Architecture des ordinateurs

GTR 1998-99

Emmanuel Viennet
IUT de Villetaneuse
Département GTR
viennet@lipn.univ-paris13.fr

Table des matières

Partie 1

Introduction à l'architecture

1.1 Présentation du sujet

Le cours d'Architecture des Ordinateurs expose les principes de fonctionnement des ordinateurs. Il ne s'agit pas ici d'apprendre à programmer, mais de comprendre, à bas niveau, l'organisation de ces machines.

Nous nous appuierons sur l'étude détaillée de l'architecture du PC, dont nous étudierons le processeur et son langage machine, les fonctions de base de son système d'exploitation (BIOS), et ses mécanismes de communication avec l'extérieur (entrées/sorties).

Nous aborderons aussi le fonctionnement de différents périphériques de l'ordinateur (écran, clavier, disques durs, CD-ROMs...), afin d'apprendre à les mettre en œuvre à bon escient, puis nous conclurons ce cours par un panorama des différentes architectures actuelles (processeurs CISC et RISC, stations de travail etc.).

1.2 Représentation des données

1.2.1 Introduction

Les informations traitées par un ordinateur peuvent être de différents types (texte, nombres, etc.) mais elles sont toujours représentées et manipulées par l'ordinateur sous forme binaire. Toute information sera traitée comme une suite de 0 et de 1. L'unité d'information est le chiffre binaire (0 ou 1), que l'on appelle *bit* (pour *binary digit*, chiffre binaire).

Le codage d'une information consiste à établir une correspondance entre la représentation externe (habituelle) de l'information (le caractère `A` ou le nombre 36 par exemple), et sa représentation interne dans la machine, qui est une suite de bits.

On utilise la représentation binaire car elle est simple, facile à réaliser techniquement à l'aide de bistables (système à deux états réalisés à l'aide de transistors, voir le cours d'électronique). Enfin, les opérations arithmétiques de base (addition, multiplication etc.) sont faciles à exprimer en base 2 (noter que la table de multiplication se résume à 0x0 = 0, 1x0 = 0 et 1x1 = 1).

1.2.2 Changements de bases

Avant d'aborder la représentation des différents types de données (caractères, nombres naturels, nombres réels), il convient de se familiariser avec la représentation d'un nombre dans une base quelconque (par la suite, nous utiliserons souvent les bases 2, 8, 10 et 16).

Habituellement, on utilise la base 10 pour représenter les nombres, c'est à dire que l'on écrit à l'aide de 10 symboles distincts, les chiffres.

En base b, on utilise b chiffres. Notons a_i la suite des chiffres utilisés pour écrire un nombre $x = a_n\, a_{n-1} ... a_1 a_0$. a_0 est le chiffre des unités.

- En décimal, $b = 10$, $a_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$;
- En binaire, $b = 2$, $a_i \in \{0, 1\}$: 2 chiffres binaires, ou bits;
- En hexadécimal, $b = 16$, $a_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ (on utilise les 6 premières lettres comme des chiffres).

Représentation des nombres entiers

En base 10, on écrit par exemple 1996 pour représenter le nombre

$$1996 = 1\ 10^3 + 9\ 10^2 + 9\ 10^1 + 6\ 10^0$$

Dans le cas général, en base b, le nombre représenté par une suite de chiffres $a_n a_{n-1} ... a_1 a_0$ est donné par :

$$a_n a_{n-1} ... a_1 a_0 = \sum_{i=0}^{n} a_i b^i$$

a_0 est le chiffre de poids faible, et a_n le chiffre de poids fort.

Exemple en base 2 :

$$(101)_2 = \quad 1\ 2^2 + 0\ 2^1 + 1\ 2^0 \quad = 4 + 0 + 1 = 5$$

La notation $(\ )_b$ indique que le nombre est écrit en base b.

Représentation des nombres fractionnaires

Les nombres fractionnaires sont ceux qui comportent des chiffres après la virgule.

Dans le système décimal, on écrit par exemple :

$$12,346 \quad = \quad 1\ 10^1 + 2\ 10^0 + 3\ 10^{-1} + 4\ 10^{-2} + 6\ 10^{-3}$$

En général, en base b, on écrit :

$$a_n a_{n-1} ... a_1 a_0, a_{-1} a_{-2} ... a-p = a_n b^n + a_{n-1} b^{n-1} + ... + a_0 b^0 + a_{-1} b^{-1} + ... + a_{-p} b^{-p}$$

Passage d'une base quelconque à la base 10

Il suffit d'écrire le nombre comme ci-dessus et d'effectuer les opérations en décimal.

Exemple en hexadécimal :

$$(AB)_{16} = 10\ 16^1 + 11\ 16^0 \quad = 160 + 11 \quad = (171)_{10}$$

(en base 16, A représente 10, B 11, et F 15).

Passage de la base 10 vers une base quelconque

Nombres entiers On procède par divisions successives. On divise le nombre par la base, puis le quotient obtenu par la base, et ainsi de suite jusqu'a obtention d'un quotient nul.

La suite des restes obtenus correspond aux chiffres dans la base visée, $a_0 a_1 ... a_n$.

Exemple : soit à convertir $(44)_{10}$ vers la base 2.

$$\begin{array}{lcl}
44 = 22 \text{ x } 2 + 0 & \Longrightarrow & a_0 = 0 \\
22 = 11 \text{ x } 2 + 0 & \Longrightarrow & a_1 = 0 \\
11 = 2 \text{ x } 5 + 1 & \Longrightarrow & a_2 = 1 \\
5 = 2 \text{ x } 2 + 1 & \Longrightarrow & a_3 = 1 \\
2 = 1 \text{ x } 2 + 0 & \Longrightarrow & a_4 = 0 \\
1 = 0 \text{ x } 2 + 1 & \Longrightarrow & a_5 = 1
\end{array}$$

Donc $(44)_{10} = (101100)_2$.

Nombres fractionnaires On multiplie la partie fractionnaire par la base en répétant l'opération sur la partie fractionnaire du produit jusqu'a ce qu'elle soit nulle (ou que la précision voulue soit atteinte).

Pour la partie entière, on procède par divisions comme pour un entier.

Exemple : conversion de $(54, 25)_{10}$ en base 2

Partie entière : $(54)_{10} = (110110)_2$ par divisions.

Partie fractionnaire :

$$\begin{array}{lcl}
0{,}25 \text{ x } 2 = 0{,}50 & \Longrightarrow & a_{-1} = 0 \\
0{,}50 \text{ x } 2 = 1{,}00 & \Longrightarrow & a_{-2} = 1 \\
0{,}00 \text{ x } 2 = 0{,}00 & \Longrightarrow & a_{-3} = 0
\end{array}$$

Cas des bases 2, 8 et 16

Ces bases correspondent à des puissances de 2 ($2^1, 2^3$ et 2^4), d'où des passages de l'une à l'autre très simples. Les bases 8 et 16 sont pour cela très utilisées en informatique, elles permettent de représenter rapidement et de manière compacte des configurations binaires.

La base 8 est appelée notation *octale*, et la base 16 notation *hexadécimale*.

Chaque chiffre en base 16 (2^4) représente un paquet de 4 bits consécutifs. Par exemple :

$$(10011011)_2 = (1001\ \ 1011)_2 = (9B)_{16}$$

De même, chaque chiffre octal représente 3 bits.

On manipule souvent des nombres formés de 8 bits, nommés *octets*, qui sont donc notés sur 2 chiffres hexadécimaux.

Opérations arithmétiques

Les opérations arithmétiques s'effectuent en base quelconque b avec les mêmes méthodes qu'en base 10. Une retenue ou un report apparait lorsque l'on atteint ou dépasse la valeur b de la base.

1.2.3 Codification des nombres entiers

La représentation (ou codification) des nombres est nécessaire afin de les stocker et manipuler par un ordinateur. Le principal problème est la limitation de la taille du codage : un nombre mathématique peut prendre des valeurs arbitrairement grandes, tandis que le codage dans l'ordinateur doit s'effectuer sur un nombre de bits fixé.

Entiers naturels

Les entiers naturels (positifs ou nuls) sont codés sur un nombre d'octets fixé (un octet est un groupe de 8 bits). On rencontre habituellement des codages sur 1, 2 ou 4 octets, plus rarement sur 64 bits (8 octets, par exemple sur les processeurs DEC Alpha).

Un codage sur n bits permet de représenter tous les nombres naturels compris entre 0 et $2^n - 1$. Par exemple sur 1 octet, on pourra coder les nombres de 0 à $255 = 2^8 - 1$.

On représente le nombre en base 2 et on range les bits dans les cellules binaires correspondant à leur poids binaire, de la droite vers la gauche. Si nécessaire, on complète à gauche par des zéros (bits de poids fort).

Entiers relatifs

Il faut ici coder le signe du nombre. On utilise le codage en *complément à deux*, qui permet d'effectuer ensuite les opérations arithmétiques entre nombres relatifs de la même façon qu'entre nombres naturels.

1. **Entiers positifs ou nuls :** On représente le nombre en base 2 et on range les bits comme pour les entiers naturels. Cependant, la cellule de poids fort est toujours à 0 : on utilise donc $n-1$ bits.

   Le plus grand entier positif représentable sur n bits en relatif est donc $2^{n-1}-1$.

2. **Entiers négatifs :** Soit x un entier positif ou nul représenté en base 2 sur $n-1$ bits
   $$x = \sum_{i=0}^{n-2} \alpha_i \, 2^i, \text{ avec } \alpha_i \in \{0,1\}$$
   et soit
   $$y = \sum_{i=0}^{n-2} (1-\alpha_i) \, 2^i + 1$$
   On constate facilement que $x + y = 2^{n-1}, \; \forall \alpha_i$

   Or sur n bits, 2^{n-1} est représenté par $n-1$ zéros, donc on à $x+y=0$ modulo 2^{n-1}, ou encore $y=-x$. y peut être considéré comme l'opposé de x.

   La représentation de $-x$ est obtenue par complémentation à 2^{n-1} de x. On dit *complément à deux*.

   Pour obtenir le codage d'un nombre x négatif, on code en binaire sa valeur absolue sur $n-1$ bits, puis on complémente (ou inverse) tous les bits et on ajoute 1.

   Exemple : soit à coder la valeur -2 sur 8 bits. On exprime 2 en binaire, soit 00000010. Le complément à 1 est 11111101. On ajoute 1 et on obtient le résultat : 1111 1110.

   Remarques :

   (a) le bit de poids fort d'un nombre négatif est toujours 1;

   (b) sur n bits, le plus grand entier positif est $2^{n-1}-1 = 011\ldots1$;

   (c) sur n bits, le plus petit entier négatif est -2^{n-1}.

1.2.4 Représentation des caractères

Les caractères sont des données non numériques : il n'y a pas de sens à additionner ou multiplier deux caractères. Par contre, il est souvent utile de comparer deux caractères, par exemple pour les trier dans l'ordre alphabétique.

Les caractères, appelés *symboles alphanumériques*, incluent les lettres majuscules et minuscules, les symboles de ponctuation (& ~ , . ; # " - etc...), et les chiffres.

Un texte, ou *chaîne de caractères*, sera représenté comme une suite de caractères.

Le codage des caractères est fait par une table de correspondance indicant la configuration binaire représentant chaque caractère. Les deux codes les plus connus sont l'EBCDIC (en voie de disparition) et le code ASCII (American Standard Code for Information Interchange).

Le code ASCII représente chaque caractère sur 7 bits (on parle parfois de code ASCII étendu, utilisant 8 bits pour coder des caractères supplémentaires).

Notons que le code ASCII original, défini pour les besoins de l'informatique en langue anglaise) ne permet la représentation des caractère accentués (é, è, à, ù, ...), et encore moins des caractères chinois ou arabes. Pour ces langues, d'autres codages existent, utilisant 16 bits par caractères.

La table page 14 donne le code ASCII. A chaque caractère est associé une configuration de 8 chiffres binaires (1 *octet*), le chiffre de poids fort (le plus à gauche) étant toujours égal à zero. La table indique aussi les valeurs en base 10 (décimal) et 16 (hexadécimal) du nombre correspondant.

Plusieurs points importants à propos du code ASCII :

- Les codes compris entre 0 et 31 ne représentent pas des caractères, ils ne sont pas affichables. Ces codes, souvent nommés *caractères de contrôles* sont utilisés pour indiquer des actions comme passer à la ligne (CR, LF), émettre un bip sonore (BEL), etc.
- Les lettres se suivent dans l'ordre alphabétique (codes 65 à 90 pour les majuscules, 97 à 122 pour les minuscules), ce qui simplifie les comparaisons.
- On passe des majuscules au minuscules en modifiant le 5ième bit, ce qui revient à ajouter 32 au code ASCII décimal.
- Les chiffres sont rangés dans l'ordre croissant (codes 48 à 57), et les 4 bits de poids faibles définissent la valeur en binaire du chiffre.

1.2.5 Représentation des nombres réels (norme IEEE)

Soit à codifier le nombre 3,25, qui s'écrit en base 2 $(11,\ 01)_2$.

On va *normaliser* la représentation en base 2 de telle sorte qu'elle s'écrive sous la forme

$$1,\ \ldots\ \text{x}\ 2^n$$

Dans notre exemple $11,\ 01 = 1,\ 101\ \text{x}\ 2^1$

La représentation IEEE code séparément le *signe* du nombre (ici +), l'*exposant* n (ici 1), et la *mantisse* (la suite de bits après la virgule), le tout sur 32 bits.

Le codage a la forme : `seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm`.

- Le signe est représenté sur le bit de poids fort `s`, + est représenté par 0 et - par 1.
- L'exposant est codé sur les 8 bits `e`. On code en binaire la valeur $n+127$.
- La mantisse est codée sur les 23 bits de poids faibles `m`.

Remarques :

1. Les exposants 00000000 et 11111111 sont interdits :
   - l'exposant 00000000 signifie que le nombre est dénormalisé;
   - l'exposant 11111111 indique que l'on n'a pas affaire à un nombre (on note cette configuration NaN, Not a Number, et on l'utilise pour signaler des erreurs de calculs, comme par exemple une division par 0).
2. Les plus petit exposant est donc -126, et le plus grand +127.

Annexe

Ce petit programme en langage C peut être utilisé pour afficher le codage d'un nombre flottant quelconque :

```
/* Affichage hexadecimal des 4 octets d'un nombre flottant IEEE */
#include <stdio.h>
main() {
    float x;
    unsigned char *p = (unsigned char *)&x;

    printf("Entrer un nombre flottant : \n");
    scanf("%f", &x);
    printf(" %f  -> %x %x %x %x\n", x,  *p, *(p+1), *(p+2), *(p+3) );
}
```

Décimal	Hexa	Binaire	Caractère	Décimal	Hexa	Binaire	Caractère
0	0	00000000	NUL	32	20	00100000	ESPACE
1	1	00000001		33	21	00100001	!
2	2	00000010	STX	34	22	00100010	”
3	3	00000011	ETX	35	23	00100011	#
4	4	00000100	EOT	36	24	00100100	$
5	5	00000101		37	25	00100101	%
6	6	00000110	ACK	38	26	00100110	&
7	7	00000111	BEL	39	27	00100111	’
8	8	00001000		40	28	00101000	(
9	9	00001001		41	29	00101001	)
10	A	00001010	LF	42	2A	00101010	*
11	B	00001011		43	2B	00101011	+
12	C	00001100		44	2C	00101100	,
13	D	00001101	CR	45	2D	00101101	-
14	E	00001110		46	2E	00101110	.
15	F	00001111		47	2F	00101111	/
16	10	00010000		48	30	00110000	0
17	11	00010001		49	31	00110001	1
18	12	00010010		50	32	00110010	2
19	13	00010011		51	33	00110011	3
20	14	00010100	NAK	52	34	00110100	4
21	15	00010101		53	35	00110101	5
22	16	00010110		54	36	00110110	6
23	17	00010111		55	37	00110111	7
24	18	00011000		56	38	00111000	8
25	19	00011001		57	39	00111001	9
26	1A	00011010		58	3A	00111010	:
27	1B	00011011		59	3B	00111011	;
28	1C	00011100		60	3C	00111100	<
29	1D	00011101		61	3D	00111101	=
30	1E	00011110		62	3E	00111110	>
31	1F	00011111		63	3F	00111111	?

Décimal	Hexa	Binaire	Caractère	Décimal	Hexa	Binaire	Caractère
64	40	01000000	@	96	60	01100000	‘
65	41	01000001	A	97	61	01100001	a
66	42	01000010	B	98	62	01100010	b
67	43	01000011	C	99	63	01100011	c
68	44	01000100	D	100	64	01100100	d
69	45	01000101	E	101	65	01100101	e
70	46	01000110	F	102	66	01100110	f
71	47	01000111	G	103	67	01100111	g
72	48	01001000	H	104	68	01101000	h
73	49	01001001	I	105	69	01101001	i
74	4A	01001010	J	106	6A	01101010	j
75	4B	01001011	K	107	6B	01101011	k
76	4C	01001100	L	108	6C	01101100	l
77	4D	01001101	M	109	6D	01101101	m
78	4E	01001110	N	110	6E	01101110	n
79	4F	01001111	O	111	6F	01101111	o
80	50	01010000	P	112	70	01110000	p
81	51	01010001	Q	113	71	01110001	q
82	52	01010010	R	114	72	01110010	r
83	53	01010011	S	115	73	01110011	s
84	54	01010100	T	116	74	01110100	t
85	55	01010101	U	117	75	01110101	u
86	56	01010110	V	118	76	01110110	v
87	57	01010111	W	119	77	01110111	w
88	58	01011000	X	120	78	01111000	x
89	59	01011001	Y	121	79	01111001	y
90	5A	01011010	Z	122	7A	01111010	z
91	5B	01011011	[	123	7B	01111011	
92	5C	01011100	\	124	7C	01111100	|
93	5D	01011101	]	125	7D	01111101	
94	5E	01011110	ˆ	126	7E	01111110	˜
95	5F	01011111	_	127	7F	01111111	

1.3 Architecture de base d'un ordinateur

Dans cette partie, nous décrivons rapidement l'architecture de base d'un ordinateur et les principes de son fonctionnement.

Un *ordinateur* est une machine de traitement de l'information. Il est capable d'acquérir de l'information, de la stocker, de la transformer en effectuant des traitements quelconques, puis de la restituer sous une autre forme. Le mot *informatique* vient de la contraction des mots *information* et *automatique*.

Nous appelons *information* tout ensemble de données. On distingue généralement différents types d'informations : textes, nombres, sons, images, etc., mais aussi les instructions composant un programme. Comme on l'a vu dans la première partie, toute information est manipulée sous forme *binaire* (ou numérique) par l'ordinateur.

1.3.1 Principes de fonctionnement

Les deux principaux constituants d'un ordinateur sont la *mémoire principale* et le *processeur*. La mémoire principale (MP en abrégé) permet de stocker de l'information (programmes et données), tandis que le processeur exécute pas à pas les instructions composant les programmes.

Notion de programme

Un programme est une suite d'instructions élémentaires, qui vont être exécutées dans l'ordre par le processeur. Ces instructions correspondent à des actions très simples, comme additionner deux nombres, lire ou écrire une case mémoire, etc. Chaque instruction est codifiée en mémoire sur quelques octets.

Le processeur est capable d'exécuter des programmes en *langage machine*, c'est à dire composés d'instructions très élémentaires suivant un codage précis. Chaque type de processeur est capable d'exécuter un certain ensemble d'instructions, son *jeu d'instructions*.

Pour écrire un programme en *langage machine*, il faut donc connaître les détails du fonctionnement du processeur qui va être utilisé.

Le processeur

Le processeur est un circuit éléctronique complexe qui exécute chaque instruction très rapidement, en quelques *cycles d'horloges*. Toute l'activité de l'ordinateur est cadencée par une horloge unique, de façon à ce que tous les circuits électroniques travaillent ensembles. La fréquence de cette horloge s'exprime en MHz (millions de battements par seconde). Par exemple, un ordinateur "PC Pentium 133" possède un processeur de type Pentium et une horloge à 133 MHz.

Pour chaque instruction, le processeur effectue schématiquement les opérations suivantes :

1. lire en mémoire (MP) l'instruction à exécuter;
2. effectuer le traitement correspondant;

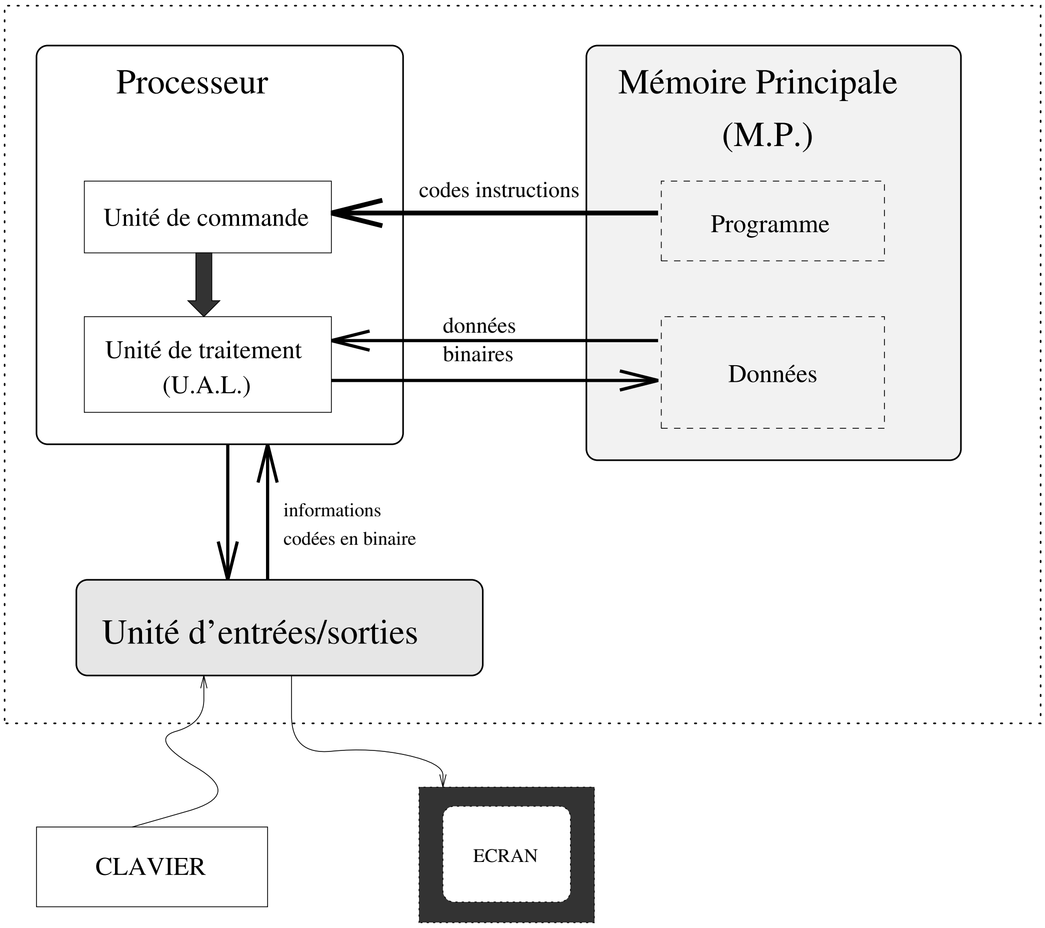


FIG. 1.1: Architecture schématique d'un ordinateur.

3. passer à l'instruction suivante.

Le processeur est divisé en deux parties (voir figure 1.1), l'unité de commande et l'unité de traitement :

- l'unité de commande est responsable de la lecture en mémoire et du décodage des instructions;
- l'unité de traitement, aussi appelée *Unité Arithmétique et Logique* (U.A.L.), exécute les instructions qui manipulent les données.

1.3.2 La mémoire principale (MP)

Structure de la MP

La mémoire est divisée en emplacements de taille fixe (par exemple 8 bits) utilisés pour stocker instructions et données.

En principe, la taille d'un emplacement mémoire pourrait être quelconque; en fait, la plupart des ordinateurs en service aujourd'hui utilisent des emplacements

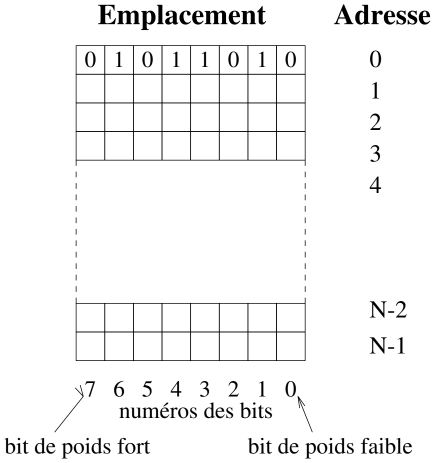


FIG. 1.2: Structure de la mémoire principale.

mémoire d'un octet (*byte* en anglais, soit 8 bits, unité pratique pour coder un caractère par exemple).

Dans une mémoire de taille N, on a N emplacements mémoires, numérotés de 0 à $N-1$. Chaque emplacement est repéré par son numéro, appelé *adresse*. L'adresse est le plus souvent écrite en hexadécimal.

La capacité (taille) de la mémoire est le nombre d'emplacements, exprimé en général en kilo-octets ou en méga-octets, voire davantage. Rappelons que le kilo informatique vaut 1024 et non 1000 ($2^{10} = 1024 \approx 1000$). Voici les multiples les plus utilisés :

1 K (Kilo)	2^{10}	= 1024
1 M (Méga)	2^{20}	= 1048 576
1 G (Giga)	2^{30}	= 1 073 741 824
1 T (Téra)	2^{40}	= 1 099 511 627 776

Opérations sur la mémoire

Seul le processeur peut modifier l'état de la mémoire[1].

