

**Architecture et  
organisation des  
systèmes**

**Juillet 1995**

**J. Ristori  
L. Ungaro**



# Table des matières

<b>Algèbre de Boole .....</b>	<b>1</b>
1- Axiomes - tables des opérations .....	1
1.1 - Introduction .....	1
1.2 - Tables des opérations booléennes .....	1
1.3 - Axiomes de l'algèbre de Boole .....	2
2- Principaux théorèmes d'algèbre de Boole .....	2
3 - Fonctions booléennes .....	4
3.1 - Définition par une table ou par une expression .....	4
3.2 - Formes canoniques d'une fonction .....	4
3.3 - Simplification de l'expression d'une fonction .....	6
 <b>Interprétation des chaînes de bits .....</b>	 9
1- Notion de représentation et d'interprétation .....	9
2- Quelques représentations usuelles .....	12
2.1 - Représentation des caractères .....	12
2.2 - Représentation des entiers naturels .....	13
2.3 - Représentation des entiers relatifs .....	13
3 - Les opérateurs sur chaînes de bits .....	14
3.1 - Opérateurs logiques .....	15
3.2 - Opérateurs arithmétiques .....	16
 <b>Les Circuits logiques .....</b>	 23
1 - Généralités .....	23
2 - Les entrées et les sorties .....	24
2.1 - Notions d'entrée et de sortie .....	24
2.2 - Aspects physiques des entrées et des sorties .....	24
3 - Assemblages de circuits logiques .....	26
3.1 - Règles de connexion .....	26
3.2 - Utilisation des sorties trois-états .....	27
4 - Quelques mots sur les technologies .....	29
 <b>Circuits combinatoires .....</b>	 33
1 - Les circuits combinatoires .....	33
1.1 - Notion de circuit combinatoire .....	33
1.2 - Assemblage combinatoire .....	33
1.3 - D'autres sortes de circuits logiques .....	34
2 - Etude de quelques circuits combinatoires .....	35
2.1 - Notations .....	35
2.2 - Les portes .....	35
2.3 - Les multiplexeurs .....	36
2.4 - Les décodeurs .....	36
2.5 - L'additionneur binaire .....	37
2.6 - Le ou exclusif .....	38
3 - Quelques méthodes de réalisation .....	41
3.1 - Critères de qualité d'une réalisation .....	41
3.2 - Méthode basée sur l'algèbre de Boole .....	42
3.2.1 - Exposé de la méthode .....	42
3.2.2 - Bon usage et limitations de ma méthode booléenne .....	43
3.2.3 - ROM et PLA .....	44
3.3 - Méthode basée sur la décomposition fonctionnelle .....	46
3.3.1 - Le découpage fonctionnel .....	46
3.3.2 - Phase d'analyse .....	46
3.3.3 - Quelques schémas de décomposition .....	49
3.3.4 - Phase d'optimisation .....	50

<b>Circuits séquentiels .....</b>	57
1- Notion de système séquentiel .....	57
1.1 - Etat interne .....	57
1.2 - Notion d'événement .....	58
1.3 - Description d'un système séquentiel .....	58
2- Circuits synchrones .....	61
3- Quelques exemples de circuits séquentiels .....	62
3.1 - Compteurs .....	62
3.2 - Bascules .....	63
3.3 - Registres .....	63
3.4 - Registres à décalage .....	64
4- Assemblages synchrones .....	65
5- Synthèse à l'aide de bascules .....	67
5.1 - Principes généraux .....	67
5.2 - Codage à un bit par état : machine "à jeton" .....	70
5.3 - Codage incorporant les sorties .....	71
6- Manipulation de signaux .....	73
6.1 - Filtrage et fusion d'horloges .....	73
6.2 - Propriétés dynamiques des portes - Aléas de commutation .....	75
7- Changement d'état asynchrone - Verrous .....	77
7.1 - Verrous (latches) .....	77
7.2 - Exemples de verrous .....	78
7.3 - Horloges à deux phases - Registres maître-esclave .....	79
7.4 - Circuits mixtes : horloge et commandes de déverrouillage.....	80
8- Prise en compte d'entrées asynchrones .....	84
8.1 - Nécessité de synchroniser les entrées asynchrones.....	84
8.2 - Le problème de la métastabilité .....	84
<b>Unités de contrôle .....</b>	87
1- Unité de contrôle et unité de traitement .....	87
1.1 - Principes généraux .....	87
1.2 - Méthode de décomposition .....	88
1.3 - Exemple de réalisation.....	89
2- Réalisation des unités de contrôle .....	91
2.1 - Structure générale d'une unité de contrôle .....	91
2.2 - Unité de contrôle à bascules .....	91
2.3 - Unité de contrôle à compteur chargeable .....	92
2.4 - Unité de contrôle microprogrammée .....	93
<b>Exercices.....</b>	96
<b>Mémoires vives .....</b>	101
1- Mémoires vives adressables .....	101
1.1 - Généralités .....	101
1.2 - Mémoires statiques .....	102
1.2.1 - Aspects externes.....	102
1.2.2 - Fonctionnement temporel .....	103
1.2.3 - Constitution de plans mémoires .....	104
1.2.4 - Structure interne d'une mémoire statique .....	104
1.3 - Mémoires dynamiques .....	105
1.3.1 - Principe de fonctionnement .....	105
1.3.2 - Chronogramme des accès .....	107
1.2.3 - Constitution d'un plan de mémoire dynamique .....	108
2- Mémoires à accès particulier.....	109
2.1 - Mémoires à adressage implicite - piles et files .....	109
2.1.1 - Pile .....	109
2.1.2 - File .....	110
2.2 - Mémoires associatives .....	112
<b>Exercices.....</b>	113

<b>Bus de processeurs .....</b>	117
1 - Interconnection de composants séquentiels .....	117
1.1 - Liaison directe point à point .....	117
1.2 - Liaison par canal centralisé : le bus .....	118
1.2.1 - Le bus adressable .....	118
1.2.2 - Le bus restreint .....	119
1.2.3 - Le bus à composant maître .....	119
2 - Architecture générale d'un ordinateur .....	120
2.1 - Les composants d'un ordinateur .....	120
2.2 - Trois aspects du processeur .....	120
2.2.1 - Interface logicielle : le modèle de programmation .....	120
2.2.2 - Un système séquentiel complexe .....	123
2.2.3 - Interface matérielle : le bus du processeur .....	124
3 - Structure et usage d'un bus.....	125
3.1 - Structure d'un bus de processeur .....	125
3.2 - L'accès aux composants .....	126
3.2.1 - Les espaces d'adressage.....	126
3.2.2 - L'implantation des composants .....	127
3.3 - Les cycles bus .....	130
3.3.1 - Les cycles bus courants .....	130
3.3.2 - L'exécution des instructions.....	132
3.3.3 - Synchronisation des accès.....	132
3.4 - Autres signaux.....	134
3.4.1 - Partage du contrôle du bus : DMA.....	134
3.4.2 - Demandes d'interruptions.....	134
4 - Exemples .....	135
4.1 - Le bus PC-XT .....	135
4.2 - Le MC68000 .....	139
Exercices.....	141
 <b>Interfaces d'entrées-sorties .....</b>	145
1 - Communication avec les périphériques .....	145
1.1 - Les convertisseurs .....	146
1.2 - L'interface.....	147
1.2.1 - Adaptation logique.....	147
1.2.2 - La synchronisation des échanges .....	147
1.2.3 - Adressage de l'environnement .....	148
2 - Synchronisation des entrées-sorties .....	149
2.1 - Consultation et affichage directs.....	149
2.1.1 - Lecture à la volée .....	149
2.1.2 - Affichage à la volée .....	150
2.2 - Entrées-sorties synchronisées par test d'état programmé .....	150
2.2.1 - Synchronisation en entrée .....	151
2.2.2 - Synchronisation en sortie .....	151
2.2.3 - Lecture du bit d'état .....	152
2.2.4 - Modification du bit d'état .....	153
3 - Compléments sur les interfaces .....	154
3.1 - Commandes de configuration .....	154
3.2 - Interfaces multiples .....	155
3.3 - Interface et port d'entrée-sortie .....	155
3.4 - Adresse mémoire et port d'entrée-sortie .....	155
Exercices.....	156
 <b>Entrées-sorties séries .....</b>	159
1 - Transmissions séries .....	159
1.1 - Généralités .....	159
1.2 - Transmissions synchrones .....	159
1.3 - Transmissions asynchrones .....	162
1.4 - Usage des transmissions séries .....	163

2 - Interface d'entrées-sorties série : USART .....	166
2.1 - Structure générale de l'USART .....	166
2.2 - Commandes de l'USART .....	167
2.3 - Mot d'état de l'USART .....	169
Exercices.....	170
<b>Mécanisme d'interruptions .....</b>	<b>173</b>
1- Principe et utilisation des interruptions .....	173
1.1 - Insuffisance du test d'état programmé .....	173
1.2 - Le mécanisme d'interruptions .....	174
1.3 - Exemple d'utilisation : traitements de données en temps réel ...	175
1.4 - Masquage des interruptions - réalisation des exclusions .....	178
1.5 - Sources d'interruptions multiples .....	180
1.5.1 - Scrutation programmée des sources .....	180
1.5.2 - Interruptions vectorisées .....	181
1.5.3 - Fonctionnement d'un contrôleur d'interruptions.....	182
2 - Interruptions du 8086 .....	185
2.1 - Table des procédures d'interruptions .....	185
2.2 - Prise en compte des interruptions - masque général .....	186
2.3 - Contrôleur d'interruptions PIC 8259 .....	187
2.4 - Exemple d'application.....	190
2.4.1 - Initialisation du PIC .....	190
2.4.2 - Utilisation .....	190
<b>Hiérarchie de mémoires .....</b>	<b>195</b>
1- Principes généraux.....	195
1.1 - Importance des performances de la mémoire .....	195
1.2 - La hiérarchie : concilier rapidité et grande taille .....	196
1.3 - Mécanismes : gestion de cache et mémoire virtuelle .....	198
2- Fonctionnement des caches.....	200
2.1 - Correspondance adresses/emplacement dans le cache .....	200
2.2 - Caches à correspondance directe.....	200
2.3 - Caches associatifs par ensembles de k emplacements .....	202
2.4 - L'écriture dans un cache .....	204
2.5 - Transferts entre cache et mémoire - mémoire entrelacée .....	205
3- Fonctionnement d'une mémoire virtuelle .....	206
3.1 - Principales caractéristiques d'une mémoire virtuelle .....	207
3.2 - Correspondance adresse virtuelle - adresse physique .....	207
3.3 - Traduction rapide des adresses - TLB .....	209
3.4 - Interactions entre cache et mémoire virtuelle.....	210
<b>Accès direct mémoire - DMA .....</b>	<b>211</b>
1- Mécanisme d'accès direct mémoire .....	211
2- Exemple de contrôleur de DMA : Intel 8237 .....	213
2.1 - Structure du contrôleur de DMA.....	213
2.2 - Programmation du DMA .....	214

## ALGEBRE DE BOOLE

## 1 - Axiomes - tables des opérations

## 1.1 - Introduction

L'algèbre de Boole formalise les opérations internes de l'ensemble à deux éléments {0,1}, parfois notés {faux,vrai}. Initialement ce système fut introduit pour manipuler la valeur de vérité/fausseté des propositions logiques, mais il peut servir à d'autres usages.

et une opération à un opérande :  $\text{NON}(x)$  également noté  $/x$ .

On peut présenter ce système de deux façons :

par énoncé d'un certain nombre d'axiomes, ou en donnant les tables des opérations.

La définition par tables a l'avantage d'être simple, mais elle est limitée à l'algèbre de Boole sur l'ensemble à deux éléments {0,1}. La présentation par axiomes, plus abstraite, s'applique à d'autres domaines que {0,1} (voir ci-après quelques exemples).

## 1.2 - Tables des opérations booléennes

Les opérations sont définis par les tables suivantes :

ET		
x	y	x.y
0	0	0
0	1	0
1	0	0
1	1	1

OU		
x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

NON	
x	/x
0	1
1	0

$x.y$  vaut 1 si  $x$  vaut 1 et si  $y$  vaut 1, 0 sinon.  
 $x+y$  vaut 1 si  $x$  vaut 1 ou si  $y$  vaut 1, 0 sinon  
 $/x$  vaut 1 si  $x$  vaut 0, 0 sinon.

### 1.3 - Axiomes de l'algèbre de Boole

<b>A1a</b>	$x + (y + z) = (x + y) + z$	+ est associatif
<b>A1b</b>	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	\cdot est associatif
<b>A2a</b>	$x + y = y + x$	+ est commutatif
<b>A2b</b>	$x \cdot y = y \cdot x$	\cdot est commutatif
<b>A3a</b>	$\exists 0 \quad x + 0 = x$	0 est élément neutre pour +
<b>A3b</b>	$\exists 1 \quad x \cdot 1 = x$	1 est élément neutre pour \cdot
<b>A4a</b>	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	\cdot est distributif sur +
<b>A4b</b>	$x + (y \cdot z) = (x + y) \cdot (x + z)$	+ est distributif sur \cdot
<b>A5</b>	$\forall x \exists /x \quad x + /x = 1$ $x \cdot /x = 0$	existence du complément

Ces axiomes sont faciles à vérifier pour les opérations définies par les tables précédentes.

### Exemples d'algèbres de Boole autres que {0,1}

Etant donné un ensemble quelconque E, l'ensemble  $P(E)$  des parties de E forme une algèbre de Boole, avec les correspondances suivantes :

$$\begin{aligned} 0 &\leftrightarrow \emptyset \text{ (ensemble vide)} & 1 &\leftrightarrow E \text{ (le total)} \\ ET &\leftrightarrow \cap \text{ (intersection)} & OU &\leftrightarrow \cup \text{ (union)} & NON &\leftrightarrow \text{complémentaire dans } E \end{aligned}$$

De même, l'ensemble des vecteurs à  $n$  composantes  $\{0,1\}^n$ , avec les correspondances suivantes :

$$\begin{aligned} 0 &\leftrightarrow (0,0,0) & 1 &\leftrightarrow (1,1,1) & \text{avec les opérations "bit à bit"} : \\ ET(X,Y) &\leftrightarrow (x_2.y_2, x_1.y_1, x_0.y_0) & OU(X,Y) &\leftrightarrow (x_2+y_2, x_1+y_1, x_0+y_0) & NON(X) &\leftrightarrow (/x_2, /x_1, /x_0) \end{aligned}$$

## 2 - Principaux théorèmes d'algèbre de Boole

### Dualité

Les axiomes vont par paires ; si on procède à l'échange des symboles :  $0 \leftrightarrow 1$  et  $\cdot \leftrightarrow +$ , on passe d'un axiome à l'autre axiome de la paire.

