Année universitaire 2003–2004

DEUG MIAS-SM Première Année
Initiation à l'informatique

Enseignants : François Denis, Vincent Lebrun, Rémi Morin
fdenis@cmi.univ-mrs.fr, Vincent.LeBrun@astrsp-mrs.fr, morin@cmi.univ-mrs.fr

Chapitre 1

Principes des ordinateurs

1.1 Introduction

Qu'est-ce qu'un ordinateur ? Que peut-on faire avec un ordinateur ? L'objectif de cette introduction est de répondre de manière informelle à ces questions.

On peut décrire un ordinateur comme un *dispositif matériel* permettant *en principe* d'effectuer *n'importe quel calcul* sur des *données discrètes*. Précisons chaque composante de cette définition.

1.1.1 Données discrètes et codage numérique

Discret vs continu.

On distingue données *discrètes* et données *continues*. Une donnée est discrète si elle est composée d'un nombre fini (ou dénombrable) d'éléments bien définis, isolés les uns des autres.

Les chiffres, les nombres, les lettres, les mots, les phrases, les notes de musiques, les nucléotides, les acides aminés ... peuvent être considérés comme des éléments bien définis, isolés les uns des autres. Tout ensemble fini, toute liste de tels éléments peut donc être considéré comme une donnée discrète. Ainsi, les ensembles de nombres entiers, les textes, les codes génétiques (composés de nucléotides) et les protéines (composées d'acides aminés), les œuvres musicales ... peuvent être considérés comme des données discrètes.

TACATCTTCTTTAAAGGTAAGGTTGCTCAACCA

FIG. 1.1 – Un fragment de code génétique.

En revanche, les nombres réels, les formes et les objets physiques en général, les sons, la peinture, l'écriture manuscrite et les calligrammes sont des données continues.

On peut objecter à cette classification qu'une lettre tracée sur une feuille peut être vue comme une image, c'est-à-dire comme un objet continu, qu'une protéine est un objet physique et donc un objet continu mais aussi que n'importe quel objet physique peut être décrit comme une suite finie d'atomes parfaitement différenciés et donc comme une donnée discrète. Cette objection est parfaitement valide et c'est une découverte importante que d'avoir compris que nombre de systèmes matériels peuvent être décrits, à des niveaux différents, à la fois comme des données discrètes et continues. Par exemple, une protéine est à la fois une suite

(discrète) d'acides aminés et une structure matérielle (continue) déployée dans l'espace à trois dimensions.

Codage numérique et analogique

Coder un objet, c'est le représenter ou représenter certaines de ses composantes essentielles dans un système particulier. On parle de codage numérique lorsqu'on représente un objet par des nombres et de codage analogique lorsqu'on représente un objet par une donnée continue.

Dans un codage analogique, on souhaite préserver directement une forme ou une caractéristique physique de l'objet. Par exemple, un paysage peut être représenté de manière analogique comme un agencement de textures chimiques sur une pellicule photographique, une chanson peut être représentée de manière analogique comme un sillon de hauteur variable imprimé sur un disque en vynil, un texte peut être recopié de manière manuscrite sur une feuille ...

Dans un codage numérique[1], on associe conventionnellement des nombres (ou des données discrètes en général) aux composantes de l'objet que l'on veut représenter de manière à pouvoir reconstituer ou seulement désigner sans ambiguité la forme ou la caractéristique physique qui nous intéresse. Les mots permettent de désigner des choses ; la suite de nombres inscrits dans un disque compact permet de reconstituer des sons.

Un codage numérique représente un objet par des données discrètes dont l'unité de base est le *bit* (binary digit) : 0 ou 1. Un bit peut être réalisé physiquement de manières très diverses : le courant passe ou non, un trou ou non (cartes perforées), une bosse ou non (micro-cuvette et zone plane : disque laser, CD-audio, CD-ROM, DVD Digital Video Disk), aimantation sud ou nord (disques et disquettes).

Un bit ne permettant pas de coder grand chose, on utilise un système d'unités basé sur l'*octet*[2], qui est une suite de 8 bits. Les unités suivantes sont le KiloOctets (1Ko $= 2^{10}$ octets $=$ 1024 octets), le MégaOctets (1Mo $= 2^{10}$ Ko), le GigaOctets Go (1Go $= 2^{10}$ Mo), le TeraOctets (1To $= 2^{10}$ Go) et le PetaOctets (1Po $= 2^{10}$ To).

Combien d'objets différents peut-on coder avec n bits ? Les codes possibles sont toutes les suites de n éléments pris parmi 0 ou 1. Il y en a 2^n. On peut donc coder 2^n objets avec n bits : 2 avec 1 bit, $2^8 = 256$ avec 1 octet. On doit souvent résoudre le problème inverse : on dispose de m objets distincts ; combien de bits sont nécessaires pour pouvoir les coder, de telle manière que deux objets différents aient deux codes différents. Réponse : le plus petit nombre entier n tel que $2^n \geq m$.

Le nombre y tel que $2^y = m$ est le logarithme en base 2 de m.

Rappels sur les logarithmes. Si b est un réel strictement supérieur à 1, la fonction $x \to b^x$ est une bijection continue dérivable définie de $\mathbb{R}$ à valeurs dans $]0, \infty[$. Sa bijection inverse est le logarithme de base b. Elle est notée $\log_b$. On s'intéressera principalement à la valeur $b = 2$, c'est-à-dire au logarithme en base 2. Quelques propriétés :

- pour tout $x > 0$, $b^{log_b(x)} = x$,
- pour tout réel x, $\log_b(b^x) = x$,
- pour tout $x > 0$, on a $\ln x = \log_e(x)$ où $\ln e = 1$,

[1] La traduction anglaise de numérique est *digital* qui provient de *digit* qui veut dire *chiffre*. En français, la racine *digit* se rapporte aux doigts et non aux chiffres et la traduction littérale *codage digital* est à proscrire !

[2] *byte* en anglais.

– si b et b' sont deux réels strictement supérieurs à 1, pour tout $x > 0$, on a

$$Log_b(x) = \frac{log_{b'}(x)}{log_{b'}(b)}. \text{ En particulier, } log_2(x) = \frac{\ln x}{\ln 2} = \frac{log_{10}(x)}{log_{10}(2)}.$$

Combien faut-il de bits pour coder 16000 objets ? Une calculatrice permet de trouver que $log_2(16000) \simeq 13,96$. Il faut donc 14 bits.

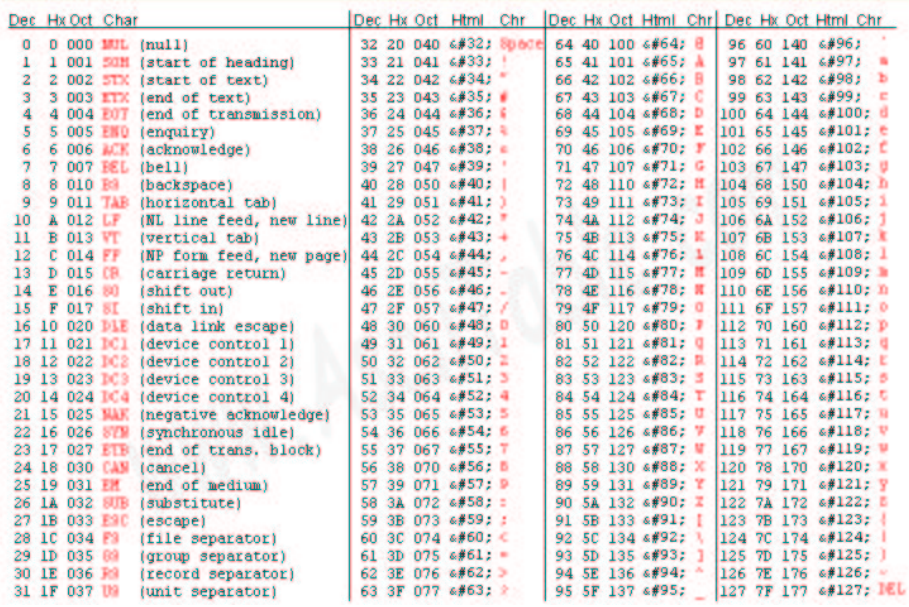
On remarquera que $2^{10} = 1024$ est proche de 1000 : cette remarque permet d'effectuer des calculs approximatifs rapidement :

$$16000 \simeq 2^4 2^{10} = 2^{14}.$$

Que peut-on représenter par des 0 et des 1 ? Réponse : toutes les données discrètes !

– *Les valeurs de vérité.* Une proposition logique est vraie ou fausse. On peut donc coder les deux valeurs de vérité possibles sur 1 bit. Le plus souvent, *Vrai* est codé par 1 et *Faux* par 0. La représentation des valeurs logiques par des nombres a conduit à la notion de *calcul logique*, fondamental dans le développement de l'informatique. Des exemples de calculs logiques seront donnés à la fin de ce cours.
– *Nombres.* Les nombres entiers sont représentables en base 2, c'est-à-dire en n'utilisant que les chiffres 0 et 1. Par exemple $23 = 2^4+2^2+2^1+2^0$ est représenté par 10111 en base 2. On peut également représenter les entiers négatifs en convenant par exemple que le premier bit représentera le signe, 0 pour un nombre négatif et 1 pour un nombre positif. Le nombre -23 sera alors codé par 010111. Un nombre rationnel $\frac{p}{q}$ peut être représenté par le couple d'entiers (p, q). Mais il ne suffit pas de juxtaposer un codage de p à un codage de q pour pouvoir reconstituer le couple : la suite 010111 désigne t-elle le couple (-2,3) ou le couple (-1,-5) ? On peut par exemple choisir de doubler les bits des codages de p et q et décider conventionnellement qu'ils seront séparés par 01. Le rationnel -2/3 sera alors représenté par 00110001111111. En revanche, il est impossible de représenter à la fois *tous* les nombres réels par des suites finies de 0 et de 1[3]. Un chapitre de ce cours est consacré aux codages usuels des nombres.
– *Lettres.* Pour coder tous les caractères utilisés dans les textes, 1 octet est amplement suffisant. Le code ASCII (pour American Standard for Communication and International Interchange) représente 128 caractères de base (codés sur 7 bits). Ce code, mis au point pour la langue anglaise, ne contient pas de lettres accentuées. Il a été étendu à 8 bits pour pouvoir coder 128 autres caractères : on parle alors de code ASCII étendu. Ce dernier n'est pas unique et dépend des lieux d'utilisation.
  – Les 32 premiers caractères ne sont pas imprimables : il s'agit de caractères de contrôle qui permettent par exemple de revenir en début de ligne (LF - Line Feed, code ASCII 10) ou de passer à la ligne suivante (CR - Carriage Return, code ASCII 13).
  – Les 26 lettres majuscules sont codées consécutivement de 65 à 90, les 26 lettres minuscules sont codées de 97 à 122. Il suffit donc d'ajouter 32 au code ASCII d'une lettre majuscule pour obtenir le code de la lettre minuscule correspondante.

[3]La raison de cette impossibilité n'est pas qu'un nombre réel peut avoir un développement décimal infini puisque les nombres 1/3 ou π sont représentables sans ambiguité par quelques caractères ; ceci n'est pas dû non plus au fait qu'il y a une infinité de nombres réels puisqu'il y a aussi une infinité de nombres entiers ou rationnels. Les réels ne sont pas codables par des suites finies de 0 et de 1 parce qu'il y en a (beaucoup) plus que de telles suites : on peut démontrer qu'il n'existe pas de bijection entre $\mathbb{R}$ et l'ensemble des suites finies de 0 et de 1.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

FIG. 1.2 – Code Ascii, téléchargé à l'adresse http ://www.asciitable.com/.

– Les 10 chiffres sont codés de 48 à 57.

– *Textes.* Comme les textes sont des suites de lettres, on peut les coder par des suites d'octets. Les caractères qui composent le mot MIAS ont pour caractère ASCII 77, 73, 65 et 83. Ce mot est donc représentable par la juxtaposition des 4 octets suivant

01001101 01001001 01000001 01010011.

Un texte de 300 pages, dont chaque page contient 40 lignes, chacunes étant composées de 80 caractères pourra donc être représenté par une suite de 300x40x80 octets soit encore 7.680.000 bits.

En réalité, les textes imprimés ont des formats particuliers (taille des marges, retraits, largeur des interlignes, paragraphes centrés ou justifiés, polices différentes, tailles de caractères différentes, casses[4] différentes, ...) qui rendent leurs représentations numériques beaucoup plus volumineuses. À titre d'exemple, les mots "DEUG MIAS" en tête d'un document HTML pourront nécessiter le codage du texte suivant :

```
<center> <h3> <b>DEUG MIAS</b></h3></center>
```

qui signifie que le texte "DEUG MIAS" doit être écrit en gras (`<b> ... </b>` : b pour **bold**), qu'il doit être écrit au format h3 (h pour *header*) et qu'il doit être centré. Les 33 caractères qui composent les instructions de formatage ne seront pas affichés tels quels : il n'empêche qu'il faut les coder comme les autres.

– *Sons.* Un son est une vibration de l'air qui peut être représentée par une fonction associant une pression à un instant. Il s'agit donc d'une donnée continue qui doit être discrétisée avant de pouvoir être codée numériquement. On appelle *échantillonnage* l'action qui consiste à découper une durée en petits intervalles de temps et à remplacer la valeur continue de la pression dans ces intervalles par une valeur constante. Plus les intervalles seront petits, plus le son pourra être restitué fidèlement. Le taux d'échantillonnage est exprimé en *Hertz*, c'est-à-dire en nombre d'intervalles par secondes. Un

[4] C'est-à-dire les caractères **gras**, *italiques* ou autres.

taux d'échantillonnage de 44000 Hz correspond à la qualité d'un CD, un taux de 8000 Hz correspond à la qualité du son transmis au téléphone[5]. Les valeurs de la pression à chaque échantillon peuvent alors être représentés par des nombres codés sur 8 ou 16 bits. Un son stéréophonique nécessite un double codage. Ainsi, une chanson de 3mn, enregistrée en stéréo à la qualité d'un CD et codée sur 16 bits, nécessitera 3x60x44000x16x2 = 253.440.000 bits pour pouvoir être représentée numériquement.

- *Images.* Les images peuvent être numérisées au moyen d'une technique d'échantillonnage analogue à celle utilisée pour numériser les sons : on obtient alors une image au format *bitmap*. Elles peuvent également être décomposées en formes géométriques simples : on dit alors qu'elles sont *vectorialisées*.
  - *codage point par point (bitmap).* L'image est découpée selon une grille dont le maillage est plus ou moins fin (résolution). Chaque élément est un pixel (pour *picture element*). Une couleur ou un niveau de gris est associé à chaque pixel. Palettes utilisées : noir et blanc, 256 niveaux de gris, 16 millions de couleurs (en fait, $2^{8*3} = 16.777.216$ couleurs). Il faut 1 bit pour coder la couleur d'un pixel en noir et blanc, 1 octet pour la coder si l'on utilise une palette de 256 niveaux de gris et 3 octets si l'on utilise une palette de 16 millions de couleurs. Par exemple, une photo haute résolution (720x480) représentée avec une palette de 65.536 couleurs pourra être codée par une suite de 3x720x480=1.036.800 octets.
  - *codage vectoriel.* Pour coder un segment, il suffit de coder ses deux extrémités. De même, un rectangle, un cercle, une ellipse, peuvent être représentés par deux points. Lorsqu'une figure peut être décomposée en figures élémentaires, le codage vectoriel permet un gain de place considérable.

À chaque fois que nous avons indiqué comment coder numériquement un objet, nous avons naturellement été amenés à parler de la taille du codage. En effet, on code un objet pour pouvoir travailler sur sa représentation et la première étape de ce *traitement* consiste toujours à la stocker dans une *mémoire informatique*[6]. Voici les capacités de quelques mémoires informatiques usuelles.

- disquette : 1,44 Mo,
- mémoire vive (RAM) : quelques dizaines de Mo,
- disques durs : quelques Go,
- CD-audio standards : 650 Mo,
- DVD (simple face, simple couche) : 4,7 Go.

Quelques outils courants permettent de passer d'un codage analogique à un codage numérique :

- Les *modems* ("modulateur/démodulateur") : pour connecter un ordinateur (numérique) au réseau téléphonique (analogique). Un signal analogique est modulé afin de coder un certain nombre de symboles, chaque symbole pouvant représenter un ou plusieurs bits. Deux unités sont utilisées : le *baud*[7] qui représente le nombre de symboles transmis par seconde et le *bps* (bit per second) qui représente le nombre de bits transmis par secondes.

[5]Ces données proviennent de l'article *Le son numérique* de Jean-François Pilou, disponible sur le site www.commentcamarche.net qui contient un grand nombre de documents concernant l'informatique au sens large et téléchargeables gratuitement.

[6]Disquettes, mémoire RAM, disques durs seront définis dans la suite.

[7]D'après Maurice Emile Baudot (1845-1903), ingénieur français qui étudia le télégraphe.

– Les *scanners* : pour numériser une image. Grâce aux scanners, on peut insérer des images, des dessins, des photos dans un document informatique.

D'autres outils transforment un codage numérique en un autre :
– *OCR* (Optical Character Recognizer) : transformation d'un caractère représenté comme une image en un caractère ASCII. C'est un outil complémentaire du scanner.
– *Compression d'information.* Un dessin enregistré au format bitmap utilise autant de place pour coder tous les pixels d'un dessin, ceux qui composent les traits comme ceux qui composent le fond blanc. Si l'on peut limiter efficacement des zones qui contiennent des pixels de même nature, on peut diminuer considérablement la taille du codage. Il en est de même des textes qui utilisent le même espace (un octet) pour coder des lettres fréquentes comme "e" que des caractères rarement utilisés comme '#'. On peut aisément réduire les données textes de 50% sans perte d'information. Voir le format MP3 pour les données musicales. Un *compresseur* réduit la taille d'une représentation numérique volumineuse,
  – sans perte d'information : cela signifie que l'opération de compression est réversible ;
  – ou sans perte d'information sensible : l'opération n'est plus réversible mais la perte ne concerne qu'une information indétectable par nos sens. Par exemple, élimination de certaines fréquences inaudibles par l'oreille humaine.

1.1.2 Qu'est-ce qu'un calcul ?

Il est plus facile de donner des exemples de calcul que de définir ce qu'est un calcul.

Exemples de calculs. Additionner deux nombres. Calculer la 10000ème décimale de π. Calculer le millionième nombre premier. Ranger les lettres d'un mot dans l'ordre croissant des lettres dans l'alphabet. Compter les fréquences d'apparition de chaque mot dans une oeuvre littéraire. Calculer si une position aux échecs est gagnante contre toute riposte[8].