Chaque emplacement mémoire conserve les informations que le processeur y écrit jusqu'à coupure de l'alimentation électrique, où tout le contenu est perdu (contrairement au contenu des mémoires externes comme les disquettes et disques durs).

Les seules opérations possibles sur la mémoire sont :
- *écriture* d'un emplacement : le processeur donne une valeur et une adresse, et la mémoire range la valeur à l'emplacement indiqué par l'adresse;
- *lecture* d'un emplacement : le processeur demande à la mémoire la valeur

[1] Sur certains ordinateurs, les controleurs d'entrées/sorties peuvent accéder directement à la mémoire (accès DMA), mais cela ne change pas le principe de fonctionnement.

contenue à l'emplacement dont il indique l'adresse. Le contenu de l'emplacement lu reste inchangé.

Unité de transfert

Notons que les opérations de lecture et d'écriture portent en général sur plusieurs octets contigüs en mémoire : un *mot* mémoire. La taille d'un mot mémoire dépend du type de processeur; elle est de

- 1 octet (8 bits) dans les processeurs 8 bits (par exemple Motorola 6502);
- 2 octets dans les processeurs 16 bits (par exemple Intel 8086);
- 4 octets dans les processeurs 32 bits (par ex. Intel 80486 ou Motorola 68030).

1.3.3 Le processeur central

Le processeur est parfois appelé CPU (de l'anglais *Central Processing Unit*) ou encore MPU (*Micro-Processing Unit*) pour les microprocesseurs.

Un *microprocesseur* n'est rien d'autre qu'un processeur dont tous les constituants sont réunis sur la même puce électronique (pastille de silicium), afin de réduire les coûts de fabrication et d'augmenter la vitesse de traitement. Les *microordinateurs* sont tous équipés de microprocesseurs.

L'architecture de base des processeurs équipant les gros ordinateurs est la même que celle des microprocesseurs.

Les registres et l'accumulateur

Le processeur utilise toujours des *registres*, qui sont des petites mémoires internes très rapides d'accès utilisées pour stocker temporairement une donnée, une instruction ou une adresse. Chaque registre stocke 8, 16 ou 32 bits.

Le nombre exact de registres dépend du type de processeur et varie typiquement entre une dizaine et une centaine.

Parmi les registres, le plus important est le registre *accumulateur*, qui est utilisé pour stocker les résultats des opérations arithmétiques et logiques. L'accumulateur intervient dans une proportion importante des instructions.

Par exemple, examinons ce qu'il se passe lorsque le processeur exécute une instruction comme "*Ajouter 5 au contenu de la case memoire d'adresse 180*" :

1. Le processeur lit et décode l'instruction;
2. le processeur demande à la mémoire la contenu de l'emplacement 180;
3. la valeur lue est rangée dans l'accumulateur;
4. l'unité de traitement (UAL) ajoute 5 au contenu de l'accumulateur;
5. le contenu de l'accumulateur est écris en mémoire à l'adresse 180.

C'est l'unité de commande (voir figure 1.1 page 16) qui déclenche chacune de ces actions dans l'ordre. L'addition proprement dite est effectuée par l'UAL.

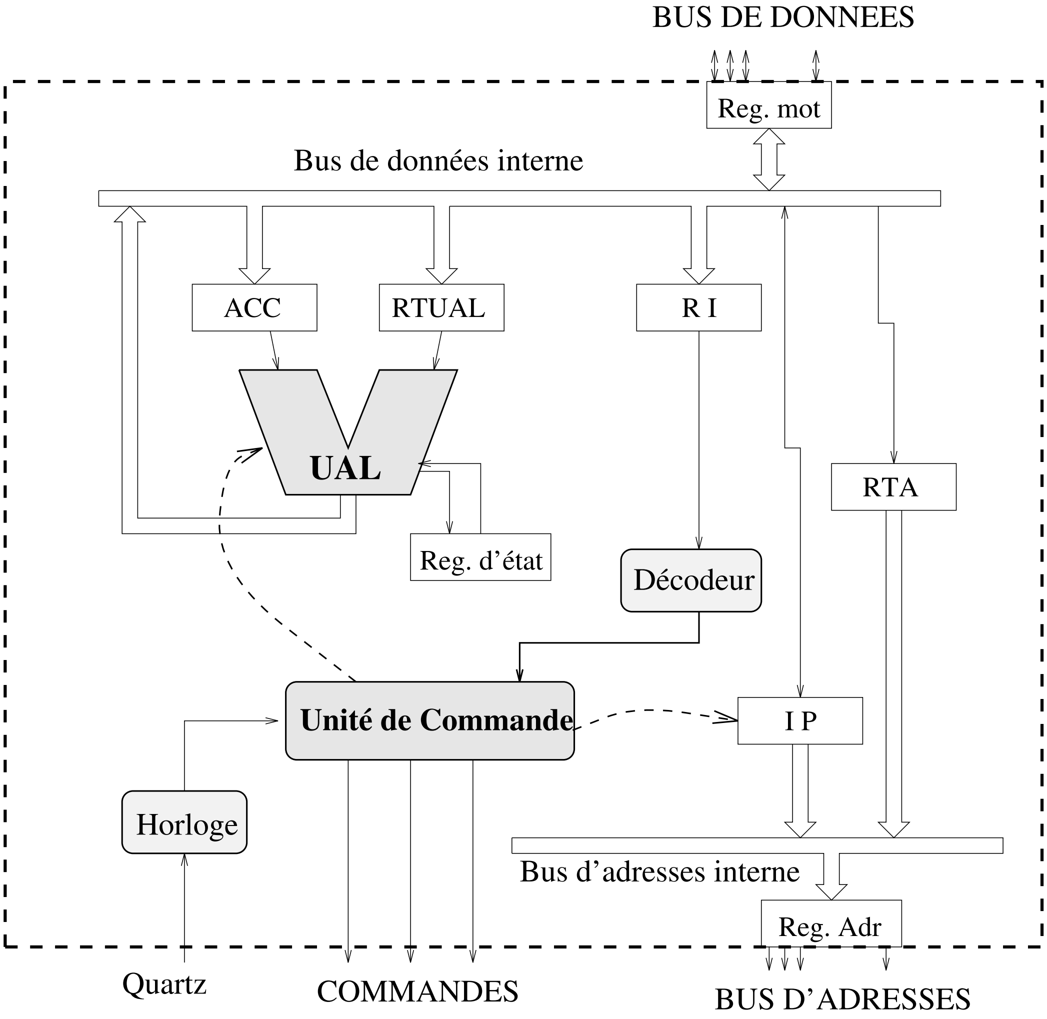


FIG. 1.3: Schéma simplifié d'un processeur. Le processeur est relié à l'extérieur par les bus de données et d'adresses, le signal d'horloge et les signaux de commandes.

Architecture d'un processeur à accumulateur

La figure 1.3 représente l'architecture interne simplifiée d'un MPU à accumulateur. On y distingue l'unité de commande, l'UAL, et le *décodeur* d'instructions, qui, à partir du code de l'instruction lu en mémoire actionne la partie de l'unité de commande nécessaire.

Les informations circulent à l'intérieur du processeur sur deux *bus internes*, l'un pour les données, l'autre pour les instructions.

On distingue les registres suivants :

ACC : Accumulateur;

RTUAL : Registre Tampon de l'UAL, stocke temporairement l'un des deux opérandes d'une instructions arithmétiques (la valeur 5 dans l'exemple donné plus haut);

Reg. d'état : stocke les *indicateurs*, que nous étudierons plus tard;

RI : Registre Instruction, contient le code de l'instruction en cours d'exécution (lu en mémoire via le bus de données);

IP : *Instruction Pointer* ou Compteur de Programme, contient l'adresse de l'emplacement mémoire où se situe la prochaine instruction à exécuter;

RTA : Registre Tampon d'Adresse, utilisé pour accéder à une donnée en mémoire.

Les signaux de commandes permettent au processeur de communiquer avec les autres circuits de l'ordinateur. On trouve en particulier le signal R/W (Read/Write), qui est utilisé pour indiquer à la mémoire principale si l'on effectue un accès en lecture ou en écriture.

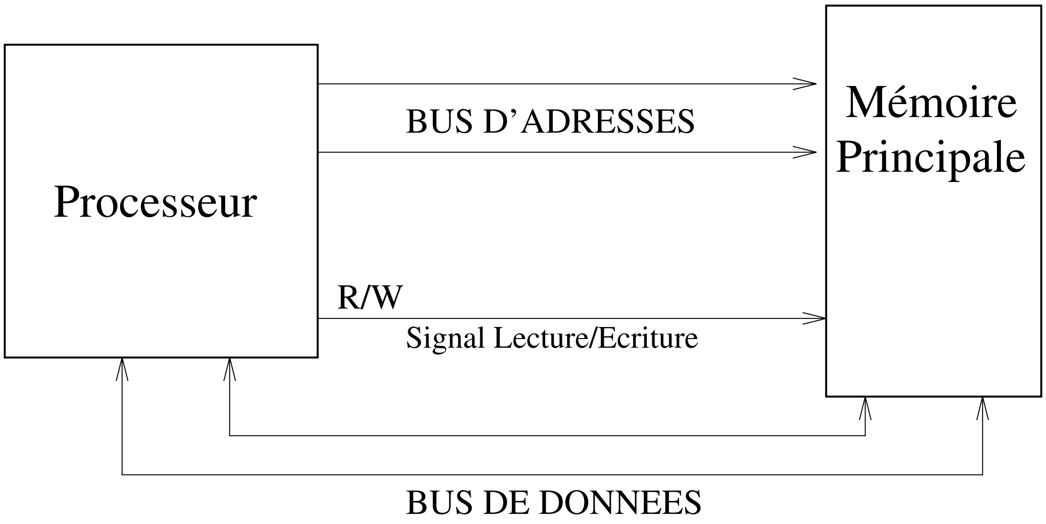


FIG. 1.4: Connexions Processeur-Mémoire : bus de données, bus d'adresse et signal lecture/écriture.

1.3.4 Liaisons Processeur-Mémoire : les bus

Les informations échangées entre la mémoire et le processeur circulent sur des *bus*. Un *bus* est simplement un ensemble de n fils conducteurs, utilisés pour transporter n signaux binaires.

Le bus d'adresse est un bus unidirectionnel : seul le processeur envoie des adresses. Il est composé de a fils; on utilise donc des adresses de a bits. La mémoire peut posséder au maximum 2^a emplacements (adresses 0 à $2^a - 1$).

Le bus de données est un bus bidirectionnel. Lors d'une lecture, c'est la mémoire qui envoie un mot sur le bus (le contenu de l'emplacement demandé); lors d'une écriture, c'est le processeur qui envoie la donnée.

Partie 2

Introduction au langage machine

Dans cette partie du cours, nous allons étudier la programmation en langage machine et en assembleur d'un microprocesseur. L'étude complète d'un processeur réel, comme le 80486 ou le Pentium fabriqués par Intel, dépasse largement le cadre de ce cours : le nombre d'instructions et de registres est très élevé. Nous allons ici nous limiter à un sous-ensemble du microprocesseur 80486 (seuls les registres et les instructions les plus simples seront étudiés). De cette façon, nous pourrons tester sur un PC les programmes en langage machine que nous écrirons.

2.1 Caractéristiques du processeur étudié

La gamme de microprocesseurs 80x86 équipe les micro-ordinateurs de type PC et compatibles. Les premiers modèles de PC, commercialisés au début des années 1980, utilisaient le 8086, un microprocesseur 16 bits[1].

Le modèles suivants ont utilisé successivement le 80286, 80386, 80486 et Pentium (ou 80586)[2]. Chacun de ces processeurs est plus puissant que les précédents : augmentation de la fréquence d'horloge, de la largeur de bus (32 bits d'adresse et de données), introduction de nouvelles instructions (par exemple calcul sur les réels) et ajout de registres. Chacun d'entre eux est *compatible* avec les modèles précédents; un programme écrit dans le langage machine du 286 peut s'exécuter sans modification sur un 486. L'inverse n'est pas vrai, puisque chaque génération a ajouté des instructions nouvelles. On parle donc de *compatibilité ascendante*.

Du fait de cette compatibilité, il est possible de programmer le 486, utilisé dans nos salles de Travaux Pratiques, comme un processeur 16 bits. C'est ce que nous ferons cette année par souci de simplification. Ainsi, nous n'utiliserons que des registres de 16 bits.

[1] Avec un bus d'adresses de 20 bits pour gérer jusqu'a 1Mo de mémoire.

[2] Les PC actuels (1996) sont équipés de Pentium, cadencé à environ 100 MHz.

Voici les caractéristiques du processeur simplifié que nous étudierons :

CPU 16 bits à accumulateur :
- bus de données 16 bits;
- bus d'adresse 32 bits;

Registres :
- accumulateur AX (16 bits);
- registres auxiliaires BX et CX (16 bits);
- pointeur d'instruction IP (16 bits);
- registres segments CS, DS, SS (16 bits);
- pointeur de pile SP (16 bits), et pointeur BP (16 bits).

Les registres IP et AX (accumulateur) ont déja été étudiés. Les autres le seront progressivement dans ce chapitre.

Nous verrons plus loin que les registres de données de 16 bits peuvent parfois être utilisés comme deux registres indépendants de 8 bits (AX devient la paire (AH,AL)) :

$$\underbrace{\overbrace{\boxed{\text{AH}}}^{\text{8 bits}}\overbrace{\boxed{\text{AL}}}^{\text{8 bits}}}_{\text{AX (16 bits)}} \qquad \underbrace{\overbrace{\boxed{\text{BH}}}^{\text{8 bits}}\overbrace{\boxed{\text{BL}}}^{\text{8 bits}}}_{\text{BX (16 bits)}} \qquad \underbrace{\overbrace{\boxed{\text{CH}}}^{\text{8 bits}}\overbrace{\boxed{\text{CL}}}^{\text{8 bits}}}_{\text{CX (16 bits)}}$$

Noter que nous évoquons ici uniquement les registres qui apparaissent explicitement dans l'écriture des instructions, et pas les registres intermédiaires tels que RI, RTUAL et RTA.

2.2 Jeu d'instruction

2.2.1 Types d'instructions

Instructions d'affectation

Déclenchent un transfert de données entre l'un des registres du processeur et la mémoire principale.
- transfert CPU $\leftarrow$ Mémoire Principale (MP) (= lecture en MP);
- transfert CPU $\rightarrow$ Mémoire Principale (MP) (= écriture en MP);

Instructions arithmétiques et logiques

Opérations entre une donnée et l'accumulateur AX. Le résultat est placé dans l'accumulateur. La donnée peut être une constante ou une valeur contenue dans un emplacement mémoire.

Exemples :
- addition : AX $\leftarrow$ AX + donnée;
- soustraction : AX $\leftarrow$ AX - donnée;

– incrémentation[3] de AX : AX $\leftarrow$ AX + 1;
– décrémentation : AX $\leftarrow$ AX - 1;
– décalages à gauche et à droite;

Instructions de comparaison

Comparaison du registre AX à une donnée et positionnement des indicateurs.

Instructions de branchement

La prochaine instruction à exécuter est repérée en mémoire par le registre IP. Les instructions de branchement permettent de modifier la valeur de IP pour exécuter une autre instruction (boucles, tests, etc.).

On distingue deux types de branchements :
– *branchements inconditionnels* : IP $\leftarrow$ adresse d'une instruction;
– *branchements conditionnels* : Si une condition est satisfaite, alors branchement, sinon passage simple à l'instruction suivante.

2.2.2 Codage des instructions et mode d'adressage

Les instructions et leurs opérandes (paramètres) sont stockés en mémoire principale. La taille totale d'une instruction (nombre de bits nécessaires pour la représenter en mémoire) dépend du type d'instruction et aussi du type d'opérande. Chaque instruction est toujours codée sur un nombre entier d'octets, afin de faciliter son décodage par le processeur.

Une instruction est composée de deux champs :
– le code opération, qui indique au processeur quelle instruction réaliser;
– le champ opérande qui contient la donnée, ou la référence à une donnée en mémoire (son adresse).

champ code opération	champ code opérande

Selon la manière dont la donnée est spécifiée, c'est à dire selon le mode d'adressage de la donnée, une instruction sera codée par 1, 2, 3 ou 4 octets.

Nous distinguerons ici quatre modes d'adressage : implicite, immédiat, direct et relatif (nous étudierons plus tard un autre mode, l'adressage indirect).

Adressage implicite

L'instruction contient seulement le code opération, sur 1 ou 2 octets.

code opération (1 ou 2 octets)

L'instruction porte sur des registres ou spécifie une opération sans opérande (exemple : "incrémenter AX").

[3] *Incrémenter* un registre (ou une variable) signifie lui ajouter 1, et le *décrémenter* lui soustraire 1.

Adressage immédiat

Le champ opérande contient la donnée (une valeur constante sur 1 ou 2 octets).

code opération (1 ou 2 octets)	valeur (1 ou 2 octets)

Exemple : "Ajouter la valeur 5 à AX". Ici l'opérande 5 est codée sur 2 octets puisque l'opération porte sur un registre 16 bits (AX).

Adressage direct

Le champ opérande contient l'*adresse* de la donnée en mémoire principale sur 2 octets.

code opération (1 ou 2 octets)	adresse de la donnée (2 octets)

Attention : dans le 80x86, les adresses sont toujours manipulées sur 16 bits, quelle que soit la taille réelle du bus. Nous verrons plus tard comment le processeur fabrique les adresses réelles sur 32 bits.

Exemple : "Placer dans AX la valeur contenue à l'adresse 130H".

Adressage relatif

Ce mode d'adressage est utilisé pour certaines instructions de branchement. Le champ opérande contient un entier relatif codé sur 1 octet, nommé *déplacement*, qui sera ajouté à la valeur courante de IP.

code opération (1 octet)	déplacement (1 octet)

2.2.3 Temps d'exécution

Chaque instruction nécessite un certain nombre de cycles d'horloges pour s'effectuer. Le nombre de cycles dépend de la complexité de l'instruction et aussi du mode d'adressage : il est plus long d'accéder à la mémoire principale qu'à un registre du processeur.

La durée d'un cycle dépend bien sur de la fréquence d'horloge de l'ordinateur. Plus l'horloge bat rapidement, plus un cycle est court et plus on exécute un grand nombre d'instructions par seconde.

2.2.4 Ecriture des instructions en langage symbolique

Voici un programme en langage machine 80486, implanté à l'adresse 0100H :

```
A1 01 10 03 06 01 12 A3 01 14
```

Ce programme additionne le contenu de deux cases mémoire et range le résultat dans une troisième. Nous avons simplement transcrit en hexadécimal le code du programme. Il est clair que ce type d'écriture n'est pas très utilisable par un être humain.

A chaque instruction que peut exécuter le processeur correspond une représentation binaire sur un ou plusieurs octets, comme on l'a vu plus haut. C'est le travail du processeur de décoder cette représentation pour effectuer les opérations correspondantes.

Afin de pouvoir écrire (et relire) des programmes en langage machine, on utilise une notation symbolique, appelée *langage assembleur*. Ainsi, la première instruction du programme ci-dessus (code `A1 01 10`) sera notée :

```
MOV AX, [0110]
```

elle indique que le mot mémoire d'adresse 0110H est chargé dans le registre AX du processeur.

On utilise des programmes spéciaux, appelés *assembleurs*, pour traduire automatiquement le langage symbolique en code machine.

Voici une transcription langage symbolique du programme complet. L'adresse de début de chaque instruction est indiquée à gauche (en hexadécimal).

```
Adresse  Contenu MP   Langage Symbolique   Explication en francais

0100     A1 01 10     MOV AX, [0110]      Charger AX avec le contenu
                                           de 0110.
0103     03 06 01 12  ADD AX, [0112]      Ajouter le contenu de 0112
                                           a AX (resultat dans AX).
0107     A3 01 14     MOV [0114], AX      Ranger AX en 0114.
```

Sens des mouvements de données

La plupart des instructions spécifient des mouvements de données entre la mémoire principale et le microprocesseur. En langage symbolique, on indique toujours la destination, puis la source. Ainsi l'instruction

```
MOV AX, [0110]
```

transfère le contenu de l'emplacement mémoire 0110H dans l'accumulateur, tandis que `MOV [0112], AX` transfère le contenu de l'accumulateur dans l'emplacement mémoire 0112.

L'instruction `MOV` (de l'anglais *move*, déplacer) s'écrit donc toujours :

```
MOV destination, source
```

Modes d'adressage

- En adressage immédiat, on indique simplement la valeur de l'opérande en hexadécimal. Exemple :
  ```
  MOV AX, 12
  ```
- En adressage direct, on indique l'adresse d'un emplacement en mémoire principale en hexadécimal entre crochets :

```
MOV AX, [A340]
```

– En adressage relatif, on indique simplement l'adresse (hexa). L'assembleur traduit automatiquement cette adresse en un déplacement (relatif sur un octet). Exemple :
```
JNE  0108
```
(nous étudierons l'instruction `JNE` plus loin).

Tableau des instructions

Le tableau 2.2.4 donne la liste de quelques instructions importantes du 80x86.

Retour au DOS

A la fin d'un programme en assembleur, on souhaite en général que l'interpréteur de commandes du DOS reprenne le contrôle du PC. Pour cela, on utilisera la séquence de deux instructions (voir tableauu 2.2.4) :

```
MOV  AH, 4C
INT 21
```

2.2.5 Utilisation du programme *debug*

debug est un programme qui s'exécute sur PC (sous DOS) et qui permet de manipuler des programmes en langage symbolique. Il est normalement distribué avec toutes les versions du système MS/DOS. Nous l'utiliserons en travaux pratiques.

Les fonctionnalités principales de *debug* sont les suivantes :

– Affichage du contenu d'une zone mémoire en hexadécimal ou en ASCII;
– Modification du contenu d'une case mémoire quelconque;
– Affichage en langage symbolique d'un programme;
– Entrée d'un programme en langage symbolique; *debug* traduit les instructions en langage machine et calcule automatiquement les déplacements en adressage relatif.
– Affichage et modification de la valeur des registres du processeur;

2.3 Branchements

Normalement, le processeur exécute une instruction puis passe à celle qui suit en mémoire, et ainsi de suite séquentiellement. Il arrive fréquemment que l'on veuille faire répéter au processeur une certaine suite d'instructions, comme dans le programme :

```
Repeter 3 fois:
   ajouter 5 au registre BX
```

En d'autres occasions, il est utile de déclencher une action qui dépend du résultat d'un test :

Symbole	Code Op.	Octets	
MOV AX, *valeur*	B8	3	AX $\leftarrow$ *valeur*
MOV AX, [*adr*]	A1	3	AX $\leftarrow$ contenu de l'adresse *adr.*
MOV [*adr*], AX	A3	3	range AX à l'adresse *adr.*
ADD AX, *valeur*	05	3	AX $\leftarrow$ AX + *valeur*
ADD AX, [*adr*]	03 06	4	AX $\leftarrow$ AX + contenu de *adr.*
SUB AX, *valeur*	2D	3	AX $\leftarrow$ AX - *valeur*
SUB AX, [*adr*]	2B 06	4	AX $\leftarrow$ AX - contenu de *adr.*
SHR AX, 1	D1 E8	2	décale AX à droite.
SHL AX, 1	D1 E0	2	décale AX à gauche.
INC AX	40	1	AX $\leftarrow$ AX + 1
DEC AX	48	1	AX $\leftarrow$ AX - 1
CMP AX, *valeur*	3D	3	compare AX et *valeur.*
CMP AX, [*adr*]	3B 06	4	compare AX et contenu de *adr.*
JMP *adr*	EB	2	saut inconditionnel (adr. relatif).
JE *adr*	74	2	saut si =
JNE *adr*	75	2	saut si $\neq$
JG *adr*	7F	2	saut si >
JLE *adr*	7E	2	saut si $\leq$
JA *adr*			saut si CF = 0
JB *adr*			saut si CF = 1
Fin du programme (retour au DOS) :			
MOV AH, 4C	B4 4C	2	
INT 21	CD 21	2	

TAB. 2.1: Quelques instructions du 80x86. Le code de l'instruction est donné en hexadécimal dans la deuxième colonne. La colonne suivante précise le nombre d'octets nécessaires pour coder l'instruction complète (opérande inclus). On note *valeur* une valeur sur 16 bits, et *adr* une adresse sur 16 bits également.

```
Si x < 0:
   y = - x
sinon
   y = x
```

Dans ces situations, on doit utiliser une instruction de *branchement*, ou *saut*, qui indique au processeur l'adresse de la prochaine instruction à exécuter.

Rappelons que le registre IP du processeur conserve l'adresse de la prochaine instruction à exécuter. Lors d'un déroulement normal, le processeur effectue les actions suivantes pour chaque instruction :

1. lire et décoder l'instruction à l'adresse IP;
2. IP $\leftarrow$ IP + *taille de l'instruction*;
3. exécuter l'instruction.

Pour modifier le déroulement normal d'un programme, il suffit que l'exécution de l'instruction modifie la valeur de IP. C'est ce que font les instructions de branchement.

On distingue deux catégories de branchements, selon que le saut est toujours effectué (sauts *inconditionnels*) ou qu'il est effectué seulement si une condition est vérifiée (sauts *conditionnels*).

