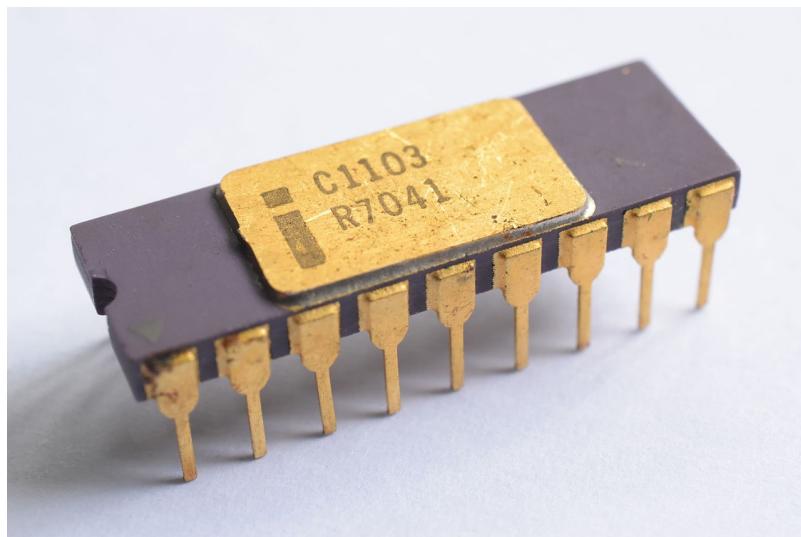


FIGURE 7.16 – Intel 1103, premier circuit intégré DRAM (1970). Capacité : 1 Kibit. (source : Wikipédia)

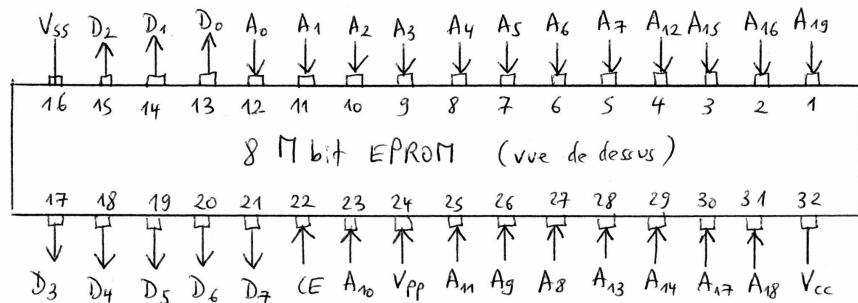


FIGURE 7.17 – EEPROM 8-Mbit typique.



FIGURE 7.18 – Exemple d'EPROM 8 Mbit.
([https://www.pinball.center/en/shop/electronics-parts/
ic/1226/ic-eprom-8-mbit](https://www.pinball.center/en/shop/electronics-parts/ic/1226/ic-eprom-8-mbit))

nées (D_0 , D_1 , D_2 et D_3) sont en entrée et sortie. Les connecteurs WE (Write Enable) et OE (Output Enable) indiquent si l'opération est une opération d'écriture ou de lecture. Comme plus haut, il n'y a que onze lignes d'adresse car les lignes et colonnes sont sélectionnées successivement.

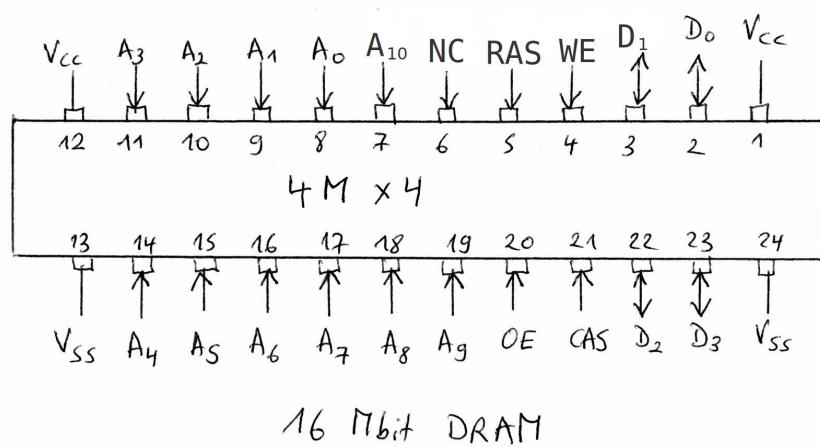


FIGURE 7.19 – DRAM 16-Mbit typique.

Si une puce de RAM ne contient qu'un bit par mot, alors il faut au moins autant de puces que le nombre de bits par mot. Par exemple, la figure 7.20 montre comment un module de mémoire de 256Ki de mots de 8 bits pourrait être organisée.

Pour 256Ki mots, il faut 18 bits d'adresse ($2^{18} = 256\text{Ki}$). Cette adresse est stockée dans un registre d'adresse mémoire (MAR). L'adresse est présentée à huit puces de $256\text{Ki} \times 1$ bit, chacune fournissant l'entrée/sortie d'un bit. Les 18 bits sont séparés en deux, 9 bits pour une ligne et 9 bits pour une colonne. Au final, huit bits peuvent être extraits.

La figure 7.21 illustre une organisation possible d'une mémoire consistant en 1Mi de mots de 8 bits. Dans ce cas, il y a quatre colonnes de puces, chaque colonne contenant 256Ki rangés comme dans la figure 7.20. Les adresses sont alors sur 20 bits. 18 de ces bits sont utilisés comme précédemment pour sélectionner les lignes et colonnes. Les deux derniers bits servent à sélectionner l'une des quatre colonnes de puces.

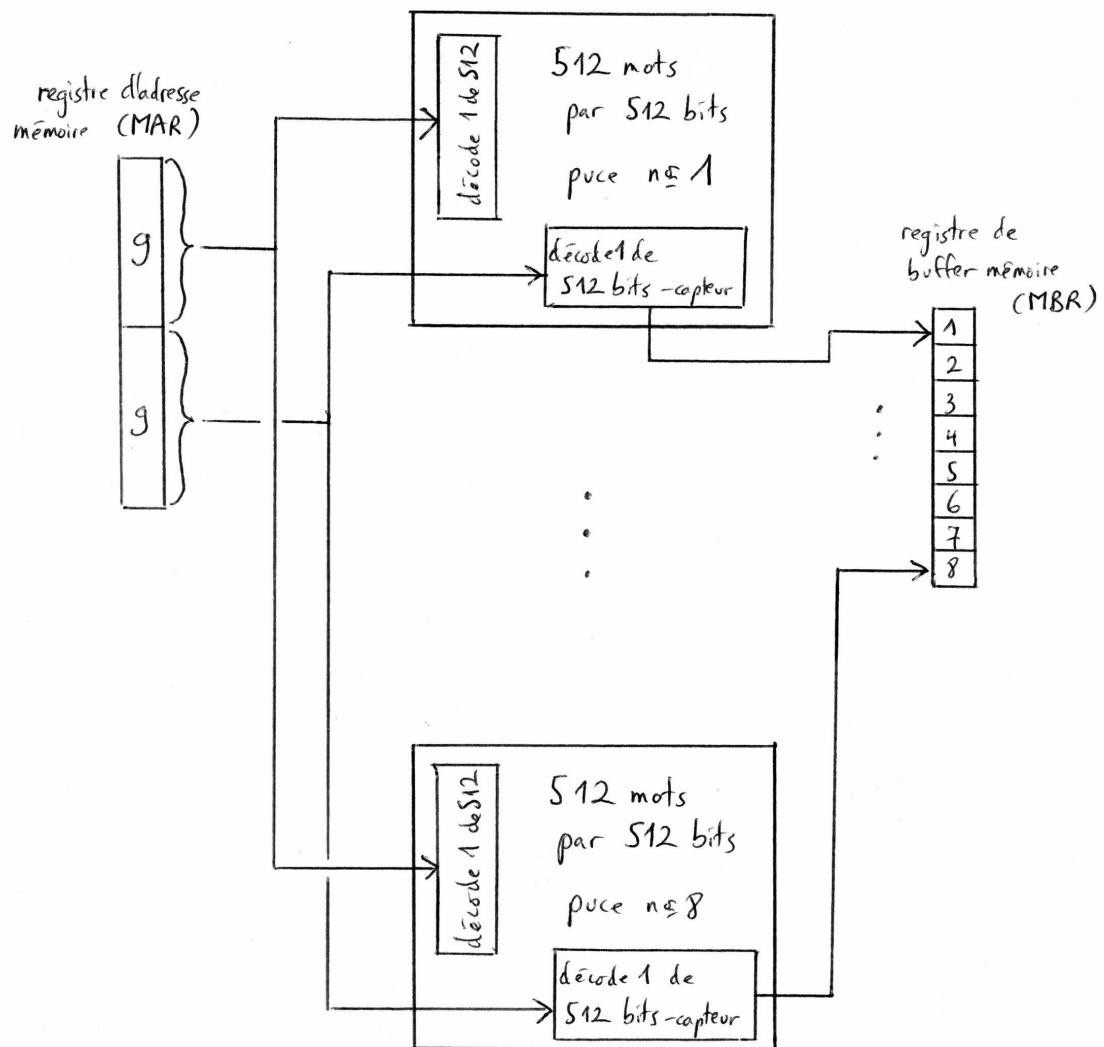


FIGURE 7.20 – Organisation d'une mémoire de 256 Kio avec 8 puces de 256K × 1 bit.

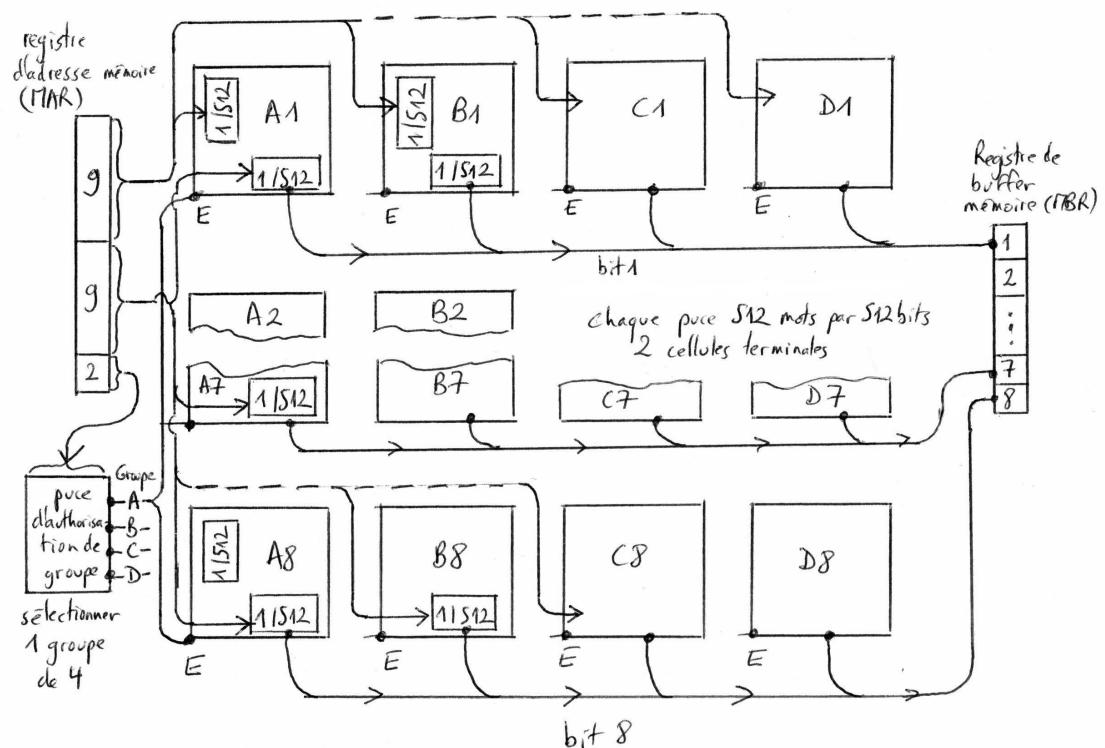


FIGURE 7.21 – Organisation d'une mémoire de 1 Mi.

7.2.3 Correction des erreurs

Cette section n'a pas encore été rédigée.

7.2.4 Améliorations de la DRAM

La DRAM n'a pas beaucoup évolué depuis le début des années 1970. Pour en améliorer les performances, on a vu qu'il était possible d'ajouter de la SRAM en cache entre la mémoire principale DRAM et le processeur. Mais la SRAM est plus coûteuse que la DRAM.

Ces dernières années, il y a eu de nouveaux types d'améliorations à la DRAM et notamment la SDRAM et la DDR-SDRAM.

La SDRAM (*Synchronous DRAM*) échange des données avec le processeur en étant synchronisée avec un signal d'horloge.

Dans une DRAM typique, lorsque le processeur présente une adresse à la mémoire afin de lire ou écrire des données, il y a un délai — le temps d'accès — avant la lecture ou l'écriture. Pendant ce délai, la DRAM accomplit certaines tâches comme l'activation des lignes ou colonnes. Dans une SDRAM, les données sont déplacées sous le contrôle d'une horloge. Le processeur peut sans risque réaliser d'autres tâches pendant que la SDRAM répond à une demande.

Dans les DDR SDRAM (*Double Data Rate SDRAM*) (figure 7.22) les transferts sont synchronisés sur les fronts montants et descendants de l'horloge et pas uniquement sur les fronts montants. Cela permet de doubler la vitesse de transmission. D'autres améliorations permettent encore de rendre la DDR SDRAM plus efficace. Il y a en particulier un mécanisme de *buffers*, aboutissant aux versions DDR2 SDRAM, DDR3 SDRAM, DDR4 SDRAM et DDR5 SDRAM.

7.2.5 Mémoires flash

Ces mémoires ont été introduites au milieu des années 1980. La mémoire flash utilise un transistor par bit et permet d'avoir une



FIGURE 7.22 – DDR SDRAM 64 Mbit. (source : Wikipédia)

grande densité. Il y a deux types différents de mémoire flash, NOR et NAND.

7.3 Mémoire externe

7.3.1 Disque magnétique

Un disque est un plateau formé de matériau non magnétique, appelé le substrat, recouvert de matière magnétique. Traditionnellement, le substrat a été en aluminium ou dans un alliage d'aluminium. Plus récemment, on a introduit des substrats en verre.

Capacité	Année	Fabricant	Modèle	Taille
5 Mo	1956	IBM	350 Ramac ³	24"
28 Mo	1962	IBM	modèle 1301	
1,02 Go	1982	Hitachi ¹⁵	H8598	
25 Go	1998	IBM	Deskstar 25 GP	
500 Go	2005	Hitachi		
1 To	2007	Hitachi	Deskstar 7K1000 ¹⁶	
2 To	2009	Western Digital ¹⁷	Caviar Green WD20EADS	
3 To	2010	Seagate		
4 To	2011	Hitachi ¹⁸	7K4000	
6 To	2013	HGST ¹⁹	WD Red Pro	
8 To	2014	Seagate ²⁰	Archive HDD	
10 To	2015	HGST	Ultrastar He10 ²¹	
14 To	2018	Seagate	Exos X14 ²²	
16 To	2019	Seagate	Exos X16 ²³	
18 To	2020	Seagate	Exos X18 ²⁴	
26 To	2022	Western Digital	Ultrastar ²⁵	

FIGURE 7.23 – Évolution des capacités des disques durs. (source : Wikipédia)

7.3.2 Disquette (1967-2011)

Une disquette est un support de stockage de données informatiques amovible. La disquette est aussi appelée disque souple (*floppy*

disk en anglais) en raison de la souplesse des premières générations (8 et 5,25 pouces) et par opposition au disque dur. (Wikipédia)

Une disquette 3,5 pouces permettait typiquement de stocker 1.44 Mio.