Idée générale. Un calcul peut toujours être réalisé de manière systématique en un nombre fini d'étapes élémentaires (ceci n'est pas une définition).

Exemple : comment faire dire à quelqu'un qui ne connaît que deux opérations élémentaires
– calculer le reste de la division d'un entier par un autre
– comparer deux nombres entiers
si un entier est premier ou non ? On peut par exemple lui donner les consignes suivantes

```
Pour savoir si un nombre N est premier :
(1) Si N est égal à 0 ou à 1, alors N n'est pas premier.
(2) Sinon, soit P=2 ;
    (3) si le reste de la division de N par P est égal à 0 alors
            si N=P alors N est premier
            sinon N n'est pas premier
        sinon àjouter 1 à P et recommencer l'étape (3)
```

Il faut démontrer que cette suite de consigne est correcte, c'est-à-dire qu'elle permet de conclure correctement pour tous les nombres entiers.

[8] Une règle stipule qu'une partie est nulle lorsque la même configuration est apparue 3 fois : cette règle implique que le nombre de parties est fini.

D'autres suites de consignes peuvent convenir et peuvent être jugées préférables à la précédente. Plus concises, plus claires, mieux structurées, il est plus facile de s'assurer qu'elles permettent de calculer ce qu'on souhaite.

```
Soit P=2.
Tant que P est plus petit que N et que P ne divise pas N
        ajouter 1 à P.
Si P=N alors N est premier sinon N n'est pas premier.
```

On appelle *algorithme*[9] la description d'un calcul à l'aide d'*instructions élémentaires*, de *tests* et autres *structures de contrôles*[10]. On peut définir précisément ce qu'est un calcul en définissant les briques de base des algorithmes constituées par les instructions élémentaires, les tests et les structures de contrôles. Mais il y a un risque alors de définir autant de notions de calculs que d'ensembles de briques de base. Plusieurs mathématiciens[11] de la première moitié du XXème siècle ont proposé, indépendamment les uns des autres, une définition de ce qu'est un calcul : toutes se sont révélées équivalentes. La *thèse de Church-Turing*, adoptée par l'ensemble des mathématiciens et des informaticiens, énonce que ces définitions équivalentes constituent une définition adéquate de la notion de calcul.

Un *langage de programmation* est constitué d'un jeu d'instructions élémentaires ainsi que d'un ensemble de règles d'assemblage de ces instructions. Un *programme* est le codage d'un algorithme dans un langage de programmation particulier. Quelques langages de programmation usuels : Pascal, C, C++, Ada, Prolog[12], Lisp, Smalltalk, Java, ...

L'algorithme ci-dessus, codé en Pascal, donne le programme suivant :

```
P:=2;
while (P<N) and (N mod P <>0) do
        P:=P+1;
If P=N then premier:=true else premier:=false;
```

La définition proposée dans les premières lignes de cette introduction stipule qu'un ordinateur doit être capable en principe d'effectuer *n'importe quel calcul* : comment peut-on en être sûr ? Turing a montré qu'il existe des machines universelles capable d'effectuer tous les calculs définis par la *thèse de Church-Turing*, autrement dit tous les calculs imaginables. Ces *machines universelles* s'appellent des *ordinateurs* ! Toutes ces questions sont étudiées par un domaine de l'informatique théorique qui s'appelle la *théorie de la calculabilité*. Un ordinateur n'est donc pas un simple calculateur ; un ordinateur a la puissance de tous les calculateurs imaginables !

Existe t-il des choses que l'on ne peut pas calculer ? La plupart des fonctions que l'on connaît sont calculables par un ordinateur. Et pourtant, la plupart des fonctions ne sont pas calculables[13] !

[9] D'après Al Khowarizmi, mathématicien persan du IXème siècle.

[10] On distingue les structures *séquentielles*, *conditionnelles* et *itératives* : voir le cours du deuxième semestre.

[11] Parmi eux figure un des plus grands noms de l'informatique, le mathématicien anglais Alan Turing.

[12] Made in Marseille par Alain Colmerauer.

[13] Esquisse de démonstration. L'ensemble des fonctions booléennes, c'est-à-dire définie de $\mathbb{N}$ à valeurs dans $\{0,1\}$, n'est pas dénombrable d'après le théorème de Cantor : un ensemble E ne peut pas être mis en bijection avec l'ensemble de ses parties $\mathcal{P}(E)$. Démonstration : supposer qu'il existe une bijection $f : E \to \mathcal{P}(E)$, considérer l'ensemble $B = \{x \notin f(x)\}$ et l'élément $b = f^{-1}(B)$ de E ; on montre alors que $b \in B$ ssi $b \notin B$.

Un exemple de fonction non calculable : les mathématiques sont composées d'un ensemble d'axiomes (des formules logiques que l'on tient pour vraies) et de règles de déduction logique qui permettent de construire d'autres formules vraies (par exemple, si P est vraie et si $P \Rightarrow Q$ est vraie, alors Q est vraie). Connaissant les axiomes et les règles de déduction, on pourrait espérer programmer un ordinateur pour qu'il dise si oui ou non une formule F est déductible des axiomes[14]. La fonction correspondante est bien définie :

$$f(F) = 1 \text{ si } F \text{ est déductible des axiomes et } 0 \text{ sinon.}$$

Un tel programme pourrait être très utile pour savoir si telle conjecture est vraie ou non ! Malheureusement, on peut prouver qu'un tel programme n'existe pas[15].

La définition proposée ci-dessus mentionne également l'atténuateur *en principe* : que signifie t-il ? Un ordinateur peut calculer tout ce qui est calculable si l'on suppose qu'il peut disposer de tout le temps nécessaire pour mener à bien ses calculs, mais aussi, de tout l'espace nécessaire (mémoire) pour noter des résultats intermédiaires. Cela n'est pas très réaliste. Le domaine de l'informatique qui s'intéresse aux calculs *pratiquement* réalisables s'appelle la *complexité algorithmique*.

Depuis l'antiquité, les hommes aiment se poser des énigmes qui n'ont pas l'air très difficile à résoudre mais dont la résolution effective met en jeu des nombres énormes, si grands qu'il semble impossible de pouvoir les traiter.

Une des anecdotes les plus célèbres a trait à l'invention du jeu d'échecs. Son inventeur ne demanda pour récompense qu'un peu de blé : un grain sur la première case, deux sur la seconde, quatre sur la troisième et ainsi de suite, en doublant le nombre de grains à chaque fois, jusqu'à la soixante quatrième. Le total dépasse, et de loin, la production mondiale de blé !

On a depuis construit des exemples de fonctions calculables mais pratiquement inatteignables bien plus redoutables encore. C'est par exemple le cas de la fonction Ack d'Ackerman, définie de $\mathbb{N} \times \mathbb{N}$ à valeurs dans $\mathbb{N}$ par

- Si $m = 0, Ack(m, n) = n + 1$
- si $m > 0$ et $n = 0, Ack(m, n)$ vaut $Ack(m - 1, 1)$
- si $m > 0$ et $n > 0, Ack(m, n)$ vaut $Ack(m - 1, Ack(m, n - 1))$

On peut facilement se convaincre que cette fonction est calculable. Calculez $Ack(2, 2)$ puis $Ack(5, 5)$.

L'ensemble des fonctions booléennes, qui peut être mis en bijection avec l'ensemble des parties de $\mathbb{N}$, n'est donc pas dénombrable. Or, il n'y a pas plus de fonctions booléennes calculables qu'il n'y a de manières d'écrire un programme ou un algorithme. Comme les programmes sont des suites finies de caractères et qu'il n'y a qu'un nombre fini de caractères disponibles, le nombre de programmes possibles est dénombrable. CQFD.

La démonstration ci-dessus n'est qu'à moitié satisfaisante car elle dit que la plupart des fonctions booléennes sont non calculables sans proposer de méthode pour en construire une ! En revanche, elle permet d'affirmer qu'une fonction booléenne dont les images sont choisies aléatoirement (en tirant à pile ou face) est certainement non calculable.

[14] La question de savoir si un tel programme peut exister a été posée, en d'autres termes, par Hilbert en 1900.

[15] Une première idée pour réaliser un tel programme consiste à faire appliquer toutes les règles de déduction à tous les axiomes de telle manière que le programme balaie toutes les démonstrations possibles. Un tel programme peut effectivement être construit ; il montrera qu'une formule déductible l'est effectivement mais il sera incapable de prouver qu'une formule *n'est pas* déductible. On peut alors imaginer de compliquer le programme en lui soumettant à la fois la formule et sa négation : comme une formule est soit vraie soit fausse (et dans ce cas, sa négation est vraie), on peut espérer conclure. Malheureusement ce n'est pas le cas. Gödel, mathématicien et logicien allemand du début du XXème siècle a montré que pour la plupart des systèmes axiomatiques utilisés, il existe des formules logiques qui ne sont pas déductibles, non plus que leur négation.

1.1.3 Un dispositif matériel

Un ordinateur est une *machine* dont les principaux composants sont :
- le *processeur* ou *Unité centrale* : c'est le cerveau de l'ordinateur, là où s'exécutent les calculs.
- la *mémoire centrale* :
  - la *mémoire vive* (RAM : Random Access Memory) : c'est là que se trouvent les données que le processeur traite et les instructions qu'il exécute.
  - la *mémoire morte* (ROM : Read Only Memory). Contenu permanent, enregistré lors de la fabrication de la machine, nécessaire pour les procédures de démarrage.
- les *périphériques d'entrée/sortie* : c'est grâce à eux qu'un ordinateur peut échanger de l'information avec son environnement. Lorsque l'information passe via le périphérique de l'environnement vers l'ordinateur, on parle de *périphérique d'entrée* ; lorsque la circulation a lieu en sens inverse, on parle de *périphérique de sortie*. Les principaux périphériques sont :
  - l'*écran*. Pour afficher les résultats d'un traitement.
  - le *clavier*. Pour *saisir* des informations.
  - l'*imprimante*. Pour imprimer le résultat d'un traitement sur du papier.
  - la *souris*. Pour sélectionner une partie de l'écran et activer les fonctions correspondantes (on dit *cliquer*).
  - le *lecteur de disquette*. Pour enregistrer et saisir des données sur ou à partir d'un support facilement transportable.
  - le *disque dur*. Mémoire permanente volumineuse qui permet de conserver des données (programmes, résultats, ...).
  - le *scanner*. Pour numériser une information analogique.
  - le *graveur*. Pour inscrire des informations sur un disque compact (souvent de manière irréversible . . . mais les choses changent vite).
  - le *modem*. Pour réaliser une connection avec le réseau téléphonique.
  - les *prises réseaux*. Pour réaliser une connection avec un réseau local puis internet.
  - . . .

Les aspects matériel des ordinateurs seront détaillés dans le prochain chapitre[16].

Nous sommes maintenant en mesure de répondre à la deuxième question posée : *que peut-on faire avec un ordinateur ?* On peut réaliser tous les calculs imaginables, pouvu que l'on dispose d'un algorithme décrivant le calcul, d'un codage des données que l'on souhaite traiter sous forme de données discrètes et pour peu que l'on dispose de suffisamment de ressources, temps et espace mémoire, pour réaliser effectivement le calcul. Quelles sont les limites à ce que l'on peut faire faire à un ordinateur ? Au début des années 50, Alan Turing[17] écrivit un article intitulé *Une machine peut-elle penser ?* qui explore l'idée selon laquelle le cerveau est représentable par une structure de données discrète et la pensée n'est rien d'autre qu'un calcul à l'intérieur de cette structure. Rien ne semble s'opposer alors à ce qu'une machine puisse penser. Le débat n'est toujours pas clos à l'heure actuelle. Cet article a été le point de départ d'un domaine toujours très actif en informatique : *l'intelligence artificielle*.

[16] Pour ceux qui s'intéressent à l'histoire des ordinateurs et de l'informatique en général, nous conseillons *Une histoire de l'informatique* de Philippe Breton, paru aux éditions du Seuil dans la collection Points Sciences. Une recherche sur internet conduit aussi rapidement à des sites très intéressants.

[17] Le principal biographe d'Alan Turing, Andrew Hodges, maintient un site web sur son oeuvre : http ://www.turing.org.uk/turing/.

De manière plus prosaïque, un ordinateur permet de faire du traitement de texte (Word, Latex, ...), du calcul numérique, du calcul formel (Maple, Mathematica), de l'aide à la gestion (Excel, SGBD Access, Oracle), des jeux, des logiciels documentaires, des logiciels éducatifs (NTE), des logiciels de recherche et d'extraction d'information, du traitement de sons et d'images, ... Cette liste n'est évidemment pas exhaustive.

1.1.4 Exercices

1. Combien d'objets sont codables (représentables de manière conventionnelle) avec 1 bit, 8 bits, n bits.
2. On souhaite pouvoir coder 1000 objets. Combien de bits sont nécessaires ?
3. Si l'on veut associer un code binaire à chacun des 60.000.000 de français, combien d'octets seront nécessaires ? Effectuez d'abord un calcul approximatif en utilisant l'approximation $2^{10} \simeq 1000$ puis vérifiez à l'aide d'un calcul exact.
4. Classez les mesures de capacité suivantes par ordre croissant : 100 bits, 10 octets, 4 Ko, 1Mo, 4000 octets.
5. Une image est un ensemble de points colorés (les pixels) : si la palette utilisée comporte 32768 couleurs, combien de bits sont nécessaires pour coder ces couleurs ?
6. Un livre comporte 500 pages, chacune composée de 80 colonnes et 40 lignes. Un caractère est codé sur un octet. Pourra-t-on, sans utiliser de logiciels de compression, stocker ce livre sur une disquette de 1,44 Mo ?
7. Une photo haute résolution comporte 720x480 pixels, la couleur de chaque pixel étant codé sur 24 bits. Pourra-t-on la stocker telle quelle (format bitmap) sur une disquette de 1,44 Mo ?
8. Imaginez des algorithmes capables de calculer les exemples décrits au début de la section 1.1.2.
9. Prouver que l'algorithme de la page 7 calcule bien ce qui lui est demandé.
10. Combien de grains de blés seront (virtuellement) sur l'échiquier à l'issue du processus décrit page 8 ? Faites un calcul exact puis un calcul approximatif.
11. Environ 25000 étudiants sont inscrits à l'université de Provence. Un numéro est attribué à chaque étudiant. Bien évidemment, deux étudiants différents ne doivent pas avoir le même numéro.
    (a) Combien de bits sont nécessaires pour coder un de ces numéros ?
    (b) Les logiciels utilisés ont l'*octet* comme unité de mémoire. Combien d'octets sont nécessaires pour coder un numéro d'étudiant ?
    (c) L'administration de l'université souhaite que les programmes mis au point pour gérer l'inscription et la scolarité des étudiants puissent servir plusieurs années. Si l'on pense que la fréquentation de l'université de Provence peut augmenter significativement (de 20 à 50%) dans les années à venir, combien d'octets faut-il réserver pour coder les numéros des étudiants ? Pourquoi ?
    (d) Pour chaque étudiant, on mémorise en plus de son numéro, son nom, son prénom, sa date de naissance, son adresse, et divers renseignements concernant sa scolarité. En supposant qu'au total un kilo-octet doit être réservé pour chaque étudiant,

pourra t-on stocker l'ensemble de ces renseignements (pour tous les étudiants) sur une disquette ? en mémoire vive (RAM) d'un ordinateur ordinaire ? sur un disque dur de capacité standard ?

(exercice donné à l'examen de deuxième session en 2002).

12. On souhaite effectuer des dessins sur une grille carrée comprenant 64x64 cases (ou pixels). Chaque point est repéré par un couple d'entiers (x, y) où x et y sont compris entre 0 et 63. On supposera que x et y sont écrits en binaire.

    (a) Combien faut-il de bits pour coder un point ?

    On suppose que les dessins seront composés de trois types d'éléments : des segments, des rectangles dont les côtés sont parallèles aux axes et qui pourront être pleins ou vides. Ainsi, le dessin de la figure ci-dessous est composé d'un rectangle vide $ADJI$, d'un rectangle plein $BCFE$ et de deux segments GK et KH.

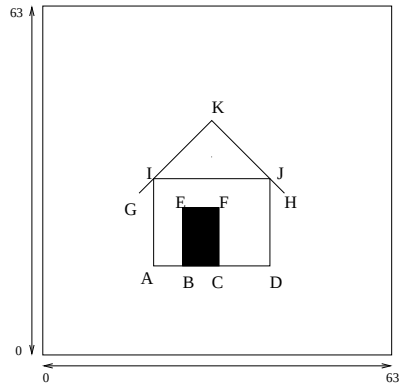


FIG. 1.3 – Un dessin

    - Un segment AB dont les extrémités ont pour coordonnées (x_A, y_A) et (x_B, y_B) sera représenté par 4 octets $01x_A$ $01y_A$ $01x_B$ $01y_B$.
    - Un rectangle vide $ABCD$ dont deux sommets opposés sont A et C sera représenté par 4 octets $10x_A$ $10y_A$ $10x_C$ $10y_C$.
    - Un rectangle plein $ABCD$ dont deux sommets opposés sont A et C sera représenté par les 4 octets $11x_A$ $11y_A$ $11x_C$ $11y_C$.
    - Un dessin est représenté par la suite des représentations des éléments qui le compose.

    On remarque donc qu'on peut savoir si un octet code un élément d'un segment s'il commence par 01, un élément d'un rectangle vide s'il commence par 10 et un élément d'un rectangle plein s'il commence par 11.

    (b) Indiquez quel dessin est représenté par le codage suivant 10001111 10001111 10101111 10101111 01001111 01001111 01101111 01101111 01001111 01101111 01101111 01001111

    (c) Combien faut-il de bits pour représenter le dessin de la figure ?

    On suppose que les $64 \times 64 = 4096$ pixels de la grille sont numérotés selon un ordre conventionnel (par exemple de gauche à droite et de haut en bas). Dans une représentation *bitmap*, un dessin (en noir et blanc) est représenté par une suite de bits $b_1 \dots b_n$ dont le i-ième bit b_i est égal à 0 si le i-ème pixel du dessin est blanc et 1 s'il est noir.

(d) Combien faut-il de bits pour représenter le dessin de la figure dans une représentation *bitmap* ?

On souhaite enrichir les codages possibles en représentant directement des lignes brisées $A_1A_2 \dots A_k$ (c'est-à-dire des réunions de segments A_1A_2, A_2A_3, $\dots$, $A_{k-1}A_k$).

(e) Indiquez comment on pourrait étendre le codage ci-dessus pour représenter de telles lignes brisées ? Indication : les extrémités de la ligne pourront être repérées par 01 et les points intérieurs par 00. Combien faudrait-il alors de bits pour représenter le dessin de la figure ?