2.3.1 Saut inconditionnel

La principale instruction de saut inconditionnel est JMP. En adressage relatif, l'opérande de JMP est un *déplacement*, c'est à dire une valeur qui va être ajoutée à IP.

L'action effectuée par JMP est :

$$\text{IP} = \text{IP} + \textit{déplacement}$$

Le déplacement est un entier relatif sur codée 8 bits. La valeur du déplacement à utiliser pour atteindre une certaine instruction est :

$$\textit{déplacement} = \text{adr. instruction visée - adr. instruction suivante}$$

Exemple : le programme suivant écrit indéfiniment la valeur 0 à l'adresse 0140H. La première instruction est implantée à l'adresse 100H.

```
Adresse  Contenu MP   Langage Symbolique   Explication en francais

0100     B8 00 00     MOV AX, 0            met AX a zero
0103     A3 01 40     MOV [140], AX        ecrit a l'adresse 140
0106     EB FC        JMP 0103             branche en 103
0107                  xxx -> instruction jamais executee
```

Le déplacement est ici égal à FCH, c'est à dire -4 (=103H-107H).

2.3.2 Indicateurs

Les instructions de branchement conditionnels utilisent les *indicateurs*, qui sont des bits spéciaux positionnés par l'UAL après certaines opérations. Les indicateurs sont regroupés dans le *registre d'état* du processeur. Ce registre n'est pas accessible globalement par des instructions; chaque indicateur est manipulé individuellement par des instructions spécifiques.

Nous étudierons ici les indicateurs nommés ZF, CF, SF et OF.

ZF *Zero Flag*
: Cet indicateur est mis à 1 lorsque le résultat de la dernière opération est zéro. Lorsque l'on vient d'effectuer une soustraction (ou une comparaison), ZF=1 indique que les deux opérandes étaient égaux. Sinon, ZF est positionné à 0.

CF *Carry Flag*
: C'est l'indicateur de report (retenue), qui intervient dans les opérations d'addition et de soustractions sur des entiers naturels. Il est positionné en particulier par les instructions ADD, SUB et CMP.
: CF = 1 s'il y a une retenue après l'addition ou la soustraction du bit de poids fort des opérandes. Exemples (sur 4 bits pour simplifier) :

```
         0 1 0 0             1 1 0 0             1 1 1 1
       + 0 1 1 0           + 0 1 1 0           + 0 0 0 1
       ---------           ---------           ---------
CF=0     1 0 1 0    CF=1     0 0 1 0    CF=1     0 0 0 0
```

SF *Sign Flag*
: SF est positionné à 1 si le bit de poids fort du résultat d'une addition ou soustraction est 1; sinon SF=0. SF est utile lorsque l'on manipule des entiers relatifs, car le bit de poids fort donne alors le signe du résultat. Exemples (sur 4 bits) :

```
         0 1 0 0             1 1 0 0             1 1 1 1
       + 0 1 1 0           + 0 1 1 0           + 0 0 0 1
       ---------           ---------           ---------
SF=1     1 0 1 0    SF=0     0 0 1 0    SF=0     0 0 0 0
```

OF *Overflow Flag*
: Indicateur de débordement[4] OF=1 si le résultat d'une addition ou soustraction donne un nombre qui n'est pas codable *en relatif* dans l'accumulateur (par exemple si l'addition de 2 nombres positifs donne un codage négatif).

```
         0 1 0 0             1 1 0 0             1 1 1 1
       + 0 1 1 0           + 0 1 1 0           + 0 0 0 1
       ---------           ---------           ---------
OF=1     1 0 1 0    OF=0     0 0 1 0    OF=1     0 0 0 0
```

Lorsque l'UAL effectue une addition, une soustraction ou une comparaison, les quatre indicateurs sont positionnés. Certaines autres instructions que nous étudierons plus loin peuvent modifier les indicateurs.

Instruction CMP

Il est souvent utile de tester la valeur du registre AX sans modifier celui-ci. L'instruction CMP effectue exactement les même opération que SUB, mais ne

[4] Débordement, ou dépassement de capacité, *overflow* en anglais.

stocke pas le résultat de la soustraction. Son seul effet est donc de positionner les indicateurs.

Exemple : après l'instruction

```
CMP AX, 5
```

on aura ZF = 1 si AX contient la valeur 5, et ZF = 0 si AX est différent de 5.

Instructions STC et CLC

Ces deux instructions permettent de modifier la valeur de l'indicateur CF.

Symbole	
STC	CF $\leftarrow$ 1 (*SeT Carry*)
CLC	CF $\leftarrow$ 0 (*CLear Carry*)

2.3.3 Sauts conditionnels

Les instructions de branchements conditionnels effectuent un saut (comme JMP) si une certaine condition est vérifiée. Si ce n'est pas le cas, le processeur passe à l'instruction suivante (l'instruction ne fait rien).

Les conditions s'expriment en fonction des valeurs des indicateurs. Les instructions de branchement conditionnel s'utilisent en général immédiatement après une instruction de comparaison CMP.

Voici la liste des instructions de branchement les plus utiles :

JE *Jump if Equal*
saut si ZF = 1;

JNE *Jump if Not Equal*
saut si ZF = 0;

JG *Jump if Greater*
saut si ZF = 0 et SF = OF;

JLE *Jump if Lower or Equal*
saut si ZF=1 ou SF$\neq$OF;

JA *Jump if Above*
saut si CF=0 et ZF=0;

JBE *Jump if Below or Equal*
saut si CF=1 ou ZF=1.

JB *Jump if Below*
saut si CF=1.

Note : les instructions JE et JNE sont parfois écrites JZ et JNZ (même code opération).

2.4 Instructions Arithmétiques et logiques

Les instructions arithmétiques et logiques sont effectuées par l'UAL. Nous avons déja vu les instructions d'addition et de soustraction (ADD, SUB). Nous abordons ici les instructions qui travaillent sur la représentation binaire des données : décalages de bits, opérations logiques bit à bit.

Notons que toutes ces opérations modifient l'état des indicateurs.

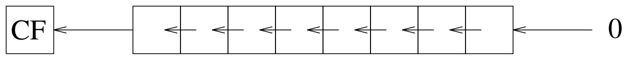
2.4.1 Instructions de décalage et de rotation

Ces opérations décalent vers la gauche ou vers la droite les bits de l'accumulateur. Elles sont utilisées pour décoder bit à bit des données, ou simplement pour diviser ou multiplier rapidement par une puissance de 2. En effet, décaler AX de n bits vers la gauche revient à le multiplier par 2^n (sous réserve qu'il représente un nombre naturel et qu'il n'y ait pas de dépassement de capacité). De même, un décalage vers la droite revient à diviser par 2^n.

Voici les variantes les plus utiles de ces instructions. Elles peuvent opérer sur les registres AX ou BX (16 bits) ou sur les registres de 8 bits AH, AL, BH et BL.

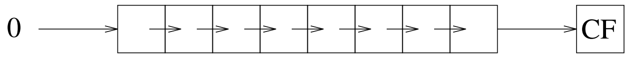
SHL *registre*, 1 *(Shift Left)*

Décale les bits du registre d'une position vers la gauche. Le bit de gauche est transféré dans l'indicateur CF. Les bits introduits à droite sont à zéro.



SHR *registre*, 1 *(Shift Right)*

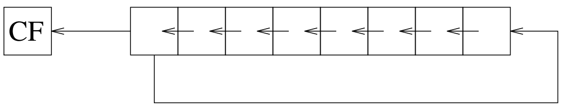
Comme SHL mais vers la droite. Le bit de droite est transféré dans CF.



SHL et SHR peuvent être utilisé pour multiplier/diviser des entiers *naturels* (et non des relatifs car le bit de signe est perdu[5]).

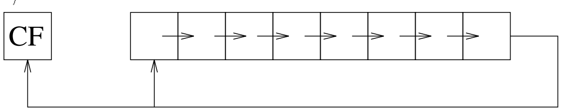
ROL *registre*, 1 *(Rotate Left)*

Rotation vers la gauche : le bit de poids fort passe à droite, et est aussi copié dans CF. Les autres bits sont décalés d'une position.



ROR *registre*, 1 *(Rotate Right)*

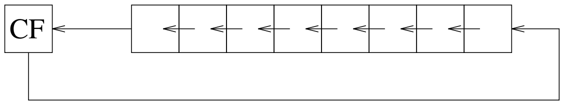
Comme ROL, mais à droite.



[5]Il existe d'autres instructions pour les relatifs, que nous ne décrirons pas ici.

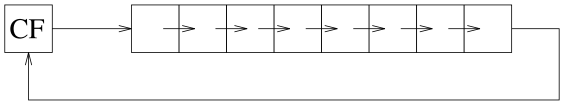
RCL ***registre*****, 1** *(Rotate Carry Left)*

Rotation vers la gauche en passant par l'indicateur CF. CF prend la place du bit de poids faible; le bit de poids fort part dans CF.



RCR ***registre*****, 1** *(Rotate Carry Right)*

Comme RCL, mais vers la droite.



RCL et RCR sont utiles pour lire bit à bit le contenu d'un registre[6].

2.4.2 Instructions logiques

Les instructions logiques effectuent des opérations logiques bit à bit. On dispose de trois opérateurs logiques : ET, OU et OU exclusif. Il n'y a jamais propagation de retenue lors de ces opérations (chaque bit du résultat est calculé indépendamment des autres).

```
     0 0 1 1            0 0 1 1             0 0 1 1
OU   0 1 0 1       ET   0 1 0 1      OU EX  0 1 0 1
-----------        -----------            -----------
     0 1 1 1            0 0 0 1             0 1 1 0
```

Les trois instructions OR, AND et XOR sont de la forme

OR *destination, source.*

destination désigne le registre ou l'emplacement mémoire (adresse) où doit être placé le résultat. *source* désigne une constante (adressage immédiat), un registre (adressage implicite), ou une adresse (adressage direct).

Exemples :

```
OR   AX, FF00   ; AX <-  AX ou FF00
OR   AX, BX     ; AX <-  AX ou BX
OR   AX, [1492] ; AX <-  AX ou [1492]
```

OR ***destination, source*** *(OU)*

OU logique. Chaque bit du résultat est égal à 1 si au moins l'un des deux bits opérande est 1.

OR est souvent utilisé pour forcer certains bits à 1. Par exemple après `OR AX, FF00`, l'octet de poids fort de AX vaut FF, tandis que l'octet de poids faible est inchangé.

[6] On pourra utiliser l'instruction JB pour brancher si CF=1 après RCL ou RCR.

AND ***destination, source*** *(ET)*

ET logique. Chaque bit du résultat est égal à 1 si les deux bits opérandes sont à 1.

AND est souvent utilisé pour forcer certains bits à 0. Après `AND AX, FF00`, l'octet de poids faible de AX vaut 00, tandis que l'octet de poids fort est inchangé.

XOR ***destination, source*** *(OU EXCLUSIF)*

OU exclusif. Chaque bit du résultat est égal à 1 si l'un ou l'autre des bits opérandes (mais *pas les deux*) vaut 1.

XOR est souvent utilisé pour inverser certains bits. Après `XOR AX, FFFF`, tous les bits de AX sont inversés.

2.4.3 Correspondance avec le langage C

Nous étudierons plus loin dans ce cours comment un *compilateur* traduit les programmes écrits en langage C en langage machine.

La table suivante établi un parallèle entre les instructions arithmétiques et logiques du 80x86 et les opérateurs du langage C (lorsque ces derniers agissent sur des variables *non signées*).

Opérateur C	Instruction 80x86	
`+`	ADD	addition;
`-`	SUB	soustraction;
`<<`	SHL	décalage à gauche;
`>>`	SHR	décalage à droite;
`|`	OR	ou bit à bit;
`&`	AND	et bit à bit;
`^`	XOR	ou exclusif bit à bit.

Partie 3

L'assembleur 80x86

3.1 L'assembleur

3.1.1 Pourquoi l'assembleur ?

Lorsque l'on doit lire ou écrire un programme en langage machine, il est difficile d'utiliser la notation hexadécimale (voir l'exemple page 26). On écrit les programmes à l'aide de symboles[1] comme MOV, ADD, etc. Les concepteurs de processeur, comme Intel, fournissent toujours une documentation avec les codes des instructions de leur processeur, et les symboles correspondant.

Nous avons déja utilisé un programme, *debug*, très utile pour traduire automatiquement les symboles des instructions en code machine. Cependant, *debug* n'est utilisable que pour mettre au point de petits programmes. En effet, le programmeur doit spécifier lui même les adresses des données et des instructions. Soit par exemple le programme suivant, qui multiplie une donnée en mémoire par 8 :

```
0100    MOV BX, [0112]  ; charge la donnee
0103    MOV AX, 3
0106    SHL BX          ; decale a gauche
0108    DEC AX
0109    JNE  0106       ; recommence 3 fois
010B    MOV [0111], BX  ; range le resultat
010E    MOV AH, 4C
0110    INT 21H
0112    ; on range ici la donnee
```

Nous avons spécifié que la donnée était rangée à l'adresse 0111H, et que l'instruction de branchement JE allait en 0106H. Si l'on désire modifier légèrement ce programme, par exemple ajouter une instruction avant `MOV BX, [0111]`, il va falloir modifier ces deux adresses. On conçoit aisément que ce travail devienne très difficile si le programme manipule beaucoup de variables.

[1] Les symboles associés aux instructions sont parfois appelés *mnémoniques*.

L'utilisation d'un *assembleur* résout ces problèmes. L'assembleur permet en particulier de nommer les variables (un peu comme en langage C) et de repérer par des *étiquettes* certaines instructions sur lesquelles on va effectuer des branchements.

3.1.2 De l'écriture du programme à son exécution

L'assembleur est un utilitaire qui n'est pas interactif, contrairement à l'utilitaire *debug*. Le programme que l'on désire traduire en langage machine (on dit *assembler*) doit être placé dans un fichier texte (avec l'extension `.ASM` sous DOS).

La saisie du programme source au clavier nécessite un programme appelé *éditeur de texte*.

L'opération d'assemblage traduit chaque instruction du programme source en une instruction machine. Le résultat de l'assemblage est enregistré dans un fichier avec l'extension `.OBJ` (*fichier objet*).

Le fichier `.OBJ` n'est pas directement exécutable. En effet, il arrive fréquemment que l'on construise un programme exécutable à partir de plusieurs fichiers sources. Il faut "relier" les fichiers objets à l'aide d'un utilitaire nommé *éditeur de lien* (même si l'on en a qu'un seul). L'éditeur de liens fabrique un fichier exécutable, avec l'extension `.EXE`.

Le fichier `.EXE` est directement exécutable. Un utilitaire spécial du système d'exploitation (DOS ici), le *chargeur* est responsable de la lecture du fichier exécutable, de son implantation en mémoire principale, puis du lancement du programme.

3.1.3 Structure du programme source

La structure générale d'un programme assembleur est représentée figure 3.1.

Comme tout programme, un programme écrit en assembleur comprend des définitions de données et des instructions, qui s'écrivent chacune sur une ligne de texte.

Les données sont déclarées par des *directives*, mots clef spéciaux que comprend l'assembleur. Les directives qui déclarent des données sont regroupées dans le *segment de données*, qui est délimité par les directives `SEGMENT` et `ENDS`.

Les instructions sont placées dans un autre segment, le *segment de code*.

La directive `ASSUME` est toujours présente et sera expliquée plus loin (section 3.2.2).

La première instruction du programme (dans le segment d'instruction) doit toujours être repérée par une étiquette. Le fichier doit se terminer par la directive `END` avec le nom de l'étiquette de la première instruction (ceci permet d'indiquer à l'éditeur de liens quelle est la première instruction à exécuter lorsque l'on lance le programme).

Les points-virgules indiquent des commentaires.

```
data         SEGMENT    ; data est le nom du segment de donnees

  ; directives de declaration de donnees

data         ENDS       ; fin du segment de donnees

             ASSUME DS:data, CS:code

code         SEGMENT    ; code est le nom du segment d'instructions

debut:                  ; 1ere instruction, avec l'etiquette debut

  ; suite d'instructions

code         ENDS

             END debut  ; fin du programme, avec l'etiquette
                        ; de la premiere instruction.
```

FIG. 3.1: Structure d'un programme en assembleur (fichier .ASM).

3.1.4 Déclaration de variables

On déclare les variables à l'aide de directives. L'assembleur attribue a chaque variable une adresse. Dans le programme, on repère les variables grâce à leur nom.

Les noms des variables (comme les étiquettes) sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre. Le nom peut comporter des majuscules, des minuscules, des chiffres, plus les caractères `@`, `?` et `_`.

Lors de la déclaration d'une variable, on peut lui affecter une valeur initiale.

Variables de 8 ou 16 bits

Les directives DB (*Define Byte*) et DW (*Define Word*) permettent de déclarer des variables de respectivement 1 ou 2 octets.

Exemple d'utilisation :

```
data    SEGMENT
entree  DW  15     ; 2 octets initialises a 15
sortie  DW  ?      ; 2 octets non initialises
cle     DB  ?      ; 1 octet non initialise
nega    DB  -1     ; 1 octet initialise a -1
data    ENDS
```

Les valeurs initiales peuvent être données en hexadécimal (constante terminée par `H`) ou en binaire (terminée par `b`) :

```
data    SEGMENT
truc    DW  0F0AH     ; en hexa
masque  DB  01110000b ; en binaire
data    ENDS
```

Les variables s'utilisent dans le programme en les désignant par leur nom. Après la déclaration précédente, on peut écrire par exemple :

```
MOV  AX, truc
AND  AL, masque
MOV  truc, AX
```

L'assembleur se charge de remplacer les noms de variable par les adresses correspondantes.

Tableaux

Il est aussi possible de déclarer des tableaux, c'est à dire des suite d'octets ou de mots consécutifs.

Pour cela, utiliser plusieurs valeurs initiales :

```
data    SEGMENT
machin  db  10, 0FH   ; 2 fois 1 octet
chose   db  -2, 'ALORS'
data    ENDS
```

Remarquez la déclaration de la variable `chose` : un octet à -2 (=FEH), suivi d'une suite de caractères. L'assembleur n'impose aucune convention pour la représentation des chaînes de caractères : c'est à l'utilisateur d'ajouter si nécessaire un octet nul pour marquer la fin de la chaîne.

Après chargement de ce programme, la mémoire aura le contenu suivant :

Début du segment `data` →	0AH	← machin
	OFH	← machin + 1
	FEH	← chose
	41H	← chose + 1
	4CH	← chose + 2
	4FH	← chose + 3
	52H	← chose + 4
	53H	← chose + 5

Si l'on veut écrire un caractère `X` à la place du `O` de `ALORS`, on pourra écrire :

```
MOV  AL, 'X'
MOV  chose+1, AL
```

Notons que `chose+1` est une constante (valeur connue au moment de l'assemblage) : l'instruction générée par l'assembleur pour
`MOV  chose+1, AL`
est `MOV [adr], AL` .

Directive dup

Lorsque l'on veut déclarer un tableau de n cases, toutes initialisées à la même valeur, on utilise la directive `dup` :

```
tab   DB  100 dup (15) ; 100 octets valant 15
zzz   DW  10  dup (?)   ; 10 mots de 16 bits non initialises
```

3.2 Segmentation de la mémoire

Nous abordons ici le problème de la segmentation de la mémoire. Nous venons de voir qu'en assembleur, les données étaient normalement regroupées dans une zone mémoire nommée *segment de données*, tandis que les instructions étaient placées dans un *segment d'instructions*. Ce partage se fonde sur la notion plus générale de *segment de mémoire*, qui est à la base du mécanisme de gestion des adresses par les processeurs 80x86.

Nous avons vu plus haut (section 2.2.2) que les instructions utilisaient normalement des adresses codées sur 16 bits. Nous savons aussi que le registre IP, qui stocke l'adresse d'une instruction, fait lui aussi 16 bits. Or, avec 16 bits il n'est possible d'adresser que $2^{16} = 64$ Kilo octets.

Le bus d'adresse du 80486 possède 32 bits. Cette adresse de 32 bits est formée par la juxtaposition d'un registre segment (16 bits de poids fort) et d'un déplacement (*offset*, 16 bits de poids faible). Les adresses que nous avons manipulé jusqu'ici sont des déplacements[2]. Le schéma suivant illustre la formation d'une adresse 32 bits à partir du segment et du déplacement sur 16 bits :

$$\underbrace{\overbrace{\boxed{\text{segment}}}^{\text{16 bits}}\overbrace{\boxed{\text{déplacement}}}^{\text{16 bits}}}_{\text{adresse 32 bits}}$$

On appellera *segment de mémoire* une zone mémoire adressable avec une valeur fixée du segment (les 16 bits de poids fort). Un segment a donc une taille maximale de 64 Ko.

3.2.1 Segment de code et de données

La valeur du segment est stockée dans des registres spéciaux de 16 bits. Le registre DS (*Data Segment*) est utilisé pour le segment de données, et le registre CS (*Code Segment*) pour le segment d'instructions.

[2]En réalité, les mécanismes de calculs des adresses sont bien plus complexes que ce que nous décrirons dans ce cours, et dépendent du type de processeur (8086, 286 ou 486), ainsi que du *mode* de travail sélectionné (réel ou protégé). Nous ignorerons ici ces aspects.

Registre CS

Lorsque le processeur lit le code d'une instruction, l'adresse 32 bits est formée à l'aide du registre segment CS et du registre déplacement IP. La paire de ces deux registres est notée `CS:IP`.

Registre DS

Le registre DS est utilisé pour accéder aux données manipulées par le programme. Ainsi, l'instruction

```
MOV AX, [0145]
```

donnera lieu à la lecture du mot mémoire d'adresse `DS:0145H`.

Initialisation des registres segment

Dans ce cours, nous n'écrirons pas de programmes utilisant plus de 64 Ko de code et 64 Ko de données, ce qui nous permetra de n'utiliser qu'un seul segment de chaque type.

Par conséquent, la valeur des registres CS et de DS sera fixée une fois pour toute au début du programme.

Le programmeur en assembleur doit se charger de l'initialisation de DS, c'est à dire de lui affecter l'adresse du segment de données à utiliser.

Par contre, le registre CS sera automatiquement initialisé sur le segment contenant la première instruction au moment du chargement en mémoire du programme (par le chargeur du système d'exploitation).

3.2.2 Déclaration d'un segment en assembleur

Comme nous l'avons vu (voir figure 3.1), les directives `SEGMENT` et `ENDS` permettent de définir les segments de code et de données.

La directive `ASSUME` permet d'indiquer à l'assembleur quel est le segment de données et celui de code, afin qu'il génère des adresses correctes.

Enfin, le programme doit commencer, avant toute référence au segment de données, par initialiser le registre segment DS, de la façon suivante :

```
MOV AX, nom_segment_de_donnees
MOV CS, AX
```

(Il serait plus simple de faire `MOV CS, nom_segment_de_donnees` mais il se trouve que cette instruction n'existe pas.)

La figure 3.2 donne un exemple complet de programme assembleur.

```
; Programme calculant la somme de deux entiers de 16 bits

data        SEGMENT
A           DW  10      ; A = 10
B           DW  1789    ; B = 1789
Result      DW  ?       ; resultat
data        ENDS

code        SEGMENT

            ASSUME DS:data, CS:code

debut:      MOV  AX, data ; etiquette car 1ere instruction
            MOV  DS, AX   ; initialise DS

            ; Le programme:
            MOV  AX, A
            ADD  AX, B
            MOV  result, AX ; range resultat

            ; Retour au DOS:
            MOV  AH, 4CH
            INT  21H
code        ENDS

            END debut ; etiquette de la 1ere inst.
```

FIG. 3.2: Exemple de programme en assembleur. On calcule la somme de deux variables A et B et on range le résultat dans la variable nommée Result.

3.3 Adressage indirect

Nous introduisons ici un nouveau mode d'adressage, l'adressage indirect, qui est très utile par exemple pour traiter des tableaux[3].

L'adressage indirect utilise le registre `BX` pour stocker l'adresse d'une donnée.

En adressage direct, on note l'adresse de la donnée entre crochets :

```
MOV  AX, [130]   ; adressage direct
```

De façon similaire, on notera en adressage indirect :

```
MOV  AX, [BX]   ; adressage direct
```

Ici, BX contient l'adressage de la donnée. L'avantage de cette technique est que l'on peut modifier l'adresse en BX, par exemple pour accéder à la case suivante d'un tableau.

Avant d'utiliser un adressage indirect, il faut charger BX avec l'*adresse* d'une donnée. Pour cela, on utilise une nouvelle directive de l'assembleur, `offset`.

```
data         SEGMENT
truc         DW   1996
data         ENDS

             ...
             MOV  BX, offset truc
             ...
```

Si l'on avait employé la forme

```
             MOV  BX, truc
```

on aurait chargé dans BX la *valeur* stockée en `truc` (ici 1996), et non son adresse[4].

3.3.1 Exemple : parcours d'un tableau

Voici un exemple plus complet utilisant l'adressage indirect. Ce programme passe un chaîne de caractères en majuscules. La fin de la chaîne est repérée par un caractère `$`. On utilise un ET logique pour masquer le bit 5 du caractère et le passer en majuscule (voir le code ASCII).

```
data         SEGMENT
tab          DB 'Un boeuf Bourguignon', '$'
data         ENDS

code         SEGMENT
             ASSUME DS:data, CS:code
```

[3] Il existe encore d'autres modes d'adressage, comme l'adressage indexé, que nous n'aborderons pas dans ce cours.