7.3.2.1 Mécanismes de lecture et d'écriture

Les données sont enregistrées et plus tard extraites du disque par l'intermédiaire d'une bobine conductrice appelée la *tête*. Dans beaucoup de systèmes, il y a deux têtes, une de lecture et une d'écriture. Pendant une opération de lecture ou d'écriture, la tête ne bouge pas tandis que le plateau sous la tête tourne.

Le principe de l'écriture repose sur le fait que l'électricité passant dans la bobine produit un champ magnétique. Des impulsions électriques sont envoyées dans la tête d'écriture et les motifs magnétiques qui en résultent sont enregistrés sur la surface sous-jacente, avec différents motifs pour les courants positifs et négatifs. La tête d'écriture est faite de matière facilement magnétisable et a la forme d'un rectangle avec une ouverture d'un côté et quelques tours d'un fil conducteur du côté opposé. Elle forme donc un électroaimant. Un courant électrique dans le fil produit un champ magnétique au niveau de l'entrefer (ouverture), ce qui magnétise une petite zone du support d'enregistrement.

La lecture exploite le fait qu'un champ magnétique se déplaçant par rapport à une bobine produit un courant dans la bobine. La structure de la tête est dans ce cas essentiellement la même que pour l'écriture et la même tête peut être employée pour la lecture et l'écriture. De telles têtes sont utilisées dans les systèmes à disques souples (*floppy disks*) et dans les systèmes à disques rigides plus anciens.

Les systèmes à disque rigide actuels utilisent un mécanisme de lecture différent, utilisant une tête de lecture séparée placée près de la tête d'écriture. Cette tête de lecture consiste en un capteur magnétorésistif partiellement protégé. La résistance de ce capteur dépend de la direction de la magnétisation du support sous-jacent



FIGURE 7.24 – Les anciens formats de disquettes. (source : Wikipédia)

et les changements de résistance sont détectés comme des signaux de tension. Ce dispositif permet des opérations de plus grande fréquence, et donc de plus grandes capacités de stockage et de vitesses d'opérations.

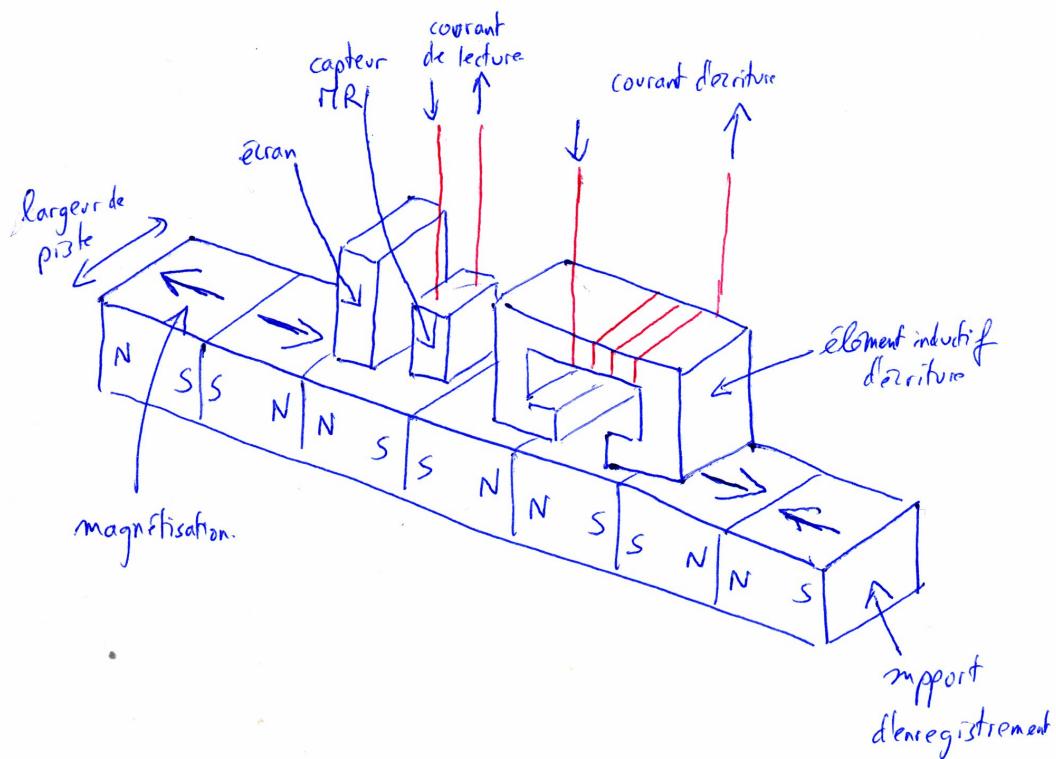


FIGURE 7.25 – Tête de lecture magnétoresistive et d'écriture inductive.



FIGURE 7.26 – Tête de lecture/écriture d'un disque Seagate de 1998.
(source : Wikipédia)

7.3.2.2 Organisation des données

Les données sur un disque sont organisées en pistes et en secteurs. Dans une portion donnée d'une piste, la densité des bits varie en fonction de la distance au centre, il y a donc plus de bits par distance près du centre qu'à la périphérie. Le nombre de bits que l'on récupère par unité de temps ne dépend donc pas de la distance au centre.

Cependant, les parties périphériques d'un disque ne sont alors pas utilisées à leur densité maximale. Les disques modernes utilisent un système appelé « enregistrement à zones multiples » (*multiple zone recording*, MZR) où la surface est divisée en un certain nombre de zones concentriques, typiquement 16, et avec une densité de bits par unité constante dans une zone. Cette densité n'est donc pas constante sur l'ensemble des pistes, mais seulement sur les pistes d'une zone. La conséquence est bien sûr que les secteurs ne sont plus alignés sur l'ensemble du disque.

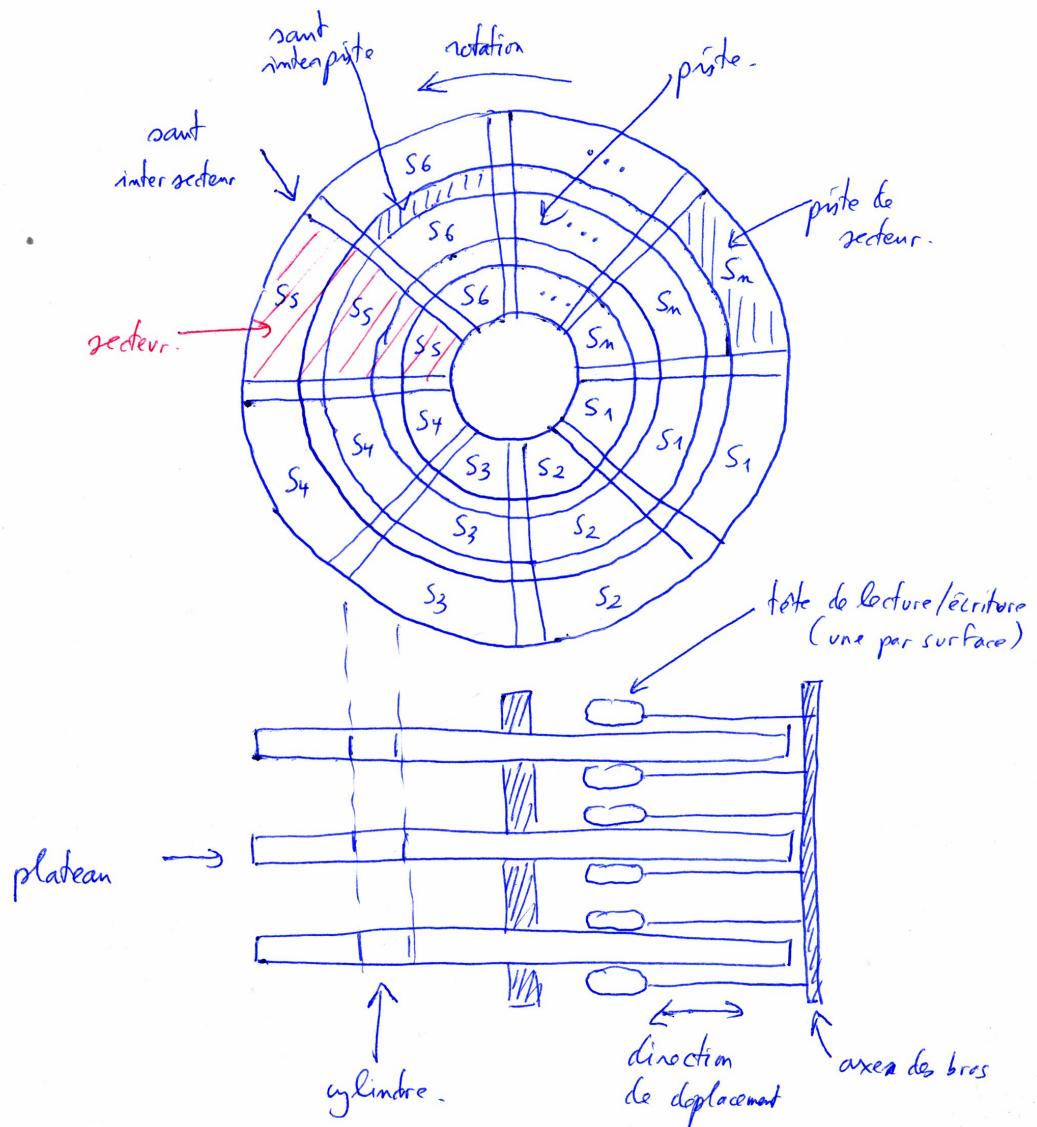


FIGURE 7.27 – Organisation des données sur un disque.

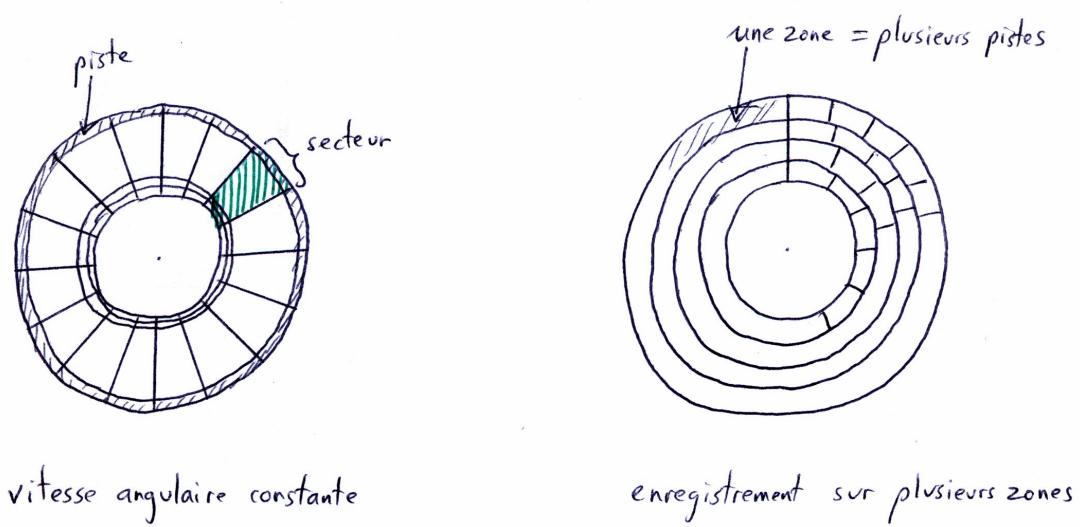


FIGURE 7.28 – Comparaison des organisations de disques.

7.3.3 SSD

Un SSD (*Solid-State Drive*, quelquefois improprement appelé « disque SSD » en français) permet le stockage de données sur de la mémoire flash. Les SSDs sur le marché utilisent de la mémoire flash NAND. Ces disques sont de plus en plus compétitifs par rapport aux disques magnétiques. Un SSD n'a aucune pièce en mouvement et ce n'est pas un disque.



FIGURE 7.29 – Un SSD de 2,5 pouces (6,35 cm). (source : Wikimedia, Vertex_2_Solid_State_Drive_by_OCZ-top_oblique_PNr°0307.jpg)

Les SSDs présentent cependant deux problèmes que l'on n'a pas avec les disques magnétiques :

- tout d'abord, en raison de la fragmentation des fichiers, la performance des SSDs a tendance à se ralentir lors de l'utilisation ;
- de deuxièmement, la mémoire flash devient inutilisable après un certain nombre d'écritures, typiquement 100000.

7.3.4 Mémoire optique

Le CD audio a été introduit en 1983. Les CDROM utilisent une technologie similaire.

Le CD enregistrable (CD-R) peut être écrit une fois, puis lu de nombreuses fois.

Le CD réenregistrable (CR-RW) peut être récrit de nombreuses fois, mais la réécriture est limitée à 500000 à un million de fois.

Les DVD permettent de stocker d'encore plus grandes capacités, jusqu'à 17 Gio en utilisant les deux faces, alors qu'une seule face peut être utilisée sur un CD.

Les disques Blu-Ray peuvent stocker 25 Gio sur une seule face.

7.3.5 Bande magnétique

Les bandes magnétiques sont un support de stockage séquentiel. Autrefois, les bandes se présentaient sous forme de disques qui devaient être déroulés sur d'autres disques. Aujourd'hui, les bandes se présentent sous forme de cartouches, un peu comme les anciennes cassettes audio. Le format actuel (2023) le plus courant est le format LTO. Une cassette LTO-8 (960 m) peut stocker jusqu'à 30 Tio.



FIGURE 7.30 – Un ensemble d'armoires de bandes magnétiques pour un ordinateur IBM 729 en 1969. (source : Wikipédia)



FIGURE 7.31 – Une bande LTO-8.
(source : [https://www.backupworks.com/
Fujifilm-LTO-8-Tape-Media-Data-Sheet.aspx](https://www.backupworks.com/Fujifilm-LTO-8-Tape-Media-Data-Sheet.aspx))

7.4 Un peu d'histoire

Core memory : <https://www.youtube.com/watch?v=AwsInQLmjXc>

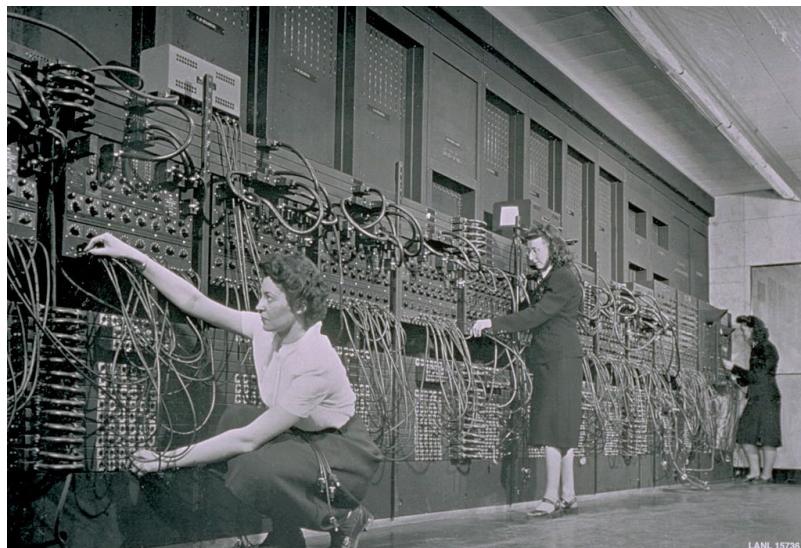


FIGURE 7.32 – La programmation de l'ENIAC. Cet ordinateur n'avait pas vraiment de mémoire, mais 20 accumulateurs. Chaque accumulateur pouvait stocker un nombre décimal de 10 chiffres.