Ceci implique que pour chaque théorème, il en existe un autre obtenu en procédant à cette transformation : ces deux théorèmes sont dits *duaux*. Ceci permet une économie de connaissances : il suffit de se rappeler de la moitié des théorèmes.

Nous présentons ci-après les principaux théorèmes, par paires *duales* (numérotés Tia et Tib ; les théorèmes qui sont simplement numérotés Ti sont leur propres *duaux*).

A titre d'exercice, le lecteur pourra facilement démontrer ces théorèmes à partir des axiomes.

<b>T1a</b>	$x + x = x$	+ est idempotent
<b>T1b</b>	$x \cdot x = x$	$\cdot$ est idempotent
<b>T2a</b>	$x + 1 = 1$	1 est absorbant pour +
<b>T2b</b>	$x \cdot 0 = 0$	0 est absorbant pour $\cdot$ .
<b>T3</b>	$\forall x /x$ est unique	unicité du complément
<b>T4</b>	$/(/x) = x$	
<b>T5a</b>	$x + x.y = x$	
<b>T5b</b>	$x \cdot (x+y) = x$	
<b>T6a</b>	$x + /x.y = x + y$	
<b>T6b</b>	$x \cdot (/x + y) = x \cdot y$	
<b>T7a</b>	$/x + y = /x \cdot /y$	Théorèmes de De Morgan
<b>T7b</b>	$/x \cdot y = /x + /y$	

Les théorèmes de De Morgan permettent de passer d'une expression à son complément.

Les théorèmes de De Morgan peuvent être appliqués plusieurs fois pour trouver une expression du complément d'une expression donnée, par exemple :

$$/((a.b + a.d).( /b + /c)) = /(a.b + a.d) + /(/b + /c) = /(a.b) \cdot /(a.d) + b.c = (/a + /b) \cdot (/a + /d) + b.c$$

On peut aussi appliquer la règle globale suivante, qui est une généralisation des théorèmes de De Morgan : on obtient le complément d'une expression en échangeant “+” et “.”, et en complémentant les variables (il faut également échanger 0 et 1, mais il est rare qu'ils apparaissent dans une expression car on peut les simplifier aussitôt). Sur l'exemple, on obtient ainsi directement :

$$/((a.b + a.d).( /b + /c)) = (/a + /b) \cdot (/a + /d) + b.c$$

<b>T8a</b>	$a.x + a./x = a$	
<b>T8b</b>	$(a+x).(a+/x) = a$	
<b>T9a</b>	$a.x + /a.y + x.y = a.x + /a.y$	
<b>T9b</b>	$(a+x).( /a+y).(x+y) = (a+x).( /a+y)$	
<b>T10</b>	$(a+b=a+c \text{ et } a.b=a.c) \Leftrightarrow b=c$	
<b>T11a</b>	$/x_1 + x_2 + \dots + x_n = /x_1 \cdot /x_2 \cdot \dots \cdot /x_n$	
<b>T11b</b>	$/x_1 \cdot x_2 \cdot \dots \cdot x_n = /x_1 + /x_2 + \dots + /x_n$	
<b>T12a</b>	$\forall f \exists f_1, f_2 \quad f(x,y,z,\dots) = x.f_1(y,z,\dots) + /x.f_2(y,z,\dots)$	
<b>T12b</b>	$\forall f \exists f_1, f_2 \quad f(x,y,z,\dots) = (x+f_1(y,z,\dots)).(/x+f_2(y,z,\dots))$	
<b>T13a</b>	$/x.f_1 + /x.f_2 = x./f_1 + /x./f_2$	
<b>T13b</b>	$/((x+f_1).( /x+f_2)) = (x+/f_1).( /x+/f_2)$	

Le théorème 12 exprime que l'on peut toujours décomposer une expression relativement aux deux cas possibles pour une variable ; le théorème 13 donne une règle simple pour complémer une expression décomposée par cas relativement à une variable.

### 3 - Fonctions booléennes

#### 3.1 - Définition par une table ou par une expression

Une fonction booléenne est une application de  $\{0,1\}^n$  dans  $\{0,1\}$ .

$$f : x_1, \dots, x_n \in \{0,1\}^n \rightarrow f(x_1, \dots, x_n) \in \{0,1\}$$

Une telle fonction peut être définie de plusieurs façons :

soit par une table qui donne les valeurs de la fonction pour chaque valeur des variables,  
soit par une expression faisant intervenir les variables.

définition par une table	x y z   f(x,y,z)	définition par une expression
0 0 0	0	$f(x,y,z) = x.y + x.z + y.z$
0 0 1	0	
0 1 0	0	
0 1 1	1	
1 0 0	0	
1 0 1	1	
1 1 0	1	
1 1 1	1	

La table d'une fonction à  $n$  variables a  $2^n$  lignes, nombre qui croît très vite avec  $n$  : il est humainement difficile d'appréhender une telle table au-delà de 6 variables (64 lignes).

#### 3.2 - Formes canoniques d'une fonction

##### Mintermes - Maxtermes

On appelle *minterme*, on dit aussi *monôme*, un ET de variables, chaque variable intervenant au plus une fois, sous forme directe ou complémentée.

On appelle *maxterme* un OU de variables, chaque variable intervenant au plus une fois, sous forme directe ou complémentée.

Un minterme ou un maxterme est dit *complet* si toutes les variables y figurent.

Exemples avec 4 variables : x,y,z,t

mintermes

x./y      y.z.t

minterme complet

/x.y.z./t

maxtermes

/x+/z+t      y+z

maxterme complet

x+/y+z+t

A chaque vecteur on peut associer un minterme complet en prenant la variable directe si la variable vaut 1 dans ce vecteur, et en prenant la variable complémentée si elle vaut 0 dans ce vecteur. Le minterme obtenu vaut 1 pour ce vecteur, et 0 pour tout autre vecteur.

On peut également associer un maxterme à un vecteur, en prenant la variable directe si la variable vaut 0 dans ce vecteur, et en prenant la variable complémentée si elle vaut 1 dans ce vecteur : ce maxterme vaut 0 pour ce vecteur et il vaut 1 pour tout autre vecteur.

vecteur x y z t	minterme associé	maxterme associé
0 1 1 0	/x.y.z./t	x+/y+/z+t
1 1 0 1	x.y./z.t	/x+/y+z+/t
1 1 1 1	x.y.z.t	/x+/y+/z+/t

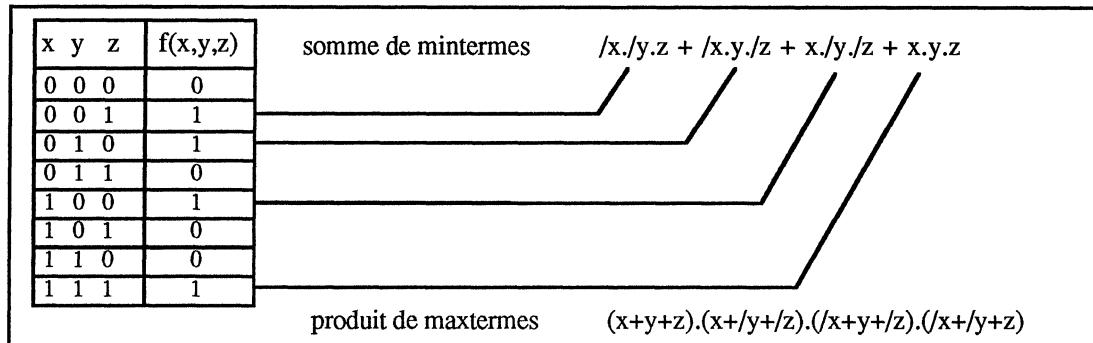
Les appellations "minterme" et "maxterme" proviennent de ce qu'on a l'habitude d'introduire un ordre entre les valeurs booléennes : "0<1" ; avec cette convention, l'opération ET entre variables correspond au minimum des variables, et le OU correspond au maximum.

### Forme canonique 1 : somme de mintermes complets

Etant donnée une fonction quelconque, on peut toujours dresser sa table. On peut exprimer alors cette fonction comme somme des mintermes associés aux vecteurs pour lesquels la fonction vaut 1. L'expression obtenue s'appelle *forme canonique 1* de la fonction.

### Forme canonique 2 : produit de maxtermes complets

On peut également exprimer la fonction comme produit des maxtermes associés aux vecteurs pour lesquels la fonction vaut 0. Cette expression est la *forme canonique 2* de la fonction.



L'existence des formes canoniques montre que toute fonction booléenne peut être donnée par une expression : l'expression obtenue ici est en fait une transcription de sa table, en énumérant les cas où elle vaut 1 (somme de mintermes) ou bien les cas où elle vaut 0 (produit de maxtermes).

### **Les fonctions de 2 variables :**

Nous avons vu que chaque fonction de  $n$  variables est entièrement décrite par une table de  $2^n$  lignes. Chaque fonction est ainsi caractérisée par un vecteur booléen de  $2^n$  éléments formés de la valeur que prend la fonction pour chacune des lignes. Il y a donc  $2^{2^n}$  fonctions différentes de  $n$  variables.

Pour 2 variables, il y a  $2^2=16$  fonctions différentes dont nous pouvons dresser la liste :

### Opérations non-et (“nand”) et non-ou (“nor”)

N'importe quelle fonction peut être exprimée en utilisant la seule opération non-et :  
 $\text{non-et}(x, y, \dots) = /(\text{x.y.}\dots)$

Pour ce faire, il suffit de partir d'une expression de la fonction sous forme de somme de monomes et d'appliquer la formule de De Morgan :

$$f = m_1 + m_2 + m_3 + \dots = /(\overline{m_1}, \overline{m_2}, \overline{m_3}, \dots)$$

Chaque  $/m_i$  est une opération non-*et* sur les variables , et le total est également un non-*et*.

De même on peut n'utiliser que l'opération non-ou :  $\text{non-ou}(x,y,\dots) = /(x+y+\dots)$   
il suffit pour cela de partir d'une expression sous forme de produit de maxtermes :

$$f \equiv M_1, M_2, M_3, \dots \equiv /(\bar{M}_1 \pm \bar{M}_2 \pm \bar{M}_3 \pm \dots)$$

### 3.3 Simplification de l'expression d'une fonction

Simplifier l'expression d'une fonction, c'est chercher une expression équivalente plus simple. Mais on ne sait pas à priori très bien ce qu'on entend par "plus simple" ; par exemple, des deux formes suivantes, laquelle est la plus simple ?

$$[1] \quad x(y+z.t) + /x.(y+z.t) \quad [2] \quad x.y + x./z.t + /x./y.z + /x./y.t$$

A première vue, les deux se valent ; quelqu'un peut cependant prétendre que [1] est plus simple, car la même sous-expression  $y+z.t$  est utilisée deux fois. Ainsi la notion de simplicité est très subjective en l'absence de critères précis.

En règle générale, lorsqu'on parle de simplification d'expression, on entend :

"obtenir une expression la plus simple possible sous forme de somme de monomes".

Ceci a pour avantage de définir un critère précis, mais limite beaucoup l'intérêt des techniques de simplification : ce critère ignorent totalement les simplifications que peuvent apporter les mises en facteur et l'usage de sous-expressions identiques.

#### Simplification sous forme de somme de monomes

Partant d'une expression sous forme de somme de monomes, par exemple la forme canonique, on peut appliquer les règles de simplification suivantes, qui diminuent soit le nombre de variables dans les monomes, soit le nombre de monomes, et parfois les deux :

$$x.m + /x.m = m$$

$$m + m = m$$

$$m + m.m1 = m$$

$$x.m + /x.m.m1 = x.m + m.m1$$

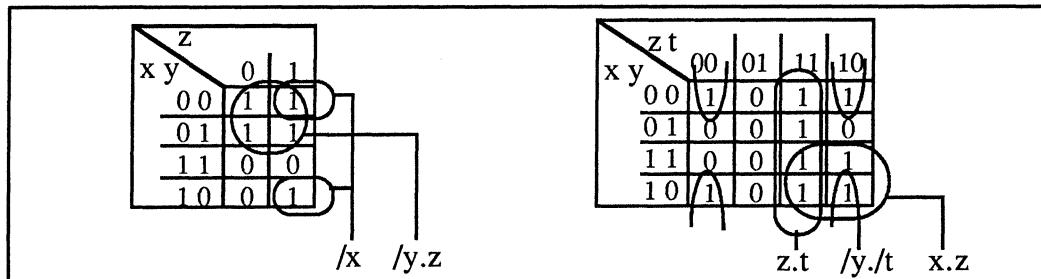
Ces règles de simplification peuvent être appliquées directement, à l'estime, sur l'expression de la fonction. Une méthode visuelle connue sous le nom de "tableau de Karnaugh" permet de les appliquer plus systématiquement et conduit à une expression simplifiée au maximum.

#### Tableaux de Karnaugh

Le tableau de Karnaugh est une table de la fonction organisée de telle façon qu'une seule variable change entre des cases voisines (verticalement ou horizontalement) ; ceci est obtenu en énumérant les vecteurs dans un ordre particulier : 00, 01, 11, 10. Le schéma suivant montre l'organisation des tableaux de Karnaugh pour 2, 3 et 4 variables, qui sont les cas les plus fréquents :

<table border="1" style="border-collapse: collapse; width: 50px; height: 50px;"> <tr><td>x</td><td>y</td></tr> <tr><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </table> 2 variables	x	y	0	0	0	1	1	1	1	0	<table border="1" style="border-collapse: collapse; width: 50px; height: 50px;"> <tr><td colspan="2"></td><td colspan="2">z</td></tr> <tr><td colspan="2"></td><td>0</td><td>1</td></tr> <tr><td>x</td><td>y</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table> 3 variables			z				0	1	x	y	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	0	<table border="1" style="border-collapse: collapse; width: 50px; height: 50px;"> <tr><td colspan="2"></td><td colspan="2">z t</td></tr> <tr><td colspan="2"></td><td>00</td><td>01</td></tr> <tr><td>x</td><td>y</td><td>00</td><td>01</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table> 4 variables			z t				00	01	x	y	00	01	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	0
x	y																																																																			
0	0																																																																			
0	1																																																																			
1	1																																																																			
1	0																																																																			
		z																																																																		
		0	1																																																																	
x	y	0	0																																																																	
0	0	0	1																																																																	
0	1	1	1																																																																	
1	1	1	0																																																																	
1	0	0	0																																																																	
		z t																																																																		
		00	01																																																																	
x	y	00	01																																																																	
0	0	0	1																																																																	
0	1	1	1																																																																	
1	1	1	0																																																																	
1	0	0	0																																																																	

Dans un tel tableau, on s'intéresse aux pavés rectangulaires de 1, 2, 4, 8 ou 16 cases contenant 1 ; à chaque pavé correspond un monome formé uniquement des variables qui ne changent pas au sein du pavé, chaque variable étant complémentée ou non selon que la valeur de cette variable est 0 ou 1 pour ce pavé. Pour constituer les pavés, il faut considérer comme voisines les deux extrémités horizontales ainsi que les deux extrémités verticales de la table.