[4] L'assembleur génère une instruction MOV AX, [*adr*] lorsqu'il rencontre un MOV AX, *etiquette*.

```
debut:        MOV  AX, data
              MOV  DS, AX

              MOV  BX, offset tab ; adresse debut tableau

repet:        MOV  AL, [BX]       ; lis 1 caractere
              AND  AL, 11011111b  ; force bit 5 a zero
              MOV  [BX], AL       ; range le caractere
              INC  BX             ; passe au suivant
              CMP  AL, '$'        ; arrive au $ final ?
              JNE  repet          ; sinon recommencer

              MOV  AH, 4CH
              INT  21H            ; Retour au DOS
code          ENDS
              END debut
```

3.3.2 Spécification de la taille des données

Dans certains cas, l'adressage indirect est ambigu. Par exemple, si l'on écrit

```
MOV [BX],  0      ; range 0 a l'adresse specifiee par BX
```

l'assembleur ne sait pas si l'instruction concerne 1, 2 ou 4 octets consécutifs.

Afin de lever l'ambiguïté, on doit utiliser une directive spécifiant la taille de la donnée à transférer :

```
MOV  byte ptr [BX], val  ; concerne 1 octet
MOV  word ptr [BX], val  ; concerne 1 mot de 2 octets
```

3.4 La pile

3.4.1 Notion de pile

Les *piles* offrent un nouveau moyen d'accéder à des données en mémoire principale, qui est très utilisé pour stocker temporairement des valeurs.

Une pile est une zone de mémoire et un pointeur qui conserve l'adresse du *sommet* de la pile.

3.4.2 Instructions PUSH et POP

Deux nouvelles instructions, PUSH et POP, permettent de manipuler la pile.

PUSH ***registre*** empile le contenu du registre sur la pile.

POP ***registre*** retire la valeur en haut de la pile et la place dans le regsitres spécifié.

Exemple : transfert de AX vers BX en passant par la pile.

```
PUSH AX    ; Pile <- AX
POP  BX    ; BX   <- Pile
```

(Note : cet exemple n'est pas très utile, il vaut mieux employer `MOV AX, BX`.)

La pile est souvent utilisée pour sauvegarder temporairement le contenu des registres :

```
; AX et BX contiennent des donnees a conserver
PUSH AX
PUSH BX

MOV  BX, truc  ; on utilise AX
ADD  AX, BX    ;   et BX
MOV  truc, BX

POP  BX        ; recupere l'ancien BX
POP  AX        ;   et l'ancien AX
```

On voit que la pile peut conserver plusieurs valeurs. La valeur dépilée par POP est la *dernière* valeur empilée; c'est pourquoi on parle ici de pile LIFO (*Last In First Out*, Premier Entré Dernier Sorti).

3.4.3 Registres SS et SP

La pile est stockée dans un segment séparé de la mémoire principale. Le processeur possède deux registres dédiés à la gestion de la pile, SS et SP.

Le registre SS (*Stack Segment*[5]) est un registre segment qui contient l'adresse du segment de pile courant (16 bits de poids fort de l'adresse). Il est normalement initialisé au début du programme et reste fixé par la suite.

Le registre SP (*Stack Pointer*) contient le déplacement du sommet de la pile (16 bits de poids faible de son adresse).

La figure 3.3 donne une représentation schématique de la pile. L'instruction PUSH effectue les opérations suivantes :

- SP $\leftarrow$ SP - 2
- [SP] $\leftarrow$ valeur du registre 16 bits.

Notons qu'au début (pile vide), SP pointe "sous" la pile.

L'instruction POP effectue le travail inverse :

- registre destination $\leftarrow$ [SP]
- SP $\leftarrow$ SP + 2

Si la pile est vide, POP va lire une valeur en dehors de l'espace pile, donc imprévisible.

[5] *stack* = pile en anglais.

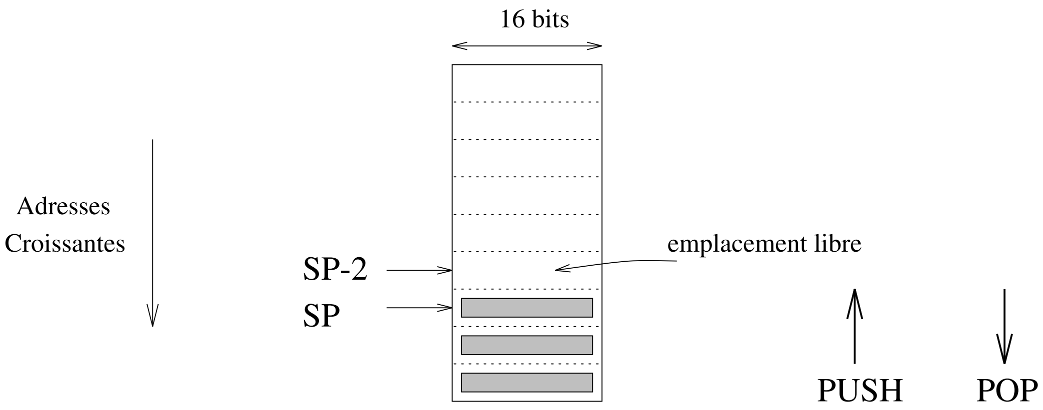


FIG. 3.3: La pile. Les adresses croissent vers le bas. SP pointe sur le sommet (dernier emplacement occupé).

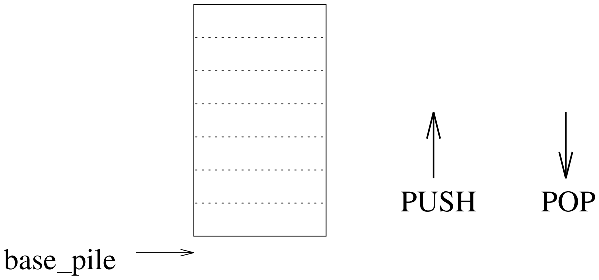


FIG. 3.4: Une pile vide. L'étiquette base-pile repère la base de la pile, valeur initiale de SP.

3.4.4 Déclaration d'une pile

Pour utiliser une pile en assembleur, il faut déclarer un segment de pile, et y réserver un espace suffisant. Ensuite, il est nécessaire d'initialiser les registres SS et SP pour pointer sous le sommet de la pile.

Voici la déclaration d'une pile de 200 octets :

```
seg_pile     SEGMENT  stack   ; mot clef stack car pile
             DW  100 dup (?)  ; reserve espace
base_pile    EQU this word    ; etiquette base de la pile
seg_pile     ENDS
```

Noter le mot clef "stack" après la directive SEGMENT, qui indique à l'assembleur qu'il s'agit d'un segment de pile. Afin d'initialiser SP, il faut repérer l'adresse du bas de la pile; c'est le rôle de la ligne

```
base_pile    EQU this word
```

(voir figure 3.4).

Après les déclarations ci-dessus, on utilisera la séquence d'initialisation :

```
ASSUME SS:seg_pile

MOV  AX, seg_pile
MOV  SS, AX        ; init Stack Segment

MOV  SP, base_pile ; pile vide
```

Noter que le registre SS s'initialise de façon similaire au registre DS; par contre, on peut accéder directement au registre SP.

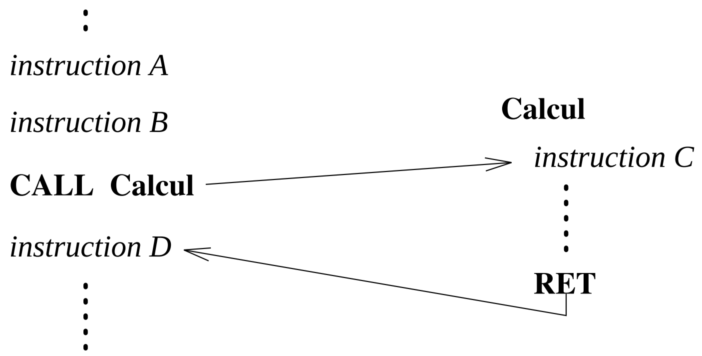


FIG. 3.5: Appel d'une procédure. La procédure est nommée **calcul**. Après l'instruction B, le processeur passe à l'instruction C de la procédure, puis continue jusqu'à rencontrer RET et revient à l'instruction D.

3.5 Procédures

3.5.1 Notion de procédure

La notion de procédure en assembleur correspond à celle de fonction en langage C, ou de sous-programme dans d'autres langages.

Une procédure est une suite d'instructions effectuant une action précise, qui sont regroupées par commodité et pour éviter d'avoir à les écrire à plusieurs reprises dans le programme.

Les procédures sont repérées par l'adresse de leur première instruction, à laquelle on associe une étiquette en assembleur.

L'exécution d'une procédure est déclenchée par un programme *appelant*. Une procédure peut elle-même appeler une autre procédure, et ainsi de suite.

3.5.2 Instructions CALL et RET

L'appel d'une procédure est effectué par l'instruction CALL.

CALL *adresse_debut_procedure*

L'adresse est sur 16 bits, la procédure est donc dans le même segment d'instructions[6]. CALL est une nouvelle instrcution de branchement inconditionnel.

La fin d'une procédure est marquée par l'instruction RET :

RET

RET ne prend pas d'argument; le processeur passe à l'instruction placée immédiatement après le CALL.

RET est aussi une instruction de branchement : le registre IP est modifié pour revenir à la valeur qu'il avait avant l'appel par CALL. Comment le processeur

[6]Il existe aussi des appels de procédure dans des segments différents, que nous n'étudierons pas dans ce cours.

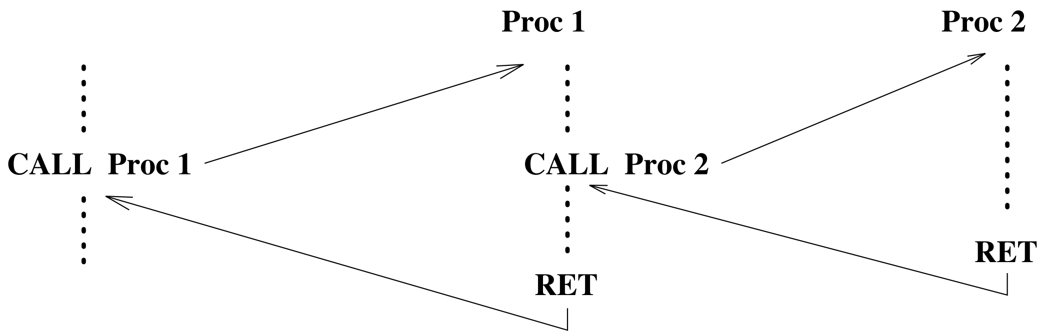


FIG. 3.6: Plusieurs appels de procédures imbriqués.

retrouve-t-il cette valeur ? Le problème est compliqué par le fait que l'on peut avoir un nombre quelconque d'appels imbriqués, comme sur la figure 3.6.

L'*adresse de retour*, utilisée par RET, est en fait sauvegardée sur la pile par l'instruction CALL. Lorsque le processeur exécute l'instruction RET, il dépile l'adresse sur la pile (comme POP), et la range dans IP.

L'instruction CALL effectue donc les opérations :
- Empiler la valeur de IP. A ce moment, IP pointe sur l'instruction qui suit le CALL.
- Placer dans IP l'adresse de la première instruction de la procédure (donnée en argument).

Et l'instruction RET :
- Dépiler une valeur et la ranger dans IP.

3.5.3 Déclaration d'une procédure

L'assembleur possède quelques directives facilitant la déclaration de procédures.

On déclare une procédure dans le segment d'instruction comme suit :

```
Calcul      PROC near        ; procedure nommee Calcul

            ...              ; instructions

            RET              ; derniere instruction
Calcul      ENDP             ; fin de la procedure
```

Le mot clef `PROC` commence la définition d'une procédure, `near` indiquant qu'il s'agit d'une procédure située dans le même segment d'instructions que le programme appelant.

L'appel s'écrit simplement :

```
            CALL  Calcul
```

3.5.4 Passage de paramètres

En général, une procédure effectue un traitement sur des données (*paramètres*) qui sont fournies par le programme appelant, et produit un résultat qui est transmis à ce programme.

Plusieurs stratégies peuvent être employées :

1. *Passage par registre* : les valeurs des paramètres sont contenues dans des registres du processeur. C'est une méthode simple, mais qui ne convient que si le nombre de paramètres est petit (il y a peu de registres).
2. *Passage par la pile* : les valeurs des paramètres sont empilées. La procédure lit la pile.

Exemple avec passage par registre

On va écrire une procédure "SOMME" qui calcule la somme de 2 nombres naturels de 16 bits.

Convenons que les entiers sont passés par les registres AX et BX, et que le résultat sera placé dans le registre AX.

La procédure s'écrit alors très simplement :

```
SOMME       PROC near       ; AX <- AX + BX
            ADD AX, BX
            RET
SOMME       ENDP
```

et son appel, par exemple pour ajouter 6 à la variable Truc :

```
            MOV  AX, 6
            MOV  BX, Truc
            CALL SOMME
            MOV  Truc, AX
```

Exemple avec passage par la pile

Cette technique met en œuvre un nouveau registre, BP (*Base Pointer*), qui permet de lire des valeurs sur la pile sans les dépiler ni modifier SP.

Le registre BP permet un mode d'adressage indirect spécial, de la forme :

```
            MOV  AX, [BP+6]
```

cette instruction charge le contenu du mot mémoire d'adresse BP+6 dans AX.

Ainsi, on lira le sommet de la pile avec :

```
            MOV  BP, SP       ; BP pointe sur le sommet
            MOV  AX, [BP]     ; lit sans depiler
```

et le mot suivant avec :

```
            MOV  AX, [BP+2]   ; 2 car 2 octets par mot de pile.
```

L'appel de la procédure "SOMME2" avec passage par la pile est :

```
PUSH  6
PUSH  Truc
CALL  SOMME2
```

La procédure SOMME2 va lire la pile pour obtenir la valeur des paramètres. Pour cela, il faut bien comprendre quel est le contenu de la pile après le CALL :

SP $\rightarrow$	IP	(adresse de retour)
SP+2 $\rightarrow$	Truc	(premier paramètre)
SP+4 $\rightarrow$	6	(deuxième paramètre)

Le sommet de la pile contient l'adresse de retour (ancienne valeur de IP empilée par CALL). Chaque élément de la pile occupe deux octets.

La procédure SOMME2 s'écrit donc :

```
SOMME2      PROC near        ; AX <- arg1 + arg2
            MOV  BP, SP      ; adresse sommet pile
            MOV  AX, [BP+2]  ; charge argument 1
            ADD  AX, [BP+4]  ; ajoute argument 2
            RET
SOMME2      ENDP
```

La valeur de retour est laissée dans AX.

La solution avec passage par la pile parait plus lourde sur cet exemple simple. Cependant, elle est beaucoup plus souple dans le cas général que le passage par registre. Il est très facile par exemple d'ajouter deux paramètres supplémentaires sur la pile.

Une procédure bien écrite modifie le moins de registres possible. En général, l'accumulateur est utilisé pour transmettre le résultat et est donc modifié. Les autres registres utilisés par la procédure seront normalement sauvegardés sur la pile. Voici une autre version de SOMME2 qui ne modifie pas la valeur contenue par BP avant l'appel :

```
SOMME2      PROC near        ; AX <- arg1 + arg2
            PUSH BP          ; sauvegarde BP
            MOV  BP, SP      ; adresse sommet pile
            MOV  AX, [BP+4]  ; charge argument 1
            ADD  AX, [BP+6]  ; ajoute argument 2
            POP  BP          ; restaure ancien BP
            RET
SOMME2      ENDP
```

Noter que les index des arguments (BP+4 et BP+6) sont modifiés car on a ajouté une valeur au sommet de la pile.

Partie 4

Notions de compilation

Après avoir étudié dans les chapitres précédents le langage machine et l'assembleur, nous abordons ici les langages plus sophistiqués, qui permettent de programmer plus facilement des tâches complexes.

Après une introduction aux notions de langage informatique et de compilation, nous étudierons plus précisement le cas du langage C sur PC.

4.1 Langages informatiques

Un *langage informatique*, par opposition aux langages *naturels* comme le français ou l'anglais, est un langage structuré utilisé pour décrire des actions (ou algorithmes) exécutables par un ordinateur.

La principale différence entre les langages informatiques et les langues naturelles réside dans l'absence d'ambiguité : alors que certaines phrases du français peuvent être interprétées différemment par différents auditeurs, tous seront d'accord pour dire ce que fait un programme donné.

Historiquement, le premier langage informatique a été l'assembleur. Or, la programmation en assembleur est souvent fastidieuse, surtout pour des programmes importants. Plus grave, un programme écrit en assembleur dépend étroitement du type de machine pour lequel il a été écrit. Si l'on désire l'adapter à une autre machine ("porter" le programme), il faut le réécrire entièrement.

C'est pour répondre à ces problèmes qu'ont été développés dès les années 50 des langages de plus haut niveau. Dans ces langages, le programmeur écrit selon des règles strictes mais dispose d'instructions et de structures de données plus expressives qu'en assembleur. Par exemple, dans certains langage comme MATLAB, on pourra écrire en une ligne que l'on désire multiplier deux matrices, alors que le programme correspondant en assembleur prendrait quelques centaines de lignes.

4.1.1 Interpréteurs et compilateurs

On distingue grossièrement deux familles de langages informatique, les langages *interprétés* et les langages *compilés*.

Un programme en langage interprété va être traduit au fur et à mesure de son exécution par un *interpréteur*. Un interpréteur est un programme chargé de décoder chaque instruction du langage et de d'exécuter les actions correspondantes.

Dans le cas de programmes compilés, la traduction en langage machine a lieu une fois pour toute. Le *compilateur* (traducteur) traduit chaque instruction du langage en une suite plus ou moins complexe d'instructions en langage machine. Les programmes compilés s'exécutent ainsi plus rapidement que les programmes interprétés, car la traduction est déja faite. On perd cependant en souplesse de programmation, car les types de données doivent être connus au moment de la compilation.

Un compilateur traduit un programme source écrit dans un langage de haut niveau (par exemple C) en un autre programme dans un langage de bas niveau (par exemple l'assembleur). Cette opération de traduction est assez complexe; les compilateurs sont des programmes sophistiqués, qui ont beaucoup progressé ces dernières années.

4.1.2 Principaux langages

Les principaux langages compilés sont :

C/C++	programmation système et scientifique;
ADA	logiciels embarqués;
Cobol	gestion;
Fortran	calcul scientifique;
Pascal	enseignement.

Quelques langages interprétés :

BASIC	bricolage;
LISP	"Intelligence Artificielle";
Prolog	idem;
Perl	traitement de fichier textes;
Python	programmation système, internet;
Java	"applets" internet;
MATLAB	calcul scientifique;
Mathematica	idem.

Notons que la distinction compilé/interprété est parfois floue. Certains langages, comme LISP, Java ou Python, peuvent subir une première phase de compilation vers un langage intermédiaire (*bytecode*), qui sera lui même interprété.

4.2 Compilation du langage C sur PC

Nous nous intéressons dans cette section à la traduction en assembleur des programmes en langage C sur PC (processeurs de la famille 80x86 que nous avons étudié dans les chapitres précédents).

Le détail de cette traduction (ou compilation) dépend bien entendu du compilateur utilisé et du système d'exploitation (DOS, Windows, UNIX,...). Il dépend aussi de divers réglages modifiables par le programmeur : taille du type `int` (16 ou 32 bits), modèle de mémoire utilisé (pointeurs sur 16 ou 32 bits, données et code dans des segments différents ou non, etc.).

Nous n'aborderons pas ces problèmes dans ce cours (voir la documentation détaillée du compilateur utilisé si besoin), mais nous étudierons quelques exemples de programmes C et leur traduction.

Le compilateur utilisé est Turbo C++ version 3 (en mode ANSI C) sous DOS, avec des entiers de 16 bits et le modèle de mémoire "small". Normalement, ce compilateur génère directement du code objet (fichier `.OBJ`) à partir d'un fichier source en langage C (fichier `.C` ou `.CPP`). Il est cependant possible de demander l'arrêt de la compilation pour obtenir du langage assembleur (fichier `.ASM`). Pour cela, utiliser sous DOS la commande :

```
tcc -S exemple.c
```

Un fichier `exemple.asm` est alors créé.

4.2.1 Traduction d'un programme simple

Considérons le programme en langage C suivant :

```
/* Programme EXEMPLE_1.c en langage C */
void main(void) {
  char X = 11;
  char C = 'A';
  int Res;
  if (X < 0)
    Res = -1;
  else
    Res = 1;
}
```

Trois variables, `X`, `C` et `Res` sont définies avec ou sans valeur initiale. Ensuite, on teste le signe de `X` et range 1 ou -1 dans la vaiable `Res`.

La figure 4.2.1 montre la traduction en assembleur effectuée par Turbo C[1].

Remarquons les points suivants :

1. La fonction `main()` est considérée à ce stade comme une procédure ordinaire (`PROC near`). C'est plus tard, lors de la phase d'édition de lien, qu'il sera indiqué que la fonction `main()` correspond au point d'entrée du programme (=première instruction à exécuter). La fonction est terminée par l'instruction RET.
2. On n'utilise pas ici de segment de données : toutes les variables sont allouées sur la pile.

[1]Nous avons légèrement simplifié le résultat obtenu par "tcc -S" pour le rendre plus lisible.

```
_TEXT  SEGMENT byte public 'CODE'
   ;
   ;    void main(void) {
   ;
       ASSUME    cs:_TEXT
_main  PROC      near
       PUSH      bp
       MOV       bp,sp
       SUB       sp, 4
   ;
   ;             char X = 11;
   ;
       MOV       byte ptr [bp-1], 11
   ;
   ;             char C = 'A';
   ;
       MOV       byte ptr [bp-2], 65
   ;
   ;             int Res;
   ;             if (X < 0)
   ;
       CMP       byte ptr [bp-1], 0
       JGE       @1@86
   ;
   ;                     Res = -1;
   ;
       MOV       word ptr [bp-4], 65535
       JMP       @1@114
@1@86:
   ;
   ;             else
   ;                     Res = 1;
   ;
       MOV       word ptr [bp-4], 1
@1@114:
   ;
   ;   }
   ;
       MOV       sp,bp
       POP       bp
       RET
_main  ENDP
_TEXT  ENDS
       END
```

Fig. 4.1: Traduction de EXEMPLE_1.C effectuée par Turbo C.

3. L'allocation des variables sur la pile s'effectue simplement en soustrayant au pointeur SP le nombre d'octets que l'on va utiliser (ici 4, car 2 variables X et C d'un octet, plus une variable (Res) de 2 octets).

4. La ligne `X = 11` est traduite par

   ```
   MOV byte ptr [bp-1], 11
   ```

   Noter l'utilisation de `byte ptr` pour indiquer que BP contient ici l'adresse d'une donnée de taille octet.

5. Le test `if (X <0)` est traduit par une instruction CMP suivie d'un branchement conditionnel, utilisant une étiquette placée par le compilateur (d'où son nom étrange : `@1@114`).

4.2.2 Fonctions C et procédures

Chaque langage de programmation doit définir une convention de passage des paramètres lors des appels de procédures ou de fonctions. Cette convention permet de prévoir l'état de la pile avant, pendant et après un appel de fonction (dans quel ordre sont empilés les paramètres ? Qui est responsable de leur dépilement ? Comment est passée la valeur de retour ?)

Etudions à partir d'un exemple simple comment sont passés les paramètres lors des appels de fonctions en langage C.

```
/* Programme EXEMPLE_2.C */
int ma_fonction( int x, int y ) {
  return x + y;
}

void main(void) {
  int X = 11;
  int Y = 22;
  int Res;
  Res = ma_fonction(X, Y);
}
```

La traduction en assembleur de ce programme (effectuée par Turbo C) est donnée dans l'encadré suivant.

```
_TEXT   SEGMENT byte public 'CODE'
   ;
   ; int ma_fonction( int x, int y ) {
          ASSUME  cs:_TEXT
_ma_fonction    PROC    near
          PUSH    bp
          MOV     bp,sp
   ;
   ; return x + y;
   ;
          MOV     ax, [bp+4]
          ADD     ax, [bp+6]
   ; }
          POP     bp
          RET
_ma_fonction    ENDP
   ;
   ; void main(void) {
   ;
          ASSUME  cs:_TEXT
_main     PROC    near
          PUSH    bp
          MOV     bp,sp
          SUB     sp,6
   ; int X = 11;
          MOV     [bp-2], 11
   ; int Y = 22;
          MOV     [bp-4], 22
   ;
   ; int Res;
   ; Res = ma_fonction(X, Y);
          PUSH    word ptr [bp-4]
          PUSH    word ptr [bp-2]
          CALL    _ma_fonction
          ADD     sp, 4
          MOV     [bp-6],ax
   ; }
          MOV     sp,bp
          POP     bp
          RET
_main     ENDP
_TEXT     ENDS
```

En étudiant cet exemple, on constate que :

1. la fonction C `ma_fonction()` a été traduite par une procédure assembleur

nommée `_ma_fonction`, qui lit ces arguments sur la pile à l'aide de la technique que nous avons vue en section 3.5.4 (page 51);

2. la fonction ne modifie pas l'état de la pile;
3. Avant l'appel de la fonction (CALL), les arguments sont empilés (PUSH). Après le retour de la fonction, le pointeur SP est incrémenté pour remettre la pile dans son état précédent (`ADD sp, 4` est équivalent à deux instructions POP 2 octets).
4. La valeur retournée par la fonction[2] est passée dans AX (d'où l'instruction `MOV [bp-6], ax`).

Le respect des conventions d'appel de procédures est bien entendu très important si l'on désire mélanger des fonctions C et des procédures en assembleur[3].