7.5 Suppléments

- How a 1-BIT Memory Works :
https://www.youtube.com/watch?v=nL7d2_4TYZk
- How does computer memory work :
<https://www.youtube.com/watch?v=7J7X7aZvMXQ>
- Fonctionnement de la DRAM (1/2) :
https://www.youtube.com/watch?v=I-9XWtdW_Co
- Fonctionnement de la DRAM (2/2) :
<https://www.youtube.com/watch?v=x3jGq0rXXc8>
- What is SRAM? :
<https://www.youtube.com/watch?v=kU2SsUUuftA>
- SRAM :
https://www.youtube.com/watch?v=GBL28_Tw6UQ

7.6 Exercices

1. Faites le schéma d'un sélecteur d'adresse sur trois bits.
2. Quelle doit être la taille minimale d'un bus d'adresse si on a :
 - 2 Gio de mémoire avec un bus de donnée de 16 bits ?
 - 8 Gio de mémoire avec un bus de donnée sur 32 bits ?
 - 32 Gio de mémoire avec un bus de donnée sur 64 bits ?
3. Quelle serait la taille d'une mémoire avec :
 - un bus d'adresse sur 20 bits et un bus de donnée de 1 octet ?
 - un bus d'adresse sur 32 bits et un bus de donnée sur 1 octet ?
 - un bus d'adresse sur 32 bits et un bus de donnée sur 4 octets ? Combien faudrait-il de barrettes mémoire de 2 Gio ?
4. Considérez une RAM dynamique qui doit être raffraîchie 64 fois par ms. Chaque opération de raffraîchissement prend 150 ns. Un cycle mémoire demande 250 ns. Quel pourcentage du temps consacré à la mémoire doit-il être donné aux raffraîchissements ?
5. la figure 7.21 indique comment construire un module de puces pouvant stocker 1 Mio à partir d'un groupe de quatre puces de 256 Kio. Disons que ce module de puces est packagé en une unique puce de 1 Mio, où la taille d'un mot est de 1 octet. Donnez un diagramme de haut niveau sur la manière de construire une mémoire de 8 Mio en utilisant 8 puces de 1 Mio. Ne pas oublier de montrer les lignes d'adresses dans votre diagramme et à quoi servent les lignes d'adresses.
6. La mémoire d'un certain microordinateur est construite à partir de DRAMs $64\text{ K} \times 1$. D'après la fiche technique, le tableau des cellules de la DRAM est organisé en 256 lignes. Chaque ligne doit être raffraîchie au moins toutes les 4 ms. Supposons que l'on raffraîchisse la mémoire de manière périodique.

- quelle est la période de temps entre deux demandes de raffraîchissements successifs ?
 - de quelle taille de compteur de raffraîchissement faut-il disposer ?
7. Concevez une mémoire 16-bits de capacité totale 8192 bits en utilisant des puces SRAM de taille 64×1 bit. Donnez la configuration du tableau des puces en montrant toutes les entrées requises et signaux de sortie. La conception devrait permettre à la fois des accès par mots d'un octet et de 16 bits.
8. On suppose qu'une unité de disque a 24 surfaces d'enregistrement. Il y a 14000 cylindres au total et 400 secteurs par piste. Chaque secteur contient 512 octets de données.
- quel est le nombre maximal d'octets qui peuvent être stockés dans cette unité ?
 - quel est le taux de transfert de données en octets par seconde avec une vitesse de 7200 tours/mn ?
9. la commande `free` affiche des informations sur la mémoire vive et la zone d'échange (*swap*) (qui sera décrite au chapitre 8); un affichage possible est

	total	used	free	shared	buff/cache	available
Mem:	16251604	5106212	575532	673960	10569860	10137876
Swap:	2097148	305944	1791204			

Sur cette machine, la mémoire vive occupe 16 Gio. Qu'en est-il sur votre machine ?

10. on peut lancer la commande `free` avec les options suivantes :
- `free -c N` pour lancer la commande *N* fois
ou encore
`free -s M` pour imposer un délai de *M* secondes entre deux lancements successifs.
- Lancez la commande `free` toutes les secondes, pendant dix secondes et observez ce qui bouge et ce qui ne bouge pas.

11. Écrivez un programme C déclarant plusieurs variables de différents types et affichez les adresses de ces variables. Lancez votre programme à plusieurs reprises. Qu'observez-vous ?

Chapitre 8

Gestion de processus

Nous donnons ici quelques éléments sur la gestion des processus par un système d'exploitation. Les questions fondamentales sont celle de la multiprogrammation, c'est-à-dire de la coexistence de plusieurs programmes simultanés, ou plutôt de plusieurs processus simultanés, et celle de l'exécution de grands programmes par rapport à la mémoire vive disponible.

8.1 Ordonnancement

Pour pouvoir exécuter plusieurs programmes simultanément, il importe de les *ordonner*. Il y a différents types d'ordonnancements :

ordonnancement à long terme : il s'agit du choix des processus devant être exécutés ;

ordonnancement à moyen terme : il s'agit du choix des processus qui sont partiellement ou entièrement en mémoire principale ;

ordonnancement à court terme : il s'agit du choix du processus disponible qui va être exécuté ;

ordonnancement d'entrée/sortie : il s'agit du choix de la demande d'entrée/sortie à exécuter.

Nous ne détaillons pas tous ces types, mais la figure 8.1 donne une idée de ces ordonnancements. Le système d'exploitation maintient un certain nombre de files d'attente. Chaque file est simplement une file d'attente de processus attendant une certaine ressource. La *file à long terme* est une liste de travaux attendant d'utiliser le système. Si les conditions sont satisfaites, l'ordonnanceur de haut niveau va allouer de la mémoire et créer un processus pour l'un des travaux en attente. La *file à court terme* consiste en tous les processus qui sont prêts à s'exécuter. C'est l'ordonnanceur à court terme qui choisit celui des processus qui va s'exécuter. En général, chaque processus a droit à son tour. Enfin, il y a la file d'attente d'entrée/sortie pour chaque périphérique. Plus d'un processus peut demander d'utiliser un même périphérique.

Le processeur alterne entre l'exécution d'instructions du système d'exploitation et l'exécution de processus utilisateur.

Quand un processus est en train de s'exécuter, il peut être interrompu pour diverses raisons. Cela peut être parce que le processus demande un accès d'entrée/sortie (par exemple un affichage), et dans ce cas il est placé dans la file d'attente correspondante. Il peut être interrompu en raison d'un *timeout* ou parce que le système d'exploitation a un travail urgent à réaliser. Il est alors mis dans l'état prêt et mis dans la file d'attente à court terme.

Le système d'exploitation gère aussi les files d'attente d'entrée et sortie. Quand une opération d'entrée/sortie est achevée, le système d'exploitation retire le processus qui a obtenu satisfaction de cette file et le met dans la file d'attente à court terme. Il sélectionne alors un autre processus en attente, le cas échéant, et demande au périphérique de satisfaire la demande du processus.

8.1.1 Swapping

Une solution pour éviter l'attente des processus est le *swapping* (figure 8.2). Dans un fonctionnement avec ordonnancement simple, on dispose d'une file d'attente à long terme pour les processus. Cette file d'attente est stockée sur le disque, c'est-à-dire que le code et l'es-

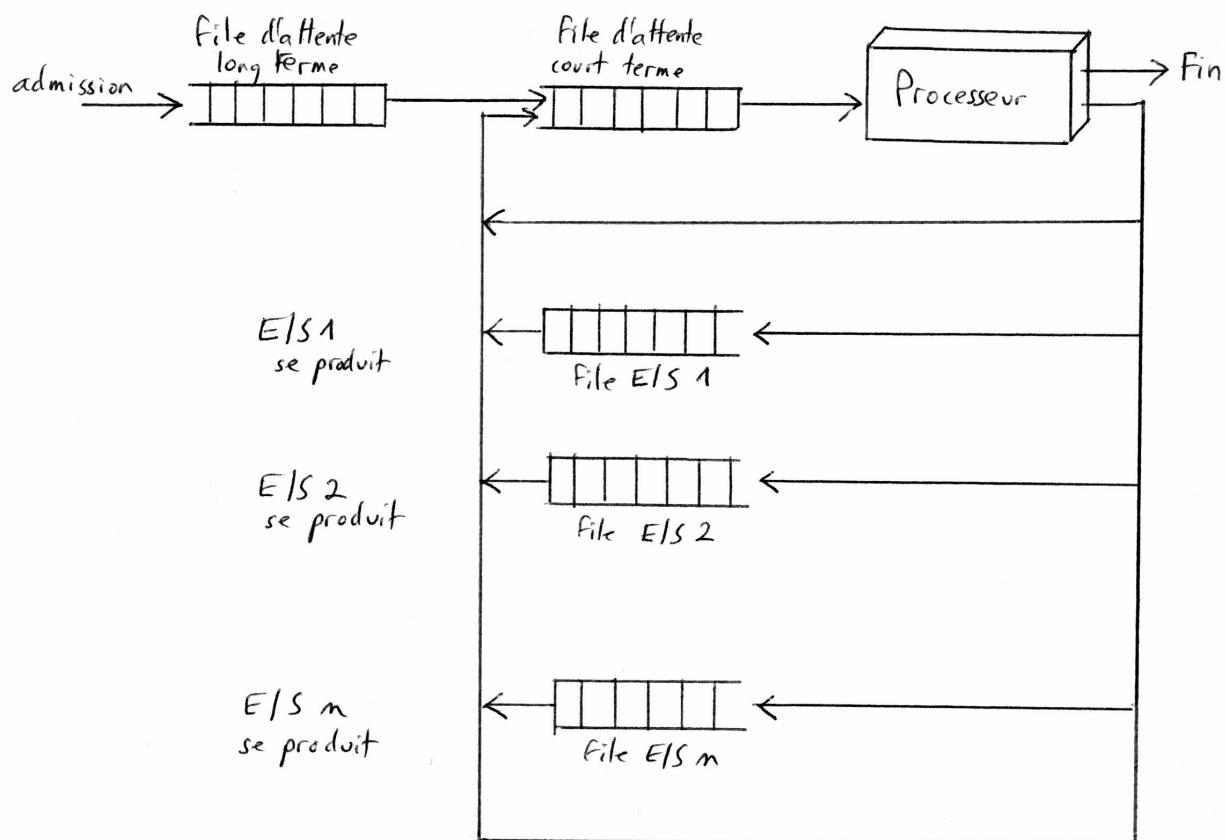


FIGURE 8.1 – Files pour l'ordonnancement des tâches pour un processeur.

pace mémoire d'un processus est stocké non pas en mémoire vive, mais sur le disque, en attendant d'être transférés vers la mémoire vive. Ces transferts se font au fur et à mesure, lorsque la mémoire devient disponible. Lorsque les processus ont fini leur travail, ils sont à nouveau retirés de la mémoire vive (et ne sont pas remis sur le disque, puisqu'ils ont fini d'exister).

Dans un fonctionnement avec *swapping* (*to swap* = échanger), si un processus est en attente d'une entrée/sortie (par exemple en attente d'une entrée clavier), ce processus peut être remis sur une file d'attente intermédiaire se trouvant sur le disque. Cela permet de libérer de la place et à un autre processus de s'exécuter. La zone où ce processus est temporairement stocké sur disque est appelée l'espace d'échange, ou le *swap*. Il peut s'agir d'une partition dédiée.

Le *swapping* est une opération d'entrée/sortie et potentiellement la situation peut empirer. En général, cependant, le *swapping* va améliorer les performances. Un mécanisme plus sophistiqué fait appel à la mémoire virtuelle et est examiné plus loin. Mais avant de l'étudier, il faut expliquer ce que sont le partitionnement et la pagination.

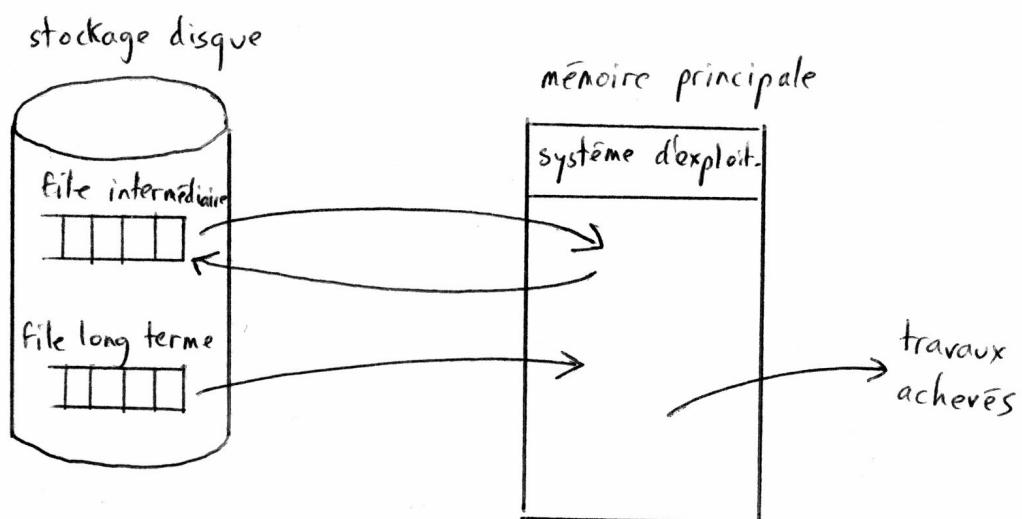
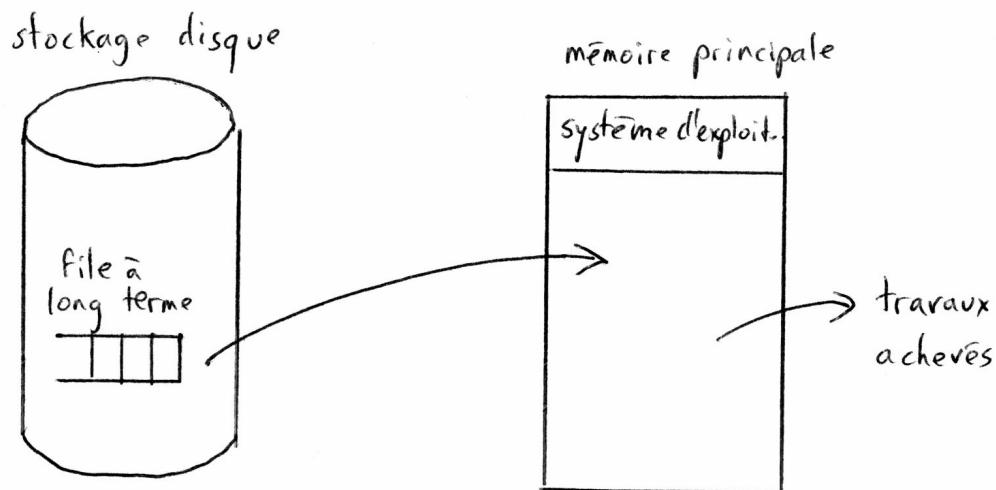


FIGURE 8.2 – En haut, un ordonnancement simple. En bas, l'emploi du *swapping*.

8.2 Gestion de la mémoire

Dans un système à un seul programme (monoprogramme), la mémoire vive est divisée en deux parties : une partie pour le système d'exploitation et une partie pour le programme qui est en train d'être exécuté. Dans un système multiprogrammé, la partie « utilisateur » est subdivisée pour gérer plusieurs processus. Cette subdivision est gérée dynamiquement par le système d'exploitation.

Une gestion efficace de la mémoire est vitale dans un système multiprogrammé. Si seulement quelques processus sont en mémoire, alors pendant une grande partie du temps tous les processus vont attendre des entrées/sorties et le processeur ne fera rien. La mémoire doit être allouée efficacement pour placer autant de processus que possible en mémoire.

Lorsque les processus ne sont pas en mémoire, ils doivent être stockés ailleurs, et ce sera par exemple sur un disque magnétique.