Pour obtenir une expression simplifiée de la fonction, il faut chercher à couvrir tous les 1 par des pavés les plus gros possibles. La fonction est alors la somme des monomes correspondants.

Un même 1 peut être recouvert par plusieurs pavés, et il faut le faire si ceci donne lieu à des pavés plus gros : au plus un pavé est gros, au plus il est simple car il correspond à un monome avec moins de variables.

On n'est pas obligé de prendre en compte tous les pavés possibles, il suffit que tous les 1 soient couverts.

Les tableaux de Karnaugh peuvent aussi être utilisés pour 5 et 6 variables :

- Pour 5 variables, le tableau est formé de deux tableaux juxtaposés, et il faut considérer comme voisines les cases symétriques par rapport à la médiane verticale.
- Pour 6 variables, le tableau est formé de 4 sous-tableaux, et il faut considérer comme voisines les cases symétriques par rapport aux deux médianes (horizontale et verticale) .

5 variables								6 variables											
z	t	u	x	y	z	t	u	x	y	z	t	u	v	x	y	z	t	u	v
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	1	0	0	0	1	0	0	1	1	0	0	1	0	1	0	
0	0	0	1	0	0	0	1	0	1	0	1	0	1	0	1	1	0	1	
0	0	1	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1	0	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	1	0	0	1	0	0	0	1	0	0	1	1	0	0	1	0	1	0	
0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	1	1	0	1	
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	0	0	0	1	0	0	1	1	0	0	1	0	1	0	
1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	1	1	0	1	
1	0	1	0	0	0	1	0	0	1	1	0	1	0	1	0	1	1	0	
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	0	0	1	0	0	0	1	0	0	1	1	0	0	1	0	1	0	
1	1	0	1	0	0	0	0	0	1	0	1	0	1	0	1	1	0	1	
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Au-delà de 6 variables, la méthode devient impraticable car la feuille de papier commence à manquer sérieusement de dimensions.

### Cas des fonctions incomplètement définies

Souvent la fonction n'est pas complètement définie, c'est-à-dire qu'il existe des valeurs des variables pour lesquelles la valeur de la fonction n'est pas spécifiée.

Cela se traduit par des tables incomplètes. Lorsqu'on utilise un tableau de Karnaugh, on place un  $\phi$  dans les cases où la fonction n'est pas définie et on s'autorise à utiliser ces  $\phi$  pour former les plus gros pavés possibles; si un  $\phi$  est recouvert par un pavé, la fonction complétée vaudra 1 en ce point, sinon elle vaudra 0, mais c'est sans importance puisque l'on a le choix.

table d'une fonction incomplète							
z	t	u	x	y	z	t	x.y
0	0	0	0	1	1	0	/x.y
0	0	1	0	1	0	1	/z
0	1	0	1	0	0	1	
0	1	1	1	1	1	0	
1	0	0	1	1	1	1	
1	0	1	0	1	1	0	

Dans la pratique, les fonctions incomplètement définies se rencontrent lorsque l'on a, par nature du problème posé, des configurations de valeurs impossibles pour les variables, c'est-à-dire que l'on sait par avance ne pas pouvoir se produire. C'est généralement le cas lorsque les variables bolléennes servent à coder des valeurs d'un autre domaine, et que tous les codes possibles ne sont pas attribués ; par exemple l'ensemble des trois couleurs, bleu, blanc et rouge, codé sur deux variables booléennes : une des 4 configurations possibles n'est pas utilisée.

**Exercice 1**

Paul dit : "je vais au cinéma s'il pleut ou s'il ne pleut pas et que le film est intéressant".

- Que fait Paul si le film est intéressant ?
- Donner une forme plus simple à l'énoncé précédent . Indiquer le théorème qui formalise cette simplification.

**Exercice 2**

Appliquer les théorèmes de De Morgan pour trouver une expression du complément de  $/x.y + x./y$

**Exercice 3**

Utiliser les théorèmes de De Morgan pour trouver une expression équivalente à  $x+y$  qui n'utilise pas l'opérateur "+".

Faire de même pour trouver une expression équivalente à  $x.y$  qui n'utilise pas le ":".

**Exercice 4**

Démontrer quelques uns des théorèmes T1...T7 ; procéder dans l'ordre où ils sont donnés, car la démonstration d'un de ces théorèmes utilise parfois un théorème qui le précède. Cet exercice, qui peut paraître sans intérêt, permet de s'habituer rapidement aux formules booléennes.

**Exercice 5**

A l'aide d'un tableau de Karnaugh, simplifier les expressions suivantes :

$$\begin{aligned} f_1(x,y,z) &= x.y + /x.z + y.z \\ f_2(x,y,z,t) &= x./z./t + x.t + /y.z + z.t \end{aligned}$$

**Exercice 6**

On considère le codage suivant, sur deux composantes booléennes, de quantités pouvant valoir 0, 1 ou 2 :

$$0 \leftrightarrow 0\ 0 \quad 1 \leftrightarrow 0\ 1 \quad 2 \leftrightarrow 1\ 0$$

On a deux variables de cette sorte  $X=x_1x_0$  et  $Y=y_1y_0$ .

Déterminer une fonction booléenne qui indique si  $X \geq Y$  (avec la relation d'ordre  $0 < 1 < 2$  ).

# INTERPRETATIONS DES CHAINES DE BITS

---

## 1 - Notions de représentation et d'interprétation

Les machines digitales travaillent sur des chaînes de bits de taille fixe :  $b_{n-1} b_{n-2} \dots b_0$ . Les tailles les plus courantes sont 4, 8, 16, 32 et 64 bits. Sur ces chaînes de bits, on peut représenter n'importe quelle sorte de valeurs :

- des nombres : 0, 1, 2, ...
- des caractères : "A", "B", "7", ";" ...
- des instructions (ordres donnés à une machine)
- des couleurs, des formes, etc...

Il suffit de définir une représentation, un code, pour chaque valeur. Sur une chaîne de  $n$  bits, on peut obtenir  $2^n$  configurations distinctes, permettant de coder  $2^n$  valeurs différentes.

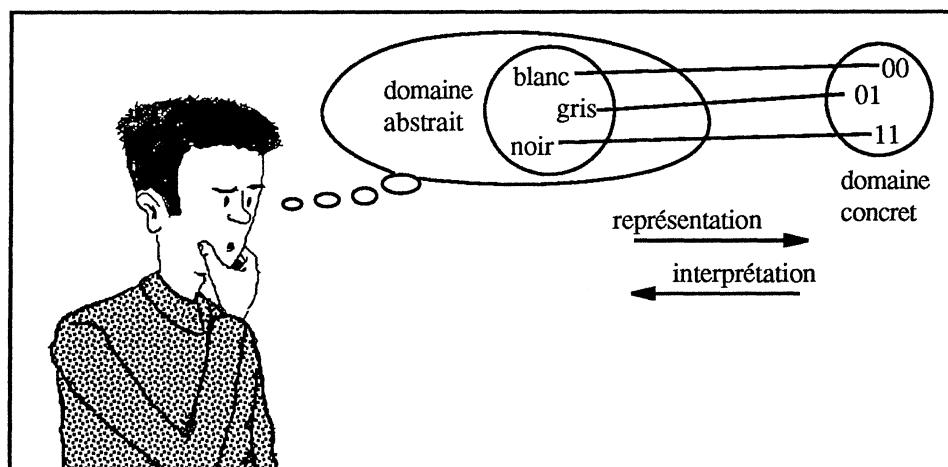
Exemple : sur deux bits, on peut représenter l'ensemble des trois couleurs {blanc, gris, noir} selon le codage suivant :

$$\text{blanc} \leftrightarrow 00 \quad \text{gris} \leftrightarrow 01 \quad \text{noir} \leftrightarrow 11$$

Il y a deux sortes de choses différentes dans un tel procédé :

D'une part il y a les chaînes de bits, qui sont des choses concrètes : elles sont effectivement manipulées par les machines.

D'autre part il y a les valeurs représentées, qui sont des choses abstraites : il n'y a rien de tel dans les machines.

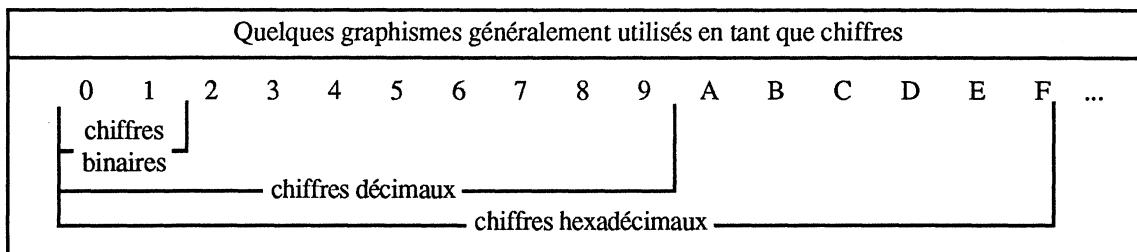


Une représentation, ou un codage, est une correspondance que l'on établit entre les choses abstraites et les choses concrètes. Pour toute représentation il y a une correspondance inverse, qui va des choses concrètes vers les choses abstraites : on appelle cela une interprétation.

Représentations et interprétations ne sont bien évidemment pas limitées aux chaînes de bits : sitôt que l'on dispose de marques concrètes bien discernées, on peut les utiliser pour représenter des choses plus abstraites ; l'exemple le plus connu, car on le pratique tous les jours, est la représentation des nombres à l'aide de chiffres que nous allons rappeler ci-dessous.

### Exemple : représentation des nombres entiers dans diverses bases

Dans un système de numération basé on utilise une collection de symboles appelés "chiffres". Ces symboles sont des choses concrètes, des marques graphiques.



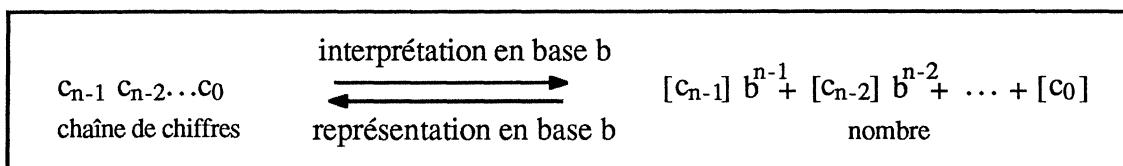
A chacun des chiffres on associe une valeur prise parmi les premiers nombres entiers. Ces valeurs sont des choses abstraites (personne n'a jamais touché un nombre, c'est une idée organisée en une théorie qui s'appelle l'arithmétique).

chiffre	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	...
valeur associée	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...

Dans ce qui suit, nous noterons  $[c]$  la valeur associée au chiffre  $c$  ; par exemple :  $[3]=3$

Pour représenter les nombres, on utilise des chaînes de chiffres, c'est-à-dire que l'on écrit des chiffres les uns à la suite des autres :  $c_{n-1} c_{n-2} \dots c_0$ .

Etant donnés une base  $b$  et une chaîne de chiffres  $c_{n-1} c_{n-2} \dots c_0$  ayant tous une valeur associée inférieure à la base,  $[c_i] < b$ , l'interprétation en base  $b$  de cette chaîne est donnée par la formule suivante :



Ainsi, la chaîne de chiffres '43' interprétée en base 10 conduit au nombre

$$[4] \times 10 + [3] = 4 \times 10 + 3 = 43$$

Et la même chaîne interprétée en base 16 conduit au nombre

$$[4] \times 16 + [3] = 4 \times 16 + 3 = 67$$

Nous conviendrons de noter  $[xxx]_b$  l'interprétation en base  $b$  de la chaîne de chiffres  $xxx$ , et de noter  $_b[n]$  la représentation en base  $b$  du nombre  $n$ .

On a l'habitude de noter les nombres par leur représentation décimale, et nous garderons cette habitude culturelle ; cependant, étant donné que nous traitons ici de la technique même de représentation des nombres, nous devons noter de façon différentes les nombres et les chaînes de chiffres car si nous notons pareillement des choses différentes, notre texte ne sera qu'un charabia incompréhensible ; ci-dessus nous avons utilisé l'italique pour distinguer les nombres ; dorénavant nous noterons les nombres normalement, et les chaînes de chiffres seront placées entre apostrophes :

chaînes de chiffres : '1' 'B' '1492' 'A05B8' '10011100101'

numbers : 1 (unité) 32 (nombre de cartes dans un jeu) 1515 (date de la bataille de Marignan)

Exemples d'énoncés exacts :

$$[358]_{10} = 358$$

$$2[12] = '1100'$$

$$[101]_2 = 5$$

$$16[12] = 'C'$$

'358' est composé de trois chiffres

58 est pair

Exemples d'énoncés faux :

$$[1001]_2 = 6$$

(ce n'est pas vrai, c'est 9)

'38' est composé de trois chiffres (ce n'est pas vrai, il y en a deux)

Exemples d'énoncés absurdes :

$$[80]_{16} = 128$$

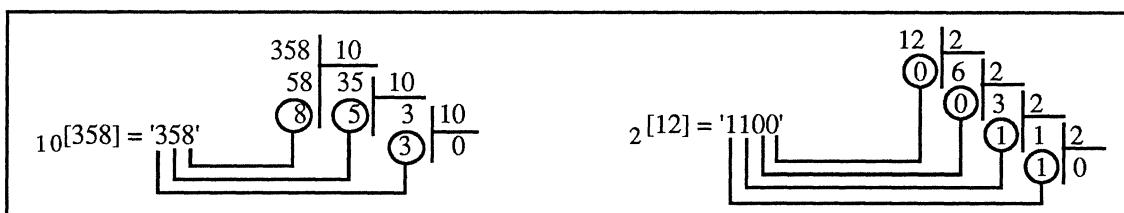
'44' est pair

542 est composé de trois chiffres

80 n'est pas une chaîne de chiffres, on ne saurait "l'interpréter en base 16"; être pair ou impair est une propriété des nombres, et non des chaînes de chiffres; les nombres tels que 542 ne sont pas "composés" de chiffres.

## Changements de base

Pour obtenir de la représentation d'un nombre en base  $b$ , on utilise la méthode suivante : on divise le nombre par la base ce qui donne comme reste la valeur du chiffre de poids faible, puis on divise à nouveau le quotient, ce qui donne comme reste la valeur du chiffre de poids supérieur, et on continue ainsi jusqu'à obtenir un quotient nul.