4.3 Utilisation d'assembleur dans les programmes C sur PC

Il est possible d'introduire explicitement des instructions assembleur dans des programmes en langage C (ou C++). Evidemment, cette pratique conduit à un programme *non portable*, car le langage assembleur diffère d'un type d'ordinateur à l'autre (on a vu que la portabilité était l'une des raisons conduisant à écrire en langage C). Cependant, lorsque l'on écrit un programme utilisant les ressources matérielles d'une machine donnée (par exemple un PC), il est souvent plus confortable d'écrire un programme C contenant quelques lignes d'assembleur que de tout écrire en assembleur.

La façon de procéder diffère suivant le type de compilateur. Nous ne mentionnerons que l'approche retenue par Turbo C++/TASM[4]

Voici un exemple en langage C :

```
void main(void) {
  int A = 20;
  asm {
    MOV AX, A
    SHL AX, 1
  }
  printf( "AX =%d\n", _AX );
}
```

Ce programme affiche `40`.

Le mot clé asm permet d'introduire des instructions assembleur. Ces instructions peuvent accéder aux variables déclarées en C (ici la variable entière A).

[2]Les fonctions C retournent toujours une seule valeur.

[3]Attention, dans d'autres langages comme Pascal, les conventions sont différentes.

[4]Le logiciel TASM est distribué séparément de Turbo C, et n'est donc pas toujours disponible.

D'autre part, les instructions en C peuvent accéder aux registres du processeur, par l'intermédiaire de "pseudo-variables" `_AX, _BX,_CX`, etc. (nom du registre précédé d'un caractère "souligné".)

Pour plus d'information, on se référera à la documentation de Turbo C (aide en ligne).

Partie 5

Le système d'exploitation

5.1 Notions générales

Le *système d'exploitation* d'un ordinateur est le programme qui permet d'accéder aux ressources matérielles de cet ordinateur. Ces ressources matérielles sont essentiellement les organes d'entrées/sorties : clavier, écran, liaisons réseau, imprimante, disque dur, etc.

Les périphériques d'entrées/sorties varient d'un modèle d'ordinateur à l'autre. Même au sein de la famille des "compatibles PC", on trouve difficilement deux modèles dotés d'exactement les mêmes périphériques (cartes d'extension par exemple). De ce fait, les instructions à exécuter pour piloter tel périphérique (par exemple pour afficher un rectangle rouge à l'écran) diffèrent d'un ordinateur à l'autre.

Le rôle principal du système d'exploitation est d'isoler les programmes des détails du matériel. Un programme désirant afficher un rectangle ne va pas envoyer des instructions à la carte graphique de l'ordinateur, mais plutôt demander au système d'exploitation de le faire. C'est le système d'exploitation qui doit connaitre les détails du matériel (dans ce cas le type de carte graphique et les instructions qu'elle comprend). Cette répartition des rôles simplifie grandement l'écriture des programmes d'application[1]

Le système d'exploitation est donc un programme complexe, lié à la configuration matérielle de la machine. Nous étudierons en deuxième année les principes de fonctionnement des systèmes d'exploitation. Notons simplement que tout système d'exploitation est divisé en plusieurs couches. La couche basse est responsable de la gestion du matériel, et change par exemple suivant le type de périphérique installé. Les couches plus hautes sont chargées de fonctions plus évoluées (gestion des fichiers sur disque par exemple), plus ou moins indépendantes du matériel.

[1] On appelle *programme d'application*, ou simplement *application*, un programme qui effectue des traitements directement utile pour l'utilisateur de l'ordinateur (traitement de texte, base de données, etc.), par opposition aux programmes qui ne sont pas directement visibles par l'utilisateur (comme le système d'exploitation ou les divers utilitaires gérant l'ordinateur).

Les systèmes d'exploitation les plus répandus sont les suivants :

Système	Type de machine	Caractéristiques
DOS	PC	simple, répandu, peu puissant.
Windows	PC	interface graphique, très répandu, peu fiable.
Window NT	PC, qq stations	multi-tâche.
VMS	Vax	multi-tâche, fiable, ancien.
UNIX	Tous	multi-tâche, fiable, flexible.

Dans ce cours, nous nous concentrerons sur les ordinateurs PC fonctionnant avec le système DOS. C'est un système très peu puissant, qui n'offre que le strict minimum de fonctionnalités. De ce fait, il est relativement simple.

5.2 Présentation du BIOS

Le BIOS (*Basic Input Output System*, système d'entrées/sorties de base) constitue la couche basse de tous les systèmes d'exploitations sur PC. Il est donc responsable de la gestion du matériel : clavier, écran, disques durs, liaisons séries et parallèles.

Le programme du BIOS se trouve en mémoire morte (ROM), c'est à dire dans une mémoire gardant son contenu lorsque l'on interrompt son alimentation électrique[2].

Chaque modèle de PC est vendu avec une version du BIOS adapté à sa configuration matérielle.

5.2.1 Les fonctions du BIOS

Du point de vue de l'utilisation, on peut considérer le BIOS comme une librairie de fonctions. Chaque fonction effectue une tâche bien précise, comme par exemple afficher un caractère donné sur l'écran. L'appel de l'une de ces fonctions constitue un *appel système*.

On pourrait envisager que les fonctions du BIOS soient simplement des procédures, que l'on appellerait avec l'instruction `CALL` en passant les paramètres nécessaires. Ce n'est pas le cas, le mécanisme d'appel est différent.

En effet, il a été prévu de pouvoir modifier le comportement du BIOS en cours d'utilisation, par exemple pour gérer un nouveau périphérique ou pour modifier la gestion d'un périphérique existant. Le code du BIOS étant en mémoire morte, il n'est pas modifiable. De plus, le BIOS étant différent d'un ordinateur à l'autre, les adresses des fonctions changent...

Prenons un exemple : soit la "fonction" du BIOS affichant un caractère (donné par son code ASCII) sur l'écran. Supposons que sur notre PC, la première instruction de cette "fonction" soit à l'adresse F1234560H. Sur le modèle d'une autre

[2]En fait, le BIOS est souvent en mémoire EEPROM ou FLASH, afin de pouvoir le remplacer plus facilement.

marque de notre voisin, cette même fonction pourrait être implantée à l'adresse F1234550H.

5.2.2 Vecteurs d'interruptions

Le problème est résolu par l'utilisation d'une table d'indirection, la table des *vecteurs d'interruptions*[3].

Cette table est placée en mémoire principale (RAM), et contient les adresses (en ROM ou en RAM) des fonctions du BIOS. Elle est implantée à partie de l'adresse 00000000H (première case mémoire) et est initialisé par le BIOS lui même au moment du démarrage du PC (boot).

Adresse	contenu
0000	adresse de la première fonction du BIOS
0004	adresse de la deuxième fonction du BIOS
...	...

Chaque élément de la table occupe 4 octets (adresse 32 bits). La table a 256 éléments (et occupe donc 1Ko).

Dans l'exemple évoqué plus haut, si l'on sait que la fonction du BIOS qui affiche un caractère est la 33ième, on va l'appeler en lisant la 33ième ligne de la table, puis en allant exécuter les instructions à l'adresse trouvée. Sur notre PC, la table contiendrait :

Adresse	contenu	
...	...	
0084H	F1234560H	(car 4x33 = 84H).
...	...	

La table des vecteurs d'interruptions contient des valeurs différentes pour chaque version de BIOS, et peut être modifiée pour pointer sur du code en mémoire principale, modifiant alors le BIOS existant.

5.2.3 Appel système : instruction INT n

L'instruction `INT n` permet d'appeler la n-ième fonction de la table des vecteurs d'interruptions.

n est un entier compris entre 0 et 255 (1 octet), car il y a 256 vecteurs d'interruptions dans la table.

L'instruction `INT n` est semblable à l'instruction CALL, sauf que l'adresse de destination est donnée par la table des vecteurs d'interruptions, et que les indicateurs sont automatiquement sauvegardés sur la pile. De plus, l'adresse de retour complète (32 bits) est empilée, car le traitant d'interruption n'est pas nécessairement dans le même segment de code que le programme appelant.

[3] On emploie abusivement le terme d'interruption car le même mécanisme est utilisé pour les interruptions matérielles que nous étudierons plus loin.

Le déroulement de `INT n` se passe comme suit :

1. sauvegarde les indicateurs du registre d'état sur la pile (les indicateurs sont regroupés dans un mot de 16 bits);
2. sauvegarde CS et IP sur la pile;
3. CS et IP sont chargés avec la valeur lue à l'adresse $4n$, n étant le paramètre de INT. L'exécution continue donc au début du traitant d'interruption.

5.2.4 Traitants d'interruptions

Un *traitant d'interruption* est une routine[4] appelée via la table des vecteurs d'interruption par l'instruction `INT n`.

Les traitants d'interruptions sont très similaires à des procédures ordinaires, sauf qu'ils doivent se terminer par l'instruction `IRET` au lieu de `RET`.

l'instruction `IRET` est très similaire à `RET`, sauf que CS et IP sont dépilés, et que tous les indicateurs sont restaurés à leur anciennes valeurs, sauvegardées sur la pile par `INT n`.

Notons que les éventuels paramètres du traitant d'interruption sont toujours passés par registre.

5.2.5 Quelques fonctions du BIOS

INT	**Fonction**	
0	Division par 0	appelé automatiquement lors de div. par 0
5	Copie d'écran	
10H	Ecran	gestion des modes vidéo
12H	Taille mémoire	
13H	Gestion disque dur	(initialiser, lire/écrire secteurs)
14H	Interface série	
16H	Clavier	(lire caractère, état du clavier)

5.3 Présentation du DOS

Le système DOS (*Disk Operating System*, système d'exploitation de disque) repose sur le BIOS, dont il appelle les fonctions pour interagir avec le matériel.

Les fonctions du DOS s'utilisent comme celles du BIOS, via des vecteurs d'interruptions. Elles offrent des fonctionnalités de plus haut niveau que le BIOS (entrées/sorties, ouverture de fichiers sur disque, etc.).

Les fonctions du DOS s'appellent toutes à l'aide du vecteur 21H. La valeur du registre AH permet d'indiquer quelle est la fonction que l'on appelle :

```
MOV AH, numero_fonction
INT 21H
```

Nous avons déja mentionné la fonction 4CH du DOS, qui permet de terminer un programme et de revenir à l'interpréteur de commandes DOS :

[4]les mots "routines", "fonctions" et "procédures" sont synonymes dans ce cours.

```
MOV AH, 4CH
INT 21H
```

5.3.1 Description de quelques fonctions du DOS

Voici à titre d'exemple quelques fonctions du DOS :

Numéro	Fonction	
01H	Lecture caractère	met le code ascii lu dans AL
02H	Affiche caractère	code ascii dans registre DL
09H	Affiche chaîne de caractères	DX=adresse début chaîne, terminée par '$'
0BH	Lit état clavier	met AL=1 si caractère, 0 sinon.

Ce programme lit un caractère au clavier et l'affiche en majuscule :

```
MOV  AH, 01H       ; code fonction DOS
INT  21H           ; attente et lecture d'un caract\'ere
AND  AL, 11011111b ; passe en majuscule
MOV  DL, AL        ;
MOV  AH, 02H       ; code fonction affichage
INT  21H           ; affiche le caractere
```

Dans la suite du cours, nous aurons l'occasion de décrire d'autres fonctions du BIOS et du DOS.

5.4 Modification d'un vecteur d'interruption en langage C

Nous allons maintenant voir comment l'on peut modifier la table des vecteurs d'interruptions.

On modifie un vecteur d'interruption pour installer un *traitant d'interruption*, "fonction" appelée par l'instruction `INT n`. L'installation de traitants permet de modifier le comportement du système d'exploitation, ou de communiquer avec des périphériques d'entrées sorties, comme nous l'étudierons dans le chapitre suivant.

En général, le système a installé un traitant pour chaque vecteur d'interruption. L'installation d'un nouveau traitant doit donc se faire avec précautions : on va sauvegarder l'ancienne adresse, de façon à pouvoir remettre le vecteur dans son état initial à la fin de notre programme.

Les vecteurs d'interruptions peuvent être modifiés en assembleur ou en langage C. Nous travaillerons ici en langage C : le principe est le même qu'en assembleur, mais les programmes sont plus intelligibles.

5.4.1 Ecriture d'un traitant d'interruption en C

Nous avons vu que le compilateur générait pour chaque fonction C un procédure en assembleur, terminée par l'instruction `RET`.

Un traitant d'interruption est similaire à une procédure, mais terminée par `IRET`. En Turbo C sur PC, on peut signaler au compilateur qu'une fonction est un traitant d'interruption grâce au mot clé `interrupt`. La déclaration

```
void interrupt un_traitant();
```

indique que la fonction nommée "`un_traitant`" est un traitant d'interruption. Les traitants ne retournent pas de valeur et ne prennent pas d'arguments.

5.4.2 Installation d'un traitant

En Turbo C, on dispose de deux fonctions qui permettent de manipuler facilement la table des vecteurs d'interruption : `setvect()` et `getvect()`[5].

`setvect()` modifie un vecteur, et `getvect()` lis la valeur d'un vecteur. La valeur d'un vecteur d'interruption est l'adresse de la fonction traitante. Une variable `TRAITANT` de ce type se déclare comme suit :

```
void interrupt (*TRAITANT) ();
```

(littéralement : "`old_handler` est un pointeur sur une fonction de type traitant d'interruption").

Exemple

```
#include <dos.h>

void interrupt ( *ancien_traitant)(...);

void interrupt un_traitant() {
   /* code C de notre traitant
           ....
    */

   /* Appelle le traitant qui etait installe */
   ancien_traitant();
}

void main(void) {
   /* Sauve l'ancien vecteur 1BH */
   ancien_traitant = getvect( 0x1B );

   /* Installe nouveau traitant pour INT 1BH */
   setvect( 0x1B, un_traitant );

   /* ... programme ... */
```

[5]Ces fonctions sont déclarées dans DOS.H.

```
   /* Restaure ancien vecteur */
   setvect( 0x1B, ancien_traitant );
}
```

Partie 6

Les interruptions

Nous étudions dans ce chapitre les interruptions *matérielles* (ou externes), c'est à dire déclenchées par le matériel (hardware) extérieur àu processeur. Nous nous appuyons ici aussi sur l'exemple du PC.

6.1 Présentation

Les interruptions permettent au matériel de communiquer avec le processeur.

Les échanges entre le processeur et l'extérieur que nous avons étudiés jusqu'ici se faisait toujours à l'initiative du processeur : par exemple, le processeur demande à lire ou à écrire une case mémoire.

Dans certains cas, on désire que le processeur réagisse rapidement à un évènement extérieur : arrivé d'un paquet de données sur une connexion réseau, frappe d'un caractère au clavier, modification de l'heure[1]. Les interruptions sont surtout utilisées pour la gestion des périphériques de l'ordinateurs.

Une interruption est signalée au processeur par un signal éléctrique sur une borne spéciale. Lors de la réception de ce signal, le processeur "traite" l'interruption dès la fin de l'instruction qu'il était en train d'exécuter[2]. Le traitement de l'interruption consiste soit :

- à l'ignorer et passer normalement à l'instruction suivante : c'est possible uniquement pour certaines interruptions, nommées *interruptions masquables*. Il est en effet parfois nécessaire de pouvoir ignorer les interruptions pendant un certain temps, pour effectuer des traitements très urgents par exemple. Lorsque le traitement est terminé, le processeur *démasque* les interruptions et les prend alors en compte.
- à exécuter un *traitant d'interruption* (interrupt handler). Un traitant d'interruption est un programme qui est appelé automatiquement lorsqu'une interruption survient. L'adresse de début du traitant est donnée par la table

[1] L'heure change en permanence... nous verrons que le circuit d'horloge, extérieur au processeur, envoie un signal d'interruption à intervalles réguliers (quelques ms).

[2] Le processeur ne peut pas réagir plus vite; imaginez les conséquences d'une instruction abandonnée à la moitié de son exécution...

des *vecteurs d'interruptions*, que nous avons déja rencontré dans le chapitre précédent. Lorsque le traitant à effectuer son travail, il exécute l'instruction spéciale `IRET` qui permet de reprendre l'exécution à l'endroit où elle avait été interrompue.

6.2 Interruption matérielle sur PC

6.2.1 Signaux d'interruption

Les processeurs de la famille 80x86 possèdent trois bornes pour gérer les interruptions : NMI, INTR, et $\overline{\text{INTA}}$ (voir figure 6.1).

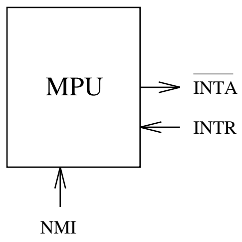


FIG. 6.1: Bornes d'interruptions.

NMI est utilisée pour envoyer au processeur une interruption non masquable (NMI, Non Maskable Interrupt). Le processeur ne peut pas ignorer ce signal, et va exécuter le traitant donné par le vecteur 02H. Ce signal est normalement utilisé pour détecter des erreurs matérielles (mémoire principale défaillante par exemple).

INTR (Interrupt Request), demande d'interruption masquable. Utilisée pour indiquer au MPU l'arrivée d'une interruption.

INTA (Interrupt Acknowledge) Cette borne est mise à 0 lorsque le processeur traite effectivement l'interruption signalée par INTR (c'est à dire qu'elle n'est plus masquée)[3].

6.2.2 Indicateur IF

A un instant donné, les interruptions sont soit masquées soit autorisées, suivant l'état d'un indicateur spécial du registre d'état, IF (Interrupt Flag).

- si IF = 1, le processeur accepte les demandes d'interruptions masquables, c'est à dire qu'il les traite immédiatement;
- si IF = 0, le processeur ignore ces interruptions.

L'état de l'indicateur IF peut être modifié à l'aide de deux instructions, CLI (*CLear IF*, mettre IF à 0), et STI (*SeT IF*, mettre IF à 1).

[3] On note $\overline{\text{INTA}}$, pour indiquer que l'état normal de cette borne est 1 et non 0 (inversé).

6.2.3 Contrôleur d'interruptions

L'ordinateur est relié a plusieurs périphériques, mais nous venons de voir qu'il n'y avait qu'un seul signal de demande d'interruption, INTR.

Le *contrôleur d'interruptions* est un circuit spécial, extérieur au processeur, dont le rôle est de distribuer et de mettre en attente les demandes d'interruptions provenant des différents périphériques.

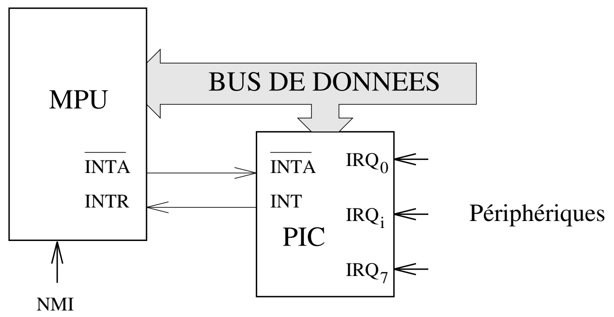


FIG. 6.2: Le contrôleur d'interruptions (PIC, pour *Programmable Interruption Controler*).

La figure 6.2 indique les connexions entre le MPU et le contrôleur d'interruptions.

Le contrôleur est rélié aux interfaces gérant les périphériques par les bornes IRQ (*InteRrupt reQuest*). Il gère les demandes d'interruption envoyées par les périphériques, de façon à les envoyer une par une au processeur (via INTR). Il est possible de programmer le contrôleur pour affecter des priorités différentes à chaque périphérique, mais nous n'aborderons pas ce point dans ce cours.

Avant d'envoyer l'interruption suivante, le contrôleur attend d'avoir reçu le signal $\overline{\text{INTA}}$, indiquant que le processeur a bien traité l'interruption en cours.

6.2.4 Déroulement d'une interruption externe masquable

Reprenons les différents évènements liés à la réception d'une interruption masquable :

1. Un signal INT est émis par un périphérique (ou plutôt par l'interface gérant celui-ci).
2. Le contrôleur d'interruptions reçoit ce signal sur une de ses bornes IRQ_i. Dès que cela est possible (suivant les autres interruptions en attente de traitement), le contrôleur envoie un signal sur sa borne INT.
3. Le MPU prend en compte le signal sur sa borne INTR après avoir achevé l'exécution de l'instruction en cours (ce qui peut prendre quelques cycles d'horloge). Si l'indicateur IF=0, le signal est ignoré, sinon, la demande d'interruption est acceptée.

4. Si la demande est acceptée, le MPU met sa sortie $\overline{\text{INTA}}$ au niveau 0 pendant 2 cycles d'horloge, pour indiquer au contrôleur qu'il prend en compte sa demande.

5. En réponse, le contrôleur d'interruption place le numéro de l'interruption associé à la borne IRQ_i sur le bus de données.

6. Le processeur lit le numéro de l'interruption sur le bus de données et l'utilise pour trouver le vecteur d'interruption. Ensuite, tout se passe comme pour un appel système (interruption logicielle, voir page 63), c'est à dire que le processeur :

   (a) sauvegarde les indicateurs du registre d'état sur la pile;

   (b) met l'indicateur IF à 0 (masque les interruptions suivantes);

   (c) sauvegarde CS et IP sur la pile;

   (d) cherche dans la table des vecteurs d'interruptions l'adresse du traitant d'interruption, qu'il charge dans `CS:IP`.

7. La procédure traitant l'interruption se déroule. Pendant ce temps, les interruptions sont masquées (IF=0). Si le traitement est long, on peut dans certains cas ré-autoriser les interruptions avec l'instruction `STI`.

8. La procédure se termine par l'instruction `IRET`, qui restaure CS, IP et les indicateurs à partir de la pile, ce qui permet de reprendre le programme qui avait été interrompu.

6.3 Exemple : gestion de l'heure sur PC

L'horloge d'un PC peut être considéré comme un "périphérique" d'un type particulier. Il s'agit d'un circuit électronique cadencé par un oscillateur à quartz (comme une montre ordinaire), qui est utilisé entre autre pour gérer l'heure et la date, que de nombreux programmes utilisent.

L'horloge envoie une interruption matérielle au processeur toutes 0,055 secondes (soit 18,2 fois par secondes). Le vecteur correspondant est le numero 08H.

Pour gérer l'heure, le BIOS installe un traitant pour l'interruption 08H. Ce traitant incrémente simplement un compteur, nombre entier codé sur 32 bits et toujours rangé à l'adresse `0040:006C` en mémoire principale.

Ainsi, si un programme désire connaitre l'heure, il lui suffit de lire cet emplacement mémoire, qui change "automatiquement" 18,2 fois par secondes. Une simple division permet alors de convertir ce nombre en heures et minutes.

Remarques :

1. Les programmes usuels utilisent des appels systèmes du DOS plus pratiques, qui se basent sur la valeur de la mémoire `0040:006C` et effectuent les conversions nécessaires.

En langage C, on pourra utiliser la fonction `time()` qui appelle elle même le DOS.

2. En modifiant le vecteur d'interruption 08H, on peut faire en sorte que le PC exécute n'importe quelle tâche de façon régulière. En pratique, il est déconseillé de modifier directement le vecteur 08H. Le traitant d'horloge standard du système BIOS appelle une autre interruption (logicielle), qui est prévue pour être déroutée par les utilisateurs.

6.4 Entrées/Sorties par interruption

En général, les périphériques qui *recoivent* des données de l'extérieur mettent en œuvre un mécanisme d'interruption : clavier, liaisons séries (modem, souris...) et parallèles (imprimantes), interfaces réseau, contrôleurs de disques durs et CD-ROMS, etc.

Nous étudierons dans le chapitre suivant le cas de la liaison série.

6.4.1 Un exemple

Etudions ici très schématiquement le cas d'une lecture sur disque dur, afin de comprendre comment l'utilisation d'une interruption permet de construire un système d'exploitation plus efficace.

Soit un programme lisant des données sur un disque dur, les traitant et les affichant sur l'écran.

Voici l'algorithme général sans utiliser d'interruption :

– Répéter :

1. envoyer au contrôleur de disque une demande de lecture d'un bloc de données.
2. attendre tant que le disque ne répond pas (*scrutation*);
3. traiter les données;
4. afficher les résultats.

Cette méthode simple est appelée entrée/sortie par *scrutation*. L'étape 2 est une boucle de scrutation, de la forme :

– Répéter:
  – regarder si le transfert du disque est terminé;
– Tant qu'il n'est pas terminé.

La scrutation est simple mais inefficace : l'ordinateur passe la majorité de son temps à attendre que les données soit transférées depuis le disque dur. Pendant ce temps, il répète la boucle de scrutation.

Ce temps pourrait être mis à profit pour réaliser une autre tâche. Très grossièrement, les entrées/sorties par interruption fonctionnent sur le modèle suivant :

1. Installer un traitant d'interruption disque qui traite les données reçues et les affiche;

2. envoyer au contrôleur de disque une demande de lecture des données;
3. faire autre chose (un autre calcul ou affichage par exemple).

Dans ce cas, dès que des données arrivent, le contrôleur de disque envoie une interruption (via le contrôleur d'interrruptions) au processeur, qui arrète temporairement le traitement 3 pour s'occuper des données qui arrivent. Lorsque les données sont traitées, le traitement 3 reprend (IRET). Pendant l'opération (lente) de lecture du disque dur, le processeur peut faire autre chose (par exemple jouer aux échecs !).