1.2 Le matériel : architecture des ordinateurs

Von Neumann rédige en 1945 un texte d'une dizaine de pages dans lequel il décrit les plans d'une nouvelle machine, l'EDVAC (Electronic Discrete Variable Computer). C'est sur cette architecture que fonctionnent encore actuellement la plupart des ordinateurs.

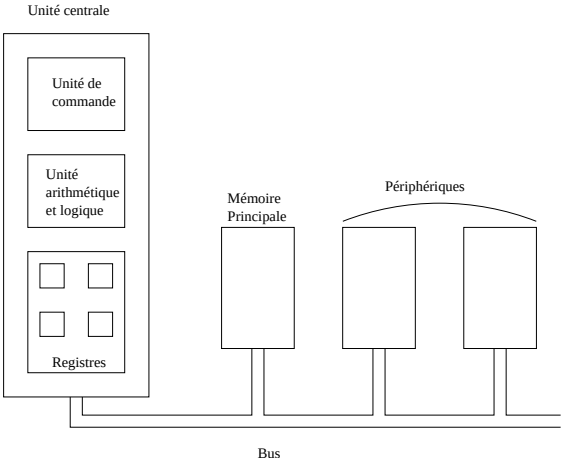


FIG. 1.4 – Organisation d'un ordinateur

Un ordinateur est composé d'un processeur, d'une mémoire principale, de divers périphériques. Ces composants sont reliés entre eux par un ensemble de fils électriques (appelé *bus*), assurant la transmission des signaux. Le processeur comprend deux unités distinctes : une unité de commande dont le rôle est d'aller rechercher les instructions situées en mémoire principale devant être exécutées et de les décoder, et une unité arithmétique et logique (UAL ou ALU), chargée d'exécuter les opérations de base décrites par les instructions.

La mémoire principale

La mémoire principale est souvent appelée *mémoire RAM* (pour Random Access Memory[18]). Il s'agit d'une mémoire *volatile* : lorsque l'ordinateur est éteint, son contenu disparaît. La mémoire principale est composée de *mots-mémoire*, d'une longueur de 2, 4 ou 8 octets, *adressables*, c'est-à-dire identifiables par une adresse qui est un nombre entier représenté en binaire. Une adresse de n bits peut désigner 2^n mots mémoires distincts. Le nombre de bits d'une adresse ne dépend que du nombre de mots mémoire directement adressables et non de leur taille.

Par exemple, une RAM de 64 Mo organisée en mots de 32 bits nécessite des adresses possédant 24 bits. Actuellement, on trouve des mémoires principales de 64, 128, 256 Mo.

Les mémoires sont constitués de circuits intégrés, c'est-à-dire, de millions de transistors placés sur une puce de silicium.

Loi de Moore : "le nombre de transistors intégrés sur une puce double tous les 18 mois". Enoncée en 1965, cette "loi" continue à décrire la réalité.

[18] Cette dénomination m'a toujours semblé un peu énigmatique. On parle aussi en français de *mémoire vive*. Quoiqu'il en soit, la mémoire principale doit être distinguée de la mémoire de masse, matérialisée par les disques durs, disques compacts ou bandes magnétiques, qui sert à mémoriser de l'information sur le long terme.

Le temps d'accès à la mémoire est une donnée critique : il est de quelques millisecondes pour le disque dur (mémoire de masse) et de quelques dizaines de nanosecondes pour la mémoire principale. On peut améliorer le temps d'exécution des algorithmes en introduisant une *mémoire cache*, proche du processeur (et donc plus rapide) de faible capacité servant à conserver les mots mémoires les plus fréquemment utilisés.

Le processeur

Le rôle du processeur ou CPU (Central Processing Unit) ou UC (Unité Centrale) est d'exécuter les programmes stockés en mémoire principale en chargeant les instructions, en les décodant et en les exécutant l'une après l'autre.

L'UC dispose d'une mémoire de travail privée qui lui permet de stocker des résultats temporaires : les *registres*. Le registre compteur ordinal (CO) ou program Counter (PC) contient l'adresse de la prochaine instruction à exécuter. Le registre instruction (RI) contient l'instruction en cours d'exécution. D'autres registres, en nombre variable, servent à stocker des résultats intermédiaires.

Exécution d'une instruction : cycle de chargement décodage exécution

1. Repérer grace au registre CO la prochaine instruction à exécuter et la charger dans le registre RI.
2. Charger dans le compteur ordinal CO l'adresse de l'instruction suivante.
3. Analyser et décoder l'instruction contenu dans le RI.
4. Localiser en mémoire les données nécessaires à l'instruction.
5. Charger ces données dans les registres généraux de l'UC.
6. Faire exécuter l'instruction par l'UAL.
7. Reprendre à l'étape 1 (sauf si l'instruction qui vient d'être exécutée est celle qui demande d'arrêter l'exécution du programme).

L'horloge

Dans un ordinateur, non seulement les données traitées sont discrètes mais le déroulement des opérations se fait aussi selon un temps discrétisé. Une horloge, le plus souvent calibrée à l'aide d'oscillateurs à quartz, émet régulièrement une suite d'impulsions, comme un métronome, qui sert à scander les opérations. Le temps d'un cycle, ou période de l'horloge, est une caractéristique des processeurs : 500 Mhz ou 1 Ghz, c'est-à-dire une impulsion tous les milliardièmes de secondes, deviennent des valeurs courantes.

Exemple

On considère une mémoire RAM contenant 16 mots mémoires d'un octet. Il suffit donc de 4 bits pour adresser un mot mémoire. On suppose l'existence d'un registre accumulateur ACC d'un octet, pouvant contenir n'importe quel mot mémoire. On suppose que les instructions peuvent être codées par les quatre premiers bits d'un mot mémoire :

ADD a. Pour additionner le contenu du registre accumulateur au mot mémoire situé à l'adresse a de la mémoire RAM. Le résultat est stocké dans l'accumulateur ; code 0000.

ADD $\#a$. Pour additionner le contenu du registre accumulateur avec la valeur a. Le résultat est stocké dans l'accumulateur ; code 0001.

JMP a. Pour charger l'adresse a dans le compteur ordinal (instruction de branchement inconditionnel) ; code 0010.

JNZ a. Si le contenu de l'accumulateur est différent de zéro, l'adresse a est chargée dans le compteur ordinal (instruction de branchement conditionnel) ; code 0011.

SHL. Pour décaler les bits du registre accumulateur d'une position vers la gauche (le dernier bit devient égal à 0) ; code 0100.

LOAD a. Pour charger le registre accumulateur avec le mot mémoire situé à l'adresse a ; code 0101.

LOAD $\#a$. Pour charger la valeur a dans le registre accumulateur ; code 0110.

STO a. Pour ranger la valeur contenue dans le registre accumulateur à l'adresse a en RAM ; code 0111.

STOP. Arrêt du programme ; code 1111.

Exemple En reprenant les trois étapes de déroulement d'une instruction décrites plus haut, décrire pour chacune des étapes le contenu du registre accumulateur, du compteur ordinal et de la mémoire lors de l'exécution de la séquence d'instructions commençant à l'adresse 0. Le contenu initial de la mémoire est le suivant :

adresse	contenu
0000	01010111
0001	00010001
0010	00110100
0011	00000110
0100	01110110
0101	11110000
0110	00011110
0111	00010100

Un tel programme est évidemment illisible : on peut améliorer sa compréhension en écrivant les adresses en notation décimale et désignant les instructions par leur nom.

adresse	contenu
0	LOAD 7
1	ADD #1
2	JNZ 4
3	ADD 6
4	STO 6
5	STOP
6	30
7	20

1.2.1 Exercices

1. "La machine de von Neumann était composée de cinq parties : la mémoire, l'unité arithmétique et logique, l'unité de contrôle et les dispositifs d'entrées et de sorties. La mémoire disposait de 4096 mots, chaque mot faisant 40 bits, c'est-à-dire deux instructions de 20 bits ou un entier signé de 40 bits. Les instructions comprenaient 2 champs : 8 bits pour le type d'instruction et 12 bits pour adresser un des 4096 mots de la mémoire". Cette citation est extraite de *Architecture de l'ordinateur*, A. Tannenbaum, Eds Dunod. Assurez-vous que vous comprenez tous ses termes.

2. Quelle est la distance parcourue par la lumière pendant un cycle d'un processeur cadencé à 1GHz ?

3. On suppose que la mémoire RAM d'un ordinateur contient 16 mots-mémoire d'un octet chacun. Chaque mot mémoire peut donc être désigné par un entier de 0 à 15, représentable sur 4 bits (son adresse). L'Unité Centrale de cet ordinateur contient
   – un registre CO (compteur ordinal) de 4 bits qui contient l'adresse de la prochaine instruction à exécuter,
   – un registre RI (registre instruction) de 4 bits qui contient l'instruction en cours d'exécution,
   – un registre ACC (accumulateur) d'un octet pouvant contenir n'importe quel mot mémoire.

   On considère 4 instructions, codée chacune sur un mot mémoire.
   – LOAD $x_1x_2x_3x_4$, codée par $0000x_1x_2x_3x_4$: cette instruction charge la valeur contenue à l'adresse $x_1x_2x_3x_4$ de la mémoire RAM dans le registre ACC.
   – STO $x_1x_2x_3x_4$, codée par $0010x_1x_2x_3x_4$: cette instruction charge la valeur contenue dans l'accumulateur à l'adresse $x_1x_2x_3x_4$ de la mémoire RAM.
   – JNZ $x_1x_2x_3x_4$, codée par $0011x_1x_2x_3x_4$: si le contenu de l'accumulateur est différent de 0, cette instruction écrit l'adresse $x_1x_2x_3x_4$ dans le compteur ordinal CO ; si le contenu de l'accumulateur est égal à 0, cette instruction ne fait rien.
   – STOP, codée par 11111111 : cette instruction arrête le déroulement du programme.

   On rappelle que le déroulement d'un programme se fait de la manière suivante :

   i. Charger le contenu du CO dans le RI

   ii. Modifier le CO pour qu'il désigne l'instruction suivante

   iii. Analyser et décoder l'instruction figurant dans le RI

   iv. Localiser en mémoire d'éventuelles données nécessaires à cette instruction

   v. Exécuter l'instruction

   vi. Reprendre à l'étape (a) si la dernière instruction exécutée est différente de STOP.

   (a) Décrivez l'état de la mémoire RAM après le déroulement du programme lorsque le contenu du CO est égal à 0000 et lorsque la mémoire RAM contient initialement les valeurs suivantes.

adresse	RAM
0000	00000000
0001	00111000
0010	00001010
0011	00101100
0100	00001011
0101	00101010
0110	00001100
0111	00101011
1000	11111111
1001	00000001
1010	11110000
1011	00001111
1100	10101010
...	...

(b) Même question si la première ligne était remplacée par

0000|00000001.

(c) Même question si en plus, la seconde ligne était remplacée par

0001|00110000.

Chapitre 2

Système d'exploitation

2.1 Définition et rôle

L'ordinateur que nous avons décrit dans le chapitre précédent serait bien difficile à utiliser. En effet, le matériel seul ne sait pas faire grand chose :

- il peut exécuter des programmes, mais comment entrer ces programmes dans la machine ? Comment les charger en mémoire ?
- il peut sauvegarder des données sur disque mais comment accède-t-on à un disque ? Comment range-t-on les informations sur le disque ?
- il peut lire ce qui est tapé au clavier à condition qu'on écrive un programme chargé de cette tâche.

Bref, on tourne en rond. Nous avons besoin d'un ensemble de programmes (un logiciel) qui permette d'exploiter les ressources matérielles (*périphériques*, *mémoire*, *processeur(s)*). Ce logiciel, c'est le *système d'exploitation*. Il possède principalement les rôles suivants :

- gestion des ressources : proposer à l'utilisateur une interface d'accès aux ressources de la machine, partager ces ressources entre plusieurs utilisateurs et/ou programmes ;

- sécurité : empêcher les programmes mal écrits de nuire au fonctionnement de la machine, empêcher les accès non autorisés à certaines ressources de la machine ;

- abstraction : dans le cadre de machines utilisables par plusieurs utilisateurs simultanément, proposer une abstraction présentant à l'utilisateur une machine virtuelle.

Notons qu'une machine n'est pas liée à un système d'exploitation. Notamment, en ce qui concerne les micro-ordinateurs, les systèmes disponibles sont variés, et coûtent plus ou moins cher suivant le système de licence qui leur est attaché. Par exemple :

PC Pour les ordinateurs PC (i.e. à base de processeurs Intel : 386, 486, Pentium I II et III, ...) :

- Windows95/98/2000/NT, produits Microsoft : assez onéreux ;
- Linux, produit de la communauté du logiciel libre[1] : disponible gratuitement ;

[1] Quelques extraits de la page "Qu'est-ce qu'un logiciel libre ? " consultable à l'adresse http ://www.gnu.org/philosophy/free-sw.fr.html.

L'expression "Logiciel libre" fait référence à la liberté pour les utilisateurs d'exécuter, de copier, de distribuer, d'étudier, de modifier et d'améliorer le logiciel. Un programme est un logiciel libre si les utilisateurs ont toutes ces libertés. L'expression « Logiciel libre » fait référence à la liberté et non pas au prix. « Logiciel libre » ne signifie pas « non commercial ».

Macintosh Pour tous les ordinateurs Apple (Mac, iMac, iBook, ...) :

- MacOS : livré avec tous les mac ... mais on le paye.
- Linux : existe aussi pour mac !

2.2 Organisation en couches

On peut voir l'ordinateur et les différents programmes qui l'accompagnent selon un modèle à couches présenté à la figure 2.1. Plus précisément, on peut décomposer les couches de la façon

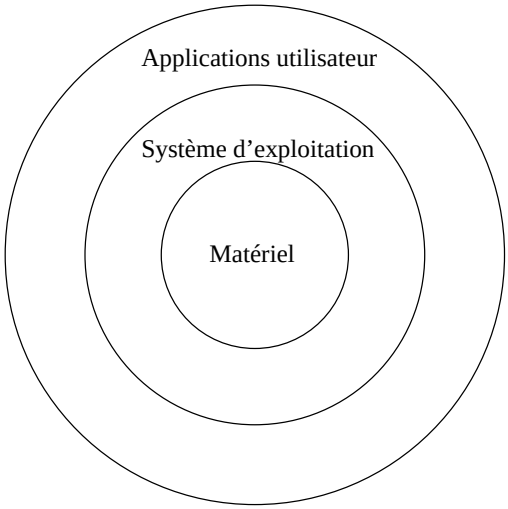
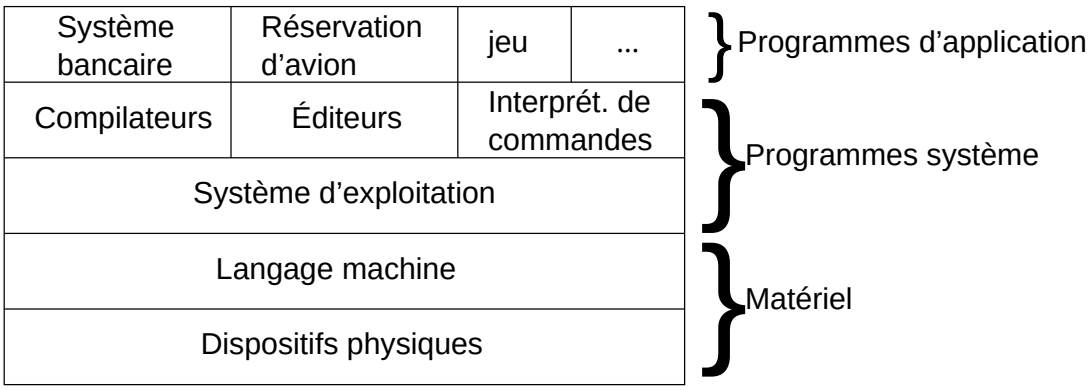


FIG. 2.1 – Le modèle en couches d'un ordinateur

suivante :



Le *matériel* se situe au niveau le plus bas et il est composé lui même de deux couches ou plus. La couche la plus basse contient les circuits physiques : circuits intégrés, fils électriques, alimentations, tubes cathodiques, ... Nous ne nous occuperons pas plus en détail de cette couche qui concerne les spécialistes de l'électronique.

Vient ensuite le logiciel de base qui contrôle ces différents dispositifs et fournit une interface plus simple à la couche suivante. Ce logiciel, appelé *microprogramme*, se situe généralement

dans des mémoires mortes[2]. C'est le programme qui charge les instructions en langage machine et les exécute les unes après les autres. L'ensemble des instructions que le microprogramme interprète constitue le *langage machine*, qui ne fait pas réellement partie du matériel, même s'il est souvent décrit comme tel par les constructeurs. Sur certaines machines (cas des micro-ordinateurs), le microprogramme est implanté directement dans le matériel et ne constitue pas à lui seul une couche distincte.

Le langage machine comprend entre 50 et 300 instructions environ, permettant essentiellement de déplacer des données, d'effectuer des calculs et de comparer des valeurs. Dans cette couche, les périphériques d'E/S sont contrôlés en chargeant des valeurs dans des registres spéciaux. Exemple : mettre un disque en mode lecture, en chargeant ses registres avec l'adresse du disque, l'adresse de la mémoire principale, le nombre d'octets à transférer et le sens du transfert (lecture ou écriture). Dans la pratique, de nombreux autres paramètres sont nécessaires.

La fonction primordiale d'un *système d'exploitation* est de masquer cette complexité et de présenter au programmeur un ensemble d'instructions plus simples à utiliser. Par exemple, LIRE UN FICHIER est conceptuellement plus simple que d'avoir à se soucier de déplacer les têtes de lecture, attendre qu'elles se positionnent, ...

Le reste du logiciel se trouve au dessus du système d'exploitation. On y trouve :

l'interpréteur de commandes (*shell*), permettant d'accéder aux fonctions du système de manière simple, à l'aide d'un *langage de commande* ;

les compilateurs, chargés de traduire des programmes écrits dans des langages de haut niveau en une suite d'instructions en langage machine ;

les éditeurs permettant de saisir et modifier du texte (par exemple des programmes) ;

et d'autres programmes qui ne dépendent pas non plus de la couche applicative.

Enfin, au dessus, on trouve les programmes d'application écrits pour résoudre des problèmes spécifiques : calcul scientifique, traitement de données commerciales, jeux, ...

2.3 Le système Unix (Linux) : présentation rapide

Unix est un système d'exploitation conçu au début des années 70 (il a beaucoup évolué depuis). Sur les machines que nous utiliserons en TP, on trouve une version particulière d'Unix appelée Linux. Nous aborderons les principales notions d'un système d'exploitation en nous appuyant sur sa terminologie.