Pour passer d'une base à une autre, c'est-à-dire trouver la représentation en base  $b_2$  d'un nombre dont on connaît la représentation en base  $b_1$ , on cherche d'abord le nombre représenté puis on applique la méthode ci-dessus.

Remarque : pour appliquer la méthode, il faut pratiquer la division ; lorsque nous pratiquons la division, nous sommes des machines et nous avons besoin de travailler sur des chaînes de chiffres (des marques concrètes) ; étant donné que nous sommes programmés depuis notre enfance pour réaliser la division sur la représentation décimale, nous sommes amenés à "passer par l'intermédiaire de la représentation décimale", mais ceci est pure coïncidence (par exemple, on pourrait faire la division avec des cailloux).

Il y a des cas où la transformation est très simple : c'est quand l'une des bases est une puissance de l'autre base :  $b_2 = b_1^p$  ; dans ce cas il suffit de regrouper les chiffres par paquets de taille  $p$  dans la représentation en base  $b_1$ , et de connaître la correspondance entre un tel groupement et le chiffre correspondant dans la base  $b_2$ .

Un exemple typique sont la représentation binaire et la représentation hexadécimale : à chaque chiffre hexadécimal correspond un paquet de 4 chiffres binaires, et le passage de la base 2 à la base 16 est un simple transcodage qui ne nécessite aucun calcul :

correspondance paquet de 4 chiffres binaires ↔ chiffre hexadécimal															
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

$[1100 \ 1000 \ 0101 \ 0111]_2$   
 $= [ \ C \quad 8 \quad 5 \quad 7 \ ]_{16}$

## 2 - Quelques représentations usuelles

Parmi les sortes de valeurs les plus fréquemment représentées dans les machines, il y a :

- les caractères, collection de symboles typographiques,
- les nombres entiers, naturels ou relatifs.

Etudions brièvement leurs représentations usuelles sur chaînes de bits de longueur fixe.

### 2.1 - Représentation des caractères

Le système de représentation le plus universel actuellement est l'ASCII ("American Standard Code for Information Interchange"), qui représente sur sept bits un jeu de 128 caractères.

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

Parmi les caractères représentés, certains ne sont pas des symboles graphiques, mais concernent des ordres de positionnement ou bien des signalisations ; dans la table, ces caractères sont désignés par un nom de baptême, par exemple :

CR ("Carriage Return") positionnement en début de ligne (retour chariot).

LF ("Line Feed") positionnement sur la ligne suivante.

VT ("Vertical tabulation") positionnement sur la ligne précédente.

BS ("Back Space") recul d'une position.

FF ("Forward feed") avance le curseur d'une position.

BEL ("Bell") signal sonore.

La représentation ASCII présente une certaine cohérence entre la structure des codes et celle des caractères, par exemples :

- Les caractères numériques, "0" ... "9", ont des codes qui se suivent dans l'ordre des valeurs numériques que l'on attribut habituellement à ces caractères.
- Les caractères alphabétiques, "A" ... "Z", ont des codes qui se suivent dans l'ordre alphabétique.
- Le passage d'une majuscule à la minuscule correspondante se fait en modifiant un seul bit, le bit 5.

## 2.2 - Représentation des entiers naturels

Les entiers naturels sont tous les entiers positifs, de zéro à l'infini.

### Représentation binaire :

Sur  $n$  bits, on ne peut représenter que  $2^n$  valeurs distinctes ; La représentation la plus fréquente est la représentation en base 2 (on dit parfois "binaire pur", ou plus simplement "binaire"). Ceci permet de représenter les entiers de 0 à  $2^n-1$ .

Représentation binaire sur 8 bits des entiers de 0 à 255							
0	1	2	3	...	254	255	
00000000	00000001	00000010	00000011		11111110	11111111	

## 2.3 - Représentation des entiers relatifs

### Représentation en complément à 2 :

La représentation habituelle est la représentation en complément à  $2^n$  (on dit également "en complément à 2", la taille  $n$  de la chaîne étant sous-entendue). Elle permet de représenter, sur  $n$  bits, les nombres de l'intervalle  $[-2^{n-1}, 2^{n-1}-1]$  de la manière suivante :

Pour les nombres positifs (de 0 à  $2^{n-1}-1$ ), c'est la représentation binaire du nombre.

Pour les nombres négatifs (de  $-2^{n-1}$  à -1), c'est la représentation binaire du nombre augmenté de  $2^n$ , c'est-à-dire la représentation binaire de  $2^n$  moins la valeur absolue du nombre, puisque le nombre est négatif.

Nous conviendrons de noter :

$_{2C}[k]$  la chaîne de bits qui est la représentation en complément à 2 du nombre relatif  $k$ .

$[x]_{2C}$  le nombre relatif qui est l'interprétation en complément à 2 de la chaîne de bits  $x$ .

Une définition précise de la représentation en complément à  $2^n$  peut alors être donnée ainsi :

$$\begin{aligned} \text{pour } 0 \leq k < 2^{n-1} : & \quad {}_{2C}[k] = {}_2[k] \\ \text{pour } -2^{n-1} \leq k < 0 : & \quad {}_{2C}[k] = {}_2[2^n + k] \end{aligned}$$

Exemples sur 8 bits :

$${}_{2C}[34] = {}_2[34] = 0010\ 0010$$

$${}_{2C}[-34] = {}_2[256 - 34] = {}_2[222] = 1101\ 1110$$

### Interprétation en complément à 2 :

L'interprétation en complément à 2 d'une chaîne de bits s'obtient ainsi :

Le bit  $x_{n-1}$  (de rang  $n-1$ , ou bit de "poids fort") est significatif du signe du nombre représenté (on l'appelle le "bit de signe") :

$x_{n-1} = 0$  : le nombre est positif ; il est égal à l'interprétation binaire de la chaîne  $x$ .

$x_{n-1} = 1$  : le nombre est négatif ; sa valeur absolue vaut à  $2^n$  moins l'interprétation binaire de la chaîne  $x$ .

Exemples sur 8 bits :

$$\text{bit de signe "positif"} \quad [0010\ 0010]_{2c} = [0010\ 0010]_2 = 34$$

$$\text{bit de signe "négatif"} \quad [1101\ 1110]_{2c} = [1101\ 1110]_2 - 256 = 222 - 256 = -34$$

Correspondance entre interprétation binaire et interprétation en complément à 2 pour des chaînes de 4 bits	chaîne de bits x	interpret. binaire $[x]_2$	interpret. complément $[x]_{2c}$
	0000	0	+ 0
	0001	1	+ 1
	0010	2	+ 2
	0011	3	+ 3
	0100	4	+ 4
	0101	5	+ 5
	0110	6	+ 6
	0111	7	+ 7
	1000	8	- 8
	1001	9	- 7
	1010	10	- 6
	1011	11	- 5
	1100	12	- 4
	1101	13	- 3
	1110	14	- 2
	1111	15	- 1

### 3 - Les opérateurs sur chaînes de bits

Pour effectuer des calculs, les machines disposent de quelques opérateurs élémentaires, au comportement parfaitement défini, qui élaborent un résultat à partir d'opérandes.

Opérandes et résultats sont de chaînes de bits.

Il faut à ce sujet se méfier des noms qu'on leur donne ; certains noms par exemple évoquent l'arithmétique : "additionneur", "multiplieur"... ; on les appelle ainsi parce que une interprétation précise des opérandes conduit à des nombres, et la même interprétation du résultat conduit à un nombre qui est la somme ou le produit de ces nombres ; il n'y a pas de nombres dans les machines.

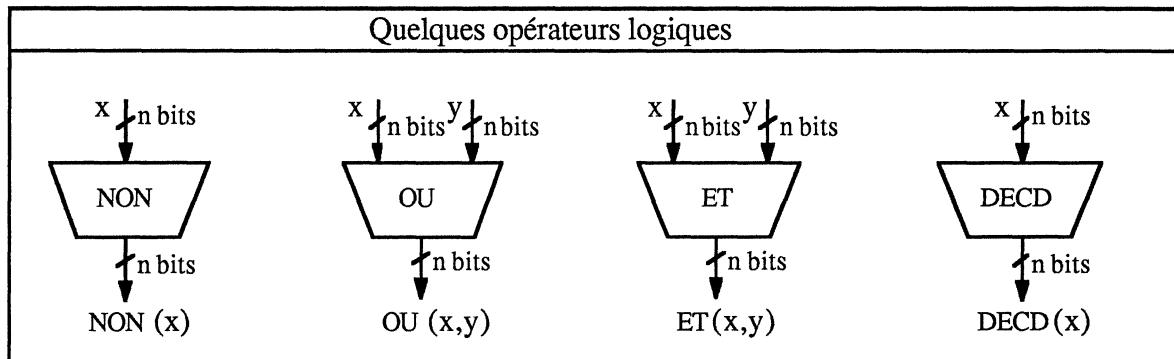
De plus il ne faut jamais oublier que représentation et interprétation sont des correspondances établies par l'esprit humain ; une chaîne de bit n'est pas "en soi" une représentation de nombre en binaire ou en complément , c'est une chaîne de bits, sans plus :

'1110' n'est pas en soi ni "14", ni "-2", ni "blanc", ni "noir", ni "gris" :

*c'est l'interprétation qu'on lui donne qui est l'une ou l'autre de ces valeurs abstraites.*

### 3.1 - Opérateurs logiques

On appelle ainsi les opérations qui ne font allusion à aucune interprétation particulière des chaînes de bit : ce sont les opérations booléennes bit à bit, les décalages ...



$$\text{NON } (X) = /x_3 /x_2 /x_1 /x_0$$

$$\text{OU } (X,Y) = x_3+y_3 \ x_2+y_2 \ x_1+y_1 \ x_0+y_0$$

$$\text{ET } (X,Y) = x_3.y_3 \ x_2.y_2 \ x_1.y_1 \ x_0.y_0$$

$$\text{DECD } (X) = 0 \ x_3 \ x_2 \ x_1$$

$$\text{DEC}G \ (X) = x_2 \ x_1 \ x_0 \ 0$$

complément bit à bit

ou bit à bit

et bit à bit

décalage d'une position à droite (complété par 0)

décalage d'une position à gauche (complété par 0)

#### Exemples

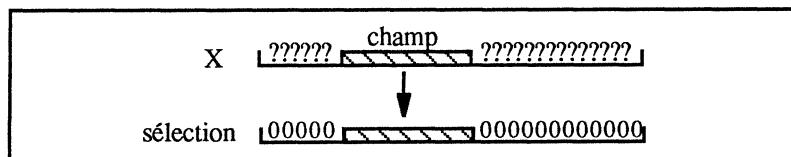
$X = 10110010$	$Y = 01001010$	$X = 10110010$
$\text{NON } (X) = 01001101$	$\text{OU } (X,Y) = 11111010$	$Y = 01001010$
		$\text{ET } (X,Y) = 00000010$

$$\text{DECD } (X) = 01011001$$

#### Exemple d'utilisation : sélection et fusion de champs

Un "champ" est un ensemble de positions au sein d'une chaîne.

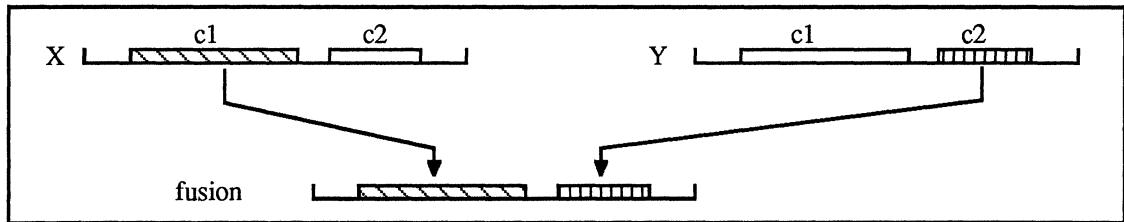
La "sélection de champ" dans une chaîne de bits X consiste à conserver la valeur des bits de X dans les positions du champ et placer une valeur fixe (généralement 0) dans les autres positions :



Le procédé de sélection habituel est le masquage, avec forçage des bits hors champ à 0 : le masque est une chaîne contenant des 1 dans les positions du champ et des 0 dans les autres. L'application de l'opérateur ET donne en résultat des 0 dans toutes les positions extérieures au champ, et conserve les valeurs de X à l'intérieur.

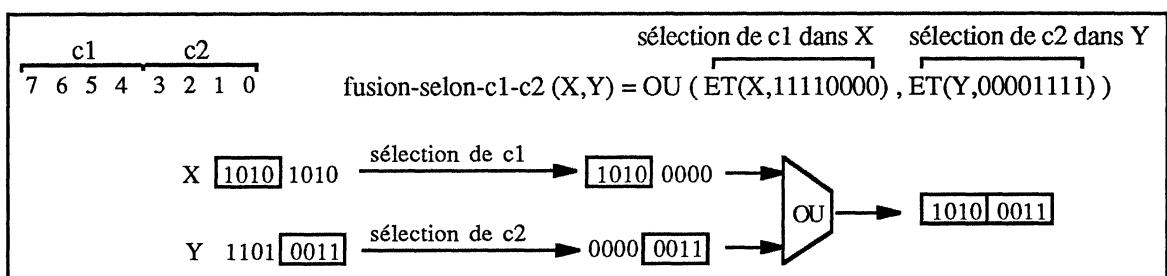
$$\begin{array}{ll} X = & x_{n-1} \dots x_{j+1} \ x_j \ \dots \ x_i \ x_{i-1} \ \dots \ x_0 \\ Y = & 0 \ \dots \ 0 \ \ 1 \ \ \dots \ 1 \ \ 0 \ \ \dots \ 0 \\ \text{ET } (X,Y) = & 0 \ \dots \ 0 \ \ x_j \ \ \dots \ x_i \ \ 0 \ \ \dots \ 0 \end{array}$$

La "fusion" de champs disjoints  $c_1$  et  $c_2$  pris dans deux chaînes  $X$  et  $Y$  est la construction d'une chaîne ayant respectivement la valeur des bits de  $X$  dans le champ  $c_1$  et de  $Y$  dans le champ  $c_2$  (les bits n'appartenant ni aux positions  $c_1$  ni aux positions  $c_2$  pouvant être quelconques).