Les questions de gestion de mémoire étudiées ici se situent donc entre la mémoire principale et la mémoire externe et on peut voir la mémoire principale comme un cache vers la mémoire externe. De ce fait, les idées présentées ici sont très proches de celles déjà mentionnées dans le cadre de la mémoire cache, laquelle fait habituellement partie du processeur.

Au problème de la place limitée en mémoire pour y placer tous les processus s'ajoute le fait qu'il est utile de faire en sorte que chaque processus soit indépendant des autres et ait l'impression d'avoir tout l'espace mémoire à lui seul. Dans ce qui suit, nous allons d'abord voir différents mécanismes pour faire cohabiter les processus.

8.2.1 Partitionnement (ou segmentation)

Le but du partitionnement (ou de la segmentation) est de définir des zones dans la mémoire afin d'éviter le gaspillage de mémoire. Dans ces zones, on placera des processus. Un processus peut consister en un ou plusieurs segments. Il peut par exemple y avoir un segment pour des données, un autre pour les fonctions d'un programme, etc.

Le moyen le plus simple de partitionner la mémoire disponible est d'utiliser des partitions de taille fixe (figure 8.3). Cependant, même si les partitions ont une taille fixe, elles n'ont pas besoin d'avoir toutes la même taille.

Quand un processus est placé en mémoire, il est placé dans la plus petite partition disponible qui peut le contenir. Mais même avec des partitions fixes de tailles différentes, il y aura du gaspillage. Dans la plupart des cas, un processus n'aura pas besoin d'exactement autant de mémoire que fournie par la partition. Par exemple, un processus qui a besoin de 3Mio de mémoire serait placé dans la partition de 4Mio du partitionnement inégal de la figure 8.3. 1Mio sont alors perdus qui pourraient servir à un autre processus.

Une approche plus efficace est d'utiliser des partitions de taille variable. Quand un processus est placé en mémoire, on lui alloue exactement autant de mémoire dont il a besoin et pas plus (figure 8.4).

Cependant, même si cette méthode commence bien, elle conduit au bout d'un certain temps à une situation avec beaucoup de petits trous en mémoire, c'est la fragmentation externe (en dehors des segments). Une solution est de faire du compactage. De temps à autre, le système d'exploitation déplace les processus en mémoire pour mettre toute la mémoire disponible dans un seul bloc. Cette procédure est coûteuse en temps.

Par ailleurs, quand un processus sort et rentre à nouveau en mémoire, du fait du *swapping*, il est rarement replacé au même endroit. Cela pourrait donc poser un problème pour les références vers la mémoire, à savoir les adresses des données et les adresses des ins-

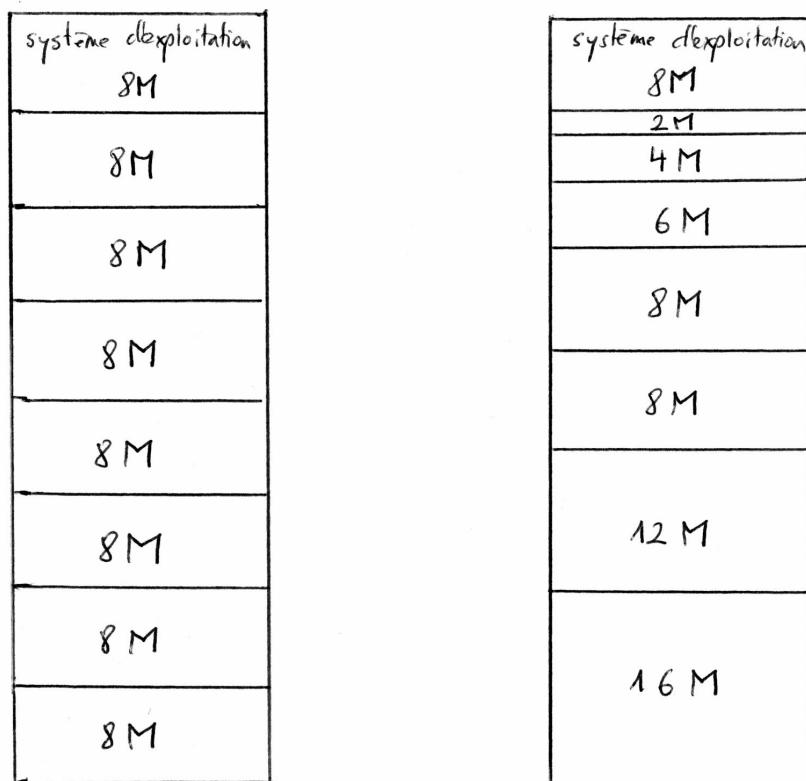


FIGURE 8.3 – Exemples de partitions fixes dans une mémoire de 64 Mi-octets. À gauche, partitions égales. À droite, partitions inégales.

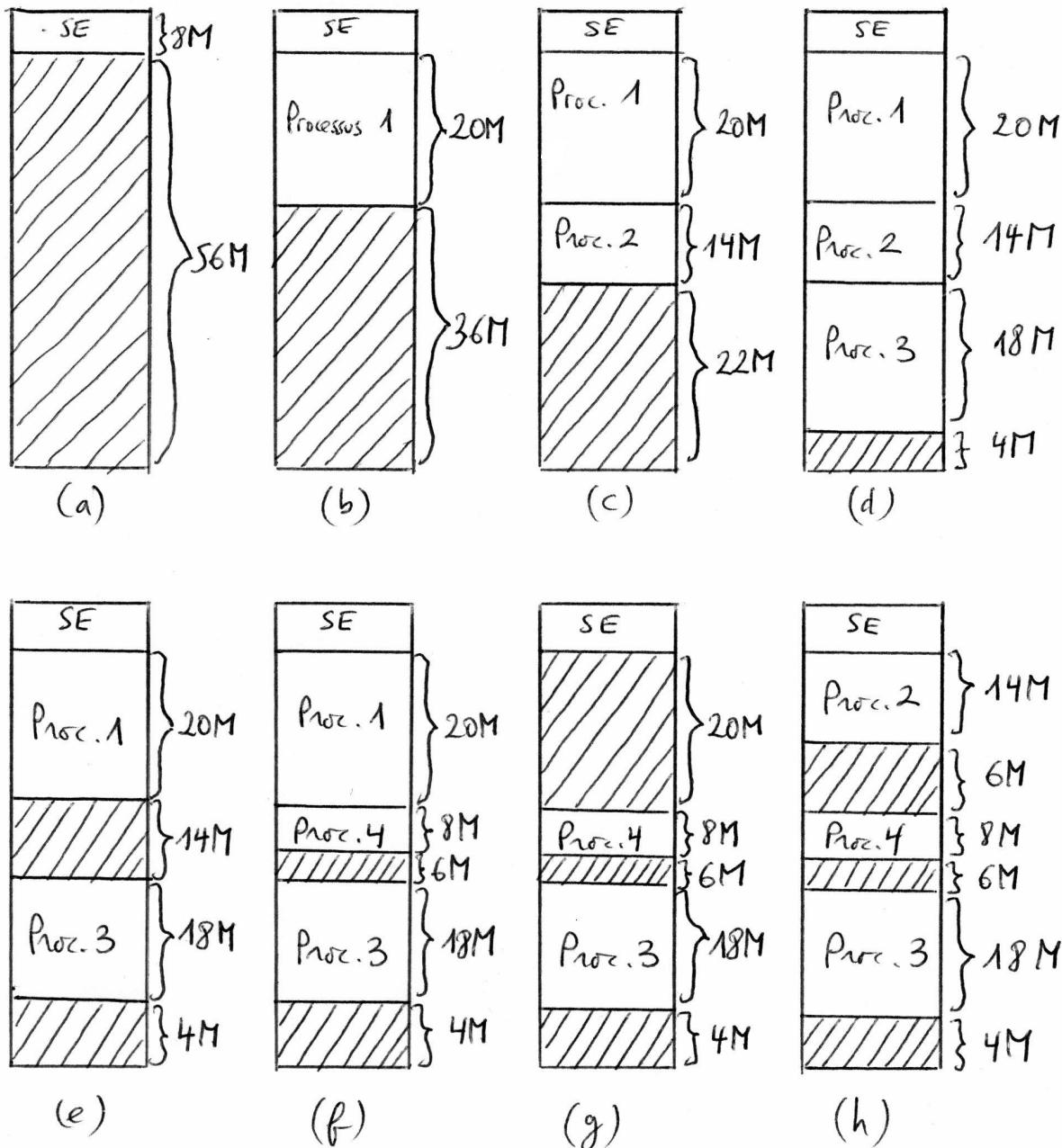


FIGURE 8.4 – L'effet du partitionnement dynamique.

tructions.

Pour résoudre ce problème, on distingue les adresses logiques et les adresses physiques.

Une *adresse logique* est exprimée comme une position relative au début du programme. Les instructions dans un programme ne contiennent que des adresses logiques.

Une *adresse physique* est un emplacement réel en mémoire.

Quand le processeur exécute un processus, il convertit automatiquement les adresses logiques en adresses physiques en ajoutant à chaque adresse logique l'adresse actuelle de début du processus, appelée *l'adresse de base*.

La traduction des adresses logiques en physiques sera examinée un peu plus loin.

8.2.2 Pagination

Les formes de partitionnement fixe et dynamique que nous venons de voir ne sont pas suffisantes parce que les processus sont toujours entièrement en mémoire et empêchent d'autres processus d'être chargés. La *pagination* permet de découper la mémoire et les processus en blocs de même taille et de ne charger que certains blocs de processus en mémoire. Les blocs de programme sont appelés *pages*, tandis que les blocs de mémoire disponibles sont appelés *cadres* ou *frames*. Au pire, l'espace perdu en mémoire en raison d'un processus est une fraction de la dernière page ou cadre (fragmentation interne).

La figure 8.5 montre un exemple d'utilisation de pages et de cadres. À gauche, on a la situation de la mémoire avant le chargement du processus A. Il y a des cadres libres (13, 14, etc.) et des cadres utilisés (16, 17 et 19). Le système d'exploitation maintient une liste des cadres libres, ici 13, 14, 15, 18 et 20. Le processus A, stocké sur disque, consiste en quatre pages. Lorsque le moment vient de charger le processus en mémoire, le système d'exploitation trouve quatre cadres libres et charge les quatre pages du processus A dans ces quatre cadres.

Si le système d'exploitation ne trouve pas quatre cadres contigus, le processus peut tout de même être chargé, mais les pages seront séparées. La partie droite de la figure 8.5 montre une telle situation où la première page du processus a été placée dans le cadre 18, et les trois autres à partir du cadre 13. Le système d'exploitation n'utilise alors pas une unique adresse de base, mais chaque page du processus a une adresse de base. Il y a une table des pages pour chaque processus et cette table indique où en mémoire se trouve la page correspondante.

La figure 8.6 montre comment se font les conversions entre adresses logiques et physiques dans ce cas. Si l'adresse logique est (numéro de page, adresse relative), par exemple (1,30), le processeur utilise la table des pages pour produire une adresse physique (numéro de cadre, adresse relative), ici (13,30).

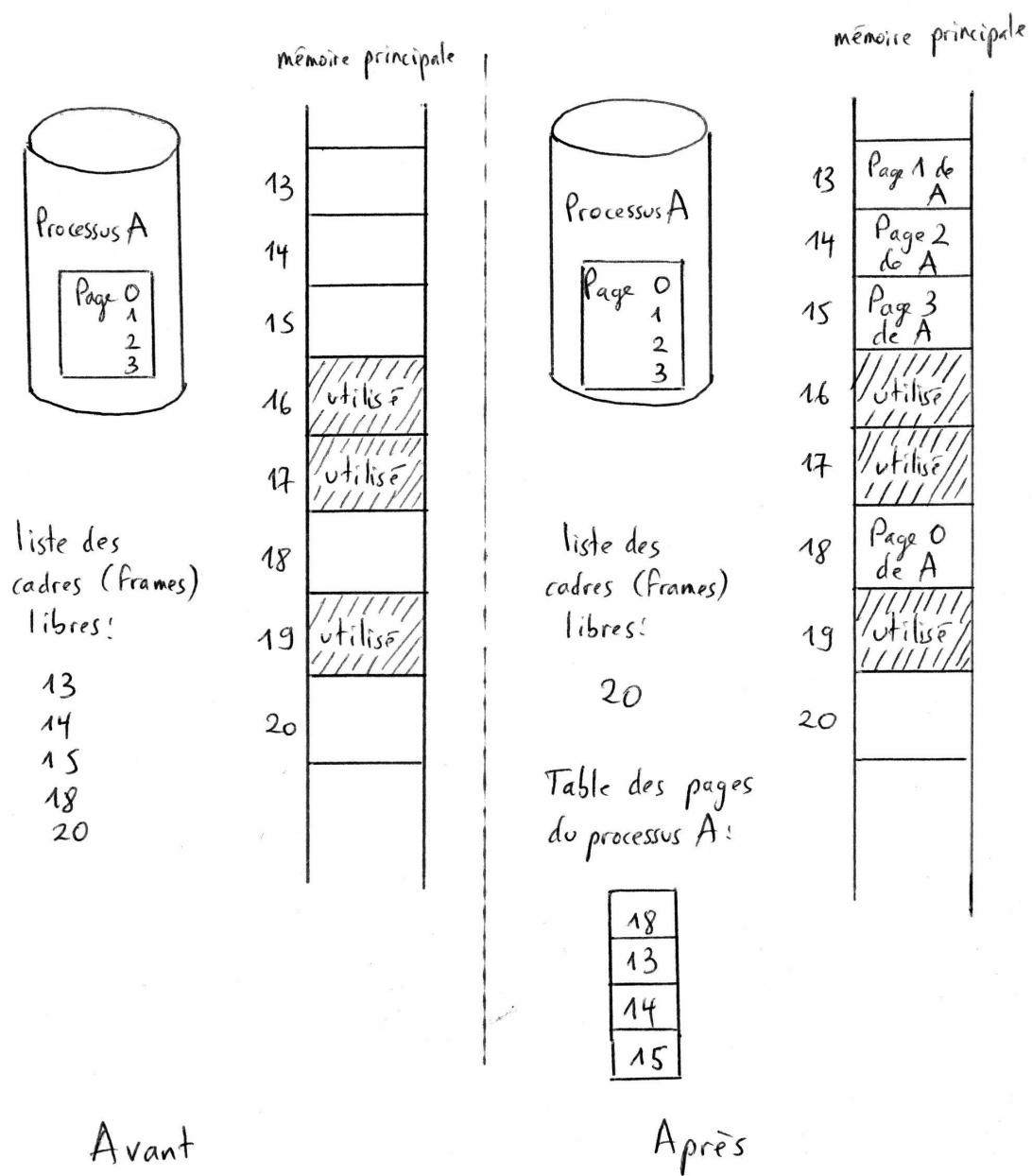


FIGURE 8.5 – Allocation de cadres (frames) libres.