Quelques «propriétés» du système Unix :

- C'est un système d'exploitation *multi-tâches* : il est capable d'exécuter plusieurs programmes en concurrence («simultanément», nous reviendrons sur cette notion) ;
- C'est un système d'exploitation *multi-utilisateurs* : plusieurs utilisateurs peuvent être connectés simultanément au système par le biais de terminaux (ensemble écran/clavier).

Par exemple, et nous y reviendrons, en TP, les étudiants utilisent le même ordinateur, le même processeur, via des terminaux différents.

Le fait qu'un système soit multi-utilisateur suppose généralement un système de droits permettant de garantir une certaine privauté des données et programmes de chaque utilisateur. Cela entraine généralement (et c'est le cas pour Linux et Unix en général) la présence d'un utilisateur particulier, l'*administrateur système*, dont le rôle est d'attribuer les droits d'accès aux différents utilisateurs.

[2]ROM : Read Only Memory. Mémoire pouvant être accédée uniquement en lecture.

2.4 Principes

L'interface entre le système d'exploitation et les programmes de l'utilisateur est constituée d'un ensemble d'instructions étendues fournies par le système d'exploitation. On parle d'*appels système*. Les appels systèmes créent, détruisent et utilisent divers objets logiciels gérés par le système d'exploitation. Les processus et les fichiers introduits dans les sections suivantes sont les plus importants de ces objets.

2.4.1 Les processus

Le *processus* est un concept clé commun à tous les systèmes d'exploitation. Fondamentalement, *un processus est un programme qui s'exécute*. Il consiste en un programme exécutable, ses données, ses registres, son compteur ordinal, et toutes autres informations nécessaires à son exécution.

Le système Unix est capable d'exécuter plusieurs processus de façon concurrente (en parallèle). On parle de système *multi-tâches*. Comment cela est-il possible alors que la machine ne possède qu'un seul processeur ? Le système d'exploitation commence par allouer le processeur à un processus pour un certain laps de temps (appelé quantum). Quand ce temps est écoulé, ou que le processus est en attente d'un périphérique, ou que le processus arrive à la fin de son exécution, le système alloue un quantum de temps à un autre processus en attente. Ce cycle se répète indéfiniment. À chaque fois qu'un processus est interrompu par le système, celui-ci sauvegarde toutes les informations nécessaires à la reprise de l'exécution de ce processus à un moment ultérieur : c'est le contexte du processus, qui est sauvegardé dans la table des processus gérée par le système. Un exemple d'exécution concurrente de processus est représenté à la figure 2.2.

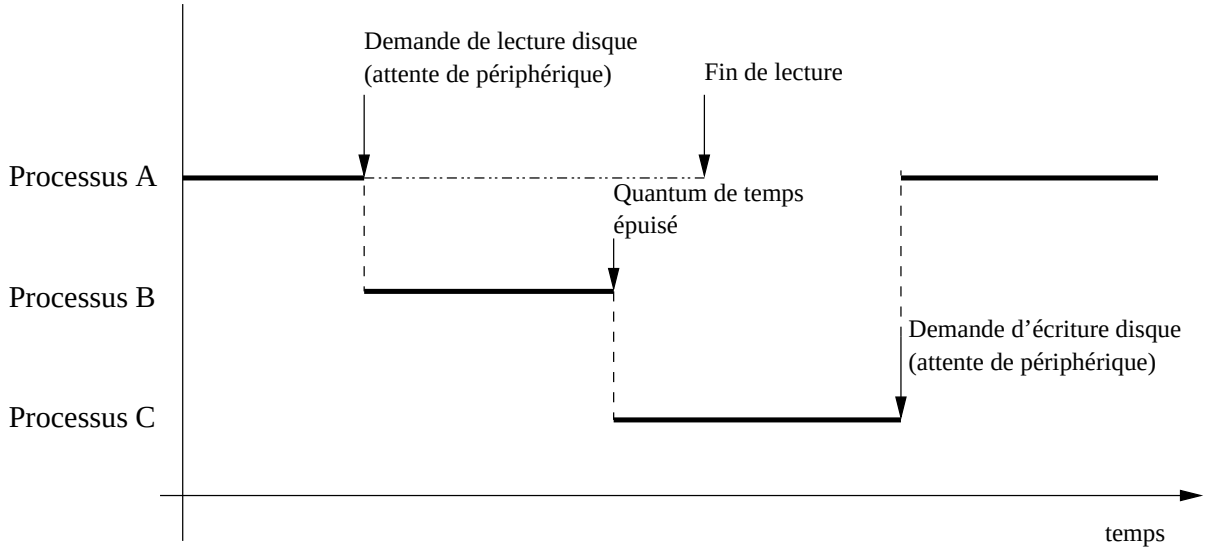


FIG. 2.2 – Exécution concurrente de trois processus.

Les appels systèmes les plus importants pour la gestion des processus sont ceux qui concernent la création et la fin d'un processus. Considérons un exemple typique. Un processus appelé *interpréteur de commandes* ou *shell* lit des commandes issues d'un terminal. Si

l'utilisateur demande l'exécution d'un programme particulier (par exemple un compilateur), le shell doit créer un nouveau processus qui lancera l'exécution du programme en question. Ce processus fera, à la fin de l'exécution du programme, un appel système pour se terminer.

Un processus peut créer d'autres processus (appelés *processus fils*), qui eux même peuvent créer d'autres processus, ... On obtient une structure en arbre, comme l'illustre la figure 2.3. Dans un système multi-tâches et multi-utilisateurs, il est important de mémoriser l'association

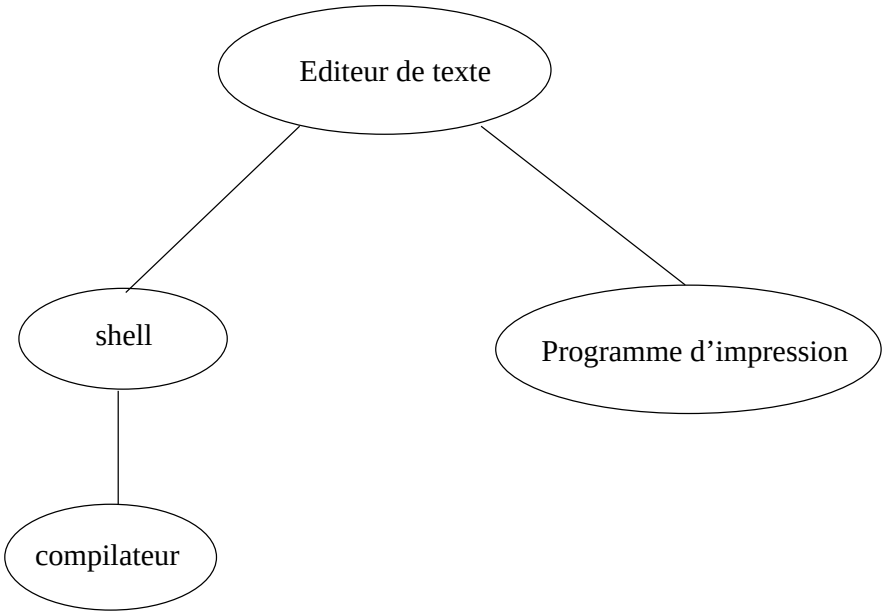


FIG. 2.3 – Arbre des processus. Le processus «éditeur de texte» a créé deux processus fils : un shell (interpréteur de commandes) et un programme d'impression. Dans le shell, l'utilisateur a lancé une commande, ce qui donne naissance à un processus fils de ce shell.

entre un processus et un utilisateur. Dans un système de ce type, on attribue à chaque utilisateur un *uid* (*user identification*) qui est un nombre entier. On attribue également à chaque processus l'*uid* de son propriétaire.

2.4.2 Les fichiers

La deuxième grande catégorie d'objets gérés par le système est le fichier. Comme mentionné précédemment, un des rôles du système d'exploitation est de masquer les spécificités des disques et des autres périphériques d'E/S et d'offrir au programmeur (et à l'utilisateur) un modèle agréable et facile d'emploi qui lui permette d'accéder aux ressources de ces périphériques. C'est le but de la notion de fichier.

Un fichier est une collection d'informations, indépendante du matériel sous-jacent. Les appels systèmes permettent de créer des fichiers, de les supprimer, de les lire et de les écrire. Il faut ouvrir un fichier avant de le lire ou de l'écrire, et le fermer après utilisation. Un fichier peut contenir à peu près n'importe quoi : un programme exécutable, un texte, une image, du son, ... Le contenu et l'organisation des informations dans un fichier dépendent de l'application qui crée ce fichier et le remplit.

La plupart des systèmes d'exploitation font appel au concept de répertoire (directory) pour regrouper les fichiers suivant leur utilisation. Par exemple, un étudiant peut avoir un répertoire

par enseignement suivi, un autre pour le courrier électronique, ... Un répertoire peut contenir d'autres répertoires : on parle de *répertoire père* et de *répertoires fils*. Ce modèle engendre donc lui aussi une hiérarchie, le système de fichiers, décrite à la figure 2.4. Chaque fichier dans

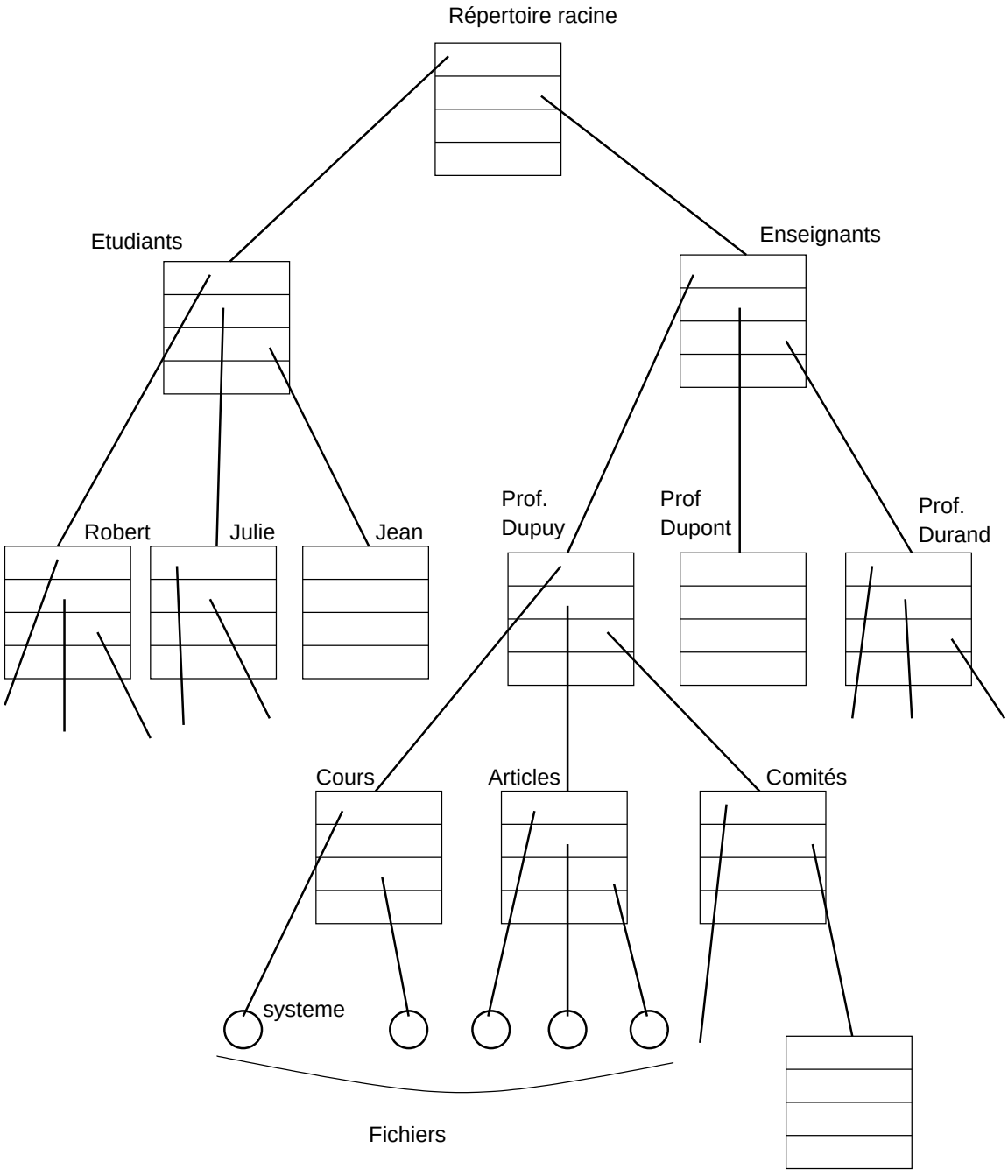


FIG. 2.4 – Un exemple de système de fichiers

un répertoire peut être atteint en spécifiant son *chemin d'accès* (*path*) à partir du haut de la hiérarchie, le *répertoire racine* (*root directory*). Ces chemins d'accès absolus sont formés de la liste des répertoires qu'il faut traverser à partir de la racine pour accéder au fichier, les

répertoires de la liste étant séparés par des barres obliques (*slash*). À la figure 2.4, le chemin d'accès au fichier `systeme` est `/Enseignants/Prof.Dupuy/Cours/systeme`. La première barre oblique indique que le chemin d'accès est absolu, c'est-à-dire qu'il commence à la racine.

Un *chemin d'accès relatif* est décrit depuis un répertoire particulier, en désignant une suite de répertoire permettant d'accéder à un fichier. On reconnaît un chemin relatif à ce qu'il ne commence pas par un slash. En UNIX, le répertoire père est désigné par ... Dans la figure 2.4, `Cours/systeme` est un chemin d'accès relatif au fichier `systeme` depuis le répertoire `/Enseignants/Prof.Dupuy`. De même, `../Prof.Dupuy/Cours/systeme` est un chemin d'accès relatif au même fichier depuis le répertoire `/Enseignants/ProfDupont`.

À tout moment, chaque processus possède un répertoire de travail particulier. C'est à partir de ce répertoire de travail que sont interprétés les chemins d'accès relatifs. En UNIX, le répertoire courant d'un processus est désigné par un point (.). Enfin, le répertoire principal d'un utilisateur est désigné par `~`.

Les processus ont la possibilité de changer de répertoire de travail au cours de leur durée de vie, via un appel système qui indique le nouveau répertoire de travail.

Plusieurs utilisateurs pouvant accéder simultanément à une machine fonctionnant sous le système d'exploitation Unix, il est important de protéger les fichiers de chaque personne. Unix utilise pour cela un code de protection de 9 bits. Ce code se décompose en trois groupes de trois bits chacun :

- le premier groupe concerne le propriétaire du fichier ;
- le second groupe concerne le groupe du propriétaire (les utilisateurs sont répartis en groupes par l'administrateur du système) ;
- le troisième groupe concerne les autres utilisateurs du système.

Chaque groupe de 3 bits s'organise ainsi :

- un bit d'autorisation de lecture (*read*) ;
- un bit d'autorisation d'écriture (*write*) ;
- un bit d'autorisation d'exécution (*execute*).

Ces trois bits sont appelés *bits rwx*. Le code `rwx|r-x|--x` signifie que :

- le propriétaire peut lire, écrire et exécuter le fichier ;
- les membres du groupe auquel appartient le propriétaire peuvent lire et exécuter le fichier (ils ne peuvent pas l'écrire).
- les autres utilisateurs ne peuvent qu'exécuter le fichier (mais ils ne peuvent ni le lire ni l'écrire).

Dans le cas d'un répertoire, le `x` indique une autorisation de recherche. Un tiret indique une interdiction. La commande «`ls -l`» permet de visualiser ces permissions. Elle les affiche sous la forme `rwxr-x--x`.

Notons qu'Unix offre une abstraction supplémentaire en permettant de traiter n'importe quel périphérique comme un fichier (par exemple, imprimer un fichier revient à lire le fichier et l'écrire sur le fichier spécial correspondant à imprimante).

Un dernier point que nous aborderons dans cette présentation générale est la notion de *tube* de communication (*pipe*, voir figure 2.5). Un tube est un pseudo-fichier qui peut être utilisé pour relier deux processus. Si un processus A veut envoyer des données à un processus B, il écrit dans le tube comme s'il écrivait dans un fichier normal. Le processus B peut lire les données du tube comme s'il lisait un fichier normal.

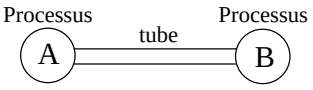


FIG. 2.5 – tube de communication entre deux processus

Exemple : La commande «`ls`» du shell permet d'afficher le contenu d'un répertoire. Si celui-ci contient plus de fichiers que l'écran peut en afficher, nous ne verrons pas tous les fichiers ! Une autre commande, «`more`», permet d'afficher page par page le texte qu'elle reçoit en entrée. Donc, pour afficher le contenu d'un répertoire page par page, il suffit d'établir un tube de communication entre un processus exécutant `ls` et un autre exécutant `more`, ce qui, dans la syntaxe du shell, s'écrit :

```
ls | more
```

Cette ligne de commande signifie que `ls` envoie ses résultats à la commande `more` au lieu de les afficher à l'écran.

2.5 L'accès au système

Cette section décrit le système utilisé en salle de TP. Les salles de TP utilisent une machine de type PC équipée de deux processeurs Intel Pentium III à 600Mhz. Sur cette machine est installé le système d'exploitation Linux (une version d'Unix libre de droits). À cette machine sont connectés des terminaux X (Tx) via un réseau de type Ethernet. Un terminal X est constitué d'un clavier et d'un écran (et de son contrôleur graphique). La dénomination X provient de X-Windows, qui est un environnement de travail graphique à fenêtres propre au monde Unix. L'architecture est décrite à la figure 2.6. Tout se passe donc comme si vous aviez chacun une machine Linux, alors que vous partagez une machine. On voit ici concrètement la tâche d'abstraction effectuée par le système d'exploitation.

Quand vous allumez un Tx, après un temps d'initialisation d'environ une minute, vous demande de vous identifier. Le système Linux étant multi-utilisateurs, chaque utilisateur possède un identifiant (un nom, appelé *login* ou *user id*) et un mot de passe.

Après avoir saisi à l'aide du clavier votre login et votre mot de passe, le système exécute un programme, l'environnement de travail X-Windows, et vous présente (entre autres) un interpréteur de commandes (shell) dans une fenêtre. En bas de l'écran, la fenêtre du gestionnaire de bureau comporte des menus permettant d'exécuter différents programmes.

Comment terminer votre TP dans la salle du SCAM au 3ième étage ?

Après le démarrage d'un ordinateur du SCAM, ouvrez une fenêtre shell. Puis, en supposant que votre login est `rmorin`, lancez `ssh -X rmorin172.22.0.41` puis entrez votre mot de passe. Vous pourrez alors terminer votre TP.