La fusion de champs peut être faite ainsi :

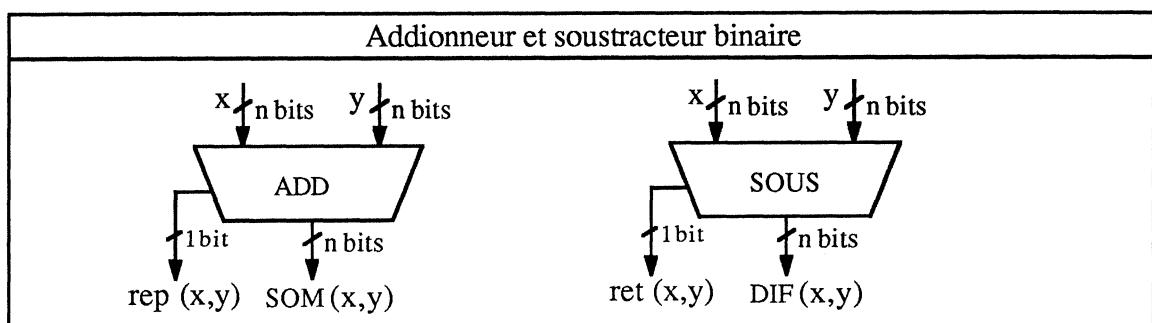
- (1) on réalise les sélections des champs  $c_1$  de  $X$  et  $c_2$  de  $Y$ ,
- (2) on réalise le OU de ces deux sélections.



### 3.2 - Opérateurs arithmétiques

Les opérateurs arithmétiques réalisent sur les chaînes de bits les opérations qui s'interprètent bien comme une opération arithmétique sur des nombres : addition, soustraction ...

Ce que fait un tel opérateur est parfaitement défini, à l'aide de tables et de règles de construction qui définissent le résultat en fonction des opérandes : mais opérandes et résultats sont bien évidemment des chaînes de bits, et non des nombres : ce n'est qu'à travers une interprétation précise, généralement le binaire ou le complément à 2, que "l'additionneur fait l'addition et le soustracteur fait la soustraction".



Le résultat des opérateurs ADD et SOUS est structuré en deux composantes :

SOM : somme $(n \text{ bits})$	DIF : différence $(n \text{ bits})$
rep : report de rang $n$ (1 bit)	ret : retenue de rang $n$ (1 bit)

## Algorithmes d'addition et de soustraction binaire

Le résultat des opérateurs ADD et SOUS se calcule selon un algorithme similaire à celui que l'on pratique habituellement sur les représentations décimales, la seule différence étant que les tables d'addition et de soustraction sont plus petites (2 chiffres au lieu de 10).

### Addition

Table d'addition binaire					
$x_i$	$y_i$	$r_i$	$r_{i+1}$	$s_i$	
bit de rang $i$ de $x$			0	0	
bit de rang $i$ de $y$		0	0	1	
report de rang $i$		0	1	0	
report de rang $i+1$		0	1	1	0
bit de rang $i$ de la somme		1	0	0	1
		1	0	1	0
		1	1	0	1
		1	1	1	1

On applique itérativement cette table, en commençant par les chiffres de rang 0 (avec  $r_0 = 0$ ). Le résultat de ADD est une chaîne de  $n+1$  bits que l'on sépare en deux composantes :

SOM sur  $n$  bits :  $s_{n-1} \dots s_0$  (la somme),      rep sur 1 bit : c'est le bit  $r_n$  (le report)

Exemples :	interprétations binaires	interprétations binaires									
$x = 10110010$ $y = 01001010$ $ADD(x,y) = \boxed{0} \underline{11111100}$ rep      SOM	<table border="1"> <tr><td>178</td></tr> <tr><td>74</td></tr> <tr><td>—</td></tr> <tr><td>252</td></tr> </table>	178	74	—	252	$x = 11110010$ $y = 01001010$ $ADD(x,y) = \boxed{1} \underline{00111100}$ rep      SOM	<table border="1"> <tr><td>242</td></tr> <tr><td>74</td></tr> <tr><td>—</td></tr> <tr><td>60</td></tr> </table>	242	74	—	60
178											
74											
—											
252											
242											
74											
—											
60											

### Soustraction

Table de soustraction binaire					
$x_i$	$y_i$	$r_i$	$r_{i+1}$	$d_i$	
bit de rang $i$ de $x$		0	0	0	
bit de rang $i$ de $y$		0	0	1	1
retenue de rang $i$		0	1	0	1
retenue de rang $i+1$		0	1	1	0
bit de rang $i$ de la différence		1	0	0	1
		1	0	1	0
		1	1	0	0
		1	1	1	1

On applique itérativement cette table, en commençant par les chiffres de rang 0 (avec  $r_0 = 0$ ). Le résultat de SOUS est une chaîne de  $n+1$  bits que l'on sépare en deux composantes :

DIF sur  $n$  bits :  $d_{n-1} \dots d_0$  (la différence),      ret sur 1 bit : c'est le bit  $r_n$  (la retenue)

Exemples :	interprétations binaires	interprétations binaires									
$x = 10110010$ $y = 01001010$ $SOUS(x,y) = \boxed{0} \underline{01101000}$ ret      DIF	<table border="1"> <tr><td>178</td></tr> <tr><td>74</td></tr> <tr><td>—</td></tr> <tr><td>104</td></tr> </table>	178	74	—	104	$x = 01001010$ $y = 10110010$ $SOUS(x,y) = \boxed{1} \underline{10011000}$ ret      DIF	<table border="1"> <tr><td>74</td></tr> <tr><td>178</td></tr> <tr><td>—</td></tr> <tr><td>152</td></tr> </table>	74	178	—	152
178											
74											
—											
104											
74											
178											
—											
152											

## Interprétation binaire de ADD et SOUS

Si on interprète en binaire les opérandes et le résultat, les opérations ADD et SOUS réalisent à peu près l'addition et la soustraction. Cependant pas exactement, car :

La somme de deux nombres de l'intervalle  $[0, 2^n - 1]$  appartient à un intervalle plus grand, l'intervalle  $[0, 2^{n+1} - 2]$ . Certains résultats ne sont donc pas représentables sur  $n$  bits.

De même, la différence appartient à l'intervalle  $[-(2^n - 1), +2^n - 1]$  : elle peut être négative, auquel cas elle n'est pas représentable sur  $n$  bits.

Les indicateurs *rep* et *ret* permettent justement de détecter ces situations et de préciser le sens que l'on peut attribuer aux résultats SOM et DIF :

- rep* = 0 : la somme est représentable en binaire sur  $n$  bits  
le résultat est "correct" :  $[\text{SOM}]_2 = [x]_2 + [y]_2$
- rep* = 1 : la somme n'est pas représentable en binaire sur  $n$  bits.  
le résultat est "incorrect" :  $[\text{SOM}]_2 = [x]_2 + [y]_2 - 2^n$
- ret* = 0 : la différence est représentable en binaire sur  $n$  bits  
le résultat est "correct" :  $[\text{DIF}]_2 = [x]_2 - [y]_2$
- ret* = 1 : la différence n'est pas représentable en binaire sur  $n$  bits.  
le résultat est "incorrect" :  $[\text{DIF}]_2 = [x]_2 - [y]_2 + 2^n$

Le report et la retenue indiquent un dépassement de capacité pour l'interprétation binaire

(Angl. report = "carry", retenue = "borrow")

Le report peut également être considéré comme le chiffre de rang  $n$  du résultat : ainsi considéré, le résultat total sur  $n+1$  bits est toujours correct en interprétation binaire.

Le report peut également être considéré comme un chiffre de rang  $n$  qu'il faudrait retrancher de DIF pour obtenir le résultat correct.

Si on considère que ADD et SOUS réalisent l'addition et la soustraction modulo  $2^n$ , les résultats SOM et DIF sont toujours corrects ; autrement dit les propriétés suivantes sont toujours vérifiées :

$$\begin{aligned} [\text{SOM}(x,y)]_2 &= ([x]_2 + [y]_2) \text{ modulo } 2^n \\ [\text{DIF}(x,y)]_2 &= ([x]_2 - [y]_2) \text{ modulo } 2^n \end{aligned}$$

## Interprétation en complément à 2 de l'opérateur ADD

L'opérateur ADD fonctionne également bien pour la représentation en complément à 2. C'est d'ailleurs ce qui fait le succès de la représentation en complément ; un même opérateur sait faire l'addition pour deux interprétations différentes des chaînes de bits :

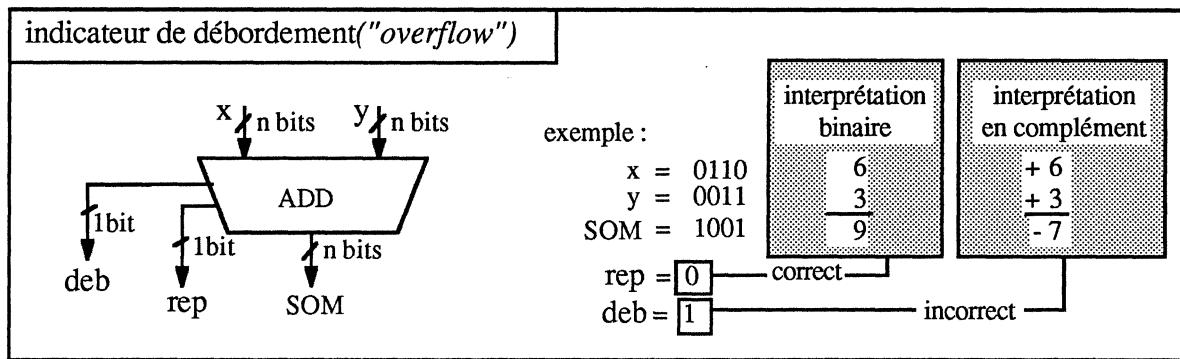
- l'interprétation binaire,
- l'interprétation en complément à 2.

La somme de deux nombres de l'intervalle  $[-2^{n-1}, 2^{n-1}-1]$  est comprise entre  $-2^n$  et  $+2^{n-2}$  : elle n'est donc pas toujours représentable en complément à 2 sur  $n$  bits. Il est facile de vérifier que lorsque la somme est représentable, l'opérateur ADD donne un résultat correct. Autrement dit, on a la propriété suivante :

$$-2^{n-1} \leq [x]_{2c} + [y]_{2c} < 2^{n-1} \Rightarrow [\text{SOM}(x,y)]_{2c} = [x]_{2c} + [y]_{2c}$$

Par contre le résultat de ADD est incorrect si la somme des deux nombres représentés sort de l'intervalle représentable : on dit dans ce cas que l'on a un débordement (Angl. "overflow").

Le report n'apporte aucune information au sujet du débordement, il est sans signification pour la représentation en complément. Pour distinguer les situations de débordement, l'opérateur ADD doit offrir un indicateur supplémentaire, *deb*, qui joue un rôle similaire à celui du report, mais pour l'interprétation en complément :



$deb = 0$  : le résultat est correct pour l'interprétation en complément :

$$[SOM(x,y)]_{2c} = [x]_{2c} + [y]_{2c}$$

$deb = 1$  : le résultat est incorrect pour l'interprétation en complément.

l'indicateur *deb* peut être obtenu ainsi :

$deb = 1$  si les deux opérandes ont le même bit de signe et si *SOM* a un bit de signe différent,

$deb = 0$  sinon.

Le débordement indique un dépassement de capacité pour l'interprétation en complément à 2

L'exemple ci-dessus illustre l'addition de 0110 et 0011 ; le résultat *SOM* est 1001 :

Dans le cadre de l'interprétation binaire, les opérandes s'interprètent en 6 et 3, et le résultat *SOM* s'interprète en 9 : le résultat est correct dans cette interprétation, et cela est indiqué par *rep* = 0.

Dans le cadre de l'interprétation en complément, les opérandes s'interprètent en +6 et +3, et le résultat *SOM* s'interprète en -7 : le résultat est incorrect dans cette interprétation, et cela est indiqué par *deb* = 1.

En résumé, l'opérateur ADD fait un certain travail, parfaitement défini, sur les chaînes de bits . Le travail de la machine s'arrête là. C'est nous qui attribuons des interprétations aux chaînes de bits ; le choix de l'interprétation dépend de nos besoins :

- si nous avons choisi l'interprétation binaire, il faut se soucier de l'indicateur *rep* pour savoir si le résultat est correct dans cette interprétation,
- si nous avons choisi l'interprétation en complément à 2, il faut se soucier de l'indicateur *deb*.

l'indicateur *rep* est sans signification particulière pour l'interprétation en complément à 2, et l'indicateur *deb* est sans signification particulière pour l'interprétation en binaire.

**Exercice 1**

Donner les représentations du nombre 55 dans les bases 2, 10 et 16.

**Exercice 2**

Le nombre représenté en base 5 par la chaîne '321' est-il pair ou impair ?

**Exercice 3**

$[303]_5$  est-il multiple de  $[111]_3$  ?

**Exercice 4**

Donner les représentations des nombres 25, 39 et 256 dans les bases 2 et 16.

**Exercice 5**

Donner la représentation binaire de  $[FE]_{16}$ , de  $[101]_{16}$  et de  $[384A]_{16}$ .

**Exercice 6**

1) Donner, quand c'est possible, la représentation binaire sur huit bits des valeurs :

12	256	-1	100	64
----	-----	----	-----	----

2) Donner l'interprétation binaire de :

0000	1111	00001111	10100010	11111111
------	------	----------	----------	----------

**Exercice 7**

Donner la représentation en complément à 2 sur huit bits (complément à  $2^8$ ) des nombres suivants :      +5      -5      -4      +126      +127      -1      -128

**Exercice 8**

Donner l'interprétation en complément à 2 des chaînes de bits :

00000101	10000101	11000000	10110010
----------	----------	----------	----------

**Exercice 9**

Une technique simple pour trouver l'opposé d'un nombre en complément à 2 est connue sous la forme : "complémenter le nombre, puis lui ajouter 1".

Par exemple, sur 4 bits, pour +5 (0101), le complément est 1010 et en ajoutant 0001 on obtient 1011 qui est bien la représentation de -5.

Prise à la lettre, cette règle est une monstruosité : on ne sait pas "complémenter un nombre", et si on ajoute 1 à un nombre, on obtient un nombre, pas une chaîne de bits !

- 1) Exprimer cette règle sous une forme plus correcte.
- 2) Essayer de justifier cette règle. Exhiber un cas où elle s'avère innexacte.
- 3) Exprimer correctement la règle exacte.

**Exercice 10**

Parmi les caractères, il y a les lettres minuscules et les lettres majuscules.

Appelons "majusc" l'opérateur qui, étant donnée une lettre minuscule, donne en résultat la lettre majuscule correspondante, par exemple:

$$\text{majusc}("a") = "A" \quad \text{majusc}("k") = "K"$$

Appelons MAJUSC l'opérateur analogue sur les représentations ASCII (7 bits) des caractères.

- 1) Exprimer cet opérateur à l'aide des opérateurs logiques et de constantes.
- 2) Dans le même esprit, réaliser l'opérateur "majminusc" qui, étant donnée une lettre quelconque donne en résultat :

pour une minuscule, la majuscule correspondante  
pour une majuscule, la minuscule correspondante.