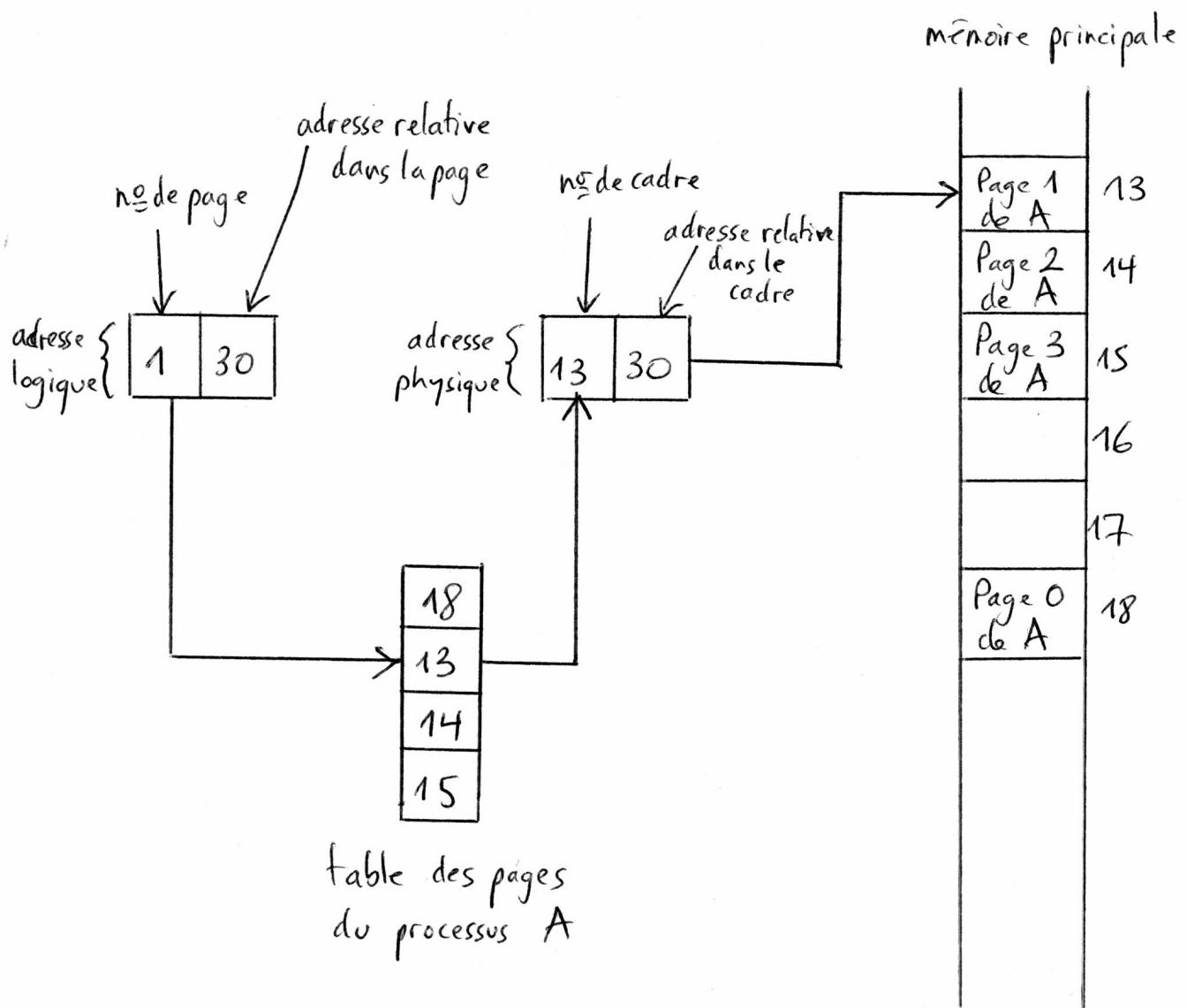


FIGURE 8.6 – Adresses logiques et physiques.



FIGURE 8.7 – L'ordinateur Atlas (1962), premier système à employer la pagination. (source : Wikipédia)

8.2.3 Pagination à la demande

La pagination a permis d'introduire la notion de mémoire virtuelle. En plus des éléments précédents, on ajoute maintenant la possibilité de chargement d'une page à la demande. Plutôt que de charger tout un processus en mémoire, on ne charge une page d'un processus que lorsqu'elle est demandée.

Lors de l'exécution d'un programme, si celui-ci veut accéder à une zone mémoire qui n'est pas chargée en mémoire, on déclenche un défaut de page (*page fault*), ce qui informe le système d'exploitation qu'il faut charger la page en question.

Ainsi, à tout moment, seules quelques pages d'un processus donné se trouvent en mémoire et par conséquent davantage de processus peuvent être conservés en mémoire. En outre, on gagne du temps parce que les pages qui ne sont pas utilisées ne sont pas échangées entre la mémoire vive et le disque.

Mais le système d'exploitation doit gérer le remplacement de pages. Quand une page est chargée en mémoire, une autre doit être éliminée, ce qui peut nécessiter une sauvegarde vers le disque, et surtout, il faut essayer de supprimer une page qui ne va pas immédiatement devoir être rechargée. Il existe un certain nombre d'algorithmes de remplacement de pages qui ne sont pas détaillés ici.

8.2.4 Espace d'adressage

En informatique, un espace d'adressage virtuel (*Virtual Address Space*, VAS) est l'ensemble des intervalles d'adresses qu'un système d'exploitation rend accessibles à un processus. Cet espace peut aller des adresses les plus petites aux plus grandes adresses autorisées par le processeur. L'un des avantages de cette allocation est la sécurité grâce à l'isolement des processus. Chaque processus a sa zone mémoire propre.

En général, la mémoire principale (physique) n'est pas aussi grande que l'espace d'adressage du processeur. Le processeur peut en principe travailler avec plus d'adresses mémoire qu'il n'y en a

physiquement de présent en mémoire principale. Si un programme est par exemple très grand, il est entièrement présent dans la mémoire virtuelle, mais seulement partiellement dans la mémoire principale. Le reste se trouve par exemple sur disque magnétique.

Cela dit, il arrive que la mémoire physique puisse, en principe, dépasser la mémoire virtuelle. Ainsi, sur l'architecture Intel64 qui implémente le jeu d'instructions x86-64, par exemple sur un Intel Core i7, il peut y avoir 48 bits pour l'adresse virtuelle et 52 bits pour l'adresse physique. Cela veut dire qu'un processus peut avoir un espace d'adressage de 2^{48} octets, soit 256Tio. La mémoire physique, elle, peut en principe atteindre 4 Pio, mais il est physiquement impossible de mettre 4 Pio de mémoire dans un PC.

L'intérêt de la mémoire virtuelle est notamment de dégager le programmeur de tout souci de mémoire. Le programmeur prépare ses programmes en utilisant tout l'espace d'adressage du processeur.

Les adresses que le processeur produit pour des instructions ou des données sont appelées des adresses virtuelles ou logiques. Ces adresses sont traduites en adresses physiques par une combinaison d'actions matérielles et logicielles. Si une adresse virtuelle fait référence à une partie du programme ou de l'espace de données qui est actuellement dans la mémoire physique, alors le contenu de cet emplacement en mémoire principale est immédiatement obtenu. Sinon, le contenu de l'adresse référencée doit être amené dans un emplacement convenable en mémoire avant de pouvoir être utilisé.

La figure 8.8 illustre une implémentation typique de mémoire virtuelle. Un composant matériel spécial, appelé MMU (*Memory Management Unit*), sait quelles parties de l'espace d'adressage virtuel est en mémoire physique. Lorsque les données ou les instructions demandées se trouvent en mémoire principale, le MMU traduit l'adresse virtuelle dans l'adresse physique correspondante. Ensuite, l'accès mémoire se poursuit de la manière habituelle. Si les données ne sont pas en mémoire principale, le MMU fait transférer par le système d'exploitation les données depuis le disque vers la mémoire.

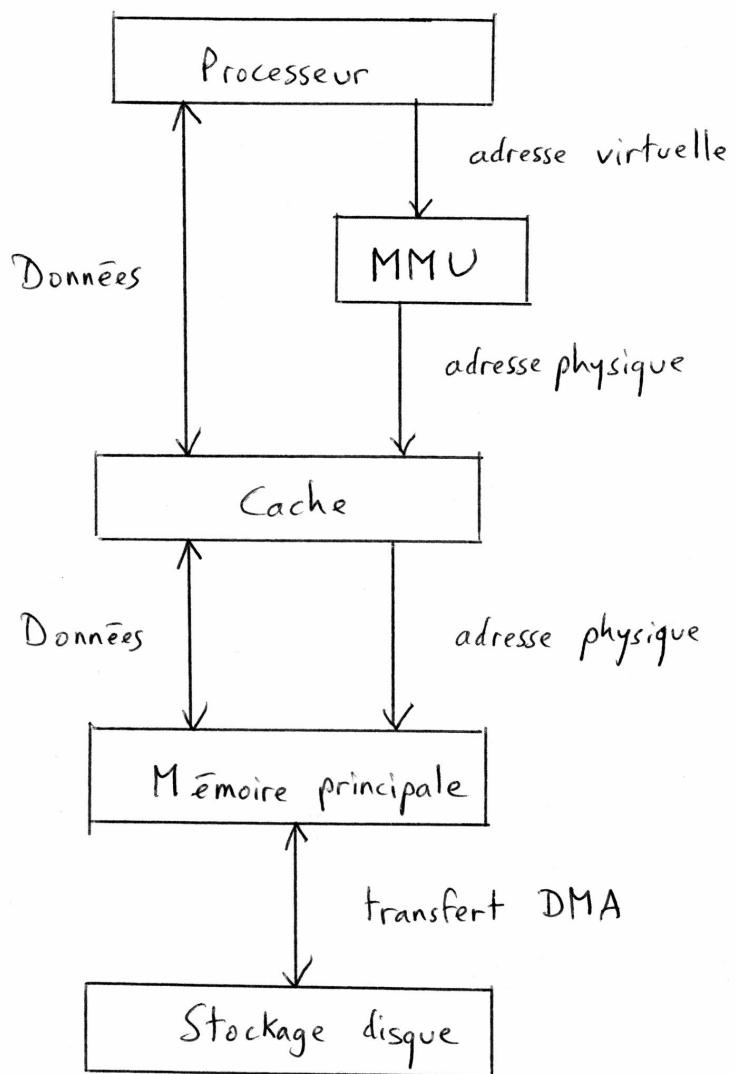


FIGURE 8.8 – Organisation de la mémoire virtuelle.

8.2.5 Traduction des adresses

Une méthode simple pour traduire les adresses virtuelles en adresses physiques est de supposer que tous les programmes et données sont composées d'unités de taille fixe appelées *pages*, chacune consistant en un bloc de mots qui occupe des emplacements contigus en mémoire principale. Les pages ont des tailles typiques de 2Kio à 16Kio. Ce sont les unités d'information de base qui sont transférées entre la mémoire principale et le disque à chaque fois que le MMU détermine qu'un transfert est nécessaire.

Les pages ne devraient pas être trop petites, parce que le temps d'accès au disque est beaucoup plus long que le temps d'accès à la mémoire principale. Il en est ainsi parce qu'il faut beaucoup de temps pour localiser la donnée sur le disque, mais ensuite le transfert est très rapide.

D'un autre côté, si les pages sont trop grandes, il se peut qu'une grande portion d'une page ne soit pas utilisée, et cela conduira à un gaspillage de l'espace de la mémoire principale.

Une méthode de traduction d'adresse de mémoire virtuelle basée sur la notion de pages de taille fixe est montrée dans la figure 8.9.

Chaque adresse virtuelle générée par le processeur, que ce soit pour une instruction *fetch* ou une opération de chargement/stockage, est interprétée comme un numéro de page virtuel suivi par un offset qui spécifie l'emplacement de cet octet (ou mot) dans une page.

Par exemple, si les pages font 4Kio, il y a 12 bits pour l'offset et les autres bits sont pour la page. Une instruction x86-64 comme

```
mov al, [0x0000100400b006ff]
```

demande de charger le contenu 0x0000100400b006ff (adresse virtuelle) dans le registre AL (un octet). Le numéro de page est alors 100400b00 et l'offset 6ff.

L'information sur la localisation de chaque page en mémoire principale est conservée dans une table de pages. Cette information inclut l'état (*status*) de la page. Une zone dans la mémoire principale qui peut stocker une page est appelée un *cadre de page* (*page*

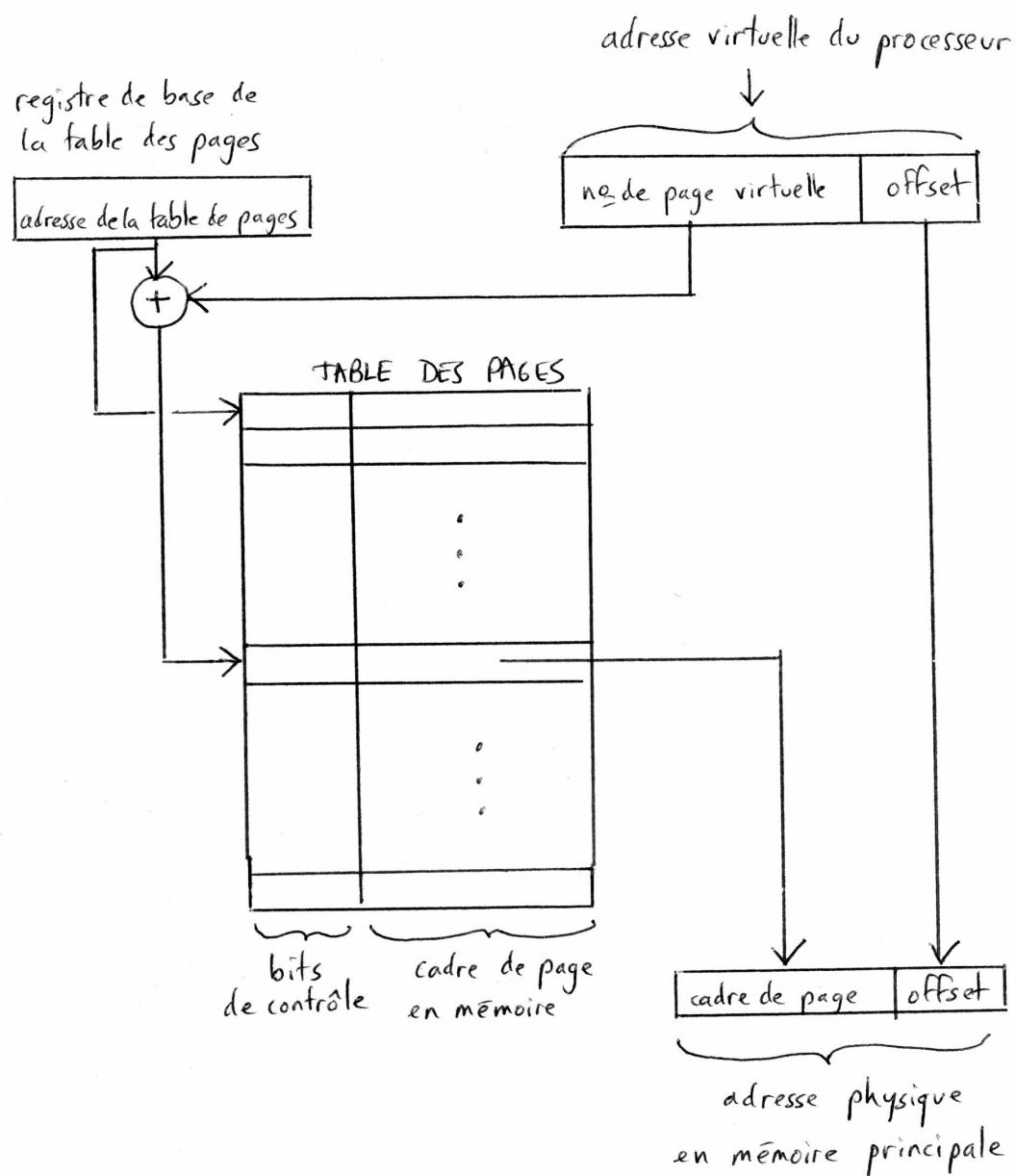


FIGURE 8.9 – Traduction d'adresse de la mémoire virtuelle.

frame). L'adresse de départ de la table des pages est conservée dans le *registre de base de la table de pages*. En ajoutant le numéro de page virtuelle au contenu de ce registre, on obtient l'adresse de l'entrée correspondante de la table des pages. Le contenu de cet emplacement donne l'adresse de départ de la page si cette page se trouve actuellement en mémoire.