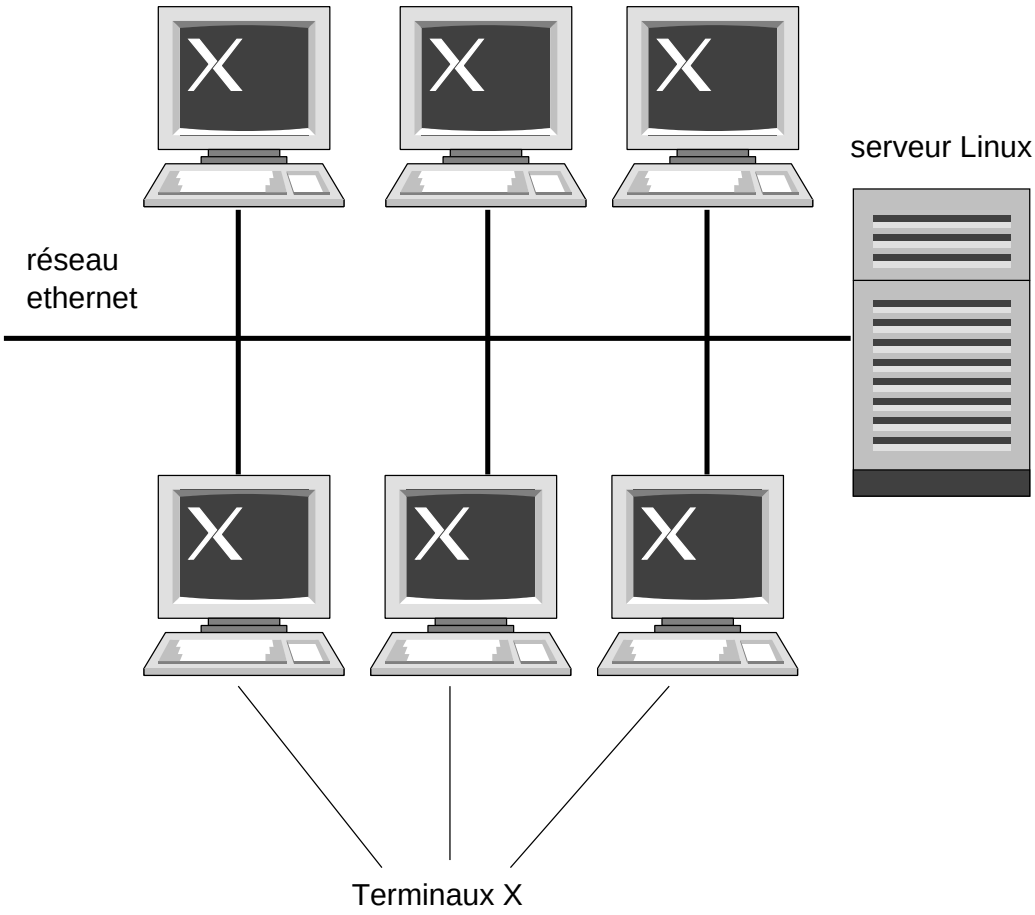


FIG. 2.6 – Architecture des salles de TP.

Chapitre 3

Le Shell

3.1 Généralités

3.1.1 Conventions typographiques

Dans ce chapitre, les noms des fichiers et des commandes UNIX seront écrit dans la fonte que voici. Lorque cette même fonte *sera utilisée en italique*, cela designera un nom générique qui devra être remplacé par un vrai nom lors de l'exécution.

Le *shell* (coquille qui entoure le "noyau" du système) est un *interpréteur de commandes*, c'est-à-dire un programme capable de comprendre et d'exécuter un ensemble de commandes paramétrables, qui sert d'intermédiaire entre l'utilisateur et le système d'exploitation. Il existe plusieurs shells pour LINUX, essentiellement *Bourne Again Shell* ou *BASH* et *TCSH*. Nous étudierons le BASH.

Structure d'une commande

Une commande comprend généralement :

1. un nom
2. des options éventuelles, précédées du caractère '-', qui paramètrent la commande
3. des arguments éventuels.

Règles de base

1. Les systèmes UNIX (et donc LINUX) différencient les minuscules et les majuscules : toto.txt, toto.TXT, TOTO.txt, TOTO.TXT, ToTo.TxT sont donc autant de fichiers différents
2. Toutes les commandes UNIX s'écrivent en minuscule
3. L'emplacement des espaces est évidemment important
4. Les commandes UNIX sont construites à partir du mot anglais (list, remove, manual, ...), en enlevant un maximum de voyelles, puis les consonnes inutiles pour différencier deux commandes, ce qui donne ls, rm, man
5. Les arguments peuvent soit être écrits explicitement, soit être traités globalement grâce aux caractères *jokers* (*wildcards* en anglais), qui permettent de désigner des noms de fichiers ayant des caractéristiques communes. Par exemple :

* : désigne toute suite de caractères (éventuellement vide). Par exemple, a* désigne tous les fichiers dont le nom commence par a ; *.txt désigne tous les fichiers dont dont les noms se terminent par .txt

? : remplace un seul caractère. Par exemple, tot ?.txt comprendra les noms toto.txt, toti.txt, tot2.txt, mais pas toto2.txt.

$[c_1 c_2 \dots c_n]$: désigne un caractère parmi $\{c_1, c_2, \dots, c_n\}$. Par exemple, file[ABC] désignera les trois noms fileA, fileB et fileC.

$[c_1 - c_2]$: désigne tous les caractères compris entre c_1 et c_n dans l'ordre ASCII. Par exemple, file[A-C] désignera aussi les trois noms fileA, fileB et fileC.

3.1.2 Variables d'environnement

Dans tous les systèmes UNIX, il existe des variables d'environnement, qui servent à configurer l'environnement de l'utilisateur. Ces variables sont toujours écrites en majuscules. On peut en obtenir la liste en utilisant la commande set. Les plus utilisées sont (entre autres) PATH, qui contient la liste des répertoires où sont situés les divers programmes du système. Outre les valeurs par défaut (/usr/bin, ...), tout utilisateur peut y ajouter ses propres répertoires (par exemple ~/bin). Cela permet de lancer une commande par son simple nom, sans se préoccuper de son emplacement exact. La variable PWD contient l'adresse du répertoire courant. On peut également citer PRINTER qui contient le nom de l'imprimante par défaut : celle qui sera utilisée sauf si on spécifie explicitement qu'on souhaite en utiliser une autre. Enfin, certains logiciels demandent également la création de variables d'environnement qui leur sont propres, et qui leur permettent de retrouver les fichiers dont ils ont besoin, et dont l'emplacement peut varier d'un système à l'autre. Par défaut, il y a moins trois variables définie pour tout nouvel utilisateur :

- HOME donne le chemin du répertoire principal de l'utilisateur
- USER donne le nom d'utilisateur
- PATH a une valeur par défaut donnée par le système

Pour afficher la valeur d'une variable V : echo $V

Attention, le nom de la variable doit être précédé du charactère '$', afin d'indiquer que la chaîne de charactère qui suit doit être traitée comme un nom de variable et non comme une chaîne normale.

Pour attribuer une nouvelle valeur à une variable V : V="valeur"

On peut aussi vouloir compléter la valeur existante : V="$Vcomplément"

Exemples :

PRINTER="*mon_imprimante*"
echo PRINTER
PRINTER
echo $PRINTER
mon_imprimante
PRINTER=*mon_imprimante*
PATH="$PATH :$HOME/bin"

3.1.3 Répertoires généraux

On retrouve certains répertoires dans toutes les hiérarchies sur tous les systèmes UNIX :

- /usr est le répertoire contenant le système proprement dit. En particulier, /usr/bin contient les commandes du système.
- /dev contient tous les fichiers relatifs aux différents périphériques de l'ordinateur, en particuliers l'écran, mais aussi les disques, les différents lecteurs (CDROM, bande, ...)
- /tmp est un répertoire qui contient les fichiers temporaires créés et utilisés par les différents programmes en cours.
- /home contient en général (mais ca n'a rien d'obligatoire) les répertoires principaux des utilisateurs du système.

3.2 Commandes de base (internes) :

3.2.1 L'aide en ligne : la commande man

Elle permet de consulter le manuel du shell à propos d'une commande.

man *commande* donne la syntaxe de la commande. Les pages de manuel contiennent entre autres les rubriques suivantes :

- NOM : donne le nom de la commande et un bref descriptif
- SYNOPSIS : donne la liste de toutes les options et arguments possibles
- OPTIONS : explicite le fonctionnement de toutes les options
- DESCRIPTION : description détaillée du fonctionnement de la commande de son bon fonctionnement. Cette valeur n'apparaît pas à l'écran, mais peut être utilisée dans un programme. Cette rubrique explicite ces différentes valeurs
- EXEMPLES : donne des exmples d'utilisation. C'est souvent une rubrique très utile ...
- VOIR AUSSI : liste des commandes, fichiers, ... sur des sujets proches. Rubrique parfois utile quand on ne sait pas exactement ce qu'on cherche.

3.2.2 Lister des fichiers dans un répertoire : ls

Exemples :

- ls : donne la liste des fichiers et répertoires du répertoire courant
- ls *rep* : idem pour le répertoire *rep*
- ls a* : donne la liste des fichiers commençant par 'a'

Options utiles :

- ls -l : liste tous les arguments des fichiers (taille, date, propriété ...)
- ls -t : liste les fichiers par date de dernière modification
- ls -a : liste tous les fichiers, y compris ceux dont le nom commence par un '.'
- ls -R : affiche aussi le contenu de tous les sous-répertoires (récursivement).

Exemple d'affichage avec ls -l :

```
drwxr-xr-x   2 dupont   informat    1024 Oct  8 10:54 AutresCours/
-rw-r--r--    1 dupont   informat    38437 oct2 3 15:09 cours.tex
```

3.2.3 Se déplacer dans les répertoires : cd

Exemples :

- cd : sans argument, retourne dans le répertoire principal (identique à cd ~)
- cd *rep* : pour aller dans le sous-répertoire *rep*
- cd .. : pour remonter dans le répertoire parent

– cd *rep/sous_rep* : pour aller dans le sous répertoire *sous_rep* du répertoire *rep*

Les répertoires peuvent être utilisés avec leurs de chemin (*path* en anglais) absolu (c'est-à-dire depuis la racine) ou relatif (au répertoire dans lequel on se trouve).

Exemple. Le chemin d'accès absolu à mon répertoire personnel est /HOME/DUPONT ; j'ai créé plusieurs sous-répertoires de telle sorte que les chemins /HOME/DUPONT/DOC/COURS/INFO et /HOME/DUPONT/PGS sont valides. Comment aller dans le second répertoire à, partir du premier. Il y a plusieurs solutions :

– cd .. , cd .. , cd .., cd pgs (on remonte de 3 niveaux, on est dans ~*username*, on va dans pgs)
– cd ../../../pgs (la même solution condensée en une seule commande)
– cd ~/pgs
– cd /home/dupont/pgs.

3.2.4 Savoir où on est : pwd

La commande pwd retourne le chemin absolu du répertoire courant.

3.2.5 Créer un répertoire : mkdir

– mkdir toto : crée le répertoire toto dans le répertoire courant
– mkdir ../toto : idem dans le répertoire parent

3.2.6 Détruire un fichier : rm

rm toto.txt détruit le fichier en question. **ATTENTION :** un fichier supprimé ne peut plus être retrouvé.

Options

-i : demande une confirmation.

-R ou -r : pour supprimer récursivement toute une hiérarchie de fichiers. Doit être utilisé avec précaution.

3.2.7 Détruire un répertoire vide : rmdir

La commande rmdir ne permet de détruire un répertoire que s'il est vide. Pour détruire un répertoire non vide, utiliser rm -R.

3.2.8 Déplacer/copier un fichier : mv/cp

cp permet de dupliquer un fichier en le renommant éventuellement.

Exemple : mv ../docs/toto.txt ~/pgs Si le répertoire ~/pgs existe, le fichier toto.txt est dupliqué dans ce répertoire en conservant son nom. Si le répertoire ~/pgs n'existe pas, le fichier toto.txt est dupliqué dans le répertoire principale de l'utilisateur (~) sous le nom pgs.

mv est utilisé dans les mêmes conditions que mv pour déplacer un fichier en le renommant éventuellement.

Exemple : la commande mv toto.txt mv toto2.txt a pour effet de renommer le fichier toto.txt en toto2.txt.

3.2.9 Visualiser un fichier : cat/less/more/head/tail

Ces quatre commandes permettent de visualiser le contenu d'un fichier ASCII. Attention, ce ne sont pas des éditeurs, vous ne pouvez donc pas modifier les fichiers. Le mode de fonctionnement est un peu différent :

- cat fait défiler tout le contenu du fichier du début à la fin. La commande cat sert également à *concaténer* deux fichiers, c'est-à-dire à les joindre (voir plus loin).
- less fait défiler le contenu écran par écran. A utiliser pour des fichiers longs. Pour afficher une ligne de plus : touche Entrée. Pour afficher un écran de plus : barre d'espace. Pour interrompre la commande : touche q.
- more presqu'identique à less, en moins bien.
- head affiche le début du fichier (nombre de lignes en option).
- tail affiche la fin du fichier (nombre de lignes en option).

3.2.10 Comparer deux fichiers : diff

diff *file1 file2* compare les deux fichiers, et liste leurs différences ligne à ligne. Consulter le manuel (man diff) pour les options.

3.2.11 Pour modifier les droits d'accès à un fichier : chmod

chmod x+y *chemin d'accès à un fichier ou à un répertoire* : pour ajouter des droits.

chmod x-y *chemin d'accès à un fichier ou à un répertoire* : pour supprimer des droits.

x prend ses valeurs parmi ugoa (u pour user, g pour group, o pour others, a pour all) et y prend ses valeurs parmi rwx (r pour read, w pour write, x pour exécuter).

3.2.12 Visualiser/arrêter un processus : ps/kill

La commande ps permet d'afficher la liste des processus en cours. De nombreuses options, dépendant du shell utilisé et de ses versions, permettent d'obtenir diverses informations : consultez le manuel.

Pour arrêter un processus, on utilise la commande kill, **si on ne sait pas faire autrement**. Exemple : kill pid. On peut envoyer un signal plus ou moins fort. Le plus fort est kill pid -9.

3.2.13 Retrouver un fichier dans une arborescence : find

Cette commande permet de retrouver un fichier par son nom dans une arborescence. Syntaxe : find *directory* -name *filename* recherche le fichier *filename* dans **tous les sous répertoires** du répertoire *directory*. L'utilisation des '*' et '?' dans le nom du fichier est autorisée.

3.2.14 Retrouver une chaîne de caractère dans un fichier : grep

Cette commande permet d'identifier un fichier contenant une chaîne de caractère donnée : grep -e *string filename* donne la liste des occurences de la chaine de caractère *string* dans le fichier *filename*. On peut aussi utiliser les '*' ou '?', pour retrouver tous les fichiers contenant la chaîne de caractère souhaitée.

3.2.15 Modifier la priorité d'exécution d'un programme : nice

Les systèmes UNIX étant multi-utilisateurs, il existe des outils permettant à chacun de faire exécuter ses programmes aveec divers niveaux de priorité : ainsi, un programme de calcul long sera executé avec une faible priorité, afin de ne pas pénaliser les activités 'immédiates', telles que la lecture de mail ou l'édition de fichiers, qui peuvent être rendues pénibles voire impossible si un programme tourne en permanence sur la machine. Pour cela on utilise la commande nice, qui permet de donner des priorités (entre 0 et 19 de la plus haute à la plus basse priorité).

Syntaxe : nice +*level command*

3.2.16 Renommer une commande : alias/unalias

Un alias est un nouveau nom pour une commande. On peut soit changer les attributions d'une commande existante, soit créer un nouveau nom de commande, qui exécute une tâche non prévue par les commandes UNIX standard. Par exemple, on peut modifier le fonctionnement de la commande rm, en lui faisant confirmer la suppression du fichier, par l'alias suivant

alias rm "rm -i"

Les guillemets sont nécessaire pour indiquer au système où commence et où s'arrête la nouvelle commande. Autre exemple, si vous désirez que la commande ls vous donne systématiquement tous les attributs des fichiers, vous tapez alias ls "ls -l". Des alias peuvent être définis automatiquement à chaque nouvelle connection en les inscrivant dans le fichier .bashrc, présent dans le répertoire principal de tout nouvel utilisateur d'un système Linux.

Pour connaître la liste des alias existants : alias.

Pour désactiver un alias : unalias *nomAlias*.

3.2.17 Quelques codes de contrôle

Lorsqu'un écran défile ou lorsqu'une commande semble s'enliser, on peut souhaiter pouvoir agir sur le processus en question. Pour cela, on peut utiliser des codes de contrôles suivants :

`<Control> s` : pour suspendre l'affichage d'un écran,
`<Control> q` : pour redémarrer un affichage suspendu,
`<Control> d` : pour indiquer la fin d'un fichier,
`<Control> c` : pour annuler une commande,
`<Control> z` : pour suspendre un processus (fg pour reprendre le processus).

3.3 Différents modes d'exécution

Les systèmes UNIX, développés pour et par des utilisateurs mutliples avec de gros besoins de ressources, admettent plusieurs types de mode d'éxecution des commandes, en fonction des spécificités des opérations.

3.3.1 Interactif

C'est le mode le plus courant : vous tapez simplement votre commande, et vous attendez le retour ...

3.3.2 Asynchrone

Très pratique sous UNIX, ce mode permet de lancer votre commande, et de récuperer immédiatement la main, pour exécuter d'autres tâches. Le système renvoie un message dès que l'exécution est terminée. Ce mode d'exécution s'obtient en faisnt suivre la commande du caractère '&'.

3.3.3 Différé/Batch

On peut aussi vouloir lancer un programme long à une heure tardive pour être sûr de ne gêner personne. UNIX offre cette possibilité grâce à la command at. La syntaxe est la suivante :

at *date command* où *date* est la date à laquelle on veut exécuter la tâche. Cette date peut s'exprimer sous plusieurs forme (date/heure, intervalle de temps depuis l'instant présent, etc ... voir le man!!!). La commande doit être donnée avec son chemin complet (Ex : /bin/ls, /usr/bin/X11/netscape, ... pour connaître le path d'une command, utiliser which). batch est une version obsolète de at.

3.3.4 Cyclique

Enfin, UNIX offre la possibilité de lancer une commande de façon répétée (tous les lundis à 3h00, à chaque heure de tous les 17 du mois, ...) via la commande crontab. Voir le man de cette commande pour des exemples de syntaxe.

3.4 Redirections et enchaînements

Par ailleurs, UNIX offre d'autres facilités d'utilisation des commandes, qui permettent d'étendre les capacités du système.