Dans la pratique, les choses sont un peu plus compliquées : il faut avoir plusieurs tâches à faire en même temps pour que l'utilisation des interruptions permettent un gain intéressant. Ce principe est surtout mis à profit dans les systèmes multi-tâches comme UNIX ou Windows NT, que nous étudierons en deuxième année.

Partie 7

Les entrées/sorties

Les ordinateurs sont utilisés pour traiter et stocker des informations. Nous avons jusqu'ici décrit le fonctionnement du processeur et la mémoire principale. Nous allons maintenant étudier comment un ordinateur peut échanger de l'information avec son environnement; ces échanges d'informations sont nommés *entrées/sorties* (ou IO, *Input/Output* en anglais).

Il peut s'agir d'un flux d'informations de l'extérieur vers l'ordinateur (acquisition via le clavier, une connexion réseau, un disque dur, etc.), ou d'un flux de l'ordinateur vers l'extérieur (écran, réseau, disque, etc.).

Les techniques d'entrées/sorties sont très importantes pour les performances de l'ordinateur. Rien ne sert d'avoir un un processeur calculant très rapidement s'il doit souvent perdre son temps pour lire des données ou écrire ses résultats.

Durant une opération d'entrée/sortie, de l'information est échangée entre la mémoire principale et un *périphérique* relié à l'ordinateur. Nous étudierons plus loin dans ce cours le fonctionnement de certains périphériques (disques durs, clavier, écran). Cet échange nécessite une *interface* ou *contrôleur*, circuit électronique gérant la connexion. L'interface réalise des fonctions plus ou moins complexes suivant le type de périphérique. Son but est surtout de décharger le processeur pour améliorer les performances du système.

A la fin de cette partie, nous décrivons pour illustrer les concepts présentés le circuit d'interface série asynchrone 8250, qui équipe tous les PC.

7.1 Les bus du PC

Nous avons dans les chapitres précédents décrit de façon simplifiée les bus reliant le processeur à la mémoire principale. Nous avons distingué le bus d'adresse, le bus de données et le bus de commandes (signaux de commandes type R/W).

En fait, la plupart des échanges d'informations dans l'ordinateur se font sur des bus : connexions processeur/mémoire, mais aussi connexions entre le processeur et les interfaces d'entrées sorties. Il existe une grande variété de bus; chacun est caractérisé par sa largeur (nombre de bits) et sa fréquence (nombre de cycles par secondes, en Méga-Hertz).

7.1.1 Bus local

Le bus local est le bus le plus rapide, sur lequel sont directement connectés le processeur et la mémoire principale. Il regroupe un bus de données un bus d'adresse et de signaux de commandes (voir le chapitre 1).

Le bus local est aussi relié aux contrôleurs des bus d'extensions, et parfois à des contrôleurs de mémoire cache (voir le chapitre 9).

7.1.2 Bus d'extension du PC

Les *bus d'extensions* (ou bus d'entrés/sorties) permettent de connecter au PC des contrôleurs d'extensions (cartes) grâce à des connecteurs spéciaux (slots sur la carte mère).

Les contrôleurs d'extensions sont utilisés pour relier le PC aux périphériques d'entrées/sorties.

Depuis l'apparition du PC au début des annés 80, plusieurs standards de bus d'extension ont étés proposés : ISA, MCA, EISA...

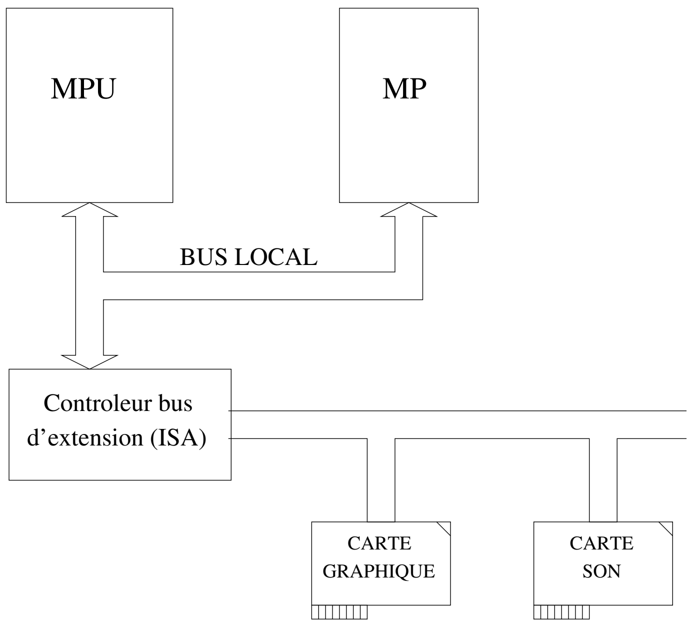


FIG. 7.1: Bus local et bus d'extension type ISA.

Le bus ISA

Le bus d'extension ISA (*Industry Standard Architecture*) est le plus répandu sur PC. De fréquence relativement basse et de caractéristiques peu puissantes, il est utilisé pour connecter des cartes relativement lentes (modems, cartes sons, ...).

Les principales caractéristiques du bus ISA (PC-AT) sont : 16 bits de données, 24 bits d'adresse, 16 lignes d'interruption, fréquence 8 MHz.

7.1.3 Bus local PCI

Les périphériques d'entrées/sorties "modernes" demandent des transferts d'information très importants entre la mémoire principale (MP) et le contrôleur. Par exemple, une carte graphique SVGA récente possède une mémoire vidéo de 1 à 8 Mo, et met en œuvre des transferts entre cette mémoire et la MP à 60 Mo/s.

Pour permettre de tels débits, il est nécessaire de connecter le contrôleur de périphérique directement sur le bus local. Le contrôleur bénéficie ainsi du haut débit de ce bus; de plus, il peut en prendre le contrôle pour effectuer des transferts directement avec la MP sans passer par le processeur.

Le premier bus PC basé sur ces principes a été le bus VLB (VESA Local Bus), qui est actuellement remplacé par le bus PCI (Peripheral Component Interface).

Le bus PCI équipe la grande majorité des PC récents. Notons qu'il n'est pas réservé au processeurs INTEL, puisqu'il est aussi utilisé sur les Macintosh à base de processeurs PowerPC. Le principe du bus PCI est justement de dissocier le processeur et les bus. Cette séparation permet d'utiliser une fréquence de bus différente de celle du processeur et facilite l'évolution des machines.

Les caractéristiques du bus PCI sont : 32 ou 64 bits de données, 32 bits d'adresse, fréquence de 33 MHz. Il permet de débits de 132 Mo/s en 32 bits, ou 264 Mo/s en 64 bits.

La figure 7.2 représente l'architecture d'un PC avec bus PCI.

Le contrôleur PCI est la plupart du temps intégré sur la carte mère (il s'agit d'un circuit intégré complexe dont les performances sont cruciales pour celles du PC).

Les connecteurs (slot) PCI sont réservés aux périphériques demandants de hauts débits : cartes vidéo, contrôleurs SCSI, cartes réseaux haut débit.

7.2 Bus de périphériques

Ces bus permettent de relier une interface (contrôleur) de l'ordinateur à un ou plusieurs périphériques (généralement à l'*extérieur* de l'ordinateur).

7.2.1 Bus SCSI

Le bus SCSI (*Small Computer System Interface*) est un bus d'entrées/sorties parallèles qui n'est pas limité aux ordinateurs PC, ni même aux micro-ordinateurs.

Il permet de connecter de 1 à 7 périphériques de toutes natures (Disques durs, lecteurs CD/ROM, digitaliseurs (scanners), lecteurs de bandes (streamers), ...).

La version SCSI 1 permet un taux de transfert de 4 Mo/s (largeur 8 bits). La version SCSI 2 permet d'obtenir jusqu'a 40 Mo/s en 32 bits.

Le bus SCSI équipe en standard tous les ordinateurs Apple Macintosh, et la grande majorité des stations de travail. Sur PC, il faut installer une carte d'interface, connectée soit au bus ISA soit au bus PCI suivant les performances désirées.

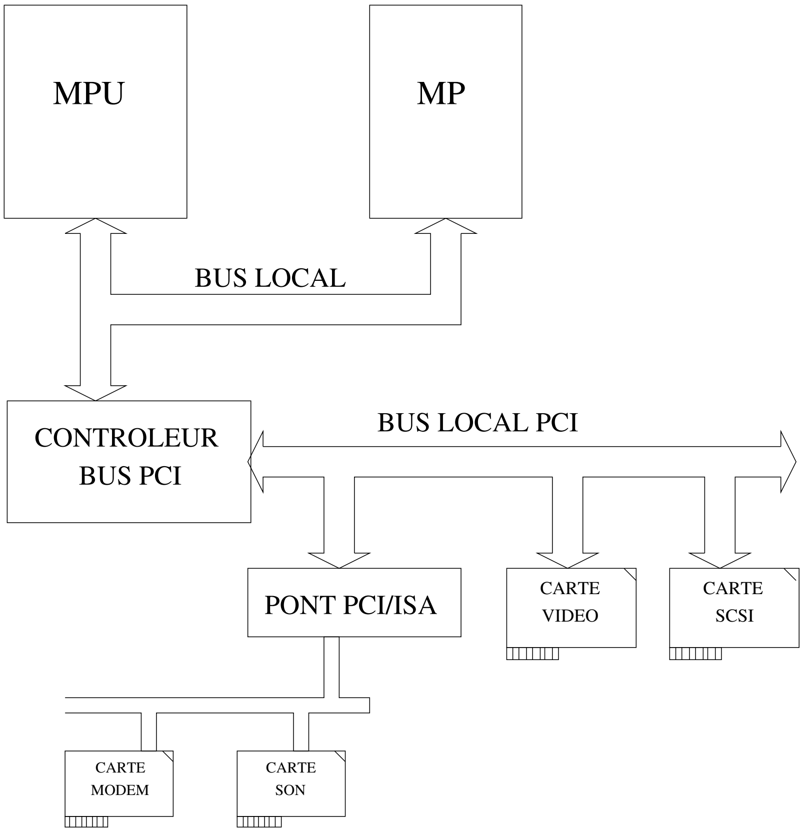


FIG. 7.2: PC avec bus PCI.

7.2.2 Bus PCMCIA

Le bus PCMCIA (*Personal Computer Memory Card International Association*) est un bus d'extension utilisé sur les ordinateurs portables. Il permet la connexion de périphériques de taille très réduite (format carte bancaire, 3 à 10 mm d'épaisseur, connecteur 68 broches).

7.3 Les entrées/sorties sur PC

7.3.1 Généralités

Les données échangées entre un périphérique et le processeur transitent par l'interface (ou contrôleur) associé à ce périphérique.

L'interface possède de la mémoire tampon pour stocker les données échangées (suivant le type d'interface, cette mémoire tampon fait de 1 seul octet à quelques méga-octets).

L'interface stocke aussi des informations pour gérer la communication avec le périphérique :

– des *informations de commande*, pour définir le mode de fonctionnement de l'interface : sens de transfert (entrée ou sortie), mode de transfert des données (par scrutation ou interruption), etc. Ces informations de commandes sont communiquées à l'interface lors de la phase d'*initialisation* de celle-ci, avant le début du transfert.
– des *informations d'état*, qui mémorisent la manière dont le transfert s'est effectué (erreur de transmission, réception d'informations, etc). Ces informations sont destinées au processeur.

On accède aux données de chaque interface par le bias d'un espace d'adresses *d'entrées/sorties*, auquel on accède par les instructions IN et OUT du 80x86.

IN AL, *adresse E/S* lit l'octet d'adresse spécifiée dans l'espace d'entrées/sorties et le transfère dans le registre AL.

OUT *adresse E/S*, AL écrit le contenu de AL à l'adresse spécifiée de l'espace d'entrées/sorties.

Lors de l'exécution des instructions IN et OUT, le processeur met à 1 sa borne IO/M et présente l'adresse E/S sur le bus d'adresse. Le signal IO/M indique aux circuits de décodage d'adresses qu'il ne s'agit pas d'une adresse en mémoire principale, mais de l'adresse d'une interface d'entrées/sorties.

7.3.2 Modes de transfert

Le transfert des données entre le processeur et l'interface peut s'effectuer de différentes manières.

On distingue les transferts *sans condition* et les transferts *avec condition* au périphérique. Les transferts sans condition sont les plus simples; ils concernent les périphériques très simples (interrupteurs, voyants lumineux, ...) qui n'ont pas de registre d'état et sont toujours prêts.

Les transferts *avec condition* sont plus complexes : avant d'envoyer ou de recevoir des informations, le processeur doit connaitre l'état du périphérique (par exemple, en réception sur une liaison réseau, on doit savoir si un octet est arrivé avant de demander la lecture de cet octet).

On distingue ici deux méthodes, les transferts par *scrutation* et les transferts par *interruption*, que nous avons déja étudié en section 6.4 (page 73).

7.4 L'interface d'entrées/sorties séries asynchrones

L'interface entrées/sorties séries équippe tous les PC et permet l'échange d'informations à faible débit avec un périphérique comme un modem, ou avec un autre PC, sur des distances inférieures à quelques dizaines de mètres.

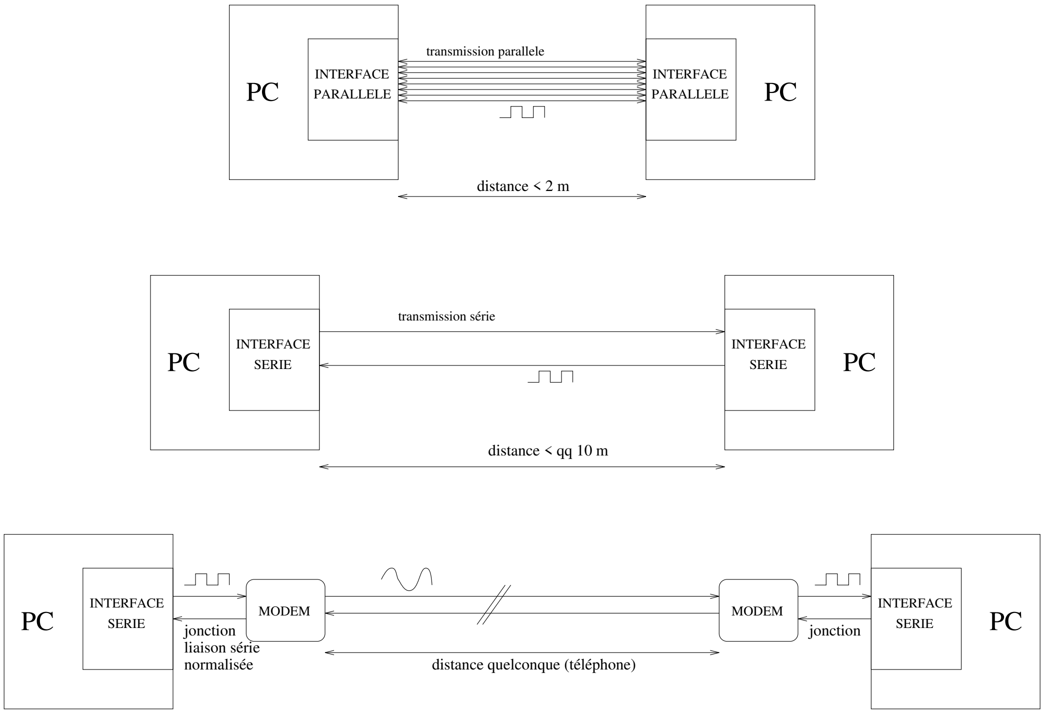


FIG. 7.3: Différents types de transmissions pour relier simplement deux PC.

7.4.1 Pourquoi une transmission série ?

Sur des distances supérieures à quelques mètres, il est difficile de mettre en œuvre une transmission en parallèle : coût du cablage, mais surtout interférences électromagnétiques entre les fils provoquant des erreurs importantes. On utilise alors une liaison série, avec un seul fil portant l'information dans chaque sens.

Sur des distance supérieures à quelques dizaines de mètres, on utilisera des modems aux extrémités de la liaison et on passera par un support de transmission public (réseau téléphonique ou lignes spécialisées) (voir figure 7.3).

7.4.2 Principe de la transmission série asynchrone

En l'abscence de transmission, le niveau de la liaison est 1 (niveau de repos).

Les bits sont transmis les un après les autres, en commençant par le bit de poids faible b_0. Le premier bit est précédé d'un bit *start* (niveau 0). Après le dernier bit, on peut transmetre un bit de parité (voir cours de réseaux), puis un ou deux bits *stop* (niveau 1).

Chaque bit a une durée de Δ, qui fixe le débit transmission. Le nombre de changements de niveaux par seconde est appelé *rapidité de modulation* (RM), et s'exprime en Bauds (du nom de Baudot, l'inventeur du code TELEX). On a

$$RM = \frac{1}{\Delta}$$

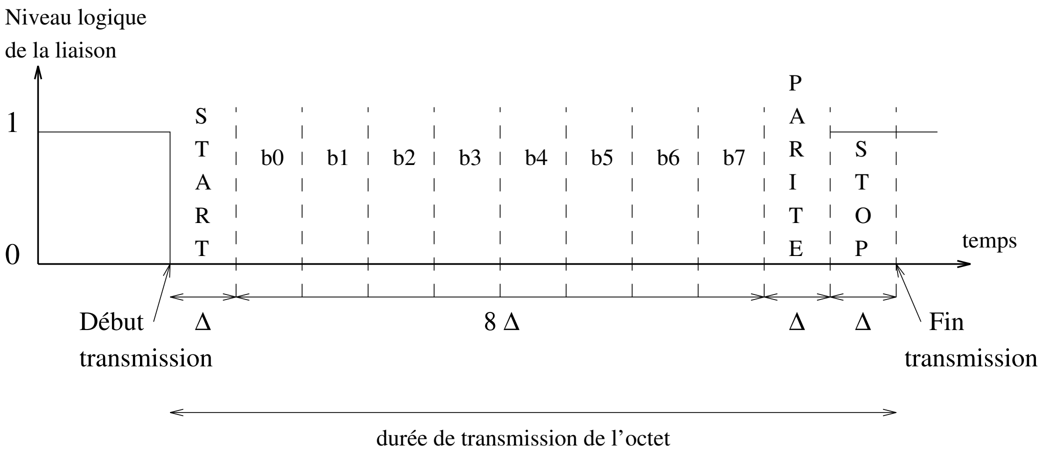


FIG. 7.4: Transmission d'un octet $b_7b_6b_5b_4b_3b_2b_1b_0$ en série.

et aussi

$$\text{débit binaire} = \frac{1}{\Delta}\text{bits/s}$$

Le récepteur détecte l'arrivée d'un octet par le changement de niveau correspondant au bit *start*. Il échantillonne ensuite chaque intervalle de temps Δ au rythme de son horloge.

Comme les débits binaires de transmission série de ce type sont faibles (< 19600 bits/s) et que les horloges de l'émetteur et du récepteurs sont suffisament stables (horloges à quartz), il n'est pas nécessaire de les synchroniser. C'est la raison pour laquelle ce type de transmission série est qualifié d'*asynchrone*.

Lorsque les débits sont plus importants, la dérive des horloges entrainerait des erreurs, et on doit mettre en œuvre une transmission *synchrone* (voir cours de réseaux).

7.4.3 L'interface d'E/S séries 8250

Le composant électronique chargé de la gestion des transmissions séries asynchrones dans les PC est appelé UART (*Universal Asynchronous Receiver Transmitter*).

Nous décrivons dans ce cours le circuit Intel 8250.

Bornes de l'interface

Les bornes de l'interface UART 8250 sont présentées sur la figure 7.5. Seules les bornes essentielles à la compréhension du fonctionnement de l'interface sont représentées.

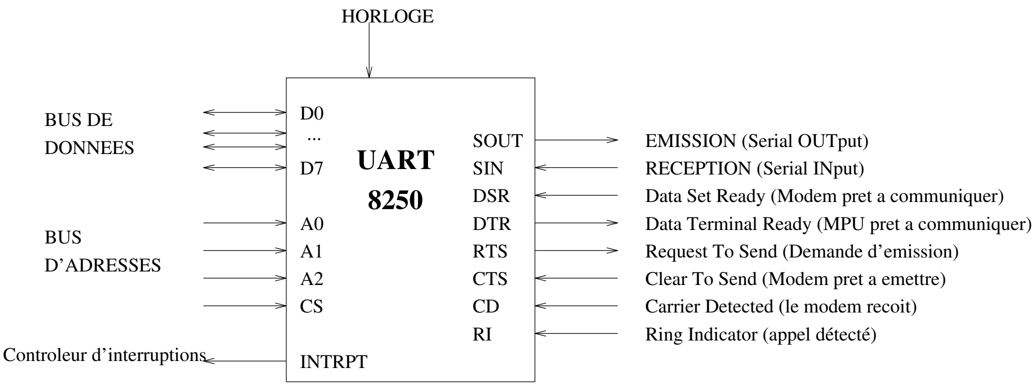


FIG. 7.5: Bornes du circuit UART 8250.

Les registres du 8250

L'interface 8250 possède un certain nombre de registres de 8 bits permettant de gérer la communication. Ces registres sont lus et modifiés par le processeur par le biais des instructions IN et OUT vues plus haut.

L'adresse de chaque registre est donnée par l'adresse de base de l'interface (fixée une fois pour toute) à laquelle on ajoute une adresse sur 3 bits $A_2A_1A_0$.

Une complication supplémentaire est introduite par le fait qu'il y a plus de 8 registres différents et que l'on ne dispose que de 3 bits d'adresse. La solution est d'utiliser un bit d'un registre spécial, DLAB. Suivant l'état du bit DLAB, on va sélectionner tel ou tel jeux de registres.

La table suivante donne l'adresse et le nom de chaque registre du 8250 :

Adresse				REGISTRES
DLAB	A2	A1	A0	
0	0	0	0	RBR : Receiver Buffer (registre de réception)
0	0	0	0	THR : Transmitter Holding Register (registre d'émission)
1	0	0	0	DLL : Divisor Latch LSB (poids faible diviseur horloge)
1	0	0	1	DLM : Divisor Latch MSB (poids fort diviseur horloge)
0	0	0	1	IER : Interrupt Enable Register
x	0	1	0	IIR : Interrupt Identification Register
x	0	1	1	LCR : Line Control Register
x	1	0	0	MCR : Modem Control Register
x	1	0	1	LSR : Line Status Register
x	1	1	0	MSR : Modem Status Register
x	1	1	1	non utilisé

Exemple : si l'adresse de base de l'interface est par exemple `3F8H`, l'adresse du registre RBR sera simplement

`3F8H + 000 = 3F8H`

et celle de IIR

`3F8H + 010b = 3F8H + 2H = 3FAH.`

On voit, comme nous l'avons dit plus haut, que les registres DLM et IER (par exemple) possèdent la même adresse (`001b`). Si le bit DLAB est 0, cette adresse permet d'accéder à DLM, et si DLAB=1 à IER.

Le bit DLAB est le bit de poids fort du registre LCR.

Notons aussi que THR et RBR ont la même adresse, ce qui n'est pas gènant car on accède toujours en *écriture* à THR et en *lecture* à RBR.

Voyons maintenant comment utiliser ces différents registres pour programmer l'interface de transmission série.

Choix de la rapidité de modulation

L'horloge de référence du 8250 est un signal à 1,8432 MHz stabilisé par un quartz.

Une première division de cette fréquence par 16 est effectuée dans l'interface. La fréquence obtenue est ensuite divisée par un diviseur codé sur 16 bits et contenu dans la paire de registres DLM (poids fort), DLL (poids faible).

On modifie donc le débit de transmission en changeant les valeurs de DLM et DLL.

Le tableau suivant donne les valeurs à utiliser pour les rapidités de modulation courantes :

Modulation (bauds)	Diviseur (hexa)
50	0900H
75	0600H
110	0417H
300	0180H
600	00C0H

Modulation (bauds)	Diviseur (hexa)
1200	0060H
2400	0030H
4800	0018H
7200	0010H
9600	000CH

Registre THR

C'est le registre d'émission. Il est chargé par le MPU en exécutant l'instruction

```
OUT THR, AL
```

(où THR est une constant initialisée avec l'adresse du registre THR).

Le contenu de THR est automatiquement copié par l'interface dans le registre à décalage d'émission, qui permettra la sortie en série des bits sur la sortie SOUT.

Registre RBR

C'est le registre de réception. Les bits qui arrivent en série sur la borne SIN du circuit entrent dans un registre à décalage. Lorsque ce dernier est complet, il est transféré dans RBR.

Le MPU peut lire le contenu de RBR avec l'instruction

```
IN AL, RBR
```

Notons que si un deuxième octet arrive avant que RBR n'ait été lu par le MPU, il remplace RBR : on perd le premier arrivé, c'est l'erreur d'*écrasement*[1].

Registre LCR (Line Control Register)

Ce registre de commande permet de définir certains paramètres de la transmission, en fonction de sa configuration binaire.

Bits 0 et 1 : spécifient le nombre de bits de la donnée à transmettre (caractères de 5, 6, 7 ou 8 bits) :

Bit 1	Bit 0	Nb de bits/caractère
0	0	5 bits
0	1	6 bits
1	0	7 bits
1	1	8 bits

Bit 2 : spécifie le nombre de bits STOP ($0 \rightarrow 1$ stop, $1 \rightarrow 2$ stops).

Bit 3 : spécifie la présence (si 1) ou l'absence (si 0) d'un bit de contrôle d'erreur (type bit de parité).

Bit 4 : s'il y a un bit de contrôle d'erreur, le bit 4 spécifie s'il s'agit d'un bit de parité paire (si 1) ou impaire (si 0).

Bit 5 : normalement à 0.