Chaque entrée de la table de page inclut aussi quelques bits de contrôle qui décrivent l'état de la page pendant qu'elle se trouve en mémoire principale. Un bit indique la validité de la page, c'est-à-dire si la page est effectivement chargée en mémoire principale. Un autre bit indique si la page a été modifiée depuis qu'elle se trouve en mémoire. D'autres bits indiquent d'éventuelles restrictions d'accès.

8.2.5.1 Translation Lookaside Buffer (TLB)

Les informations de la table des pages sont utilisées par le MMU pour chaque accès en lecture et écriture. Idéalement, la table des pages devrait être stockée dans le MMU. Malheureusement la table des pages peut être très grande. Comme le MMU fait normalement partie du processeur, il est impossible d'inclure la table complète dans le MMU.

Au lieu de cela, une copie de seulement une petite partie de la table est stockée dans le MMU et la table complète est conservée en mémoire principale. La portion conservée dans le MMU consiste en les entrées correspondant aux pages accédées le plus récemment. Elles sont stockées dans une petite table, habituellement appelée *Translation Lookaside Buffer* (TLB). Le TLB fonctionne comme un cache pour la table de pages qui se trouve en mémoire principale. Chaque entrée du TLB inclut une copie de l'information dans l'entrée correspondante de la table de pages. De plus, elle inclut l'adresse virtuelle de la page, qui est nécessaire pour rechercher une page particulière dans le TLB. La figure 8.10 montre une organisation possible pour un TLB utilisant une technique de mapping associatif.

La traduction des adresses est réalisée comme suit. Étant donnée

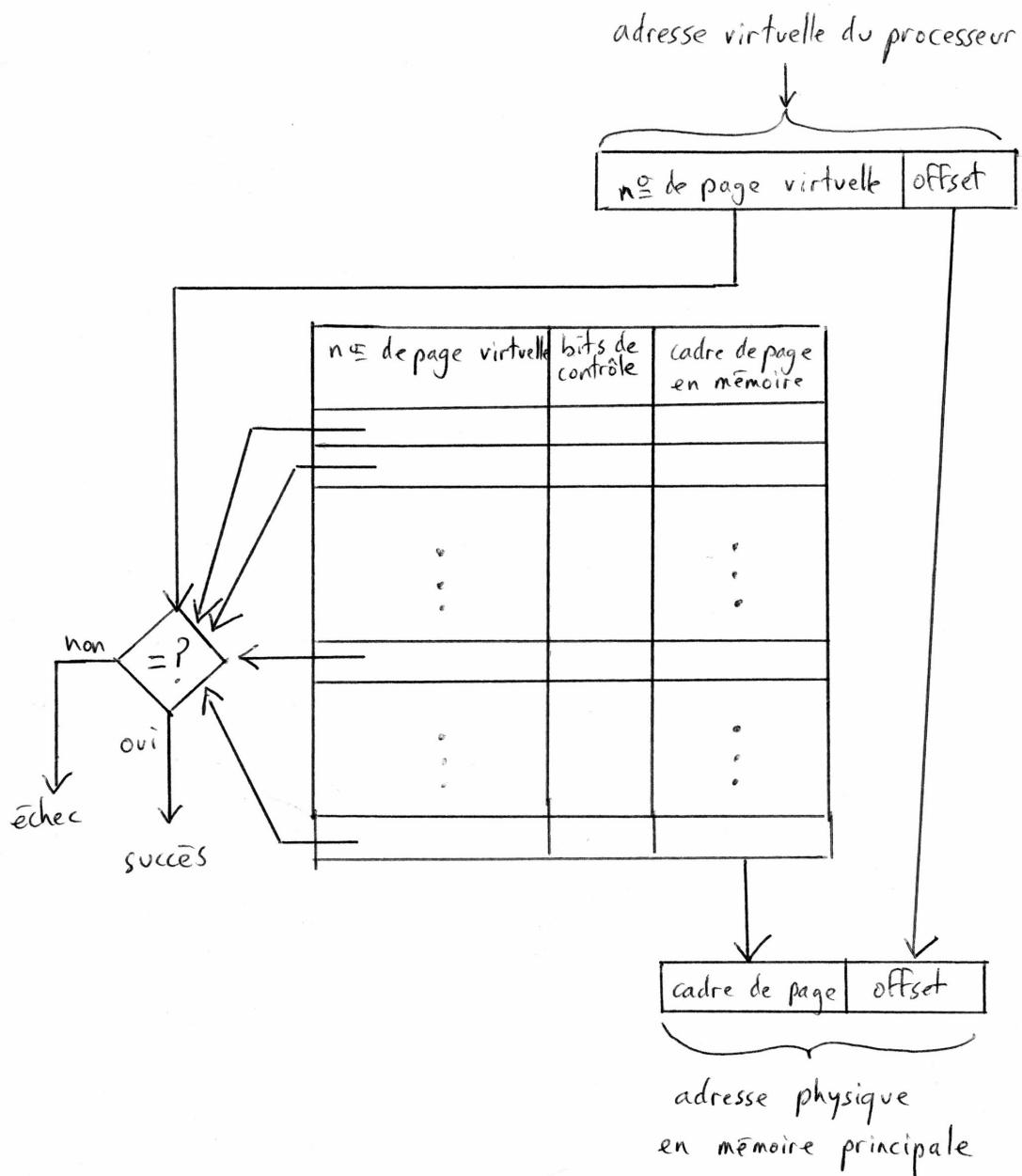


FIGURE 8.10 – Utilisation d'un TLB associatif.

une adresse virtuelle, le MMU cherche dans le TLB la page demandée. Si l'entrée de la table de pages pour cette page est trouvée dans le TLB, l'adresse physique est obtenue immédiatement. Si elle n'est pas trouvée, l'entrée requise est obtenue de la table de pages dans la mémoire principale et le TLB est mis à jour.

Il est impératif d'assurer que le contenu du TLB soit toujours le même que les contenus des tables de page dans la mémoire. Quand le système d'exploitation change le contenu d'une table de pages, il doit simultanément invalider les entrées du TLB. L'un des bits de contrôle du TLB sert à cela. Quand une entrée est invalidée, le TLB obtient la nouvelle information de la table des pages en mémoire par le fonctionnement normal des défauts d'accès.

8.2.5.2 Défauts de pages

Lorsqu'un programme génère une demande d'accès à une page qui n'est pas dans la mémoire principale, on dit qu'il se produit un *défaut de page*. Toute la page doit être chargée en mémoire depuis le disque avant que l'accès ne puisse se faire. Lorsque le MMU détecte un défaut de page, il demande au système d'exploitation d'intervenir en levant une exception. Le déroulement du programme qui a généré le défaut de page est interrompu et c'est le système d'exploitation qui prend le contrôle. Le système d'exploitation copie la page demandée depuis le disque dans la mémoire principale. Comme cette opération prend du temps, le système d'exploitation peut commencer l'exécution d'un autre programme dont les pages sont en mémoire. Lorsque le transfert est achevé, l'exécution du programme interrompu reprend.

Lorsque le MMU lève une interruption pour indiquer un défaut de page, l'instruction qui a demandé l'accès mémoire peut avoir été partiellement exécutée. Deux cas de figure sont possible, ou bien l'exécution continue au point d'interruption, ou bien l'instruction doit être redémarrée.

Si une nouvelle page est amenée du disque lorsque la mémoire principale est pleine, elle doit remplacer une des pages résidentes.

Le problème est le même qu'avec les caches. Une méthode simple est d'utiliser un bit de contrôle indiquant si une page a été récemment accédée et ces bits peuvent être périodiquement remis à 0 par le système d'exploitation.

Une page qui a été modifiée doit être écrite sur le disque avant qu'elle ne soit supprimée de la mémoire principale.

Le fait de consulter le TLB introduit un certain délai qui ralentit l'opération du MMU. On remédié à cela en conservant les entrées les plus récentes du TLB dans quelques registres spéciaux qui peuvent être accédés rapidement.

8.3 Exercices

- Supposons que la mémoire physique d'un ordinateur soit découpée en 4 pages (de 0 à 3) de 256 octets. C'est-à-dire, la mémoire physique fait 1 Kio ($256 \times 4 = 1$ Kio). Supposons qu'un processus a un espace virtuel de 3 Kio (soit 12 pages, de 0 à 11) et fait successivement référence aux adresses virtuelles suivantes : 249, 255, 300, 500, 540, 600, 650, 700, 750, 800, 900, 1200. Trouvez pour chaque adresse virtuelle, le numéro de page virtuelle et le déplacement (*offset*).
- Supposons que l'on a un schéma de partitionnement fixe avec des partitions de 2^{16} octets et une mémoire principale totale de 2^{24} octets. Une table de processus est maintenue et inclut un pointeur vers la partition correspondant à chaque processus. Combien faut-il de bits pour ce pointeur ?
- Supposons que la table des pages du processus qui est en train de s'exécuter ressemble à la table suivante. Toutes les valeurs sont décimales, tout est numéroté à partir de 0 et toutes les adresses sont des adresses d'octets. La taille d'une page est 1024 octets.

num. page virtuelle	bit de validité	bit de référence	bit de modification	num. de cadre
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

- expliquez comment, en général, une adresse virtuelle générée par le CPU est traduite en une adresse physique de la mémoire principale ;
- à quelles adresses physiques correspondent les adresses virtuelles suivantes : 1052, 2221, 5499 ?

4. considérez la table de pages suivante pour un système avec des adresses virtuelles et physiques sur 12 bits et des pages de 256 octets :

Page logique	Page physique
0	–
1	2
2	C
3	A
4	–
5	4
6	3
7	–
8	B
9	0

I.e., il y a 10 pages en mémoire virtuelle qui renvoient à 7 pages en mémoire physique. La page 2 renvoie à la page C, etc. On suppose par ailleurs que la liste des pages libres est D, E, F. Si on a besoin d'une page, on la mettra en mémoire en D. Si on a besoin d'une seconde page, on la mettre en E, etc. Considérons maintenant les adresses virtuelles (logiques) hexadécimales suivantes : 9EF, 111, 700 et 0FF. Si elles sont utilisées dans cet ordre, quelles seront les adresses physiques correspondantes ?

Chapitre 9

Entrées/sorties

En plus du processeur et de modules de mémoire, le troisième élément clé d'un ordinateur est un ensemble de modules d'entrée/sortie (E/S). Chaque module a une interface avec le bus système et contrôle un ou plusieurs périphériques. Un module E/S n'est pas simplement un ensemble de connecteurs qui relient un appareil au bus système, mais il contient des éléments pour accomplir la fonction de communication entre le périphérique et le bus.

Il y a différentes raisons pour lesquelles on ne connecte pas les périphériques directement sur le bus système, notamment la grande variété des périphériques, les vitesses de transfert souvent inférieures pour les périphériques que pour la mémoire ou le processeur, etc. Pour toutes ces raisons et d'autres, on utilise des modules E/S.

Dans ce qui suit, nous commençons par passer en revue les périphériques.

9.1 Appareils externes

Un appareil externe se connecte à un ordinateur par une liaison à un module E/S (figure 9.1). Un tel appareil est appelé un périphérique.

On peut classer les périphériques en différentes catégories, sui-

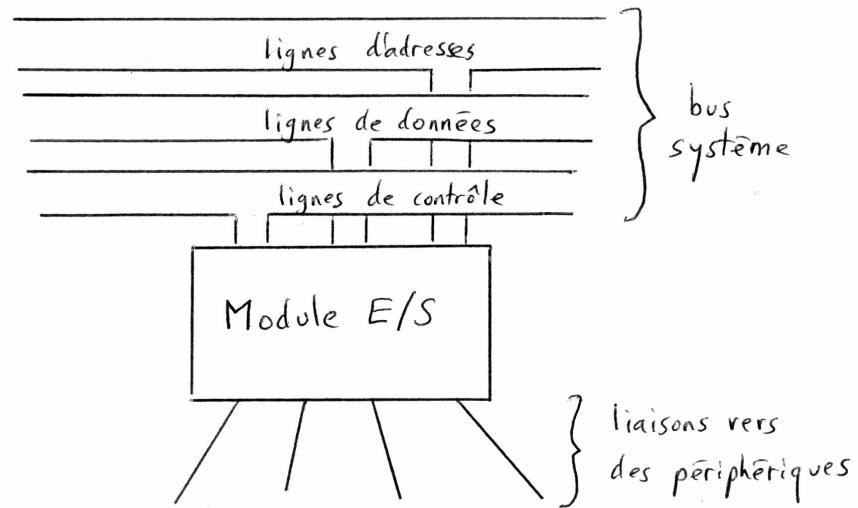


FIGURE 9.1 – Modèle générique d'un module E/S.

vant qu'ils servent à communiquer directement avec un utilisateur (par exemple un écran), avec un équipement (par exemple des disques ou des capteurs) ou avec des appareils distants. Les dispositifs de stockage comme les disques sont en fait des périphériques, bien que nous les ayons considérés plus haut comme des éléments de la hiérarchie mémoire.

La figure 9.2 montre la nature générale d'un périphérique. Le module E/S est situé vers le haut et l'environnement extérieur vers le bas. L'interface avec le module E/S consiste en l'échange de données, le module E/S envoie des signaux de contrôle au périphérique et le périphérique envoie au module E/S des signaux d'état.

Un périphérique contient un bloc pour gérer ces signaux. Un transducteur convertit les données entre une forme électrique et une autre forme (par exemple un déplacement mécanique). Un buffer est d'habitude associé au transducteur.

Nous n'étudierons pas en détail l'interface entre le périphérique et l'extérieur. Nous dirons simplement quelques mots sur des péri-

phériques courants.

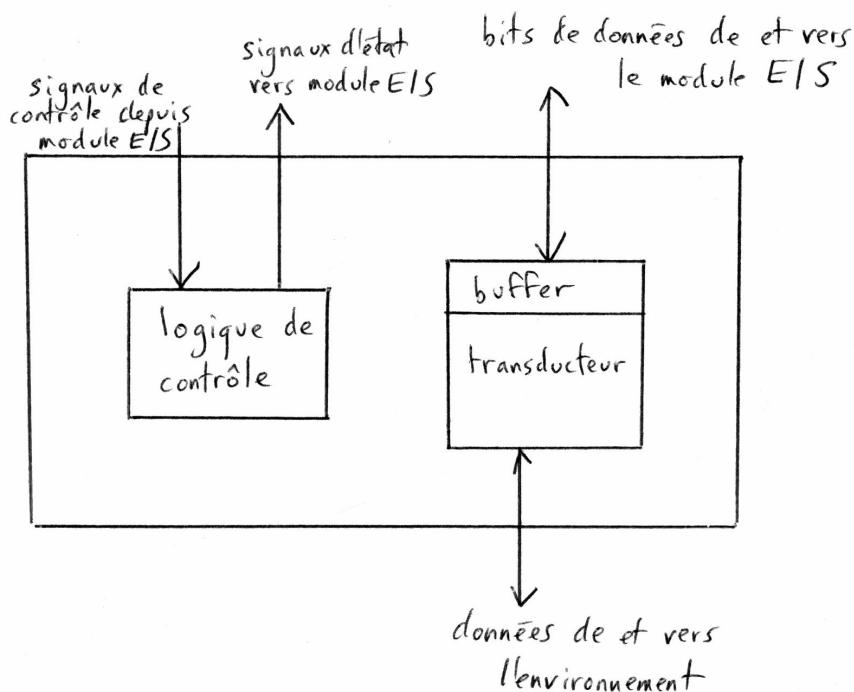


FIGURE 9.2 – Diagramme de bloc pour un appareil externe.