3.4.1 Les tubes (ou pipe)

Le *pipe* permet d'envoyer directement le résultat d'une commande comme entrée d'une seconde, sans pour cela créer de fichier intermédiaire de stockage. Le caractère utilisé est | (Alt Gr 6).

L'exemple d'utilisation le plus courant consiste à enchaîner la commande less à une commande dont le résultat s'affichera sur plusieurs écrans. Exemple : ls -l | more.

Voici un exemple plus complexe. Imaginons qu'on veuille obtenir la liste des répertoires contenus dans le répertoire courant, et seulement les répertoires. Il n'y a pas d'options de la commande ls qui permette de faire cela. Par contre, si on regarde le résultat de la commande ls -l, on constate que le premier caractère prend la valeur d pour les répertoires et eux seulement. La commande qui permet de rechercher une chaine de caractères dans un fichier est grep (voir Sect. 3.2.14). Le manuel associé à cette commande nous apprend que pour rechercher une chaîne *en début de ligne*, il faut la faire précéder du caractère ^. La commande ls -l | grep ^d permet de n'afficher que les répertoires. Et si cet affichage dépasse le contenu d'un écran, on peut lancer la commande ls -l | grep ^d | less.

3.4.2 Redirections

Les redirections permettent de stocker le résultat d'une commande dans un fichier, ou d'utiliser un fichier comme entrée à une commande. Les systèmes UNIX ont une sortie standard, utilisée par défaut, qui est l'écran. Pour une commande, il y a deux types de sortie : la première correspond aux messages 'normaux' issus de la commande, c.-à-d. son résultat (une liste de fichier, une chaîne de caractères), la seconde correspond aux messages d'erreurs générés par la commande, si un quelconque problème survient lors de l'éxécution. Le caractère réservé pour les redirections est le '>', la syntaxe est donc la suivante :

ls -l > liste.txt

Cette commande permettra de stocker le résultat de la commande dans le fichier liste.txt. Par contre, les messages d'erreur éventuels seront affichés à l'écran. Si on veut rediriger l'intégralité des messages de la commande vers le fichier, il faudra utiliser

ls -l 2> liste.txt

alors que la ligne

ls -l > liste.txt 2> err.txt

redirigera la sortie standard dans liste.txt, et les messages d'erreur dans un autre fichier, err.txt.

Pour rediriger la sortie vers la fin d'un fichier existant, il faut utiliser les caractères '»'.

Exemple :

ls /home/machin -l > liste.txt

ls /home/truc -l » liste.txt

Le fichier liste.txt contiendra le contenu des deux répertoires,machin et truc.

On peut également faire en sorte que l'entrée d'une commande soit non plus le clavier mais le contenu d'un fichier. Le caractère réservé est '<'.

Exemple : pour envoyer le contenu du fichier essai par mail à une personne dont l'adresse est *dupont@machin.fr*, on peut lancer la commande mail dupont@machin.fr < essai.

Chapitre 4

Scripts shell

4.1 Généralités

Tous les shells UNIX permettent de programmer des enchaînements complexes de commandes au moyen d'un langage de programmation *interprété* (le programme est traduit en langage machine au cours de son exécution) et *récursif* (un programme peut s'appeler lui-même). Un *script* est un fichier ASCII qui contient un tel programme, interprétable par le shell. Un script être considéré comme une nouvelle commande, qui pourra être exécutée en tapant son nom au clavier.

Un script particulier et présent sur tous les comptes utilisateurs, le fichier .bashrc., contient toutes les définitions concernant l'environnement de l'utilisateur et peut-être personnalisé selon les goûts de chacun. Ce fichier contiendra par exemple des alias des commandes, ainsi que la définition de différentes variables d'environnement (PATH, PRINTER, ...). Ce script est exécuté à chaque ouverture d'une fenêtre de commande.

Il existe plusieurs shell (bash, csh, tcsh) et le langage de programmation des scripts diffère d'un shell à l'autre. La première ligne d'un script doit **toujours** indiquer le shell qui doit être utilisé pour l'interpréter. Par exemple, pour le bash, la première ligne d'un script doit être :

#! /bin/bash

Il est nécessaire de donner l'adresse absolue du shell, à partir de la racine.

Un script peut s'écrire à l'aide d'un éditeur de textes comme emacs ou, s'il est très court, à l'aide de la commande cat.

4.2 Exemples simples

Pour afficher un message :

```
#!/bin/bash
# afficher un message
echo "Bonjour"
```

A part la première ligne, les lignes commençant par le caractère # sont des *commentaires*, c'est-à-dire des textes qui doivent aider à comprendre ce que fait un programme. Ces lignes ne sont pas interprétées par le shell.

Pour afficher le contenu d'une variable :

```
#!/bin/bash
# afficher le contenu d'une variable
echo $PATH
```

Pour afficher un message personnalisé :

```
#!/bin/bash
# afficher un message personnalisé
echo "Bonjour, $USER"
```

Quelques compléments sur le formatage d'un message à l'aide de la commande echo : l'option -e permet d'utiliser les caractères de contrôle suivants :

```
\b : backspace
\n : newline
\r : carriage return
\t : tabulation horizontale
```

Exemple : `echo -e "aze\bnjour\rbo\n"`

Pour renommer une commande :

```
#!/bin/bash
# OuSuisJe
pwd
```

On peut forcer l'utilisation d'une commande à l'aide des "back quotes" (touche altgr 7).

```
#!/bin/bash
# OuSuisJe
echo "Je suis là : `pwd`"
```

4.3 Exemples avec paramètres

Il est possible de faire passer plusieurs paramètres à un script. Les paramètres sont placés à la suite du nom du script, et séparés par des espaces, comme pour une commande standard. A l'intérieur du script, la valeur de ces paramètres est récupérée grâce aux variables '1' à '9'. Par exemple, la ligne

`echo $3`

renverra la valeur du troisième paramètre si il existe, rien sinon ...

Par ailleurs, `$*` contient la liste de tous les paramètres, et $# contient le nombre de paramètres (utile si ce nombre n'est pas fixe, et si on veut traiter le dernier).

Enfin `$0` contient le nom de la commande invoquée (utile si on veut que la commande s'appelle elle même, sans pour autant connaître le nom du script).

```
#!/bin/bash
# utilisation des paramètres
echo "$0 est le nom de la commande"
echo "$1 est le premier argument"
echo "$2 est le second"
echo "$# est le nombre total d`arguments"
echo "$* est la liste des arguments"
```

On peut maintenant renommer des commandes complexes :

```
#!/bin/bash
# afficher
ls $*
```

On peut créer un outil facilitant l'écriture des scripts

```
#!/bin/bash
# créer un script
echo "#!/bin/bash" > $1
chmod u+x $1
echo "Ecrivez votre script et terminez par C-d"
cat >> $1
```

Structure conditionnelle

L'instruction if permet d'effectuer des opérations si une condition est réalisée, et d'autres sinon. La structure la plus simple est la suivante :

```
if <condition>
then
        <liste d'instructions>
fi
```

On peut écrire des conditions à l'aide de la commande test qui peut être utilisée pour comparer deux entiers : n1 op n2 où op peut prendre les valeurs

```
-eq pour =
-ne pour différent
-lt pour <
-le pour < ou =
-gt pour >
-ge pour > ou =
```

Exemple d'utilisation :

```
#!/bin/bash
# estPositif
if test $1 -gt 0
then echo ‘‘l'argument est positif''
```

4.4 Utilisation de variables

Une variable est un emplacement mémoire, repéré par un nom et dans lequel, le programme peut mémoriser certaines valeurs. Un nom de variable est une suite quelconque de caractères (à l'exception de quelques symboles réservés comme *, ?, , ...).

Voici un exemple de manipulation de variables :

```
#!/bin/bash
a=3
debut=to
echo debut
echo "le $ est necessaire pour afficher le contenu d'une variable, debut vaut $debut"
nom=$debutto
echo "nom vaut $nom"
nom=${debut}to
echo "maintenant nom vaut" $nom
#Et maintenant tentons différentes opérations :
c=$a+4
d= `expr $a + 4 `
# Et l'affichage
echo 'c= ' $c
echo 'd= '$d
```

Le résultat de ce programme sera le suivant :

```
debut
le $ est necessaire pour afficher le contenu d'une variable, debut vaut to
nom vaut
maintenant nom vaut toto
c=3+4
d=7
```

Explication de la troisième ligne : le système cherche une variable appelée debutto, qui n'existe pas, d'ou l'absence de sortie. Les acollades sont nécessaires pour séparer le nom de la variable de la chaîne de charactère qui le suit.

4.4.1 La commande test

Comme son nom l'indique, cette commande permet d'effectuer différents tests sur des fichiers, des répertoires ou des expressions arithmétiques.

-r *fichier* : vrai si le fichier existe et est accessible en lecture (R)

-w *fichier* : vrai si le fichier existe et est accessible en écriture (W)

-x *fichier* : vrai si le fichier existe et est exécutable (X)

-f *fichier* : vrai si le fichier existe et est un fichier régulier

-d *fichier* : vrai si le fichier existe et est un répertoire

-s *fichier* : vrai si le fichier existe et a une taille non nulle

s1 = s2 : vrai si les deux expressions sont égales

s1 != s2 : vrai si les deux expressions sont différentes

e1 -eq e2 : vrai si les deux entiers e1 et e2 sont égaux (autres opérateurs de comparaison : -ne , -gt , -ge , -lt , -le)

4.4.2 Structures de programmation plus complexes

If

L'instruction if permet d'effectuer certaines opérations si une condition est réalisée, et d'autres sinon. La structure est la suivante :

if *cmde*
then
    liste_commandes
[else
    liste_commandes]
fi

Exemples :

le programme suivant affiche le contenu d'un fichier dont le nom est passé en argument s'il existe et rien sinon.

```
if test -f $1
then
    cat $1  fi
```

Le programme suivant se comporte comme le précédent si le fichier existe ; il affiche un message particulier si le fichier n'existe pas.

```
if test -f $1
then
    cat $1
else
    echo " le fichier n'existe pas "
fi
```

Un exemple un peu plus compliqué. Que fait ce script ? Réponse plus bas.

```
#! /bin/bash
a=1                 # a est une variable initialisée à 1
b=0                 # b est initialisée à zéro
until test $a = 10  # Jusqu'à ce que a=10, faire
do
  b=‘expr $a + $b‘  # evalue a+b et le place dans b
  a=‘expr $a + 1‘   # evalue a+1 et le place en 1
done
echo $b             # Affiche la valeur de b
fini
```

For

Cette instruction permet de répéter une instruction pour un ensemble de paramètres.

for *para [in liste]*
do
    liste_commandes
done

La variable *para* prend successivement les valeurs de la liste. Si la liste est omise, *para* prend alors les valeurs passées en paramètres du script. Exemple :

```
for i in ‘ls‘
do
    cp $i /dir/$i
    echo "$i copie "
done
```

Dans cet exemple, ne pas oublier les “backquote” ‘ qui forcent l'exécution de la commande ls, sinon le script va chercher à copier un fichier nommé 'ls'.

While/Until

Ces deux commandes permettent de faire exécuter une liste d'instruction tant qu'une (ou jusqu'à ce qu'une) certaine condition soit réalisée :

```
while condition
do
    liste_commandes
done
```

ou

```
until condition
do
    liste_commandes
done
```

La différence entre les deux concerne l'endroit où s'effectue la verification de la condition : avant l'exécution de la liste de commande dans le cas de while, après pour until. En conséquence, dans le cas de until, la liste de commande est toujours exécutée au moins une fois.

Exemple :

```
#!/bin/bash
a=$1
while test $a -ne 0
do
echo "un "
a=‘expr $a - 1‘
done
```

ou

```
#!/bin/bash
a=$1
until test $a -eq 0
do
echo "un "
a=‘expr $a - 1‘
done
```

Attention : dans le cas du until, la commande echo "un" sera exécutée au moins une fois, quelle que soit la valeur de l'argument.

4.4.3 Exercices

Dans les exercices qui suivent, nous supposons que le répertoire courant `scripts` contient les fichiers suivants :

```
[rmorin@serveurtx scripts]$ ls -al
total 32
drwxr-xr-x    2 rmorin   users        4096 nov  6 11:01 ./
drwxr-xr-x   14 rmorin   users        4096 nov  6 10:55 ../
-rwxr-xr-x    1 rmorin   users          90 nov  6 10:19 affiche*
-rwxr-xr-x    1 rmorin   users          73 nov  6 10:27 multiaffiche*
-rwxr-xr-x    1 rmorin   users         321 nov  6 10:57 mystere*
-rwxr-xr-x    1 rmorin   users          86 nov  6 10:23 recopie*
-rw-r--r--    1 rmorin   users        1239 nov  6 11:01 test
-rw-r--r--    1 rmorin   users          50 nov  6 10:20 toto.txt
```

De plus le script `affiche` contient les commandes suivantes :

```
[morin@moorea scripts]$ more affiche
#! /bin/bash
if test -f $1
  then
    cat $1
  else
    echo "Ce fichier n'existe pas"
fi
```

Exercice. Quelles vont être les réponses du shell aux commandes suivantes :

```
[rmorin@serveurtx scripts]$ affiche toto.txt
[rmorin@serveurtx scripts]$ affiche toto.ps
```

Solution.

```
[rmorin@serveurtx scripts]$ affiche toto.txt
Bonjour,
Nous sommes le 31 avril, n'est-ce pas ?

[rmorin@serveurtx scripts]$ affiche toto.ps
Ce fichier n'existe pas
```

Dans le premier cas, `affiche toto.txt` affiche le contenu du fichier toto.txt.

Le fichier `recopie` contient le script suivant :

```
[morin@moorea scripts]$ more recopie
#! /bin/bash
for i in `ls`
do
  cp $i /tmp/$i
  echo $i " a été recopié en /tmp"
done
```

Exercice. Quelles vont être les actions réalisées au lancement de la commande suivante :

```
[rmorin@serveurtx scripts]$ recopie
```

Solution.

```
[rmorin@serveurtx scripts]$ recopie
affiche  a été recopié en /tmp
multiaffiche  a été recopié en /tmp
mystere  a été recopié en /tmp
recopie  a été recopié en /tmp
test  a été recopié en /tmp
toto.txt  a été recopié en /tmp
```

Tous les fichiers du répertoire courant sont recopiés dans le répertoire `/tmp`

Le fichier `multiaffiche` contient le script suivant :

```
[morin@moorea scripts]$ more multiaffiche
#! /bin/bash
while test $# != 0
do
  affiche $1
  shift
done
echo "fini"
```

Exercice. Quelles vont être les actions réalisées au lancement de la commande suivante :

```
[rmorin@serveurtx scripts]$ multiaffiche toto.txt toto.ps mystere
```

Solution.

```
[rmorin@serveurtx scripts]$ multiaffiche toto.txt toto.ps mystere
Bonjour,
Nous sommes le 31 avril, n'est-ce pas ?

Ce fichier n'existe pas
#! /bin/bash
```

```
a=1                 # a est une variable initialisée à 1
b=0                 # b est initialisée à zéro
until test $a = 10  # Jusqu'à ce que a=10, faire
do
  b=‘expr $a + $b‘  # evalue a+b et le place dans b
  a=‘expr $a + 1‘   # evalue a+1 et le place en 1
done
echo $b             # Affiche la valeur de b
fini
```

Le script affiche d'abord le contenu de `toto.txt` Il écrit ensuite que `toto.ps` n'existe pas. Puis il affiche le contenu du fichier `mystere`.
Exercice. Quelle va être la réponse au lancement du script `mystere` ?
Solution.

```
[rmorin@serveurtx scripts]$ mystere
45
```

En effet, $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$.

Chapitre 5

Représentation des Nombres

5.1 Representation des entiers

5.1.1 Principe des représentations en base b

Base 10 L'entier écrit 3404 correspond a 3 × mille + 4 ×cent + 0× dix + 4. Plus généralement $a_n a_{n-1} \dots a_1 a_0$ $_{(10)}$ correspond à $a_n \times 10^n + \dots + a_1 \times 10^1 + a_0 10^0$. On a donc n chiffres pour un entier x de l'ordre de 10^n et un rapport logarithmique entre la représentation et l'écriture de l'entier x c.a.d. $log_{10}(x) \approx n$.

Cas général : base b On a b chiffres pour représenter $0, 1, \dots, b-1$ et $x = a_n a_{n-1} \dots a_1 a_0$ $_{(b)}$ correspond à $x = a_n \times b^n + a_{n-1} b^{n-1} + \dots + a_1 \times b^1 + a_0 \times b^0$.

Exemples :

base 10	chiffres $0, 1, \dots, 9$	
base 2	chiffres 0 et 1	8 s'ecrit 1000 7 s'ecrit 111
base 8 (octal)	chiffres $0, \dots, 7$	9 s'écrit 11 25 s'écrit 31
base 16	plus assez de chiffres ! A pour 10, B pour 11,...,F pour 15	? ? ? 167 s'écrit $A7_{(16)}$

Changer de base.

Représentation décimale vers représentation en base 2.

13	| 2		
1	6	| 2	
	0	3	| 2
		1	**1**

d'ou $13_{(2)} = 1101$

Représentation binaire vers représentation en base 10.

$1101_{(2)} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$

Principe pour passer de base 10 vers base b : faire les divisions, la suite des restes et du dernier quotient **inversée** forme l'écritude.

Principe pour passer de base b vers base 10 : faire l'évaluation à partir de la définition $a_n \dots a_1 a_0 = a_n \times b^n + \dots + a_1 \times b^1 + a_0 b^0$.

Exercice : montrer que les deux principes sont corrects.

5.1.2 Opérations arithmétiques

Addition Cours élémentaire première année : une difficulté majeure, la retenue.

$$\begin{aligned} 99 + 77 &= (9 \times 10 + 9) + (7 \times 10 + 7) \\ &= 9 \times 10 + 7 \times 10 + \mathbf{1} \times 10 + 6 \\ &= \mathbf{1} \times 100 + 7 \times 10 + 6 \end{aligned}$$

Conclusion : en base 10 il faut apprendre les tables d'addition. En base 2 c'est plus simple : voici une machine à additionner :

2 états dans la machine.

Etat 0 pas de retenue

Etat 1 une retenue (qui vaut forcement 1)

Les vecteurs sont de la forme $\begin{pmatrix} op1 \\ op2 \\ res \end{pmatrix}$ avec $op1$ est le n^{me} bit du premier opérande, $op2$ est le n^{me} bit du second opérande, res est n^{me} bit du résultat.