Bit 6 : normalement à 0.

Bit 7 : bit DLAB, permettant l'accès aux registres DLL et DLM dans la phase d'initialisation de l'interface.

Registre IER

Ce registre est utilisé pour les entrées/sorties par *interruption*.

Bit 0 : interruption lorsque donnée reçue dans RBR;

Bit 1 : interruption lorsque registre THR vide;

Bit 2 : interruption lorsque l'un des 4 bits de poids faible de LSR passe à 1;

Bit 3 : interruption sur état du modem.

Registre LSR

Ce registre d'état rend compte du fonctionnement de la liaison en réception (bits 0 à 4) et en émission (bits 5 et 6).

Bit 0 : passe à 1 lorsqu'une donnée a été reçue et chargée dans RBR.

Bit 1 : signale erreur d'écrasement. 1 si donnée reçue et chargée dans RBR alors que la précédente n'avait pas été lue. Remis automatiquement à 0 par la lecture du registre LSR.

[1] Le même type d'erreur peut se produire en émission si le processeur écrit THR avant que le caractère précédent ait été transféré.

Bit 2 : passe à 1 à la suite d'une erreur de parité. Remis à 0 par la lecture de LSR.

Bit 3 : passe à 1 lorsque le niveau du bit STOP n'est pas valide (erreur de format). Remis à 0 par la lecture de LSR.

Bit 4 : passe à 1 lorsque le niveau de la liaison est resté à 0 pendant la durée d'émission de la donnée (problème de l'emetteur). Remis à 0 par la lecture de LSR.

Bit 5 : passe à 1 lorsqu'une donnée est transférée de THR vers le registre à décalage d'émission (THR est alors libre pour le caractère suivant).

Bit 6 : passe à 1 lorsque le registre à décalage d'émission est vide.

Registre IIR

IIR est utilisé pour les E/S par interruption. Son contenu permet d'identifier la cause de l'interruption émise par l'interface 8250.

Registre MCR

MCR est le registre de commande du modem.

Bit 0 : commande le niveau de la borne DTR qui informe le modem que le MPU est prêt à communiquer;

Bit 1 : commande la borne RTS qui demande au modem d'émettre.

Registre MSR

MSR est le registre d'état du fonctionnement du modem.

7.4.4 Programmation de l'interface en langage C

Nous avons vu que les deux instructions assembleur permettant au processeur de communiquer avec les interfaces d'entrées/sorties étaient IN et OUT.

Il est possible d'accéder simplement à ces instructions en langage C, grâce aux fonctions `inportb( adr )` et `outportb( adr )`.

- `unsigned char inportb( int address )`
  lit un octet à l'adresse (de l'espace d'entrées/sorties) indiquée et le retourne.
- `void outportb( int address, unsigned char *data )`
  écrit l'octet (argument `data`) à l'adresse (E/S) indiquée.

Voici un exemple de configuration de l'interface 8250 en langage C. On configure ici l'interface pour un débit de 300 bauds, en mode scrutation, parité paire, 1 bit stop, 8 bits de données :

```
#include <dos.h>

/* Quelques constantes pour ameliorer la lisibilite:
 */
```

```
#define PORT (0x3F8) /* adresse de l'interface */
#define RBR  PORT
#define THR  PORT
#define LSR  (PORT+5)
#define IIR  (PORT+2)
#define LCR  (PORT+3) /* DLAB ... */
#define DLL  PORT     /* DLAB = 1 */
#define DLM  (PORT+1) /* DLAB = 1 */
#define IER  (PORT+1)
#define MCR  (PORT+4)
#define MSR  (PORT+6)

/* Initialise l'interface 8250
 */
void init_8250(void) {
  /* 1- Rapidite de modulation */
  outportb( LCR, 0x80 ); /* DLAB = 1 */
  outportb( DLM, 0x01 );
  outportb( DLL, 0x80 ); /* 300 bauds */

  /* 2- Format des donnees
   *  DLAB = 0, parite paire, 1 stop, 8 bits
   *    LCR = 00011011  = 1BH
   */
  outportb( LCR, 0x1B );

  /* 3- Mode de transfert: scrutation */
  outportb( IER,  0 );
}
```

7.4.5 Normes RS-232 et V24

Ces normes spécifient les caractéristiques *mécaniques* (les connecteurs), *fonctionnelles* (nature des signaux) et *électriques* (niveaux des signaux) d'une liaison série asynchrone avec un longueur maximale de 15m et une rapidité de modulation maximum de 20kbauds.

L'EIA (*Electrical Industry Association*) a été à l'origine aux USA de la norme RS-232, dont la dernière version est RS-232C. Le CCITT (Comité Consultatif International pour la Téléphonie et la Télégraphie) a repris cette norme qu'il a baptisé V24.

Deux autres normes permettent des débits plus élevés et des distances plus importantes : RS-423 (666m, 300kbauds), et RS-422 (1333m, 10Mbauds).

La norme V24 utilise le connecteur DB25, de forme trapézoïdale à 25 broches, représenté figure 7.6.

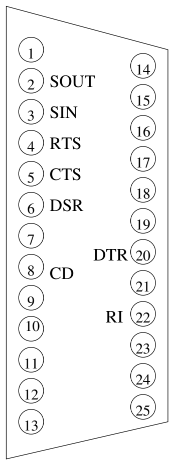


FIG. 7.6: Connecteur DB25, avec les bornes correspondantes du circuit UART 8250.

Les niveaux électriques des bornes 2 et 3 (signaux d'information) sont compris entre +3V et +25V pour le niveau logique 0, et -3V et -25V pour le niveau logique 1 (niveau de repos).

Cable NULL-MODEM

On peut connecter deux PC par leur interface série. Si la distance est courte (< quelques dizaines de mètres), il n'est pas nécessaire d'utiliser un modem. On utilise alors un cable *Null-Modem*, qui croise certains signaux comme le montre la figure 7.7.

Lorsque les signaux de dialogues ne sont pas nécessaires, il suffit de croiser les signaux SIN et SOUT, ce qui donne le cable Null Modem simplifié (3 fils) représenté sur la figure 7.8.

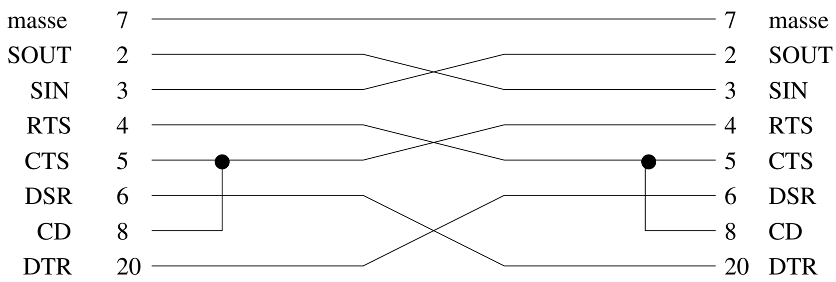


Fig. 7.7: Cable Null Modem complet.

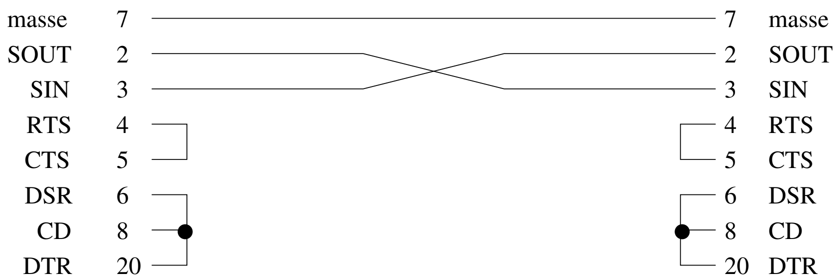


Fig. 7.8: Cable Null Modem complet.

Partie 8

Les périphériques

Nous étudions dans cette partie les périphériques d'entrées/sorties les plus couramment utilisés : clavier, écran et gestion des modes graphiques, disques durs et autres mémoires secondaires. Pour chaque type de périphérique, nous décrivons le principe de fonctionnement et mentionnons les performances des modèles actuellement en vente.

Si les principes fondamentaux de fonctionnement restent les mêmes, il faut noter que les performances (vitesse, capacité) de la plupart des périphériques informatiques évoluent très rapidement; les chiffres donnés ici sont donc à prendre comme des ordres de grandeur typiques du matériel utilisé à la fin des années 90.

8.1 Terminaux interactifs

Les micro-ordinateurs possèdent tous, sauf exception, un clavier et un écran uniques. Ce n'est pas le cas des ordinateurs plus gros, qui sont habituellement reliés à plusieurs *terminaux* (quelques dizaines ou centaines). Un terminal interactif est un périphérique permettant à un usager (humain) de communiquer avec un ordinateur. La communication se fait par l'intermédiaire d'un *écran* (ou moniteur), d'un *clavier* et éventuellement d'une souris.

Le terme "interactif" indique que l'échange utilisateur/ordinateur à lieu en temps réel, de façon interactive (l'ordinateur répond immédiatement aux commandes de l'utilisateur). Dans le passé, on utilisait aussi des terminaux non interactif, par exemple à base de cartes perforées, et l'on devait attendre plusieurs minutes (ou heures) avant de prendre connaissance des résultats d'une commande par le bias d'une imprimante.

8.1.1 Claviers

Le clavier est le périphérique le plus commode pour saisir du texte.

La figure 8.1 représente le principe de fonctionnement d'un clavier. Chaque touche est un interrupteur, normalement en position ouverte. Lorsque qu'une touche est appuyée, un signal électrique est envoyée vers le *codeur*, circuit élec-

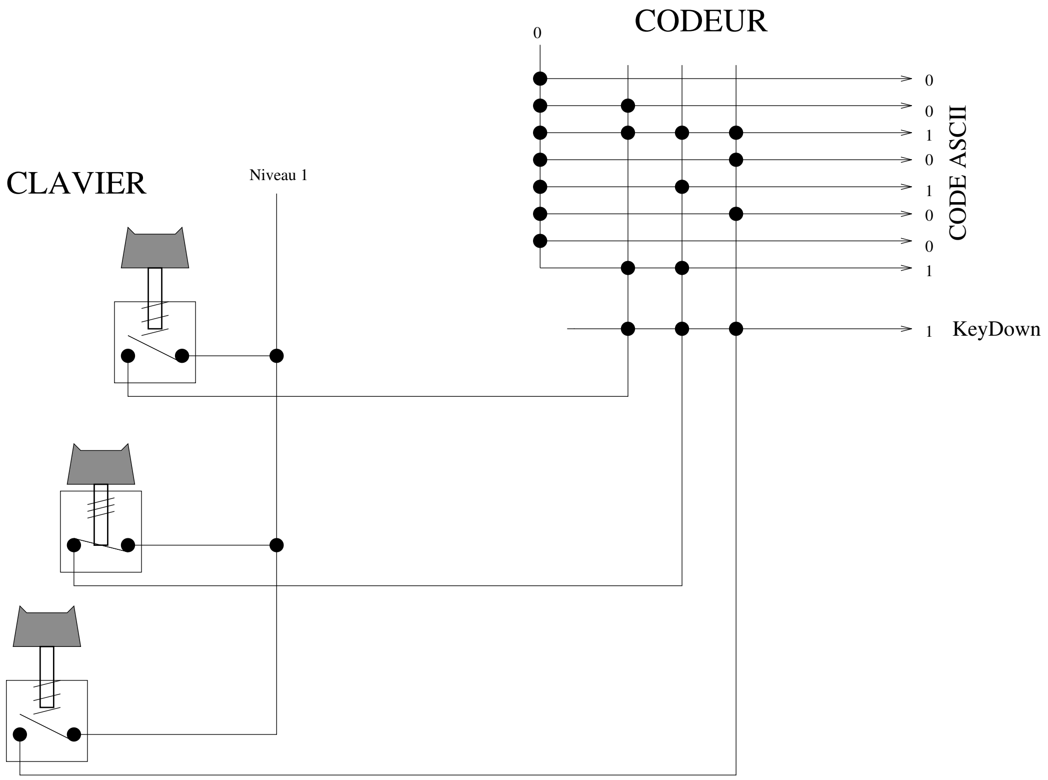


FIG. 8.1: Principe de fonctionnement d'un clavier. La pression d'un touche fait passer à 1 le signal *KeyDown*, et le code ASCII correspondant est présenté sur le bus de sortie du codeur. Seules trois touches sont représentées.

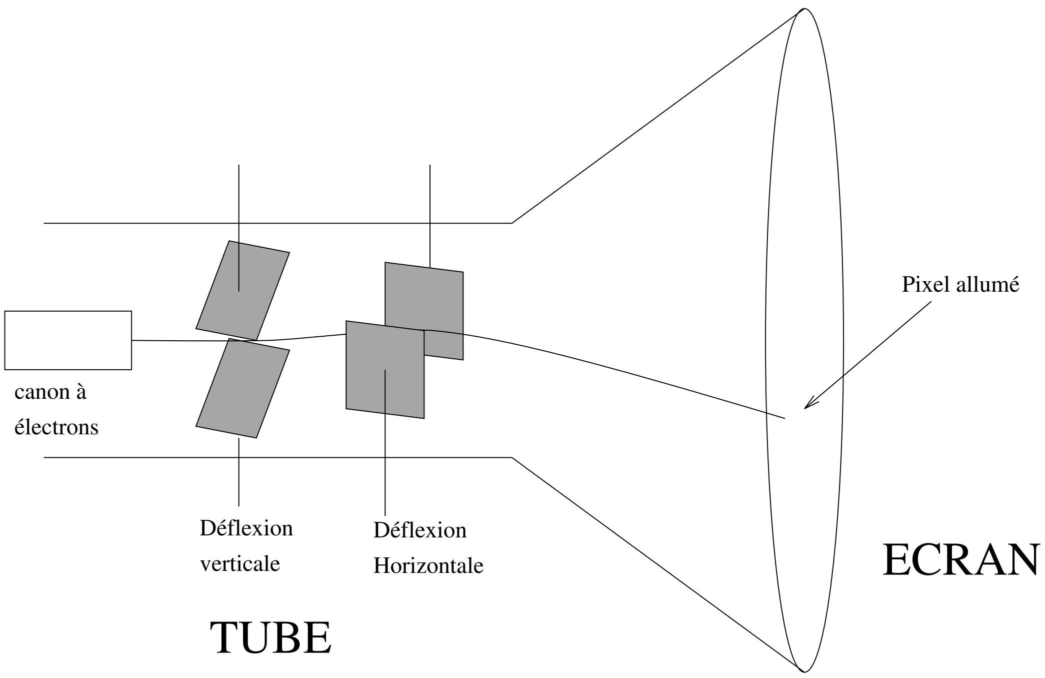


FIG. 8.2: Tube cathodique : un faisceau d'électrons accélérés est défléchi verticalement puis horizontalement par des champs électriques; l'impact de ces électrons sur l'écran, constitué d'une fine couche de phosphore sur du verre, allume un petit point.

tronique très simple qui associe à chaque signal un code (par exemple le code ASCII de la touche). Le code est associé à chaque touche par le bias de connexions ouvertes ou fermées dans la matrice du codeur.

Le codeur est relié à un bus d'entrées/sorties. Il génère aussi un signal *KeyDown* pour indiquer qu'une touche est appuyée. Ce signal peut être utilisé pour envoyer une interruption au processeur afin qu'il traite l'information.

Les codeurs réellement utilisés assurent des fonction supplémentaires, comme la répétition automatique des touches appuyées longtemps, la gestion d'une mémoire tampon de quelques dizaines de caractères, l'allumage de voyants, le verrouillage des majuscules, etc.

8.1.2 Ecrans et affichage

L'écran de l'ordinateur, aussi appelé *moniteur*, est le périphérique de sortie le plus répandu. Suivant les cas, il est utilisé pour afficher du texte ou des graphiques.

Un écran est constitué d'un tube cathodique[1], dont le principe est le même que celui d'un tube d'oscilloscope (voir figure 8.2).

[1]c'est pourquoi les écrans sont parfois nommés CRT, pour *Cathodic Ray Tube*, tube cathodique.

Le faisceau d'électron agit comme un pinceau, controlé par le signal vidéo émis par le contrôleur d'affichage de l'ordinateur. Chaque point de l'écran ne reste allumé qu'un court instant; le pinceau doit donc "repeindre" en permanence l'écran, environ 50 fois par seconde. Ce processus, appelé *balayage*, démarre du coin en haut à gauche de l'écran, et descend ligne par ligne jusqu'à arriver en bas à droite. C'est le contrôleur d'affichage ("carte graphique") de l'ordinateur qui génère le signal de balayage [2].

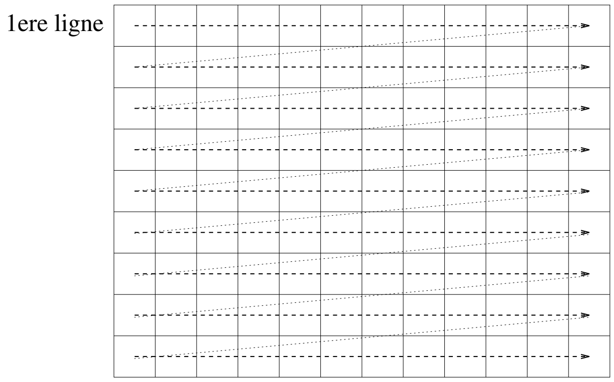


FIG. 8.3: Balayage des points de l'écran.

Le signal vidéo est donc défini par deux fréquences importantes : la *fréquence de balayage horizontal*, qui mesure combien de fois par seconde le faisceau revient en début de ligne, et la *fréquence de balayage vertical*, qui indique combien de fois par seconde le faisceau revient en haut à gauche de l'écran.

Les écrans ont des caractéristiques variables :

- taille : comme pour les télévisions, on mesure la diagonale de l'écran, dont la longueur est exprimée en pouces[3]. Les écrans d'entrée de gamme mesurent 14 pouces; les ordinateurs dits "multimédia" et les stations de travail utilisent des écran 17", et les graphistes des écrans 21".
- finesse (ou *pitch*) : indique le nombre de points par unité de longueur de l'écran, qui est donné par la finesse de la grille et la précision des faisceaux d'électrons. Plus la finesse est grande, plus l'image est précise, et plus on pourra afficher de pixels.
- fréquence maximale de balayage : plus l'écran est rafraichi fréquemment, plus l'image va apparaitre stable. Une fréquence de rafraichissement de 50 Hz, utilisé sur les écrans bas de gamme, donne une image avec des "battements" presque imperceptibles, mais qui fatiguent les yeux de l'utilisateur et sont très gênant si l'on visualise des images animées. Bien entendu, augmenter cette fréquence suppose d'utiliser une électronique plus couteuse; les écrans haut de gamme arrivent à une fréquence de balayage verticale de 120 Hz.

[2]Remarque : il existe deux modes de balayage, dits entrelacé et non-entrelacé; nous ne décrivons que le mode non-entrelacé.

[3]un pouce (", inch) mesure 2,54 centimètres.

8.1.3 Mode alphanumérique et mode graphique

L'affichage sur écran peut s'effectuer en mode texte (alphanumérique), ou bien en mode graphique.

Affichage en mode texte

Le mode texte est le plus simple, tant d'un point vue programmation que d'un point de vue implémentation. Par contre, il donne des résultats très pauvres. Ce mode d'affichage était le seul en usage jusqu'à une époque récente.

En mode texte, on affiche 24 lignes de 80 caractères. Chaque caractère est codé sur 8 bits, ce qui permet d'afficher les caractères ASCII (7 bits), plus un certain nombre de caractères spéciaux dont les codes ne sont pas normalisés (donc l'effet va varier d'un type d'ordinateur à l'autre). Ces caractères spéciaux permettent d'afficher des pseudo-graphiques (barres horizontales, obliques, symboles divers).

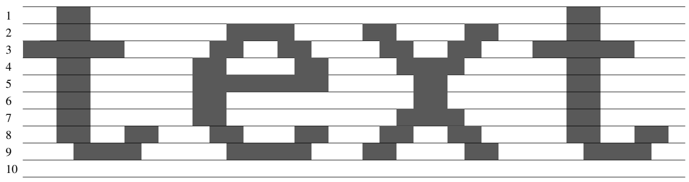


FIG. 8.4: Affichage des caractères en mode texte.

Le contrôleur graphique doit maintenir une mémoire de 24x80 octets, qui permet de générer le signal vidéo. Chaque caractère est affiché sur un nombre fixe de lignes (une dizaine), comme indiqué sur la figure 8.4. La mémoire vidéo est lue en séquence par le contrôleur pour générer le signal de chaque ligne.

La gestion d'attribus comme la couleur, l'inversion vidéo, le clignotement peut se faire en ajoutant des bits supplémentaires au codage des caractères en mémoire vidéo.

Affichage en mode graphique

L'affichage en mode graphique en apparu durant les années 70 sur certaines stations de travail haut de gamme, mais ne s'est généralisé qu'avec l'apparition des ordinateurs Apple Macintosh en 1983, puis du système Windows sur PC au début des années 90. Le PC garde la trace de cette évolution, puisque le DOS fonctionne en mode texte, ainsi que de nombreuses applications encore en usage en 1997 (par exemple Turbo C version 3).

Dans ce mode, les logiciels d'affichage ne manipulent plus des caractères mais des *pixels*. Chaque pixel correspondant à un point sur l'écran, et est caractérisé par sa couleur.

L'affichage est défini par le nombre de lignes, le nombre de colonnes, et le nombre de couleurs différentes. Le nombre de lignes et colonnes varie de 640x480 (mode VGA sur PC) à 1600x1200, voire d'avantage.

Le nombre de couleurs est lié au nombre de bits par pixels. En noir et blanc (2 couleurs), 1 bit par pixel suffit. Pour afficher simultanément 65536 couleur, il faut 16 bits par pixel. Les modes les plus sophistiqués utilisent 32 bits par pixel, l'intensité trois couleurs fondamentales (Rouge, Vert, Bleu) étant codée chacune sur 8 bits, les 8 bits restant pouvant coder la transparence de la couleur.

Le passage du codage d'un pixel sur n bits au signal de couleur vidéo peut être effectué de différentes manières. En 24 ou 32 bits/pixel, on travaille en "couleur directe" : chaque valeur code une couleur unique (spécifiée par les composantes RVB). Par contre, en 8 ou 16 bits, on utilise une table associant à chaque valeur (entre 0 et 2^n) une couleur RVB. Cette table (nommée *Look Up Table* (LUT) ou palette) peut être changée en cours d'utilisation; à chaque instant, on peut afficher 2^n couleurs parmi 2^{24}.

Mémoire vidéo La taille de la mémoire vidéo est ici beaucoup plus importante qu'en mode texte : le mode 1600x1200 en 256 couleurs demande environ 1.8 Mo, contre moins de 2 Ko pour un affichage texte. De ce fait, le volume d'information échangé entre la mémoire principale et le contrôleur graphique est très important, et l'on doit utiliser des bus d'entrées/sortie très performants pour atteindre une vitesse d'affichage satisfaisante (voir section 7.1.3).

8.2 Mémoires secondaires

Les mémoires secondaires (ou auciliaires) sont des périphériques permettant de stocker et de retrouver de l'information de manière durable : l'information est conservée même en l'absence d'alimentation électrique, contrairement à ce qui se passe pour la mémoire principale (RAM).

Les mémoires secondaires ont généralement une capacité stockage plus importante que les mémoires principales.

Parmi les mémoires secondaires les plus courantes, citons les disquettes et les disques durs, basés sur un enregistrement magnétique, les CD-ROM, utilisant une lecture optique, et divers types de bandes magnétiques.

8.2.1 L'enregistrement magnétique

Le principe de l'enregistrement magnétique est utilisé pour les cassettes audio et vidéo, ainsi pour les disquettes et disques durs informatiques. Il consiste à *polariser* un milieu magnétique (couche d'oxyde de fer déposée sur la bande ou le disque) à l'aide d'un champ électromagnétique créé par une bobine.

Un matériau magnétique (comme un aimant) possède la propriété intéressante de conserver durablement sa polarisation (orientation des particules magnétiques). La polarisation ne peut prendre que deux directions; chaque aimant peut donc être utilisé pour stocker 1 bit d'information.

L'enregistrement consiste à exploiter l'information rémanente (durable) créée par une tête de lecture/écriture. Cette tête comporte une bobine qui crée un champ

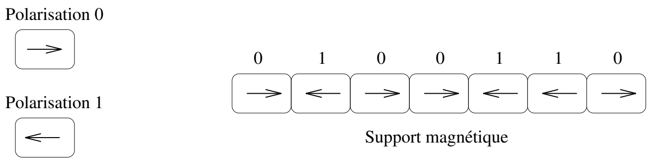


FIG. 8.5: Chaque cellule magnétique se comporte comme un aimant, et peut coder un bit.

magnétique dont l'orientation dépend du sens de circulation du courant électrique qui la parcourt.

La surface du support (bande ou disque) est divisée en petits emplacements qui vont se comporter individuellement comme des aimants (figure 8.5). Chaque emplacement code un bit. Pour lire l'information, on fait défiler le support sous la tête de lecture, qui mesure l'orientation du champ magnétique (qui crée un courant induit dans une bobine), de laquelle on déduit l'information stockée sur chaque emplacement.

Densité d'enregistrement magnétique

Le volume d'information (nb de bits) que l'on peut stocker sur une longueur donnée de surface magnétique dépend de la densité longitudinale d'enregistrement, que l'on mesure en BPI (*bits per inchs*, bits par pouces). Cette densité est limitée par le nombre maximum de renversements d'orientation de la polarisation par unité de longueur, qui dépend du type de couche magnétique, et par la taille de la tête de lecture.