Dans le cas de l'interaction avec un clavier et un écran, l'unité de base de l'échange est le caractère. À chaque caractère, on associe un code, typiquement de 7 ou 8 bits. Le code le plus couramment utilisé est le code IRA (*International Reference Alphabet*), dont la version américaine est le code ASCII. Ce code contient des caractères imprimables et des caractères de contrôle.

Lors d'une entrée au clavier, lorsque l'utilisateur appuie sur une touche, il génère un signal électrique qui est traduit dans la suite de bits du code IRA correspondant. Cette suite de bits est ensuite transmise au module E/S de l'ordinateur. Dans l'ordinateur, le texte peut être stocké dans le même code IRA.

9.2 Modules E/S

Les principales fonctions d'un module E/S entrent dans les catégories suivantes :

- contrôle et timing;
- communication avec le processeur;
- communication avec le périphérique;
- buffering de données;
- détection d'erreurs.

Lors d'un intervalle de temps quelconque, le processeur est susceptible de communiquer avec un ou plusieurs appareils externes d'une manière imprévisible. Les ressources comme la mémoire principale et le bus système doivent être partagés entre un certain nombre d'activités, y compris des entrées/sorties de données. Il est par conséquent nécessaire de coordonner le flux de traffic entre des ressources internes et des appareils externes.

Par exemple, le transfert de données depuis un appareil externe au processeur pourrait faire intervenir les étapes suivantes :

- le processeur interroge le module E/S pour vérifier l'état (*status*) de l'appareil externe;
- le module E/S renvoie l'état;
- si l'appareil est opérationnel et prêt à transmettre, le processeur demande le transfert de données, au moyen d'une commande envoyée au module E/S;
- le module E/S obtient une unité de données (par exemple 8 ou 16 bits) de l'appareil externe;
- les données sont transférées du module E/S au processeur.

Un module E/S doit donc communiquer à la fois avec le processeur et avec l'appareil externe.

Un aspect essentiel d'un module E/S est le buffering de données. Quand des données sont envoyées de la mémoire à un module E/S, cet envoi est très rapide, mais les données sont alors mises dans un

buffer pour être envoyées à l'appareil externe à la vitesse qui lui convient.

9.3 Opérations E/S

Trois techniques sont possibles pour les opérations d'entrée/sortie. Nous ne ferons que les mentionner :

- 1) E/S programmées (*programmed I/O*) : le processeur exécute un programme qui lui donne un contrôle direct des opérations d'entrée/sortie ; si le processeur est plus rapide que le module E/S, c'est un gaspillage de temps de processeur ;
- 2) E/S déclenchées par interruption : le processeur donne un ordre d'entrée/sortie, puis continue à exécuter d'autres instructions ; il est ensuite interrompu par le module E/S lorsque ce dernier a fini son travail ;

Dans les deux cas précédents, le processeur doit extraire les données de la mémoire principale pour la sortie, ou doit les stocker en mémoire principale pour l'entrée. Une troisième technique est

- 3) l'accès direct mémoire (DMA, *Direct Memory Access*) : dans ce mode, le module E/S et la mémoire principale échangent des données directement, sans que le processeur ne soit impliqué.

Nous ne détaillerons pas davantage ces techniques qui seront étudiées dans un autre cours.

9.4 Un peu d'histoire



FIGURE 9.3 – L'ordinateur UNIVAC 1 (1951). (source : Wikipédia)



FIGURE 9.4 – L'interface de programmation de l'IBM 650 (vers 1958).
À droite, le périphérique de lecture de cartes perforées.



FIGURE 9.5 – L'imprimante IBM 1403, introduite en 1959. (source : Wikipédia)



FIGURE 9.6 – L'interface d'entrée/sortie de l'ordinateur d'Apollo (Apollo Guidance Computer) (1969). (source : Wikipédia)

9.5 Exercices

1. de quand date la première calculatrice électronique de poche ? qui en est l'inventeur ?
2. recherchez qui a inventé la souris ; comment faisait-on avant la souris ?
3. recherchez de quand date le premier ordinateur avec un écran ;
4. qu'est-ce qu'un terminal ?
5. quel est le premier ordinateur personnel ?

Chapitre 10

Petit mémento de C

Certains des exercices des chapitres précédents font appel au langage C et nous faisons ici rapidement un survol de ce langage.

Nous avons choisi d'utiliser le C pour illustrer certaines questions de codage ou en rapport avec la mémoire, car ce langage permet d'accéder très précisément aux emplacements mémoire. Ce n'est par exemple pas le cas pour Java où les programmes fonctionnent dans une machine virtuelle. Par ailleurs, le C est un langage historiquement important et utile, il a grandement influencé Java et d'autres langages et un informaticien sérieux se doit de le connaître.

Le C est un langage compilé, c'est-à-dire qu'un fichier source est transformé en code objet (langage machine) à l'aide d'un compilateur. Les exemples qui suivent feront appel au compilateur gcc sous Linux, mais on peut aussi utiliser gcc sous Windows, ou d'autres compilateurs. Nous recommandons toutefois de travailler sous Linux, afin de pouvoir répliquer les différents exemples de ce support.

Ce qui suit ne se veut pas un cours complet de C. Pour cela, on pourra consulter de nombreux documents en ligne, y compris des vidéos. Ici, il s'agit simplement de donner les éléments essentiels utiles à ce cours. Beaucoup d'aspects étant presque identiques en Java, nous passerons rapidement sur ces points.

10.1 Édition et compilation

Un programme C est contenu dans un ou plusieurs fichiers textes. Ici, pour simplifier, nous n'utiliserons à chaque fois qu'un seul fichier. Ce fichier C, que l'on appelle un fichier source, a pour extension .c. Il faut donc l'éditer avec un éditeur et le sauvegarder quelque part. Le mieux est de créer un dossier pour ces programmes. Pour cela, on ouvre un terminal et on écrit :

```
> cd # pour aller dans le HOME (# est un commentaire)
> mkdir progC # pour les programmes C
> cd progC # on va dans ce dossier
> gedit bonjour.c & # on ouvre (crée) bonjour.c avec gedit
```

Le > est l'invite et peut être différent chez vous. Nous avons utilisé gedit, mais on peut utiliser emacs, Visual Studio Code, ou n'importe quel autre éditeur de fichier texte. Il faut simplement prendre garde à ne pas utiliser un outil qui ajouterait des caractères à votre insu dans le fichier. (En réalité, il y a presque toujours de tels caractères, ne seraient-ce que les fins de lignes.)

Le caractère « & » dans la dernière ligne précédente permet de lancer gedit en tâche de fond, c'est-à-dire permet de continuer de travailler dans le terminal après le lancement de l'éditeur.

Une fois l'éditeur lancé, on peut écrire le programme.

```
#include <stdio.h>
int main(void) {
    printf("Bonjour !\n");
}
```

Une fois ce programme écrit, on le sauvegarde, puis on le compile dans le terminal avec gcc :

```
> gcc -o bonjour bonjour.c
```

Ceci produit un exécutable appelé `bonjour`. Pour exécuter le programme, il faut aller là où il se trouve et écrire son nom, précédé de `./` :

```
> ./bonjour
```

Au total, il doit s'afficher ceci à l'écran :

```
> gcc -o bonjour bonjour.c
> ./bonjour
Bonjour !
>
```

10.2 La structure d'un programme C

Le programme précédent ne contient que quelques lignes, mais il peut sembler déjà complexe. Il y a d'abord une première ligne spéciale, commençant par `#`. Elle indique au compilateur de charger le fichier `stdio.h` qui le renseigne sur les profils des fonctions d'entrée/sortie, comme celle de l'affichage. Cette ligne sera presque toujours présente dans vos programmes.

Ensuite, on définit une fonction, d'une manière assez proche du Java. La fonction s'appelle `main`, elle ne prend pas de paramètre, elle rend un entier `int` (ou du moins elle le prétend) et son corps se limite à l'appel de la fonction `printf`.

Concernant le type de retour de la fonction `main`, celui-ci n'impose pas que la fonction rende quelque chose, mais par défaut toutes les fonctions d'un programme C ont comme type de retour `int`, qui est le type des entiers.

Tout programme C est constitué en gros d'une suite de fonctions. Il n'y a pas de classes comme en Java. Les fonctions se suivent au même niveau. Pour qu'un programme soit exécutable, il doit contenir une fonction `main`, de même qu'un programme Java doit contenir une méthode `main` (avec un profil bien particulier) dans une classe.

Dans le corps de cette fonction `main`, il y a donc un appel de la fonction `printf`. C'est la fonction standard d'affichage. On lui donne en paramètre une chaîne. À la fin de la chaîne, `\n` permet d'aller à la ligne. Les instructions s'achèvent toutes par un «`;`».

10.3 Types et variables

Les principaux types que nous utiliserons ici sont

- `char` : le type pour les caractères, c'est un type sur un octet ;
- `int` : le type pour les entiers signés, en général sur 4 octets ;
- `short` : le type pour les entiers courts ;
- `long` : le type pour les entiers longs ;
- `float` : le type pour les réels ;
- `double` : le type pour les réels double-précision.

Les déclarations des variables sont similaires à celles en Java, si ce n'est que l'on ne crée pas d'objets. Voici quelques exemples de déclarations et d'affectations. Il est préférable, pour la lisibilité d'un programme, de faire les déclarations au début d'un bloc.

```
#include <stdio.h>
int main(void) {
    char c;
    int m,n=10; // n est initialisé //=commentaire, comme en Java
    short s;
    long l;
    float f;
    double d;
    f=17.3;
    printf("Bonjour !\n");
}
```

Dans la suite, l'en-tête du programme (la ligne `#include`) ne sera pas systématiquement répété. De même, on donnera souvent un fragment de programme qui devra être inséré dans un ensemble plus complet.

Les calculs se font comme en Java.

10.4 Entrées/sorties

Pour afficher le contenu de la variable `n`, on utilisera aussi `printf` :

```
printf("Bonjour ! n=%d\n",n);
```

Ici, on affiche la valeur de `n` à l'aide d'une commande formatée `%d`. Cela signifie que la valeur de `n` va être mise à la place du `%` et cette valeur sera affichée en décimal.

On peut aussi afficher les entiers en hexadécimal (`%x`) ou en octal (`%o`).

En règle générale, pour connaître la syntaxe d'une fonction, on peut se servir du manuel disponible dans le terminal en tapant `man printf` ou mieux, `man 3 printf` dans ce cas. Une page de manuel s'affiche et on peut la parcourir. On ressort de cette page en tapant « `q` » (comme *quit*).

Nous ne ferons pas beaucoup d'entrées clavier, mais elles pourraient se faire avec `scanf`.

10.5 Structures de contrôle

Les structures de contrôle, `if`, `for`, `while`, sont pour l'essentiel les mêmes qu'en Java. Il faut cependant noter qu'il n'y a pas de type booléen en C. Une condition est simplement vraie si elle n'est pas nulle.

10.6 Appels de fonction

Les fonctions sont toutes au même niveau et une fonction ne peut normalement faire appel à une autre fonction que si celle-ci a été déclarée auparavant.

```
#include <stdio.h>

void f1() {
    // ...
}

int f2() {
    f1();
    // ...
}

int main() {
    f1();
    f2();
    // ...
    f3(); // NE FONCTIONNE PAS
}

int f3() {
    f1();
    // ...
}
```

10.7 Un exemple un peu plus complexe

Dans l'exemple qui suit, on calcule la factorielle de n :

```
int main() {
    int n,f,i;
    n=8; // on veut calculer n!
    f=1; // initialisation
    for (i=2;i<=n;i++) { // accolades inutiles...
        f=f*i;
    }
    printf("%d!=%d\n",n,f);
}
```

Les accolades de la boucle peuvent être retirées si le corps de la boucle ne contient qu'une instruction (comme en Java).

À la fin, on fait un affichage un peu plus complexe. `printf` a trois paramètres, tout d'abord une chaîne qui a deux formateurs (`%d` et `%d`), et puis deux valeurs entières, `n` et `f`. Ceci conduit normalement

à afficher

$8!=40320$

10.8 Valeurs signées et non signées

En plus des types précédents, on peut définir des variantes non signées. Considérons l'exemple suivant :

```
int main() {
    int a;
    unsigned int b;
    short s;
    a=-5;
    b=-5;
    s=5;
    printf("a=%d, b=%d, b=%u, s=%d\n",a,b,b,s);
    printf("sizeof(a)=%ld, sizeof(s)=%d\n",sizeof(a),(int)(sizeof(s)));
}
```

L'exécution de ce programme affiche :

a=-5, b=-5, b=4294967291, s=5
sizeof(a)=4, sizeof(s)=2

On notera tout d'abord que l'on peut afficher la taille mémoire occupée par une variable avec `sizeof`. Cette fonction peut aussi être employée sur un type. On voit que `a` occupe quatre octets et que `s` occupe deux octets. Ceci peut éventuellement être différent sur certaines plate-formes.

La variable `b` occupe aussi quatre octets, mais quelle est la différence entre `a` et `b`? Sur quatre octets, `a` peut stocker une valeur entre 2^{-31} et $2^{31} - 1$, c'est le codage en complément à deux qui est étudié à la section 5.2.1.

`b`, par contre, permet normalement de stocker des valeurs uniquement positives, entre 0 et $2^{32} - 1$.

On peut donc être étonné que l'affichage précédent de `b` donne `-5`, mais bien sûr on peut aussi s'étonner du fait qu'il a été possible de donner la valeur `-5` à la variable. Ce dernier point s'explique par les conversions de type. En écrivant `b=-5`, la valeur `-5` est d'abord un entier, puis est convertie en entier non signé sur quatre bits. Cette valeur est `4294967291`, qui est justement $2^{32} - 5$. En fait, elle contient exactement la même valeur, mais elle ne sera pas manipulée de la même manière lors des calculs. Contentons-nous pour l'instant d'admettre qu'il peut y avoir des différences et qu'il faut y prêter garde.

On peut voir en tous cas que pour afficher l'interprétation non signée de `b`, il faut utiliser le formateur `%u`. Le fait d'afficher `b` avec `%d` conduit à afficher à nouveau `b` sous sa forme signée.

Enfin, l'opérateur `sizeof` produit en fait un entier long et il faut ou bien utiliser le formateur `%ld`, ou bien faire un *cast* et transtyper la sortie de `sizeof` en entier avec `(int)`. C'est ce que nous avons fait dans la dernière ligne.

10.9 Tableaux

Les tableaux sont déclarés d'une manière assez similaire au Java, mais, une fois de plus, il n'y a aucun objet et donc aucun `new`.

Pour créer un tableau de 10 entiers `int` appelé `tab`, on peut écrire :

```
int tab[10];
```

Les variables de ce tableau seront `tab[0]`, `tab[1]`, ..., `tab[9]`. On peut utiliser les éléments du tableau comme toute autre variable.

Il est important de noter que contrairement au langage Java, un tableau n'inclut pas sa taille. On ne peut pas écrire `tab.length` ou des choses de ce genre. Par ailleurs, rien n'interdit de sortir du tableau. On peut très bien écrire

```
tab[25]=12;
```

avec l'exemple précédent, même si cela n'est pas souhaitable.