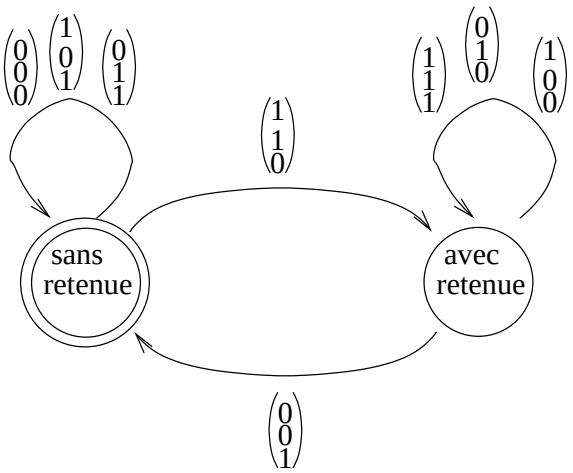


FIG. 5.1 – Un automate pour l'addition

Les nombres sont lus de la droite vers la gauche (ou bien on considère qu'ils sont écrits à l'envers), on complète l'ecriture avec des 0 pour que les deux nombres et leur somme aient même longueur (on peut rajouter autant de 0 que voulu, le résultat ne change pas), la valeur du résultat est le seul bit possible correspondant aux valeurs de $op1$ et $op2$ dans l'état ou on est.

10 s'écrit 1010, en inversant on a 0101, complété par 0 on a 010100... 7 s'écrit 111, en inversant on a 111, complété par 0 on a 111000.... La machine donne :

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

d'où en rétablissant le bon sens de lecture $1010 + 111 = 10001$

Soustraction Comment gérer $a - b$ si $b > a$? Problème de signe. Si le problème est réglé, alors soustraire c'est ajouter l'opposé. On verra comment faire plus loin.

Multiplication Opération difficile : apprendre les tables de multiplication.

Division Pour que le résultat soit entier, il faut considérer la division entière. C'est encore plus compliqué que la multiplication. On ne parlera pas ici ni de la réalisation de la division ni même de la multiplication.

5.2 Représentation des entiers machines

Les entiers vont être représenté par des emplacement mémoires formés d'octets (byte), un octet est un groupe de 8 cases élémentaires appelées bits dont la valeur est 0 ou 1 (ceci est physiquement réalisé paar une valeur de tension positive ou négative, une aimantation,...). Le mot machine est l'entité de stockage de base est est formé de $1, 2, 3, 4, 8, \ldots$ octets selon les machines, c.a.d. $8, 16, 32, 64, \ldots$ bits. La plupart des PC actuels utilisent 32 bits. Avec N bits on a 2^N suites possibles de 0 et 1.

Conséquence : les nombres représentables sont en nombre fini. On ne peut pas tout représenter ni calculer. Avec 32 bits on est limité à 2^{32} écritures possibles et donc de l'ordre de quelques milliards de nombres (et le plus grand nombre est de cet ordre de grandeur).

Question : sachant qu'on dispose de N bits pour représenter un entier,

1. comment représenter les entiers **signés** ?
2. comment calculer $+, -$ avec la représentation choisie ?

Representation numéro 1 :bit de signe et valeur absolue Le premier bit indique le signe 0 pour +, 1 pour −. le reste est la valeur absolue (en base 2). Sur 3 bits cela donne :

```
   -3 111     000 +0
-2 110             001 +1
 -1 101          010 +2
    -0  100    011 +3
```

Inconvénient : 2 représentation pour 0, additionner demande à comparer les deux nombres

$$\begin{array}{lccc} & 001 & 001 & 100 \\ \text{avant :} & 110 & 001 & 101 \\ & \overline{\quad ? \quad} & \overline{\quad 010 \quad} & \overline{\quad 101 \quad} \end{array}$$

Représentation numéro 2 : complément à 1 Représentation de $-x$: prendre le complément des bits de x ($0 \to 1$,$1 \to 0$). Ex $-3_{(2)} = 011$ donne 100. D'ou sur 3 bits on a :

```
   -0 111     000 +0
-1 110            001 +1
 -2 101         010 +2
    -3  100    011 +3
```

Si $x = \Sigma_{i=0}^{i=N-1} b_i \times 2^i$ alors $\bar{x}$ correspondant au complément à 1 vaut : $\bar{x} = \Sigma_{i=0}^{i=N-1}(1-b_i)\times 2^i = -\Sigma_{i=0}^{i=N-1} b_i \times 2^i + \Sigma_{i=0}^{i=N-1} 2^i = -x + (2^N - 1)$.

Problème : deux représentations pour 0 : on perd une représentation et l'addition faite par ajout bit à bit ne marche pas : sur 3 bits $111 + 001 = 1000$ alors que le résultat est $0 + 1 = 1$ et on doit réétablir 1000 en 001.

Representation numéro 3 : complément à 2 On prend le complément à 1 de x et on ajoute 1. On a tous les nombres de $[-2^{N-1}, \ldots, 2^{N-1} - 1]$, l'addition est l'ajout bit à bit. Alors $\bar{x} = -x + 2^N$

```
   -1 111     000 +0
-2 110            001 +1
 -3 101         010 +2
    -4  100    011 +3
```

Le calcul de $\bar{x}$ est plus compliqué mais on gagne en représentation et simplicité de l'addition.

$$\begin{array}{lccc} & 001 & 001 & 111 \\ & 110 & 001 & 110 \\ & \overline{111} & \overline{010} & \overline{101} \end{array}$$

Exercice : pourquoi le résultat est-il correct ?

Débordement Avec une reprśentation sur 3 bits on ajoute deux entiers. Que se passe-t-il si on a un résultat plus grand que 3 ou plus petit que -4 ? Par exemple $3 + 2 = 5$ $\begin{array}{r} 011 \\ 010 \\ \hline 101 \end{array}$ le résultat est aberrant (il est négatif), il y a eu débordement. Il faut donc s'assurer que les opérations arithmétiques qu'on effectue restent dans les limites des représentations (y penser quand on écrit ses programmes).

Terminologie : bit de poids fort : celui de la plus grande puissance, bit de poids faible : celui des unités.

5.3 Représentation des réels

Même problème en plus compliqué : place finie pour une infinité de réels, mais en plus on ne sait pas représenter complètement un réel.

- 165686979878979678568008998 grand mais complètement déterminé.
- $1/3 = 0.3333333...$ pas de représentation décimale finie,
- $\sqrt{2} = 1.414...$ pas de représentation rationnelle,
- $\pi = 3.14159....$ pas de représentation algébrique

Seuls les nombres décimaux pas trop grands peuvent se représenter en machine. Par conséquent toute représentation de nombre réel sera imparfaite. De plus comme pour les entiers, les nombres sont représentés par des suites de bits donc en base 2. Cela a des conséquences inattendues : le nombre 0.1 est un décimal en base 10 mais pas en base 2 !

Représentation 1 : virgule fixe (la virgule a une place fixe)

On code le nombre x par l'entier $x\ b^p$ où p est fixé correspond au nombre voulu de chiffres de la partie fractionnaire.

Exemple : $p = 4$ alors la suite 1100 représente $x = 0.1100$ (en base 2) c'est à dire $x = 0.5 + 0.25 + 0.0625 = 0.8125$ en base 10.

Problème : l'échelle est fixe et on ne peut pas la modifier. Impossible de combiner des nombres très différents. D'où l'abandon au profit de la représentation suivante.

Représentation 2 : virgule flottante (représentation utilisée par les machines)

$$x = \underbrace{m}_{mantisse} * \underbrace{b}_{base}{}^{\overbrace{e}^{exposant}}$$

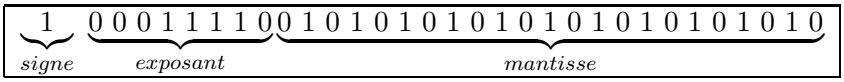
et la représentation de x est (m, e) (b étant fixée une fois pour toute). Comme pour les entiers, un réel peut être trop grand ou trop petit, ce qui causera un débordement (overflow si trop grand ou underflow si trop petit).

Des représentations approchées de π sont : $(0.031, 2)$, $(3.142, 0)$, $(0.003, 3)$ et on voit qu'elles ne donnent pas la même précision. Pour éviter ce problème et garder la meilleure précision, on utilisera une mantisse *normalisée* c'est à dire qu'elle ne contiendra pas de 0 en tête (donc le premier bit de la mantisse sera 1). Par contre 0 devra être représenté de manière spéciale. De plus on choisira le facteur d'echelle de mani'ere à ce que $1 \leq |m| < b$ (donc mantisse comprise en valeur absolue entre 1 et 2 en base 2). Comme cette représentation commence toujours par 1, on n'écrit pas ce 1 ce qui permet d'économiser un bit.

La comparaison de deux nombres se fera par comparaison de leurs exposants d'abord ce qui est plus compliqué s'ils sont signés. En base 2 on utilise des exposants biaisés : si on a N bits pour représenter l'exposant, on ajoute $2^{N-1} - 1$ à l'exposant. Tout exposant entre $-2^{N-1} + 1$ et 2^{N-1} est représentable ainsi. Pour assurer la compatibilité entre les machines, un standard a été edicté par l'IEEE (Institute of Electrical and Electronics Engineers), c'est la norme 754.

1. pour une représentation 32 bits : 1 bit de signe, exposant sur 8 bits biaisé à $127 = 2^{8-1}-1$, mantisse sur 23 bits
2. pour une représentation 64 bits : 1 bit de signe, exposant sur 11 bits biaisé à $1023 = 2^{11-1} - 1$, mantisse sur 52 bits

Exemple :



- signe : bit à 1 donc le nombre est négatif.
- exposant biaisé vaut 00011110 = 30 donc l'exposant est $= 30 - 127 = -97$
- mantisse : (ne pas oublier le premier 1) vaut :
  $1+2^{-2}+2^{-4}+2^{-6}+2^{-8}+2^{-10}+2^{-12}+2^{-14}+2^{-16}+2^{-18}+2^{-20}+2^{-22}+2^{-24}+2^{-26} \simeq 4/3$

Le nombre vaut donc $-\ 4/3\ 2^{-97} \simeq -\ 4/3\ (2^{10})^{-10} \cdot 2^3 \simeq -32/3 \cdot .10^{-30}$.

Opération inverse : trouver la représentation sur 32 bits du réel 278.

Le nombre est positif d'où 0 comme bit de signe. On doit trouver l'exposant e tel que $x = m * 2^e$ avec $1 \leq m < 2$. D'où $e = 8$ (car $2^8 < x < 2^9$) et donc $m = 278/256$. L'exposant est biaisé à 127 d'ou $e = 127 + 8 = 135$ représenté comme 10000111

Un manière simple de trouver la représentation de m est de voir que $x = 2^8 + 2^4 + 2^2 + 2 = 2^9(2^{-1} + 2^{-5} + 2^{-7} + 2^{-8}$ d'où $m = 1.0001011$ en base 2 ce qui donne la représentation du nombre (ne pas oublier que la partie entière de la mantisse est omise) :

Effectuer les opérations arithmétiques sur ces représentations n'est pas une tache facile et met usuellement en oeuvre des algorithmes qui sont réalisés en durs par des circuits électroniques (voir la presentation de l'ALU plus loin). Rappelons que le celèbre bug du pentium vient d'une mauvaise réalisation d'un algorithme de division sur les réels.

Exercices

Exercice 1: Dans tout l'exercice, on considère le codage des entiers sur 4 bits (4 chiffres binaires).

1. Dans un premier temps, on ne code que des entiers positifs ou nuls. Quelle est le codage binaire (sur 4 bits) de l'entier $(12)_{10}$? Quel nombre en base 10 correspond au nombre en base 2 $(1010)_2$? Quel est le plus grand nombre représentable par ce codage (donnez sa valeur en base 2 et en base 10) ?
2. Un nombre négatif n est codé par le complément à un de son opposé $-n$. Rappel : le complément à un d'un nombre binaire consiste à inverser tous les chiffres de ce nombre. Exemple : le complément à 1 de $(0100)_2$ est $(1011)_2$. Quel est le codage de l'entier $(-3)_{10}$ en utilisant le complément à un ? A quel nombre en base 10 correspond $(1100)_2$. A quel nombre correspond $(1111)_2$? Quel est l'inconvénient de la représentation des entiers négatifs par complément à 1 ?.
3. Le complément à deux d'un nombre binaire consiste à ajouter 1 à son complément à un. Le décalage à gauche d'un nombre binaire est une opération consistant à décaler tous les chiffres (bits) de ce nombre d'une position vers la gauche. exemple Le décalage à gauche de $(0010)_2$ est $(0100)_2$. Que réalise l'opération de décalage des nombres binaires sur les entiers en base 10 correspondants ? Sur une représentation par complément à 2, quel est est le plus grand nombre binaire représentant un entier positif auquel on peut appliquer cette opération sans risque de débordement ? Que vaut cet entier en base 10.

Exercice 2: On suppose qu'on dispose de 3 bits pour représenter les entiers signés en base 2 en utilisant le complément à deux.

1. Donner les représentations de 2, -1 et -2. Calculer la somme des trois représentations (expliquer en détail vos calculs). Quel est le résultat ? De quel nombre est-ce la représentation ?
2. Pourrait-on faire de même avec 2,1, 3 sans problème ?

Exercice 3: La representation d'un nombre flottant sur 32 bits est telle que :
- le bit de signe est 0,
- l'exposant est 10001001,

– la mantisse est 00010110000000000000000

Expliquer ce que cela signifie. Donner la valeur du nombre flottant en base 10 (ne pas oublier que l'exposant est biaisé à 128). Indication : le résultat est un entier dont la valeur absolue ne dépasse pas 300.

Exercice 4: Les flottants sont représentés de manière normalisée sur 32 bits.

– Calculer la représentation sur 32 bits du nombre réel $0,1$. Même question pour $0,2$ puis $3,125$.
– Quel est le plus petit réel représentable ? le plus grand ?

Chapitre 6

Calcul Booléen et Circuits Logiques

6.1 Traitement Logique et Machine

6.1.1 Exemple

Nos raisonnement sont usuellement simples :
si ma voiture ne marche pas **et** il pleut **alors** je prends le metro
formé de

1. propositions
   (a) ma voiture marche **v**
   (b) il pleut **p**
   (c) je prends le metro **m**

   à valeur Vrai ou Faux (1 ou 0)
2. connecteurs logiques : $\wedge, \vee, \neg, \Rightarrow, \ldots$ permettant de combiner les propositions

$$p \wedge \neg v \Rightarrow m$$

3. Règles de calcul pour donner une sémantique : Vrai et Vrai donne Vrai (ou $1 \wedge 1 = 1$)

Les raisonnements simples comme ci dessus peuvent s'exprimer dans un cadre formel qui est un calcul particulier appelé calcul booléen.

6.1.2 Calculs logique et machines

Calcul booléen inventé par le logicien George Boole basé sur 2 valeurs 0 et 1.

Représentable physiquement par des mécanismes électroniques (transistors) : une tension >0 représente 1, une tension nulle 0. Ouvrir ou fermer le circuit revient à passer de 1 à 0 et réciproquement.

Représenté par :
(l'interrupteur est un modèle idéalisé du transistor)

En réalité on a plutot :

Que peut-on réaliser :
- 18eme siècle : Leibniz pense que toutes les mathématiques pourront se ramener à un calcul.
- 19eme siècle : Boole invente le calcul Booléen (aussi appelé calcul propositionnel) que nous allons voir.
- fin 19eme 20eme siècle :
  - Frege invente le formalisme de la logique du premier ordre (quantification $\forall, \exists$, variables,...)
  - Turing invente un modèle de machine et la thèse de Church-Turing postule que toutes les fonctions calculables par une machine le sont par une machine de Turing.
  - Godel montre que les mathématiques contiennent des fonctions non-calculable (théorèmes d'incomplétude).
  - Von Neuman décrit les principes de fonctionnement des ordinateurs (séquentiels).

Le calcul booléen est réalisable par des composants matériels : les circuits logiques (qu'on fabrique avec des transistors) qui sont les composants de base du microprocesseur. Nous verrons comment ils sont combinés dans l'Unité Arithmétique et Logique qui est la partie du microprocesseur qui effectue les calculs arithmétiques et logiques en lequel la majorité des opérations complexes de l'utilisateur se réduisent.

6.2 Calcul Booléen et circuits logiques

6.2.1 Calcul Booléen

Objets :
- valeurs de vérité : $B = \{0, 1\}$
- variables propositionnelles : $x_1, x_2, \ldots$ à valeurs dans B
- fonctions booléennes à n variables $n = 0, 1, \ldots : f : B_n = B \times \ldots B \to B$
- connecteurs logiques : ce sont des fonctions particulières de $B \times B \to B$ ou $B \to B$

Sémantique : $\varphi : B_n \to B$ déterminée par sa table de vérité (le graphe de la fonction qui est fini car le domaine de départ est fini).

Exemple : Table du ET logique (noté ici comme une multiplication .).

x	y	$z = x.y$
0	0	0
0	1	0
1	0	0
1	1	1

Règles de calcul : Notation arithmétique des connecteurs logiques.
. le ET
+ le OU
$^-$ le non
Alors $(B, +, ., ^-)$ est une algèbre booléenne :
- $+, .$ sont des lois internes associatives et commutatives, 0 neutre pour $+$, 1 neutre pour $.$,
- $x + \bar{x} = 1$ et $x.\bar{x} = 0$,

– Distributivité : $(x + y).z = x.z + y.z$ et $(x.y) + z = (x + z).(y + z)$

Noter la similitude entre les opérations logiques et les opérations ensemblistes union, intersection, complément.

Remarque : les éléments n'ont pas d'inverse pour + (ni pour .). Pour avoir un inverse il faut utiliser une autre addition le **ou exclusif** $\oplus$ que nous verrons plus loin.

6.2.2 Fonction booléennes de bases et circuits

Les fonctions booléennes de base peuvent se réaliser par des circuits logiques combinatoires. Ces circuits se représentent de manière conventionnelle avec des fils d'entrée (un par variable de la fonction) transmettent la valeur de la variable correspondante et un fil de sortie qui transmet la valeur calculée par le circuit. Pour simplifier, on suppose que le calcul de la fonction par le circuit est instantanné (En réalité, il y a un délai -quelques nanosecondes- entre le moment où les entrées arrivent et celui où la sortie est disponible, délai que les fabriquant de circuits intégrés doivent prendre en compte). On parlera de portes logiques pour ces circuits de base (porte ET, porte OU...). D'un point de vue physique, les signaux d'entrée sont transformés en le signal de sortie.