**Exercice 11**

On considère la représentation sur 2 bits des trois teintes {blanc, gris, noir} selon le codage :

$$\text{blanc} \leftrightarrow 00 \quad \text{gris} \leftrightarrow 01 \quad \text{noir} \leftrightarrow 11$$

Appelons "mélange" l'opération suivante sur les teintes :

x	y	mélange(x,y)	x	y	mélange(x,y)	x	y	mélange(x,y)
blanc	blanc	blanc	gris	blanc	gris	noir	blanc	gris
blanc	gris	gris	gris	gris	gris	noir	gris	gris
blanc	noir	gris	gris	noir	gris	noir	noir	noir

Appelons MELANGE l'opérateur analogue sur les chaînes de 2 bits représentant les teintes.

- 1) Exprimer MELANGE à l'aide des opérateurs sur 2 bits ET, OU et de constantes.
- 2) Imaginer, définir et réaliser des opérateurs :

NEGATIF, CONTRASTE, ECLAIRCI et ASSOMBRI

**Exercice 12**

Considérons des pièces de jeux d'échec positionnées sur l'échiquier. Chaque pièce est caractérisée par :

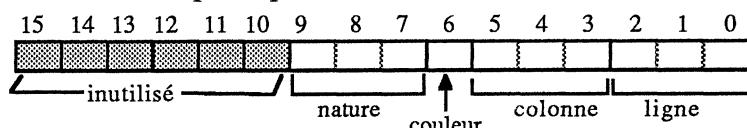
- sa nature : {pion, tour, cavalier, fou, roi, reine}
- sa couleur : {blanc, noir}
- sa ligne : {0,1,2,3,4,5,6,7}
- sa colonne : {0,1,2,3,4,5,6,7}

Exemple : <tour, blanc, 2, 5> est une tour blanche située ligne 2 colonne 5.

Appelons "prise" l'opérateur qui, étant données deux pièces-positionnées x et y délivre en résultat la pièce positionnée qui correspond à la prise de y par x, c'est à dire qui a la nature et la couleur de x et la position de y.

Exemple: prise (<tour,blanc,2,5>, <pion, noir,7,3>) = <tour, blanc,7,3>

On représente comme suit une pièce-positionnée sur 16 bits :



Exprimer, à l'aide des opérateurs logiques et de constantes sur 16 bits, l'opérateur PRISE analogue à "prise", mais travaillant sur les représentations.

**Exercice 13**

La méthode proposée pour réaliser la fusion de champs utilise les propriétés suivantes des opérateurs logiques :

$$x.0 = 0 \quad x.1 = x \quad x+0 = x$$

Trouver une méthode analogue pour réaliser la fusion de champs, mais qui utilise les propriétés duales des précédentes.

**Exercice 14**

Donner les résultats de ADD et SOUS pour les couples d'opérandes 8 bits suivants :

10100101	10100101	10100101	10000000
00010010	00010110	01100001	10000000

11111111	00000000		
00000001	00000001		

**Exercice 15**

- 1) Donner les résultats (SOM,rep,deb) de l'opérateur ADD pour les couples d'opérandes 4 bits suivants :

0011	1001	0010	1100
0010	0101	1010	1101

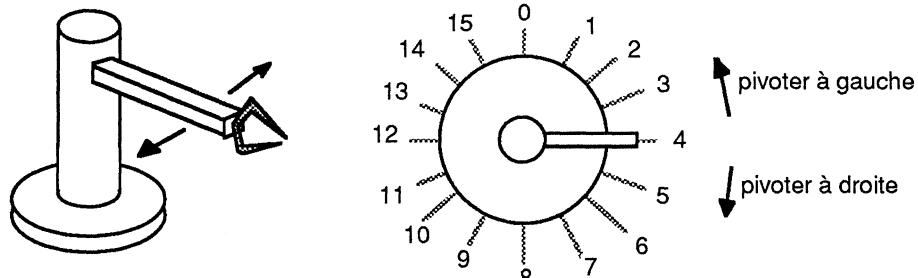
- 2) Donner l'interprétation binaire de ces opérandes et du résultat.
- 3) Donner l'interprétation en complément à 2 de ces opérandes et du résultat.

**Exercice 16**

On considère un bras monté sur un axe qui peut pivoter à gauche ou à droite, avec 16 positions possibles.

Appelons “pivotéAdroite( $p,n$ )” la position du bras lorsqu’il pivote à droite de  $n$  emplacements à partir de la position  $p$ , et “pivotéAgauche( $p,n$ )” sa position lorsqu’il pivote à gauche.

Par exemple: pivotéAdroite(14,5) = 3      pivotéAgauche(9,3) = 6



- 1) Choisir une représentation sur 4 bits pour les positions et les déplacements et exprimer, à l'aides des opérateurs ADD et SOUS sur 4 bits, les opérations représentant pivotéAdroite et pivotéAgauche sur ces représentations.
- 2) On ne dispose que des opérateurs sur chaines de 8 bits. Choisir une représentation sur 8 bits pour les positions et les emplacements et exprimer à nouveau les opérateurs précédents.

**Exercice 17**

On a donné précédemment une règle effective de calcul du débordement : il y a débordement ( $\text{deb}=1$ ) si, et seulement si, les opérandes  $x$  et  $y$  ont le même bit de signe et que la somme SOM a un bit de signe différent.

Une autre règle, plus généralement appliquée par les machines dit qu'il y a débordement si, et seulement si, dans l'algorithme d'addition binaire, le report de rang  $n-1$  est différent du report de rang  $n$ .

- 1) Vérifier cette règle sur quelques exemples.
- 2) Montrer l'équivalence des deux règles, en envisageant tous les cas possibles pour les bits de rang  $n-1$  des opérandes et de SOM, et pour les reports de rang  $n-1$  et de rang  $n$ .

**Exercice 18**

On a coutume de dire que le décalage à gauche d'une position binaire, DECG, réalise la multiplication par 2.

- 1) Montrer, à l'aide de la table d'addition binaire, que  $\text{DECG}(x) = \text{SOM}(x,x)$ .
- 2) Déterminer les limites de validité de la coutume pour la représentation binaire.
- 3) Déterminer les limites de validité de la coutume pour la représentation en complément.

**Exercice 19**

On dit souvent que le décalage à droite d'une position, DECD, réalise la division par 2.

- 1) Expérimenter la coutume.
- 2) Donner les limites de validité dans la représentation binaire, puis celle en complément (DECD introduit un 0 en poids fort).
- 3) On rencontre souvent dans les machines l'opérateur DECAD : “décalage arithmétique à droite”. Cet opérateur décale d'une position à droite, mais plutôt que d'introduire un 0 en poids fort, il y laisse le bit de poids fort de l'opérande. Expliquer l'intérêt de cet opérateur.

# LES CIRCUITS LOGIQUES

---

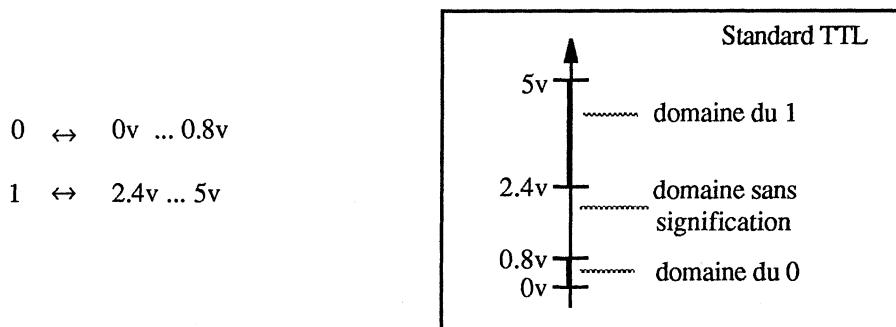
## 1 - Généralités

Les circuits logiques sont des appareillages électroniques\* destinés à manipuler des grandeurs binaires : le 0 et le 1.

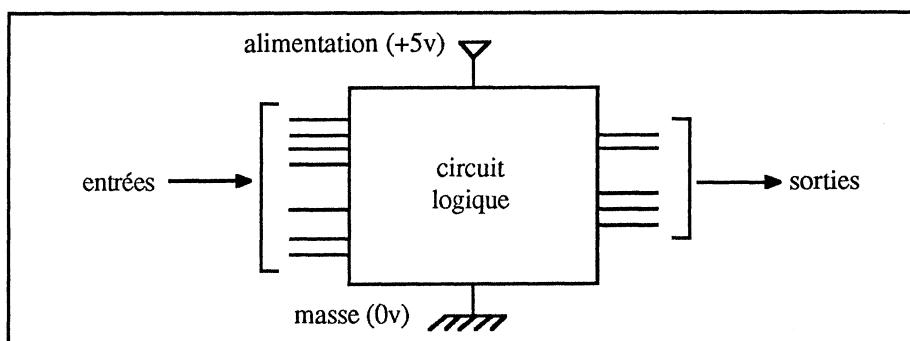
Dans un circuit logique, chaque variable logique est matérialisée par un conducteur et sa valeur logique par la tension électrique du conducteur. La correspondance entre les valeurs logiques et les tensions électriques est généralement :

0	$\leftrightarrow$	0v
1	$\leftrightarrow$	+5v

En fait, aucun appareillage physique n'est capable ni de mesurer ni de fournir une grandeur physique avec une précision infinie. C'est pourquoi les valeurs logiques correspondent à des domaines de tension, séparés par un domaine de sécurité (sans signification logique) pour empêcher toute ambiguïté. La convention la plus répandue, appelée "standard TTL" est :



Les circuits nécessitent une alimentation (généralement +5v) qui donne l'énergie nécessaire au fonctionnement du circuit. On la fournit par un générateur placé entre une broche d'alimentation et une broche de masse.



\*Il existe des appareillages logiques qui utilisent d'autres phénomènes que ceux de l'électronique (optique, hydraulique, mécanique...). Même en électronique, on peut réaliser des dispositifs logiques en exploitant d'autres grandeurs que les tensions des conducteurs (les courants, l'état magnétique ...). En fait, sitôt que l'on a la maîtrise d'un phénomène physique, que l'on peut faire dépendre convenablement telles grandeurs de telles autres et que l'on sait "forcer" ces dernières, on peut exploiter ce phénomène pour faire des appareillages logiques.

## 2 - Les entrées et les sorties

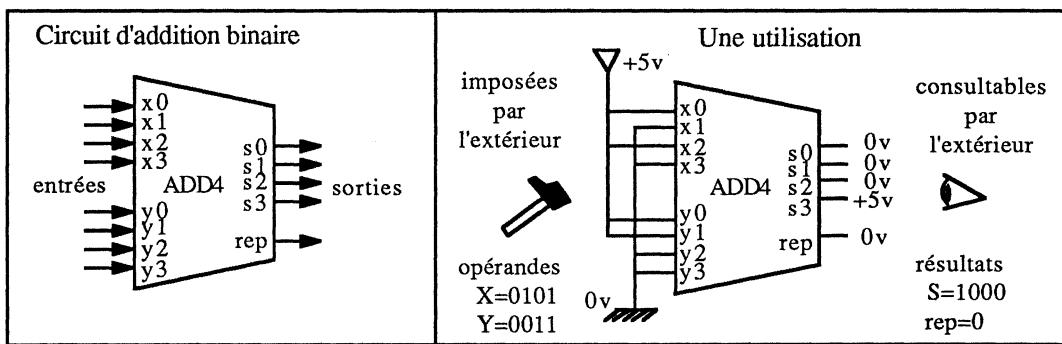
### 2.1 - Notions d'entrée et de sortie

Un circuit logique réalise un calcul à partir d'opérandes, et il délivre des résultats :

- les entrées sont les conducteurs qui permettent à l'utilisateur de présenter les opérandes,
- les sorties sont les conducteurs qui permettent à l'utilisateur de consulter les résultats.

Pour utiliser un circuit, il faut imposer sur les entrées les tensions correspondants aux valeurs logiques des opérandes ; on peut alors observer sur les sorties les tensions correspondants aux valeurs logiques des résultats.

Exemple : le schéma suivant montre les aspects externes d'un circuit d'addition binaire et une utilisation particulière de ce circuit :



Le circuit ADD4 possède huit entrées, sur lesquelles il admet deux opérandes de quatre bits, X et Y, et cinq sorties sur lesquelles il donne en résultats la somme S et le report rep de l'addition binaire des opérandes. L'utilisation particulière illustrée ici consiste à lui faire calculer 0101 plus 0011 : il faut imposer les tensions correspondantes sur les entrées du circuit.

Pour imposer une tension sur un conducteur, il faut le connecter à une source de tension. Ici les opérandes sont connus, ce sont les constantes 0101 et 0011 ; il suffit donc de connecter les entrées à des sources de tension fixes 0v et +5v (masse et alimentation par exemple). Le résultat du calcul est alors disponible sur les sorties, sous forme de tensions qu'il suffit d'observer.

### 2.2 - Aspects physiques des entrées et des sorties

Pour ce qui nous concerne, l'état physique d'un conducteur peut être :

- *Libre*, son potentiel électrique n'est pas imposé avec force (ex : un fil en l'air).
- *Forcé*, son potentiel est imposé avec force (ex : fil connecté à une source de tension).

Dans ce cas, le potentiel peut être :

- *sans signification logique* (ne correspond ni à 0 ni à 1),
- *avec signification logique*, 0 (0v...0.8v) ou bien 1 (2.4v...5v).

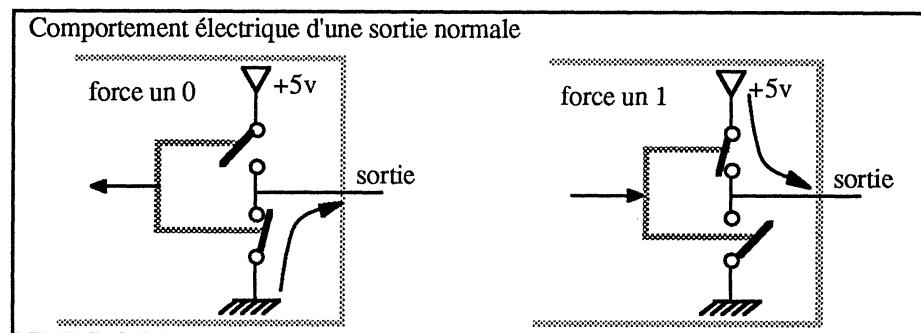
Dans un circuit, on est amené à rencontrer essentiellement 4 sortes de broches : des entrées, des sorties normales, des sorties trois-états, et éventuellement des sorties collecteur ouvert :

#### Les entrées :

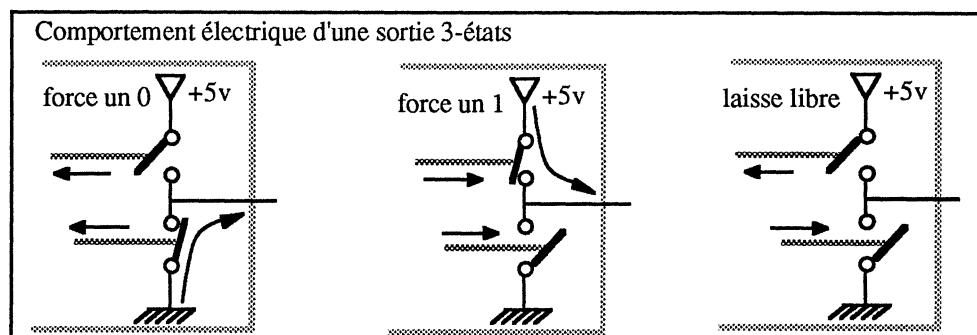
Un circuit logique laisse ses entrées libres ; c'est l'extérieur qui force les entrées pour présenter des opérandes.