Les densités typiques sont de l'ordre de 10 000 BPI. la distance entre la tête de lecture/écriture et le support est alors de l'ordre de 0,2 μm, ce qui impose une très grande propreté de la surface (voir figure 8.6).

Les disques durs, qui utilisent une densité d'enregistrement très élevée, sont scellés afin éviter toute entrée de poussière.

8.2.2 Les disques durs

Les premiers disques durs ont été développés par IBM en 1957 et ont connu un grand succès jusqu'à maintenant. Ils permettent en effet de stocker de grands volumes d'information tout en conservant un temps d'accès assez faible, et un rapport prix/capacité avantageux. Les micro-ordinateurs sont tous équipés de disques durs depuis la fin des années 80.

La capacité d'un disque dur typique de coût 4000 francs est passé de de 20 Mo en 1988 à 3 Go en 1996, soit une multiplication par 150 en dix ans !

Principe d'un disque dur

Une unité de disque dur est en fait constituée de plusieurs disques, ou plateaux, empilés et en rotation rapide autour du même axe (figure 8.7).

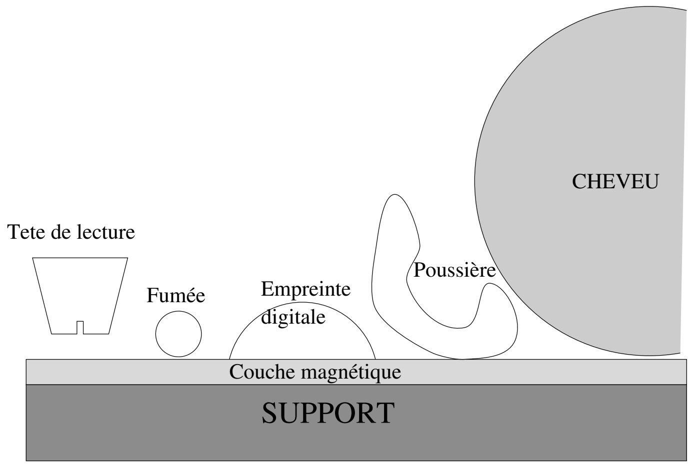


FIG. 8.6: Les diverses particules de poussières sont très gènantes pour l'utilisation d'un support magnétique : sont ici représentés à la même échelle la tête de lecture, une particule de fumée, une trace d'empreinte digitale, une poussière et un cheveu.

Chaque face d'un plateau est lue ou écrite par une tête de lecture. Afin de simplifier le mécanisme, toutes les têtes se déplacent en même temps, radialement (seule la distance tête-axe de rotation varie).

Les disques sont structurés en pistes et en secteurs, comme indiqué sur la figure 8.8.

Le nombre de pistes est fixé par la densité *transversale* (nombre de pistes par unité de longueur radiale). Cette densité dépend essentiellement de la précision du positionnement de la tête sur le disque.

Chaque piste ou secteur contient le même nombre d'octets (en fait, toutes les pistes n'ont pas la même longueur, mais la densité est plus grande sur les pistes du centre, de façon à obtenir le même volume d'information sur chaque piste). L'unité de lecture ou d'écriture sur le disque est le secteur.

Le système complet est constitué d'une ensemble de disques empilés, comme représenté sur la figure 8.7. Le contrôleur du disque doit être capable d'écrire ou de lire n'importe quel secteur. Pour repérer un secteur, il faut connaitre son plateau, le numéro de sa piste, et le numéro du secteur dans la piste. La plupart des systèmes introduisent la notion de *cylindre* : un cylindre est formé par l'ensemble des pistes de même position sur tous les plateaux.

Un secteur est alors repéré par (figure 8.9) :
– numéro de cylindre (donnant la distance tête-axe de rotation);
– numéro de piste (en fait le numéro de tête de lecture à utiliser);
– numéro du secteur (lié à l'angle).

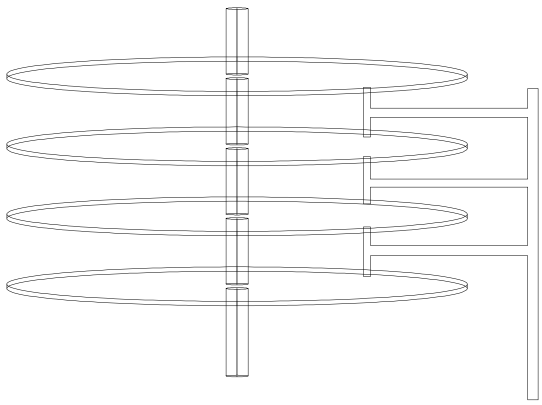


FIG. 8.7: Les plateaux d'un disque dur et les têtes de lectures (à droite), qui se déplacent toutes en même temps.

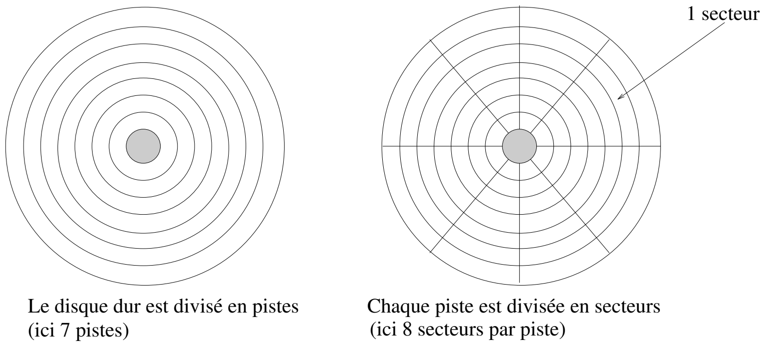


FIG. 8.8: Division d'un plateau de disque dur en pistes et en secteurs.

Temps d'accès

Le temps d'accès pour lire ou écrire un secteur du disque dur dépend de la vitesse de rotation du disque, de la vitesse de déplacement des têtes et de la dimension du disque.

Chaque transfert (lecture ou écriture d'un secteur) demande les opérations suivantes :

1. si les têtes ne sont pas déjà sur le bon cylindre, déplacement des têtes. On définit le temps de positionnement *minimum* (passage d'un cylindre au cylindre voisin), et le temps de positionnement *moyen* (passage à un cylindre quelconque, donc parcours en moyenne de la moitié du rayon).

2. attendre que le début du secteur visé arrive sous la tête de lecture : en moyenne, il faut que le disque tourne d'un demi-tour. Ce temps est appelé *demi délai rotationnel*.

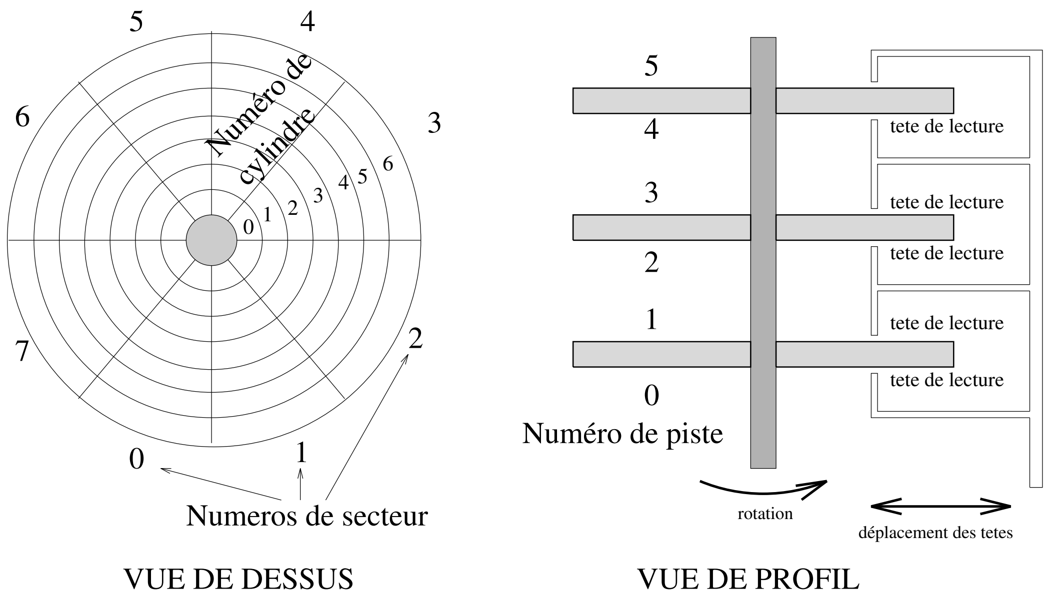


FIG. 8.9: Repérage d'un secteur du disque dur.

3. transfert des données, qui dure le temps nécessaire pour faire défiler le secteur entier sous la tête de lecture.

Le débit d'information maximal est déterminé par la vitesse de rotation du disque, la densité d'enregistrement longitudinale, et parfois limitée par le débit du bus d'entrées/sorties reliant le disque à l'ordinateur.

Les fabriquants de disques durs indiquent en général le temps d'accès moyen et le taux de transfert maximum (débit).

Cas des disquettes

Les lecteurs de disquettes fonctionnent sur les mêmes principes que les disques durs, mais il n'y a que deux faces, la vitesse de rotation est beaucoup plus faible et la densité d'écriture moindre.

Les disquettes actuelles ont une capacité de 1,4 Mo; il est probable qu'elles soient remplacées dans les années à venir par des disques durs extractibles miniaturisés, pouvant stocker plusieurs centaines de Mo.

8.2.3 Lecteurs de CD-ROM

Les CD-ROM (*Compact Disc Read Only Memory*), se sont imposés ces dernières années comme des mémoires secondaires en lecture seule. Leur capacité est de 650 Mo (soit l'équivalent de 450 disquettes). Le format de stockage est identique à celui utilisé pour les disques audio.

Leur (relativement) grande capacité en fait le support idéal pour livrer les logiciels de grande taille, mais aussi pour stocker des bases de données et programmes de toute nature (édition électronique,...).

La spécificité du CD-ROM est que l'on ne peut pas y modifier les informations, inscrites en usine.

Un disque CD-ROM est constitué d'une piste en spirale qui est lue par un faisceau laser de faible puissance. La piste est recouverte d'une fine couche de métal réfléchissant, sur laquelle sont percés des trous. La lecture s'effectue en mesurant le reflet du faisceau laser sur la piste, ce qui permet la détection des trous, donc la reconnaissance des bits 0 ou 1.

Le temps d'accès et le débit des lecteurs de CD-ROM sont essentiellement déterminés par la vitesse de rotation du disque, qui est elle même limitée par la difficulté à guider le laser sur la piste. ces informations sont souvent exprimées relativement à un lecteur de première génération; on parle ainsi de lecteur "double-vitesse", "quadruple-vitesse", voire "x12" ou "x24".

8.2.4 Autres supports optiques : WORM, magnéto-optiques

Outre le CD-ROM, il existe plusieurs autres types de support optiques. Les disques WORM (*Write Once, Read Many*, écrire une fois, lire plusieurs) utilisent un second laser plus puissant qui permet de former des "trous" sur la surface réfléchissante; on peut ainsi écrire de l'information une seule fois. Les disques WORM sont vendus vierges, sous l'appellation CD-R (CD enregistrable).

Les disques *magnéto-optiques* sont basés sur une technologie différente. Les trous sont remplacés par des différences de magnétisation d'un milieu spécial, ayant la propriété de modifier la polarité de la lumière suivant le sens de magnétisation. On associe donc un champ magnétique et un faisceau laser.

Ces disques sont ré-inscriptibles à volonté; ils ont été présentés à la fin des années 80 comme les successeurs des disques durs, mais leur faible vitesse d'accès en limite actuellement l'usage aux applications d'archivage ou de sauvegarde.

8.2.5 Bandes magnétiques

Les bandes magnétiques peuvent aussi être utilisées comme mémoire secondaires. La principale différence avec les disques est que l'accès à une bande est nécessairement *séquentiel* : si un fichier est enregistré à la fin d'une bande, il faut la rembobiner entièrement avant de d'y accéder. De ce fait, le temps d'accès moyen à un disque dur est de l'ordre de 10 ms, tandis que le temps d'accès sur une bande est de quelques secondes (100 à 1000 fois plus lent).

Les bandes sont cependant très utilisées car elle permettent un stockage à très faible coût de très grandes quantité d'information (exemple : une cassette de 8mm contient 7 Go et vaut environ 100 F).

Il existe un très grand nombre de standards différents pour les lecteurs de bandes, de cartouches ou de cassettes. La capacité d'une bande est déterminée par sa longueur, son nombre de pistes, et la densité d'écriture (voir plus haut, 8.2.1).

Notons enfin que certains lecteurs de bandes effectue une compression automatique des données avant écriture, ce qui permet de gagner un facteur 2 environ sur des données non préalablement compressées.

Partie 9

La mémoire

Nous revenons dans cette partie sur les différents types de mémoires utilisés dans les ordinateurs. Dans la partie précédente, nous avons traité les mémoires secondaires; nous nous intéressons maintenant au fonctionnement des mémoires *vives* (ou *volatiles*), qui ne conservent leur contenu que lorsqu'elles sont sous tension.

Ce type de mémoire est souvent désigné par l'acronyme RAM, *Random Access Memory*, signifiant que la mémoire adressable par opposition aux mémoires secondaires séquentielles comme les bandes.

Nous mentionnerons aussi différents types de mémoires mortes, qui sont des circuits accessibles uniquement en lecture (ROM, *Read Only Memory*).

9.1 Mémoire vive

La mémoire vive (RAM) est utilisable pour écrire ou lire des informations. Elle constitue la plus grande partie de la mémoire principale d'un ordinateur.

9.1.1 Technologie des mémoires vives

On peut réaliser des mémoires RAM avec deux technologies différentes, les RAM *dynamiques* (DRAM), et les *RAM statiques* (SRAM).

Mémoires vives dynamiques (DRAM)

Ce type de mémoire est très utilisé car peu couteux.

Les boitiers de mémoire dynamique enferment une pastille de silicium sur laquelle sont intégrées un très grand nombre de cellules binaires. Chaque cellule binaire est réalisée à partir d'un transistor relié à un petit condensateur. L'état chargé ou déchargé du condensateur permet de distinguer deux états (bit 0 ou bit 1).

L'inconvénient de cette technique simple est que le condensateur se décharge seul au cours du temps (courants de fuite). Il est donc nécessaire de rafraichir tous les condensateurs du boitier périodiquement, environ 1000 fois par seconde. Cette opération est effectuée par un circuit de rafraichissement intégré dans le boitier :

le circuit lit l'état de chaque cellule et le ré-écrit, ce qui recharge le condensateur. Notons que cette opération empèche l'accès à une cellule mémoire durant quelques cycles d'horloge.

Les mémoires DRAM sont utilisées en informatique, mais leur usage se répand aussi pour des objets grand public, comme la télévision numérique. Les boitiers sont fabriqués en très grandes séries dans des usines spécialisées, qui demandent des investissement énormes (une nouvelle chaine de fabrication coûte de l'ordre d'un milliard de dollars). Lorsque des usines de nouvelle génération sont mises en service, les prix des mémoires baissent dans le monde entier. La demande de mémoires augmentant sans cesse, les prix remontent, avant la construction de nouvelles usines, etc. Les prix des mémoires subissent ainsi des *cycles* économiques.

On trouve des boitiers DRAM de 256k x 1 bit, 256k x 4bits, 1M x 1 bit, jusqu'à 16M x 4bits, et bientôt d'avantage.

Mémoires vives statiques (SRAM)

Les mémoires statiques n'utilisent pas de condensateurs : chaque cellule binaire est réalisée à l'aide de 4 transistors formant un *bistable*, circuit restant d'un l'état 0 ou 1 tant qu'il est alimenté électriquement.

Les SRAM permettent des temps d'accès plus court que les DRAM, mais sont plus coûteuses car leur construction demande 4 fois plus de transistors que les DRAM.

Les SRAM sont utilisées lorsque l'on désire maximiser les performances, par exemple pour construire des mémoires caches (voir plus loin, section 9.3).

Notons aussi l'existence des boitiers SRAM CMOS, caractérisé par leur très faible consommation en énergie : une petite pile électrique suffit à les maintenir en activité plusieurs années. Ces mémoires, de petite capacité, sont utilisées par exemple dans certains agendas électroniques pour le grand public.

9.1.2 Modules de mémoire SIMM

Les modules SIMM *Single In-line Memory Module* sont des groupes de boitiers de mémoires dynamiques montés sur un circuit imprimé rectangulaire allongée, appelé *barette*. Chaque barette SIMM offre un capacité importante (1 à 16 Mo), et s'enfiche sur des connecteurs prévus à cette effet sur la carte mère de l'ordinateur.

Les barettes SIMM, utilisées au départ sur les stations de travail et les Macintosh, équipent aujourd'hui tous les PC.

9.2 Les Mémoires mortes

Les mémoires mortes ne sont normalement accessibles qu'en lecture. On distingue différents types de circuits de mémoires mortes :

ROM : circuit intégré dont le contenu est déterminé une fois pour toute au moment de la fabrication.

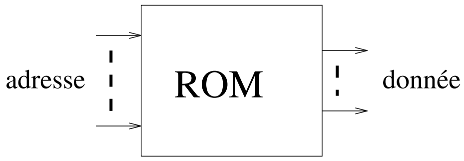


FIG. 9.1: Mémoire ROM.

Le coût relativement élevé de leur fabrication impose une fabrication en grandes séries, ce qui complique la mise à jour de leur contenu. Au départ, ces mémoires étaient utilisées pour stocker les parties bas-niveau du système d'exploitation de l'ordinateur (BIOS du PC par exemple).

PROM (Programmable ROM) : Alors que la mémoire ROM est enregistrée de manière irréversible lors de sa fabrication, la mémoire PROM est configurée par l'utilisateur en utilisant un *programmateur de PROM*, utilisé pour enregistrer son contenu. Le circuit PROM ne peut plus être modifié par la suite.

EPROM (Erasable PROM) : Les mémoires EPROM sont des PROM reconfigurables : il est possible de les effacer pour les reprogrammer. L'effaçage se produit en exposant le boitier à un fort rayonnement ultraviolet (UV). Pour cela, le boitier est percé d'une fenêtre transparente permettant l'exposition du circuit intégré.

L'opération d'effacement nécessite de retirer l'EPROM de son support et entraine sont immobilisation pendant environ 30 minutes.

EEPROM (Electricaly Erasable PROM) : Même principe qu'une EPROM, mais l'effacement se fait à l'aide de signaux électriques, ce qui est plus rapide et pratique.

FLASH EPROM Les mémoires FLASH sont similaires aux mémoires EEPROM, mais l'effacement peut se faire par sélectivement par blocs et ne nécéssite pas le démontage du circuit.

Le temps d'écriture d'un bloc de mémoire FLASH est beaucoup plus grand que celui d'écriture d'une mémoire RAM, mais du même ordre que celui d'un disque dur. L'accès en lecture à une EEPROM est à peu près aussi rapide qu'a une DRAM. On utilise donc parfois des cartes de mémoire FLASH comme mémoire secondaires, par exemple pour les ordinateurs portables.

9.3 Mémoires caches

9.3.1 Hierarchie mémoire

Chacun des différents types de mémoires primaires et secondaires que nous avons décrit est caractérisé par un *temps d'accès* et une *capacité* caractéristiques. Plus l'on s'éloigne du processeur, plus la capacité et le temps d'accès augmentent :

	Taille	Temps d'accès
Registres du processeur	10 octets	10^{-8} s
Mémoire principale	10^6 octets	10^{-7} s
Disque dur	10^9 octets	10^{-2} s

On constate que les processeur est nettement plus rapide que la mémoire principale. Dans les premières parties de ce cours, nous avons supposé que presque chaque instruction du processeur effectuait un accès, en lecture ou en écriture à la mémoire principale. Si c'était réellement le cas, le processeur passerait la majeure partie de son temps à attendre les accès mémoire, et l'on n'utiliserait pas pleinement ses possibilités.

9.3.2 Principe général des mémoires caches

L'introduction de *mémoires caches* peut pallier à ce problème. L'idée est d'intercaler entre le processeur et la mémoire principale un circuit de mémoire statique, plus rapide que la mémoire dynamique constituant la mémoire principale mais de petite taille. Dans cette mémoire, on va essayer de garder les informations normalement en mémoire principale dont le processeur se sert le plus souvent à un instant donné.

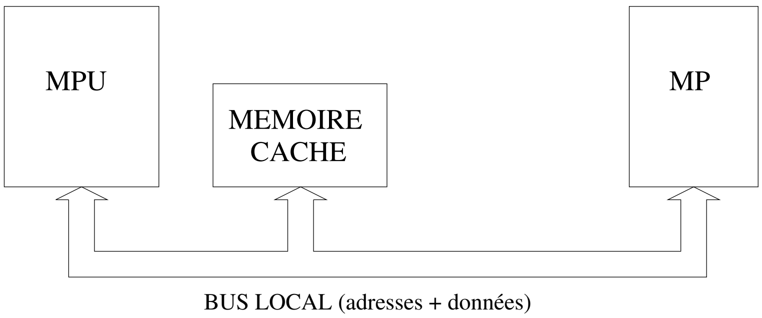


FIG. 9.2: Mémoire cache placée sur le bus local.

Lors d'un accès par le processeur à un mot en mémoire, deux cas peuvent se rencontrer :

1. le mot est présent dans le cache : le cache, plus rapide, envoie la donnée demandée sur le bus de données;
2. le mot n'est pas présent dans le cache : l'accès mémoire se déroule normalement (le cache peut en profiter pour copier la donnée afin d'en disposer pour la prochaine fois).

9.3.3 Mémoires associatives

Dans les mémoires ordinaires, on peut accéder à un mot par son adresse, et les adresses doivent être contigües : un boitier donné va par exemple stocker tous les mots dont les adresses sont comprises entre 100000H et 200000H.

Les mémoires caches doivent utiliser un principe différent, car les mots qu'elles vont stocker ont des adresses quelconques, qui ne se suivent pas forcément. Ce type de mémoire, appelé *mémoire associative*, est constitué d'un ensemble de paires (*clé, valeur*). La clé va ici être l'adresse du mot en mémoire.

Exemple : la mémoire associative suivante :

clé	valeur
00AA0004	1
01068C0B	78
00ABF710	789

contient trois éléments, d'"adresses" respectives :

00AA0004, 01068C0B et 00ABF710.

Les circuits de mémoire associative sont capables d'effectuer la recherche d'une clé en parallèle, donc en un temps très court (la clé demandée est comparée d'un seul coup à toutes les clés stockées).

La taille des mémoires caches associative est habituellement de l'ordre de quelques centaines de Ko.

9.3.4 Efficacité d'un cache : principe de localité

L'utilisation d'une mémoire cache n'est efficace que s'il arrive fréquement qu'un mot demandé par le processeur se trouve déja dans le cache.

Nous avons vu, sans entrer dans les détails, qu'une donnée entrait dans le cache lorsqu'elle était lue en mémoire principale. Le premier accès à une donnée est donc lent, mais les accès suivants à la même adresse vont être plus rapides.

On conçoit aisément que plus le programme que l'on exécute fait des accès mémoires variés, moins le cache sera efficace.

Rappelons que le processeur effectue deux types d'accès mémoires : lecture des instructions, et lecture (ou écriture) des données. Les accès aux instructions sont toujours différents (on passe à chaque fois à l'adresse de l'instruction suivante), sauf lorsque le programme effectue des répétitions (boucles), ce qui se produit en fait très fréquemment (les long calculs sont souvent effectué par répétition d'une suite d'instructions assez courte).

Les accès aux variables se répètent eux aussi fréquemment (les variables locales d'une fonction par exemple sont souvent accédées chacune un grand nombre de fois).

On appelle *localité* d'un programme sa tendance à effectuer des accès mémoires répétés; cette tendance augmente le gain de vitesse apporté par le cache.

9.3.5 Autres aspects

La gestion des caches est très importante pour les performances des ordinateurs modernes. Nous n'avons fait qu'effleurer ce vaste sujet, en particulier nous n'avons pas expliqué comment mettre le cache à jour.

Les processeurs récents possèdent des caches intégrés, et utilisent souvent deux caches différents, l'un pour les données, l'autre pour les instructions. A l'extérieur du processeur, on trouve un autre cache, dit de niveau 2, ou externe.

Notons enfin que la même problématique se retrouve pour la gestion des accès aux mémoires secondaires : la différence de vitesse entre les mémoires vives et les disques durs rend l'utilisation d'un cache encore plus avantageuse (malheureusement, les accès aux mémoires secondaires se répètent moins que ceux à la mémoire principale).

Partie 10

Architectures actuelles

Non rédigé

Dans cette dernière partie, nous passons brièvement en revue les différents types d'architectures employés dans les ordinateurs actuels.

10.1 Microprocesseurs

10.1.1 Architectures RISC et CISC

10.1.2 Famille de processeurs Intel

Du 8086 au Pentium II...
Processeurs "compatibles Intel" : AMD, Cyrix, ...

10.1.3 Famille Motorola 68k

Du 68000 au 68040...
Différences principales avec la famille 80x86.

10.1.4 PowerPC et autres RISCs

PowerPC, Sparc, Alpha, MIPS, ...
Bref tour des constructeurs et des performances.

10.2 Micro-ordinateurs

Etat du marche de la micro (PC, Macintosh, NC).

10.3 Stations de travail

Définition, quelques exemples.

10.4 Superordinateurs

Définition, quelques exemples.

Index