Not Cette fonction prend la négation de son entrée, le circuit correspondant est un inverseur.

x	$z = not\ x$
0	1
1	0

représenté par

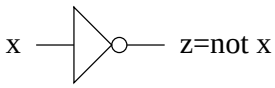


FIG. 6.1 – l'inverseur

AND Cette fonction prend le ET logique de ses entrées, le circuit correspondant est une porte ET.

x	y	$z = x.y$
0	0	0
0	1	0
1	0	0
1	1	1

représenté par

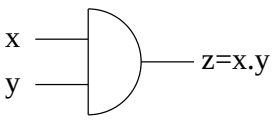


FIG. 6.2 – porte AND

OR Cette fonction prend le OU logique de ses entrées, le circuit correspondant est une porte OU.

x	y	$z = x + y$
0	0	0
0	1	1
1	0	1
1	1	1

représenté par

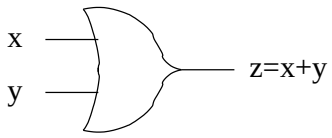


FIG. 6.3 – porte OR

Not AND Cette fonction prend la négation du ET logique de ses entrées, le circuit correspondant est une porte AND suivie d'un inverseur.

x	y	$z = x.y$
0	0	0
0	1	0
1	0	0
1	1	1

représenté par

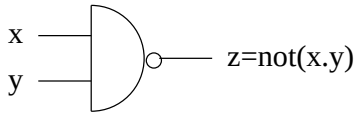


FIG. 6.4 – porte NAND

Not OR Cette fonction prend la négation du OU logique de ses entrées, le circuit correspondant est une porte OR suivie d'un inverseur.

x	y	$z = x.y$
0	0	0
0	1	0
1	0	0
1	1	1

représenté par

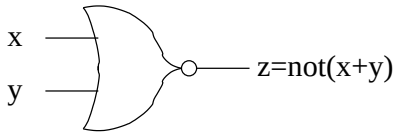


FIG. 6.5 – porte NOR

XOR Cette fonction prend le ou exclusif de ses entrées : elle n'est vrai que si une seule de ses entrées est vraie (on peut voir XOR comme $\neq$). Le circuit correspondant est une porte XOR.

x	y	$z = x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

représenté par

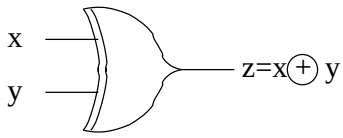


FIG. 6.6 – porte XOR

6.3 Manipulation des fonctions booléennes

6.3.1 Construction

Une fonction booléénne définie par sa table de verité est aussi représentable par une expression algébrique formée par composition d'autres fonctions booléennes. Couramment on utilisera les fonctions x_i, et ., ou +, non $\bar{}$.

Une technique simple (mais pas efficace) est la suivante :

1. pour chaque ligne de la table de vérité telle que la valeur de φ est 1 écrire $x_1^{\epsilon_1} \dots x_n^{\epsilon_1}$ avec $\epsilon_i = 1$ si x_i vaut 1, sinon $\epsilon_i = -1$ et la convention $x_i^1 = x_i$ et $x_i^{-1} = \bar{x_i}$. Le produit obtenu m_i est appelé monôme.
2. Faire $\varphi(x_1, \dots, x_n) = \Sigma_{i \mid \varphi=1} m_i$ la somme des monômes obtenus

On peut simplifier le résultat par exemple avec $x + \bar{x} = 1$.

Exemple pour le OU on obtient $\varphi(x_1, x_2) = \bar{x_1}.x_2 + x_1.\bar{x_2} + x_1.x_2$ qu'on simplifie en $x_1 + x_2$ par factorisation et utilisation de la règle précédente et de $x + y = x + \bar{x}.y$. Des méthodes plus systématiques (tableaux de Karnaugh, algorithme de Quine,...) on été developpées pour obtenir les formes algébriques les plus simples possibles. Trouver la meilleure forme est un problème crucial car les fonction booléennes sont codées par des circuits à partir de leur forme algébriques. Il suffit de combiner les circuits vus précédemment pour obtenir toutes les sommes, produits et négation.

Exercice 5: Pour n variables, montrer qu'il y a 3^n monômes possibles en considérant comme monôme un produit de x_i ou $\bar{x_i}$ (certaines variables peuvent ne pas être présentes). Donner le nombres de fonctions booléennes à n variables, et le nombre de somme de monômes. Conclusion.

6.3.2 Un peu de logique

Bien comprendre les règles de calcul sur les connecteurs usuels $ET, OU, NON, IMPLIQUE$ est essentiel aussi pour programmer et utiliser les machines correctement. En effet on est amené souvent a écrire des combinaisons complexes de variables propositionnelles qui codent des conditions ou des raisonnements simples. Il faut donc être capable de traduire les raisonnement et de manipuler correctement par le calcul les formules obtenues.

Une difficulté classique est de comprendre l'implication logique. Par définition $p \Rightarrow q$ est équivalent (c'est 'a dire a la même table de vérité) à $\neg p \vee q$. Quelles sont les implications suivantes qui sont vraies (en considérant les propositions uniquement de leur point de vue de valeur de vérité) :

1. *New-York capitale de la Russie* implique *Moscou capitale des Etats-Unis*
2. *New-York capitale des Etats Unis* implique *Paris capitale de la Russie*
3. *New-York capitale de la Russie* implique *Paris capitale des Etats-Unis*
4. *New-York capitale des Etats-Unis* implique *Paris capitalede la France*

Il est instructif de comparer $p \wedge q$ et $p \Rightarrow q$ en dressant leur table de vérité et de réaliser que ce sont deux énoncés totalement différents.

Bien comprendre les combinaisons logiques est utile pour l'informaticien quand il programme (écriture de conditions pour des constructions **if ... then ... else ..** ou des itération **while ... do ...** qu'on verra au deuxième semestre) ou quand il utilise des logiciels : recherche de motifs, interrogation de bases de données (moteur de recherche sur le Web par exemple).

Par exemple vous dites à votre binome de lancer Google (moteur de recherche sur internet) et de *chercher un sujet d'examen de math ou d'informatique et un corrigé de TP de physique*. Votre énoncé est ambigu et plusieurs interprétations sont possibles. Lesquelles sont valables dans la liste suivante et indiquer à quelle formule booléenne elles correspondent :

1. un sujet de math,
2. un sujet d'informatique,
3. un corrigé de TP de physique,
4. un sujet de math et un corrigé de TP de physique,
5. un sujet d'informatique et un corrigé de TP de physique,
6. un sujet de math et un sujet d'informatique,
7. un sujet de math et un sujet d'informatique et un corrigé de TP de physique

Le calcul booléen permet de résoudre des puzzles logiques (en fait il s'agit d'évaluer des fonctions booléennes. En voici deux exemples :

- le règlement du club *le logicien distrait* est donné par :
  - Les membres de la direction financière sont choisis parmi ceux de la direction générale,

- personne ne peut être membre de la direction g{enérale et de la direction de la bibliothèque s'il n'est membre de la direction financière,
- Aucun membre de la direction de la bibliothèque ne peut être membre de la direction financière.

Montrer qu'on peut simplifier les deux derniers points du règlement en *aucun membre de la direction générale ne peut être membre de la direction de la bibliothèque*

- A la recherche d'un trésor, vous arrivez à un embranchement et deux chemins sont possibles. L'un mène à la cachette, l'autre dans l'antre du dragon gardien du trésor. Deux lutins sont au croisement pour renseigner le voyageur. Mais l'un dit toujours la vérité et l'autre ment toujours. Le bruit de la conversation reveillant le dragon, vous ne pouvez poser qu'une question. Laquelle faut-il poser pour gagner le trésor ?

6.4 Circuits plus complexes

Les circuits logiques que nous verrons sont des circuits **combinatoires** : la fonction en sortie ne dépend des valeurs des entrées. Les circuits **séquentiels** sont plus complexes : la sortie peut dépendre aussi des valeurs des entrées à un instant précédent.

6.4.1 Additionneur

Demi-additionneur : deux entrées et en sortie on a leur somme et une retenue (carry) réalisé avec 1 porte ET et une porte XOR.

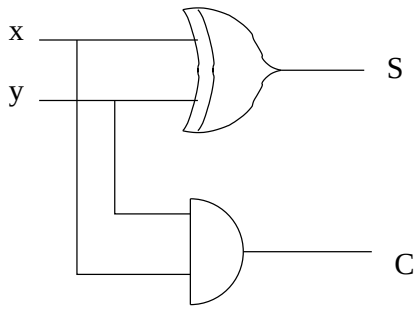


FIG. 6.7 – Demi-additionneur

$S = A \oplus B$ et $C = A.B$.

L'additionneur complet est un peu plus complexe et simule ce qu'on fait à la main (report des retenues). Pour 4 bits on a (noter la mise en cascade des additionneurs partiels) :

$$\begin{array}{ccccccccccc} CI & A_0 & B_0 & & A_1 & B_1 & & A_2 & B_2 & & A_3 & B_3 \\ & + & & & + & & & + & & + & & \\ & S_0 & C_1 & & S_1 & C_2 & & S_2 & C_3 & & S_3 & CO \end{array}$$

L'additionneur a 3 entrées A, B et la retenue CI (carry in) et deux sorties S et CO. Une implémentation est :

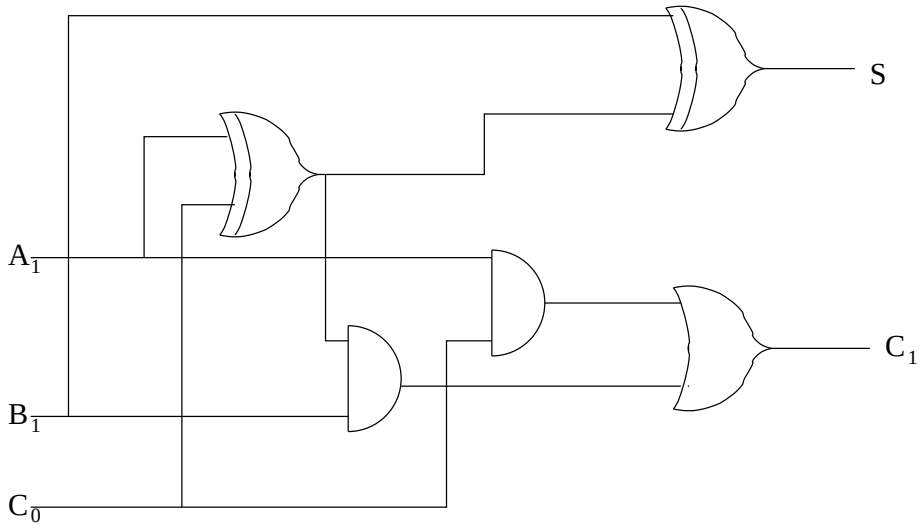


FIG. 6.8 – Additionneur

6.4.2 Multiplexeur

Le multiplexeur a 2^n entrées $A_0 \dots A_{2^n-1}$ et n entrées de sélection $S_0, \dots, S_{n-1}$ et une sortie Z qui recoit la valeur de l'entrée dont l'indice est codé par $S_0, \dots, S_{n-1}$.

Exemple pour 2 entrées de sélection et $4 = 2^2$ entrées.

S_0	S_1	Z
0	0	A_0
0	1	A_1
1	0	A_2
1	1	A_3

Alors $Z = \bar{S_0}.\bar{S_1}.A_0 + \bar{S_0}.S_1.A_1 + S_0.\bar{S_1}.A_2 + S_0.S_1.A_3$ ce qui donne :

Dans ce schéma on utilise des portes à plus de deux entrées qui s'obtiennent aisèment avec plusieurs portes élémentaires.

6.5 Une unité arithmétique et logique très simple

L'unité arithmétique et logique (UAL ou ALU en anglais) réalise les opérations logiques et arithmétiques de base. C'est le coeur de tous les ordinateurs et de la plupart des systèmes matériels digitaux. Nous allons voir un modèle d'UAL simple mais qui contient l'essentiel.

– l' UAL travaille sur n bits. Elle prend deux mots d'entrée $A = A_{n-1} \dots A_0$ et $B = B_{n-1} \dots B_0$ et a une sortie $F = F_n F_{n-1} \dots F_0$ où F_n est le bit de retenue.

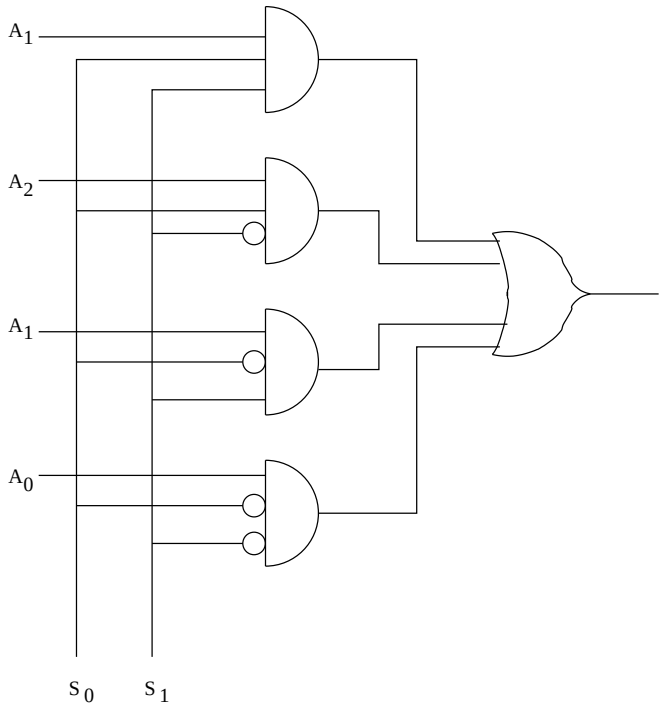


FIG. 6.9 – Multiplexeur

– Une autre entrée est un sélecteur de mode M avec $M = 1$ indique un mode arithmétique et $M = 0$ un mode logique.
– L'UAL est formée de n tranches correspondant chacune au traitement d'un bit qui sont mis en cascade pour avoir une UAL traitant n bits.

Spécification de la tranche i :

6 entrées : $A_i, B_i, C_i, M, S_0, S_1$
2 sorties : $F_i, C_i + 1$
Opérations :
$M = 0$ opérations logiques

S_0	S_1	Fonction	Commentaire
0	0	$F_i = A_i$	entrée A_i émise en sortie
0	1	$F_i = not\ A_i$	complément de A_i
1	0	$F_i = A_i \oplus B_i$	calcule le XOR
1	1	$F_i = A_i\ nxor\ B_i$	calcule le XNOR (équivalence)

$M = 1, C_i = 0$, opérations arithmétiques

S_0	S_1	Fonction	Commentaire
0	0	$F_i = A_i$	entrée A_i émise en sortie
0	1	$F_i = not\ A_i$	complément de A_i
1	0	$F_i = A_i + B_i$	Addition
1	1	$F_i = (not\ A_i) + B_i$	B_i plus complément de A_i

$M = 1, C_0 = 1$, opérations arithmétiques

S_0	S_1	Fonction	Commentaire
0	0	$F_i = A_i + 1$	incrémenter A_i
0	1	$F_i = (not\ A_i) + 1$	incrémenter le complément de A_i
1	0	$F_i = A_i + B_i + 1$	incrémenter l'addition de A_i et B_i
1	1	$F_i = (not\ A_i) + B_i + 1$	A_i moins B_i

L'UAL finale sera la mise en cascade de n exemplaires de l'UAL sur 1 bit précédente. Sa performance dépend des propagations des retenues. Trouver un circuit qui réalise cette fonction n'est pas une tache facile. On peut utiliser des outils automatiques de conception ou bien essayer à la main. La table de vérité de la fonction est la suivante (x signifie que la valeur est indiférente) :

M	S_0	S_1	C	A	B	F	C'
0	0	0	x	0	x	0	x
	0	1	x	0	x	1	x
	1	0	x	0	0	0	x
			x	0	0	0	x
			x	1	0	1	x
			x	1	1	0	x
	1	1	x	0	0	1	x
			x	0	1	0	x
			x	1	0	0	x
			x	1	1	1	x
1	0	0	0	0	x	0	x
		0	1	x	1	x	
	0	1	0	0	x	1	x
			0	1	x	0	x
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
			0	1	1	0	1
	1	1	0	0	0	1	0
			0	0	1	0	1
			0	1	0	0	0
			0	1	1	1	0
1	0	0	1	0	x	1	0
			1	1	x	0	1
	0	1	1	0	x	0	1
			1	1	x	1	0
	1	0	1	0	0	1	0
			1	0	1	0	1
			1	1	0	0	1
			1	1	1	1	1
	1	1	1	0	0	0	1
			1	0	1	1	1
			1	1	0	1	0
			1	1	1	0	1

et un circuit qui convient est ainsi :

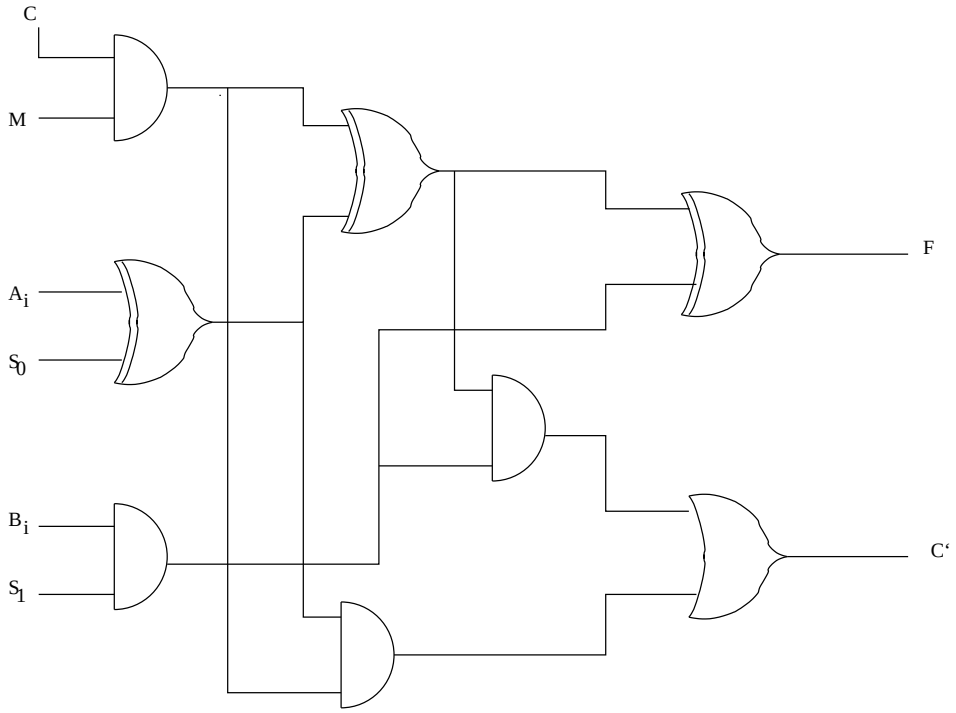


FIG. 6.10 – Tranche i de l'UAL