10.10 Chaînes de caractères

Une chaîne de caractères est simplement un tableau de char :

```
char ch[100];
```

Mais en manipulant des chaînes, on a souvent besoin des mêmes fonctionnalités, comme l'affichage, la concaténation, la recherche de caractères, etc. Il y a pour cela une bibliothèque de fonctions que l'on peut charger en ajoutant `#include <string.h>` au début du programme. L'une des fonctions de la bibliothèque est celle qui donne la longueur d'une chaîne. Cette longueur n'est pas la place occupée par la variable, mais la place occupée par le contenu utile. Voyons un exemple. Tout d'abord, comment donne-t-on une valeur à une chaîne ?

```
#include <stdio.h>
#include <string.h>

int main() {
    char c;
    char ch[100];
    c='X'; // un seul caractère, cotes simples
    strcpy(ch,"Bonjour"); // chaîne, cotes doubles
    printf("c=%c\n",c);
    printf("ch=%s\n",ch);
}
```

On n'écrit pas `ch="Bonjour"`, mais on utilise la fonction `strcpy`. Cela dit, on aurait pu donner une valeur initiale à la chaîne `ch` au moment de sa déclaration et seulement à ce moment-là :

```

int main() {
    char c;
    char ch[100] = "Toto"; // (1)
    c='X'; // un seul caractère, cotes simples
    printf("ch=%s\n",ch);
    strcpy(ch,"Bonjour"); // (2) chaîne, cotes doubles
    printf("c=%c\n",c);
    printf("ch=%s\n",ch);
}

```

Maintenant, les deux affectations (1) et (2) font quelque chose qui est implicite, à savoir qu'elles ajoutent un caractère spécial à la fin de la partie utile de la chaîne. On peut voir quel est ce caractère en affichant le contenu des 10 premiers caractères de la chaîne :

```

int main() {
    int i;
    char c;
    char ch[100] = "Toto";

    for (i=0;i<10;i++) {
        printf("ch[%d]=%c\n",i,ch[i]);
    }
}

```

Ceci affiche

```

ch[0]=T
ch[1]=o
ch[2]=t
ch[3]=o
ch[4]=
ch[5]=
ch[6]=
ch[7]=
ch[8]=
ch[9]=

```

Cela nous permet de voir que nous affichons les éléments d'une chaîne comme des caractères, mais nous ne voyons pas bien ce qui

vient après « Toto ». En fait, un char n'est rien d'autre qu'un entier sur un octet et nous pouvons l'afficher en tant qu'entier, il suffit de remplacer %c par %d :

```
int main() {
    int i;
    char c;
    char ch[100] = "Toto";

    for (i=0; i<10; i++) {
        printf("ch[%d]=%d\n", i, ch[i]);
    }
}
```

et nous obtenons :

```
ch[0]=84
ch[1]=111
ch[2]=116
ch[3]=111
ch[4]=0
ch[5]=0
ch[6]=0
ch[7]=0
ch[8]=0
ch[9]=0
```

Nous voyons ici deux choses. Tout d'abord, nous voyons que « T » est codé par 84, « o » par 111 et « t » par 116, ce qui correspond tout simplement au code ASCII de ces caractères. Signalons en passant que si la chaîne que l'on stocke contient des caractères qui ne tiennent pas sur un octet, le nombre réel de caractères occupé peut être plus grand que le nombre visible de caractères.

Ensuite, il y a une suite de 0. En fait, le 0 est un caractère spécial de fin de chaîne. C'est cette valeur qui permet aux fonctions travaillant sur les chaînes de caractères de savoir où s'arrête la partie utile de la chaîne. Cela veut aussi dire que si l'on écrit ses propres fonctions sur les chaînes, il faut faire attention à ce caractère et ne pas l'oublier.

Après la valeur initiale « Toto », mettons maintenant un seul caractère avec `strcpy` et affichons le contenu, cette fois-ci de manière un peu plus compacte :

```
#include <stdio.h>
#include <string.h>

int main() {
    int i;
    char c;
    char ch[100] = "Toto";
    for (i=0;i<10;i++) printf("ch[%d]=%d\n",i,ch[i]);
    strcpy(ch,"X");printf("\n");
    for (i=0;i<10;i++) printf("ch[%d]=%d\n",i,ch[i]);
}
```

Et nous obtenons :

```
ch[0]=84
ch[1]=111
ch[2]=116
ch[3]=111
ch[4]=0
ch[5]=0
ch[6]=0
ch[7]=0
ch[8]=0
ch[9]=0
```

```
ch[0]=88
ch[1]=0
ch[2]=116
ch[3]=111
ch[4]=0
ch[5]=0
ch[6]=0
ch[7]=0
ch[8]=0
ch[9]=0
```

Ce que nous voyons, c'est que la fonction `strcpy` à ajouter « X » (codé par 88) et un 0, mais n'est pas allé plus loin. Il reste donc dans le tableau `ch` des caractères de la chaîne initiale, mais cela ne présente aucun danger, tant que l'on respecte la convention du 0 de fin de chaîne. Ainsi, on peut afficher la longueur de la partie utile de la chaîne avec `strlen` qui rend aussi un entier long (d'où `%d`) :

```
#include <stdio.h>
#include <string.h>

int main() {
    int i;
    char c;
    char ch[100] = "Toto";
    printf("longueur(ch)=%ld\n", strlen(ch));
    strcpy(ch, "X");
    printf("longueur(ch)=%ld\n", strlen(ch));
}
```

Et nous obtenons :

```
longueur(ch)=4
longueur(ch)=1
```

Parmi les autres fonctions utiles sur les chaînes de caractères, citons `strcat` qui permet de concaténer des chaînes de caractères et `strcmp` qui permet de les comparer. On pourra trouver des détails sur ces fonctions dans le manuel en ligne.

10.11 Manipulation de bits

Il est quelquefois utile de trouver les bits qui composent une variable, par exemple un entier. Dans le cas d'un entier positif, on peut obtenir ces bits en faisant des décalages avec l'opérateur `>>` :

```
#include <stdio.h>
int main() {
    int i;
    unsigned int a;
    a=0xb0c6a563;
    i=0;
    while (a!=0) {
        printf("%d",a&1);
        a=a>>1;
    }
    printf("\n");
}
```

Ce programme affiche les bits correspondant à la valeur hexadécimale B0C6A563, mais du dernier au premier. On peut les afficher dans un autre ordre en les mettant par exemple dans un tableau. Dans cet exemple, le type `unsigned int` est essentiel, sans quoi `a` est considéré comme négatif et le décalage vers la droite conserve le signe. La boucle ne s'arrête alors jamais. Si on veut connaître les bits d'un entier négatif, on peut le mettre d'abord dans une variable `unsigned`.

On verra plus loin comment décoder un flottant.

10.12 Accès à la mémoire

Chaque variable occupe un certain emplacement mémoire, ou plutôt une suite d'octets en mémoire. Et chaque octet a une adresse. Il est possible de connaître l'adresse d'une variable et il est possible de savoir quelle valeur se trouve dans un certain octet. Voyons quelques exemples :

```
#include <stdio.h>
int main() {
    unsigned long a; // pour l'adresse
    unsigned short s;
    s=1000;
    a=(unsigned long)&s;
    printf("adresse(s) = %ld = %lx = %p\n",a,a,&s);
    printf("adresse(a) = %p\n",&a);
}
```

Ceci affiche par exemple :

adresse(s) = 140721020194014 = 7ffc2a6bccde = 0x7ffc2a6bccde
 adresse(a) = 0x7ffc2a6bcce0

(chez vous les valeurs seront différentes)

On a défini deux variables, a et s, et on a affiché leurs adresses en utilisant l'opérateur de *déréférencement* &. Dans un cas, on a mis l'adresse de s dans a, mais on aurait pu s'en passer, puisque les adresses peuvent être affichées directement en hexadécimal avec %p.

Les valeurs hexadécimales affichées avec %p commencent par 0x, mais celles affichées avec %lx (1 car long) ne commencent pas par 0x. La toute première valeur affichée est la valeur décimale de l'adresse.

Nous savons donc maintenant afficher des adresses et même l'adresse de s. Ce que nous pourrions faire, c'est afficher le contenu de l'octet se trouvant à l'adresse donnée en premier, et de l'octet suivant. Cela peut se faire ainsi :

```
#include <stdio.h>
int main() {
    unsigned long a; // pour l'adresse
    unsigned short s;
    s=1000;
    a=(unsigned long)&s;
    printf("%d\n",((unsigned char*)a)[0]);
    printf("%d\n",((unsigned char*)a)[1]);
}
```

C'est un peu compliqué, c'est vrai, mais le but est de voir que l'on peut accéder à la mémoire octet par octet. Ce programme affiche donc :

232

3

Pourquoi ? Tout simplement parce que $1000 = 3 \times 256 + 232$.

La valeur 1000 est stockée sur deux octets, l'un valant 3, l'autre 232. La valeur 232 est stockée en premier.

Ce qu'a fait le programme précédent, c'est simplement de stocker l'adresse de s dans un entier non signé. Ensuite, on a prétendu que cette adresse était celle d'un tableau de caractères non signés, c'est-à-dire de valeurs entières sur un octet, puis on a accédé aux deux premières valeurs de ce tableau.

On peut de cette manière voir comment sont stockés les réels et n'importe quel autre type de variable.

Une autre manière de décoder une variable, comme par exemple un float qui occupe quatre octets, est la suivante :

```
#include <stdio.h>
int main() {
    int i;
    float f;
    unsigned char c;
    f=-17.3;
    for (i=0;i<4;i++) {
        c=((char*)(&f))[i];
        printf("%X ",c);
    }
    printf("\n");
}
```

Le programme précédent affiche

66 66 8A C1

et on pourra vérifier que la représentation IEEE754 de -17.3 est C18A6666 en hexadécimal. (Vérifiez-le à la main, et contrôlez-le avec

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>)

10.13 Les adresses et les tableaux

Un tableau, lorsqu'il est créé, n'est pas initialisé à zéro. On pourra s'en convaincre en lançant le programme suivant :

```
#include <stdio.h>
int main() {
    int ti[10],i;
    for (i=0;i<10;i++) {
        printf("t[%d]=%d\n",i,ti[i]);
    }
}
```

Par ailleurs, on notera que le nom d'un tableau est en fait l'adresse de la première case du tableau. Les deux affichages suivants seront identiques :

```
int t[10];
printf("adresse(t)=%p\n",t);
printf("adresse(t[0])=%p\n",&(t[0]));
```

Enfin, il est facile d'imaginer que les adresses étant des entiers, il est possible de faire des calculs sur ces adresses. Les adresses des variables sont des entiers longs, que l'on appelle en C *pointeurs*. On peut déclarer des variables de ces types en utilisant le caractère « * ». Dans l'exemple suivant, pi est un pointeur et plus précisément un pointeur sur un entier. Cela veut dire que pi peut contenir l'adresse d'une variable entière, mais en fait peut contenir n'importe quelle adresse. Le C ne va pas vérifier qu'il y a effectivement un entier à l'adresse indiquée. Et une fois que l'on a une adresse, on peut obtenir la *valeur pointée* en préfixant le pointeur de « * ». Toujours dans cet exemple, on donne à pi initialement l'adresse de la première case du tableau ti, qui est identique au nom du tableau. Enfin, lorsque l'on incrémente pi, on n'ajoute en fait pas 1 à l'adresse, mais la longueur d'un entier, donc 4 en général. Cela permet d'aller de case en case, donc d'entier en entier.

```
#include <stdio.h>
int main() {
    int ti[10], i;
    int *pi;
    pi=ti;
    for (i=0;i<10;i++) {
        printf("t[%d]=%d\n", i, *pi);
        pi++;
    }
}
```

10.14 Compléments sur la compilation

On peut produire le code objet d'un programme C en utilisant l'option `-c` de `gcc`. Le résultat est le fichier `fichier.o`.

```
gcc -c fichier.c
```

On peut aussi produire le code en assembleur (`fichier.s`) :

```
gcc -S fichier.c
```

Ce fichier est un fichier texte qui comporte des lignes comme :

```
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $64, %rsp
movq %fs:40, %rax
movq %rax, -8(%rbp)
xorl %eax, %eax
```

On reconnaît des instructions du langage machine.

L'obtention d'un fichier exécutable se scinde en fait en quatre phases :

- l'exécution du préprocesseur qui traite les lignes commençant par `#`, notamment (mais pas seulement) les `#include` ;

- (ce n'est pas l'instruction `#include`, mais l'instruction `include` du préprocesseur)
- la production du code d'assemblage (`gcc -S`);
 - l'assemblage du code précédent avec un assembleur comme `as`;
 - enfin, l'édition de liens, qui complète le code précédent par diverses bibliothèques (par exemple pour les entrées sorties, les chaînes de caractères, etc.); cette étape peut se faire avec le programme `ld`.

Toutes ces phases sont gérées automatiquement lorsque l'on appelle `gcc` sans options particulières ou avec l'option `-o`.

10.15 Autres sujets

Il reste encore beaucoup d'autres sujets à traiter, comme les structures (`struct`), la définition de nouveaux types (`typedef`), les entrées au clavier (`scanf`, `gets`), l'utilisation de fichiers (`fopen`, etc.), l'allocation dynamique de mémoire (`malloc`), la structuration de code, l'automatisation des compilations (`Makefile`), etc., etc., mais ces sujets seront développés dans un autre cours.

10.16 Un peu d'histoire

- Ken Thompson et Dennis Ritchie expliquent UNIX (vers 1980) :
<https://www.youtube.com/watch?v=JoVQTPbD6UY>
- History and spirit of C : <https://www.youtube.com/watch?v=xGVRF-Y--hI>

Ken Thompson a commencé à concevoir le premier système UNIX en 1969. Il lui fallait un langage, il est parti du langage BCPL (1967), lui-même une évolution de CPL (1963), puis l'a simplifié, ce qui a donné le langage B, prédecesseur immédiat du C.

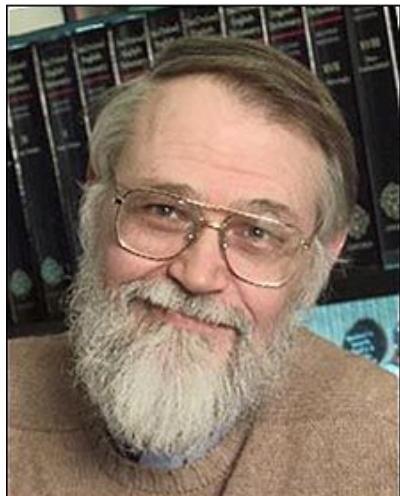
Un exemple de programme en BCPL :

```
GET "LIBHDR"  
  
LET START () BE  
$(  
    WRITES ("Hello world!*N")  
$)
```

Un exemple de programme en B (faisant autre chose, évidemment) :

```
main() {
    extrn putchar, n, v;
    auto i, c, col, a;

    i = col = 0;
    while(i<n)
        v[i++] = 1;
    while(col<2*n) {
        a = n+1;
        c = i = 0;
        while (i<n) {
            c += v[i] *10;
            v[i++] = c%a;
            c /= a--;
        }
        putchar(c+'0');
        if(!(++col%5))
            putchar(col%50?' ': '*n');
    }
    putchar('*n*n');
}
v[2000];
n 2000;
```



Brian Kernighan



Dennis Ritchie

FIGURE 10.1 – Brian Kernighan (né en 1942) et Dennis Ritchie (1941-2011).



FIGURE 10.2 – Dennis Ritchie debout à côté de Ken Thompson (né en 1943) travaillant sur le PDP-11 en 1972. (source : CNN)

10.17 Exercices

1. tester tous les programmes C donnés plus haut;
2. écrire (et tester) une fonction longueur qui fait la même chose que `strlen`, mais sans utiliser `strlen`; le profil de la fonction sera le suivant :

```
int longueur(char ch[]) {  
    // à compléter  
}
```

et dans la fonction `ch` est le nom de la chaîne ; on peut accéder aux différents caractères avec `ch[0]`, `ch[1]`, `ch[2]`, etc.

3. de même, écrire (et tester) une fonction concatene qui fait la même chose que `strcat`, mais sans utiliser les fonctions standard.
4. Écrivez un programme qui affiche toutes les puissances 2^i de 2, pour i de 0 à 64, et cela sans faire de multiplication ni addition, ni utiliser la fonction `pow`.
5. Faites un programme qui, partant d'une variable sur 32 bits, récupère les 24 bits de poids fort et les stocke dans une autre variable. On affichera les deux variables en hexadécimal.