**Les sorties normales (Angl. "totem-pole")**

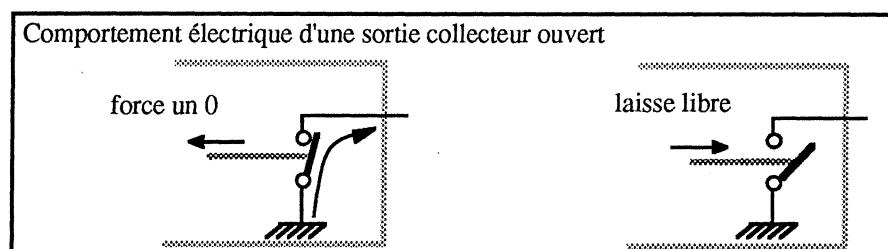
Les sorties normales sont toujours forcées par le circuit. Le comportement électrique d'une sortie normale est analogue à celui d'un commutateur : deux interrupteurs couplés qui connectent la sortie à la masse, pour forcer un 0, ou à la source d'alimentation, pour forcer un 1 (ces interrupteurs sont des transistors).

**Les sorties trois-états (Angl. "tri-state")**

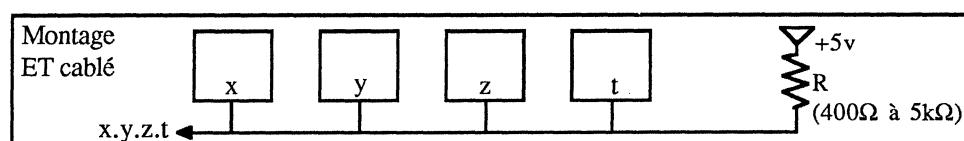
Les sorties 3-états sont des sorties qui sont laissées libres en certaines circonstances :

**Les sorties collecteur ouvert (Angl. "open collector")**

Ce sont des sorties qui peuvent être laissées libres ou forcées à 0.



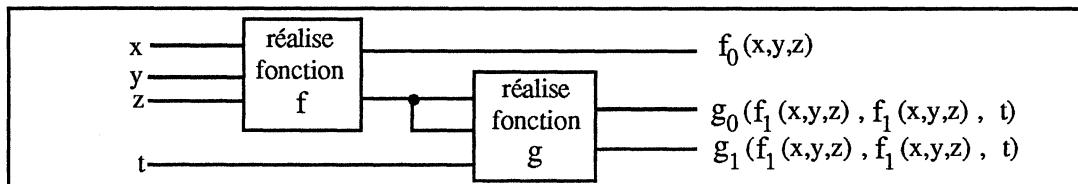
On les appelle ainsi parce que la sortie est prélevée sur le collecteur d'un transistor qui n'est pas connecté internement ; on les appelle également "sorties à drain ouvert" (Angl. "open drain"), parce qu'en technologie MOS c'est le drain d'un transistor à effet de champ qui joue ce rôle. Ces sorties sont réservées à des usages assez particuliers : par elles-mêmes, elles ne peuvent forcer un 1 ; on les utilise toujours en connectant entre-elles plusieurs sorties de ce genre et en reliant la connexion par une résistance au +5v : si au moins une des sorties force 0, la connexion est à 0, et si toutes les sorties sont laissées libres, la connexion est à 1 (grâce à la résistance) : ceci réalise ce qu'on appelle un "ET câblé" ; on s'en sert pour transmettre un signal susceptible d'être activé simultanément par plusieurs sources.



### 3 - Assemblages de circuits logiques

Les sorties d'un circuit sont des générateurs de tension qui assurent la même correspondance entre tension et valeur logique que celle exigée sur les entrées : on dit dans ce cas que les entrées et les sorties sont "compatibles" ; ceci autorise de construire des ensembles en connectant certaines sorties de circuits à certaines entrées d'autres circuits. Ainsi, avec quelques circuits logiques et en respectant quelques règles d'assemblage très simples, on peut créer d'autres circuits, sans avoir à connaître l'électronique, bien que les appareils que l'on construise soient des appareillages électroniques.

Si les circuits utilisés réalisent certaines fonctions, le nouveau circuit ainsi assemblé réalise une composition de ces fonctions :



#### 3.1 - Règles de connexion

Les quelques règles simples qui suivent doivent être respectées pour avoir quelque espoir que l'assemblage réalisé fonctionne.

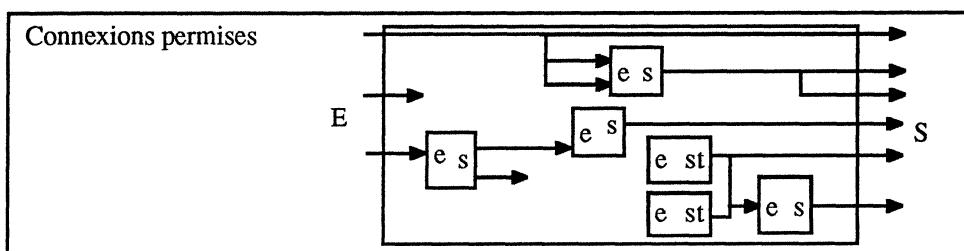
Lorsqu'on veut réaliser un circuit, on distingue :

- les entrées et les sorties du circuit que l'on réalise (abréviations :  $E$  et  $S$ ),
- les entrées et les sorties (subdivisées en sorties normales et sorties trois-états) des circuits que l'on utilise (abréviations :  $e$ ,  $s$ , et  $st$ ).

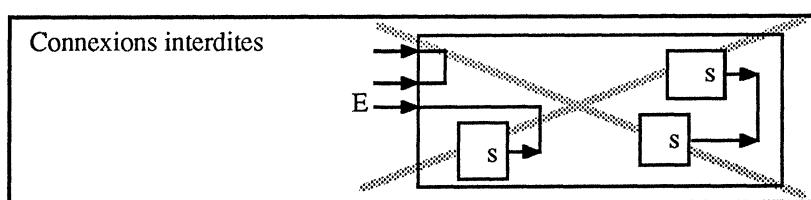
Règle 1 : une  $E$ , des  $e$  et des  $S$  peuvent être connectées ensemble.

Règle 2 : une  $s$ , des  $e$  et des  $S$  peuvent être connectées ensemble.

Règle 3 : des  $st$ , des  $e$  et des  $S$  peuvent être connectées ensemble, sous réserve que, à tout moment, au plus une de ces  $st$  force une valeur.

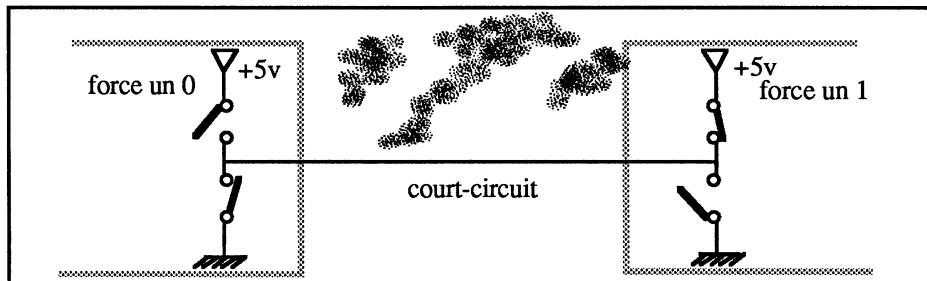


Par contre, il est interdit de connecter ensemble des  $s$ , ou des  $E$ , ou des  $s$  et des  $E$ .



Les raisons de cette interdiction sont doubles :

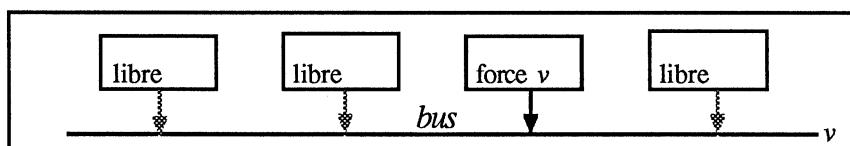
- D'un point de vue logique, c'est un non-sens que de vouloir forcer une variable simultanément à deux valeurs potentiellement différentes.
- D'un point de vue électrique, on risque le court-circuit, dans le cas où l'on cherche à imposer des potentiels différents : l'une des sorties essaye de forcer une tension 0v, et l'autre une tension +5v.



Pour des motifs analogues, il ne faut pas connecter ensemble deux  $E$  dans le circuit, car on empêche alors l'extérieur de communiquer des valeurs ; si par exemple il désire présenter 0 sur l'une des entrées et 1 sur l'autre, on provoque également un court-circuit chez l'utilisateur du circuit : on vient de fabriquer un piège.

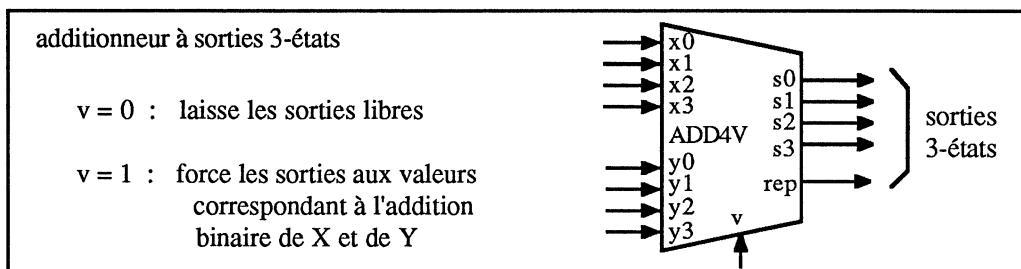
### 3.2 - Utilisation des sorties trois-états

La règle 3 permet de connecter ensemble des sorties 3-états, à condition d'assurer que au plus une de ces sorties force une valeur : la valeur portée par la connexion sera celle de cette sortie. Les sorties 3-états servent essentiellement à réaliser ce qu'on appelle des "bus", c'est-à-dire des lignes sur lesquelles transitent des informations en provenance de diverses sources : à tout moment une seule source affiche de l'information sur le bus, et les autres sources laissent leurs sorties libres :

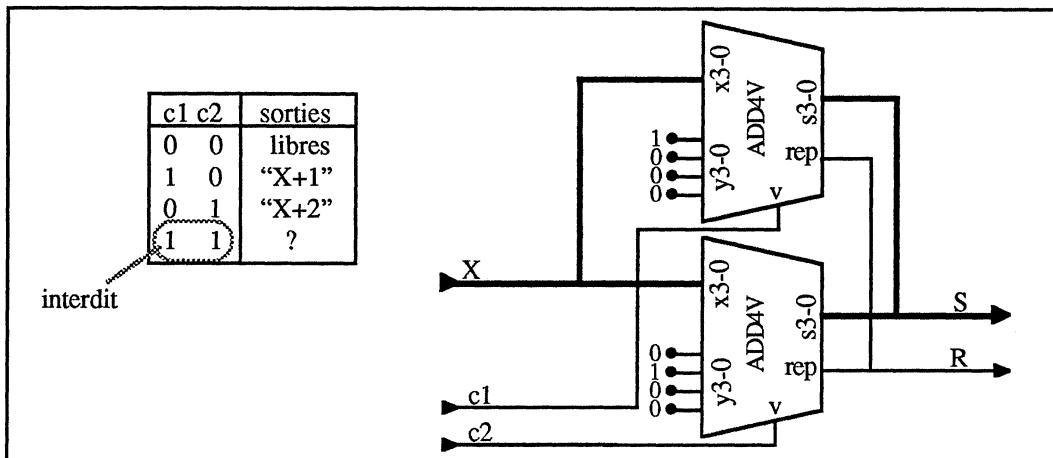


Exemple :

Le circuit suivant (ADD4V) est un additionneur binaire quatre bits, avec une entrée de validation des sorties ; toutes les sorties sont ici des sorties 3-états ; lorsque l'entrée de validation  $v$  est à 0, le circuit laisse les sorties libres et lorsque  $v$  est à 1, il force sur les sorties le résultat de l'opération.



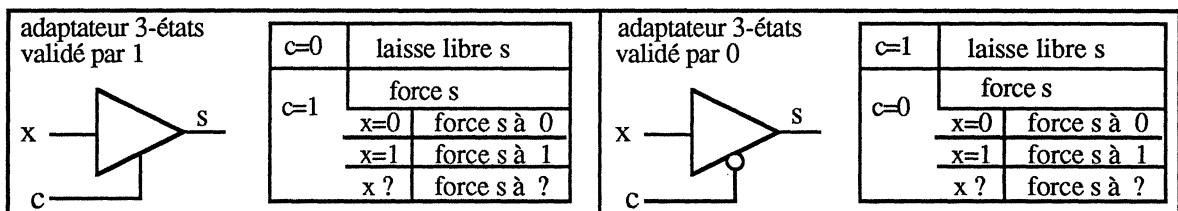
Le circuit suivant utilise deux exemplaires du précédent pour calculer “X+1”, ou “X+2”, selon la valeur présentée sur deux entrées de commande c1 et c2.



Lorsque c1 et c2 valent 0, les sorties du circuit restent libres : ce sont encore des sorties 3-états du circuit total. L'utilisateur de ce circuit ne doit pas présenter la configuration d'entrée c1c2=11, sinon les deux circuits ADD4V essayent simultanément de forcer leurs sorties (court-circuit).

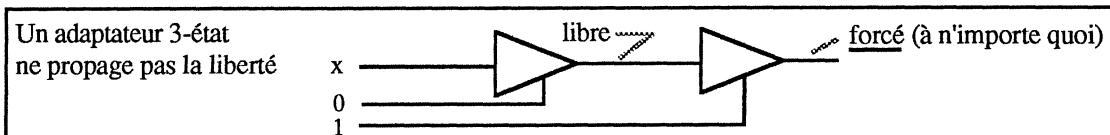
### Les adaptateurs 3-états

Les adaptateurs 3-états (Angl. "*tri-state buffers*") sont les plus simples des circuits à sortie 3-états : selon la valeur d'une commande c, ils permettent soit de transmettre en sortie une valeur d'entrée x soit de ne rien transmettre en sortie (la sortie reste libre).



#### Attention !

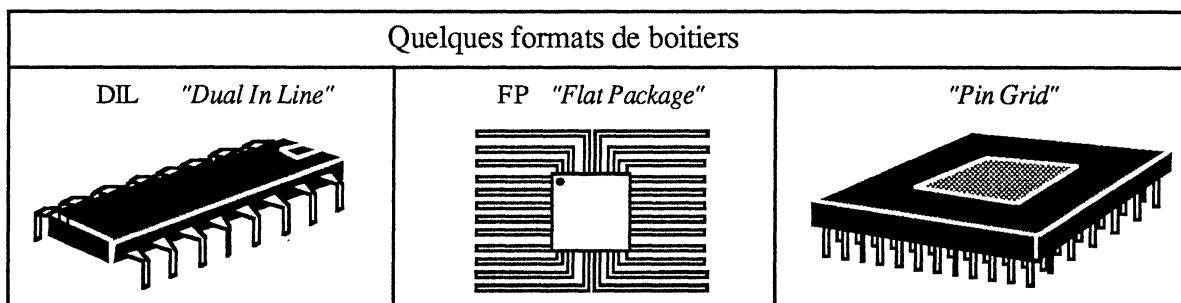
Une confusion est fréquemment commise à propos des adaptateurs 3-états : on s'imagine, à tort, que ce circuit est capable de “transmettre la liberté” (si l'entrée est libre, la sortie le serait aussi, même quand l'adaptateur est validé). Il n'en est rien ! Les tables spécifient bien que lorsqu'il est validé par sa commande c, il force sa sortie dans tous les cas. Si son entrée x est libre, la sortie est forcée à “n'importe quoi”, mais elle est forcée. Il serait électroniquement possible de fabriquer des supers adaptateurs 3-états, capables de propager la liberté, mais ce serait complexe et coûteux ; c'est pourquoi de tels circuits n'existent pas.



## 4 - Quelques mots sur les technologies

Les circuits logiques offerts par les constructeurs se présentent sous la forme de circuits intégrés : ce sont des pastilles de quelques mm<sup>2</sup> (des "puces", angl. "chip"), généralement en silicium, sur lesquelles ont été gravés des circuits électroniques. Ces pastilles sont emballées dans des boîtiers (Angl. "package") dotés de broches (Angl. "pin") pour les connexions avec l'extérieur.

Il existe de nombreuses formes de boîtiers, adaptés à diverses contraintes : encombrement, évacuation de la chaleur, nombre de broches ...



Les formats standards de boîtiers ont longtemps été du type "*dual in line*" ; ce format, assez encombrant, autorise de 14 à 64 broches ; pour des nombres de broches plus importants, pouvant aller jusqu'à 200 broches, la tendance est de plus en plus aux boîtiers de type "*Pin Grid*".

Les circuits électroniques peuvent être réalisés selon diverses technologies qui diffèrent par :

- la sorte de transistors employés (transistors bipolaires, transistors à effet de champ),
- l'organisation électronique de ces transistors pour réaliser les dispositifs élémentaires,
- le procédé de fabrication.

On distingue quatre grandes familles parmi les technologies actuelles (1988), ce sont :

TTL ("Transistor Transistor Logic")  
 ECL ("Emitter Coupled Logic")  
 nMOS ("type n Metal Oxyde Silicium")  
 CMOS ("Complementary Metal Oxyde Silicium")

Les technologies ECL et TTL utilisent des transistors bipolaires ; les technologies nMOS et CMOS par contre utilisent des transistors à effet de champ.

### Caractéristiques externes des diverses technologies

En tant qu'utilisateurs de composants, nous ne sommes intéressés que par les aspects externes des diverses technologies proposées, c'est-à-dire les choses qui nous concernent lorsque on réalise un assemblage de composants ; ce sont essentiellement :

la rapidité, les niveaux logiques acceptés, les précautions à prendre pour les interconnexions, la fiabilité, la consommation en énergie, les températures de fonctionnement, les tolérances sur les tensions d'alimentations.

Le tableau suivant résume les principales caractéristiques de ces diverses technologies. La rapidité et la consommation indiquée sont à peu-près ceux d'une "porte" utilisable extérieurement.

	Rapidité (ns)	Consommation (mW)	Niveaux logiques (V)	Alimentation (V)	Connexions	Intégrabilité
ECL	1	30	< -1, > -2	-3,+5 ±10%	adaptation au-delà de 5 cm	faible
TTL	S	4	<0.8, >2.4	5 ±10%	sans problèmes jusqu'à 15 cm	moyenne
	LS	10	<0.8, >2.4	5 ±10%	sans problèmes jusqu'à 30 cm	moyenne
nMOS	20	1	<0.8, >2.4	5 ±10%	sans problèmes jusqu'à 30 cm	grande
CMOS	3	0 à 2	<0.8, >2.4	3...6	sans problèmes jusqu'à 20 cm	très grande

La technologie ECL est difficile à utiliser : forte consommation de courant, double source d'alimentation, nécessité d'adapter les interconnexions (la plupart des signaux sont transmis de façon différentielle sur deux fils, et il faut placer des résistances en bout de ligne pour absorber les reflexions d'onde dues à la rapidité d'évolution du signal); elle est réservée aux applications spéciales nécessitant une grande rapidité. De plus ses niveaux logiques ne sont pas compatibles avec les autres technologies, qui respectent le standard TTL : toute intéconnexion entre des circuits en ECL et des circuits en d'autres technologies nécessite un interface d'adaptation.

La technologie TTL est d'usage très répandu ; elle se subdivise en de nombreuses sous-familles qui diffèrent par la rapidité et la consommation ; les deux principales sont la sous famille S, rapide mais à forte consommation, et la famille LS, plus lente mais à faible consommation.

La technologie nMOS fut une des premières à permettre une forte intégration (microprocesseur sur une seule puce) ; elle est plutôt lente, mais ceci est surtout du au fait que c'est une technologie en voie d'extinction, que les laboratoires ne cherchent plus à faire progresser, car elle est remplacée par la technologie CMOS

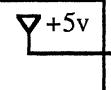
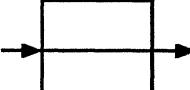
La technologie CMOS est la technologie la plus sollicitée actuellement (1988) ; elle cumule presque tous les avantages sur les technologies concurrentes : rapidité pouvant être supérieure au TTL-S, consommation faible, tolérance considérable sur la tension d'alimentation (la plupart des circuits CMOS fonctionnent correctement avec une source d'alimentation de 3v à 6v), et surtout elle autorise une très forte intégration (plusieurs millions de transistors sur une puce). C'est de plus une technologie qui continue de progresser : elle n'a pas atteint, contrairement aux autres, le maximum de ses possibilités. Un autre avantage du CMOS est la possibilité d'avoir des circuits qui consomment une énergie quasi-nulle lorsqu'on s'en sert très lentement (dissipation statique nulle). Ceci permet de construire des appareils dont l'alimentation peut être coupée en l'absence de sollicitation (passage en veille, "stand by").

**Exercice 1**

On désire réaliser un circuit d'incrémentation binaire INC4 , qui admet un opérande de quatre bits, et donne en résultat sur cinq bits la somme binaire de cet opérande et de 1. Dessiner ses aspects externes et réaliser ce circuit en utilisant le circuit ADD4.

**Exercice 2**

Voici quelques circuits logiques de base : ils sont parmi les plus simples que l'on puisse imaginer (ils sont si simples que l'on oublie parfois que ce sont des circuits logiques) ; la figure montre également comment ils peuvent être faits à l'intérieur (avec de l'électronique connue de tous) :

circuits	constante 1 S →	constante 0 S →	$x \rightarrow E S \rightarrow x$
symbolisations plus légère	1 ● —	0 ● —	—
réalisations possibles			

En combinant ces circuits et éventuellement l'additionneur ADD4 et des adaptateurs 3-états, réaliser les circuits suivants :

- 1) Un circuit qui génère la constante 4 bits 1001.
- 2) Un circuit qui, étant donné un opérande 4 bits X donne un résultat quatre bits s3s2s1s0 formé des deux bits du milieu de X encadrés de deux 0 : s3 s2 s1 s0 = 0 x2 x1 0.
- 3) Un circuit qui, étant donné un opérande de quatre bits X donne en résultat un bit qui vaut 0 si l'interprétation binaire de X est un nombre pair, et 1 si elle est impaire.
- 4) Un circuit qui, étant donné un opérande de 4 bits X=x3x2x1x0 donne en résultat quatre bits qui sont le décalage à gauche de X : s3 s2 s1 s0= x2 x1 x0 0.
- 5) Un circuit qui donne en résultat, sur quatre bits, la représentation binaire de 3+5.
- 6) Un circuit qui, étant donnés un opérande de 4 bits X et une commande c, donne en résultat, pour c=0, X lui même et pour c=1, le décalé de X d'une position à gauche.

**Exercice 3**

Réaliser un circuit d'addition binaire de 3 opérandes de 4 bits. Le résultat doit être la représentation binaire de la somme des trois nombres correspondants (indiquer d'abord combien de bits sont nécessaires pour le résultat).

**Exercice 4**

- 1) A l'aide des circuits présentés jusqu'ici, réaliser un circuit qui, à partir de deux opérandes 4 bits X, Y et de commandes en nombre suffisant, délivre en résultat soit "X+Y", soit "2X+Y", soit "X+2Y" (le tout modulo 16).
- 2) Indiquez les configurations d'entrée interdites pour l'utilisateur de votre circuit.



# CIRCUITS COMBINATOIRES

---

## 1 - Les circuits combinatoires

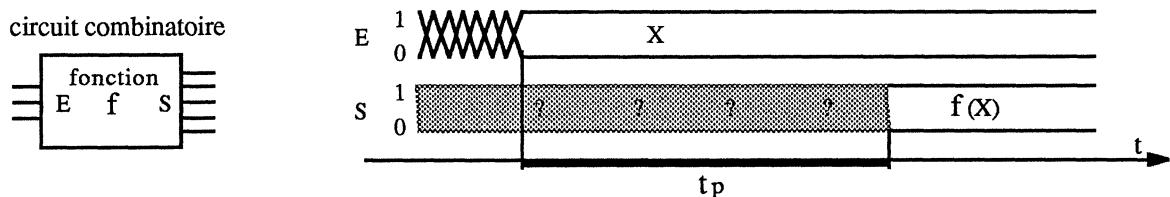
### 1.1 - Notion de circuit combinatoire

Le fonctionnement d'un circuit logique a nécessairement un aspect temporel : il s'y passe des choses au cours du temps. Un circuit combinatoire a un comportement temporel très simple :

- Il réalise sur ses sorties une *fonction de ses entrées*,
- Il fait son calcul en un *temps connu* borné.

Un circuit combinatoire se comporte ainsi : si on présente des opérandes sur les entrées et on les maintient, sans les modifier, pendant une certaine durée connue  $tp$ , on peut observer alors sur les sorties les résultats de la fonction indiquée pour ce circuit.

Ainsi, les sorties d'un circuit combinatoire sont une fonction des valeurs présentées couramment sur les entrées, et uniquement de cela. Elles ne dépendent pas notamment de l'historique des valeurs présentées au cours du temps ; un circuit combinatoire n'a pas de "mémoire" interne.



La durée  $tp$ , appelée "délais de propagation", dépend de l'organisation interne du circuit. Il est par exemple de  $30\text{ns}$  ( $30 \times 10^{-9}$  seconde) pour un additionneur 4 bits réalisé en technologie courante.

Dans le cas général on ne sait rien de ce qui se passe pendant la durée  $tp$  ; le circuit est en cours de calcul, et les sorties peuvent prendre des valeurs quelconques. Le résultat peut être correct avant la fin de  $tp$ , mais on n'en sait rien et on ne peut pas en profiter. Le résultat n'est sûr qu'après  $tp$ .

### 1.2 - Assemblage combinatoire de circuits combinatoires

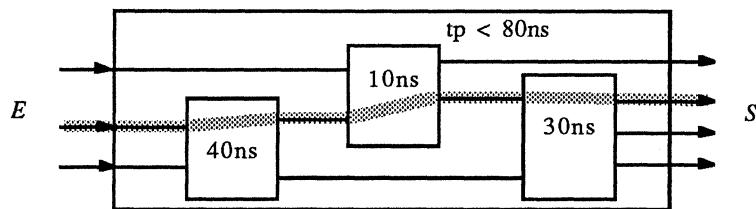
Si on réalise un assemblage *sans boucles* utilisant exclusivement des circuits combinatoires, on obtient encore un circuit combinatoire.

Le terme "sans boucle" signifie que, partant d'une entrée de circuit, en cheminant le long des connexions sortie-entrées, on ne retombe pas sur une entrée de ce même circuit.

Ainsi, aucune variable interne du circuit ne dépend d'elle-même ; toutes les variables internes dépendent uniquement des entrées externes.

Cette règle nous permet de construire des circuits combinatoires, en toute certitude :

- On sait qu'il réalise bien une fonction des entrées : c'est la composée des fonctions utilisées, de structure analogue à celle du circuit.
- On connaît une borne maximum à la durée de calcul de ce circuit : c'est le maximum des sommes des durées de calcul des circuits rencontrés sur les chemins des entrées vers les sorties.

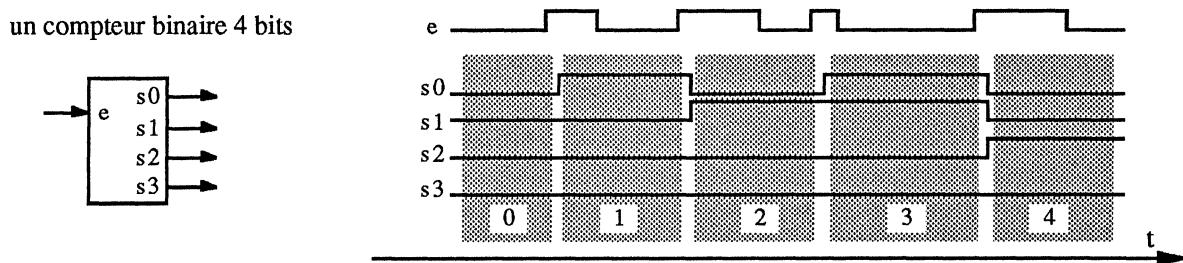


### 1.3 - D'autres sortes de circuits logiques

#### Circuits séquentiels

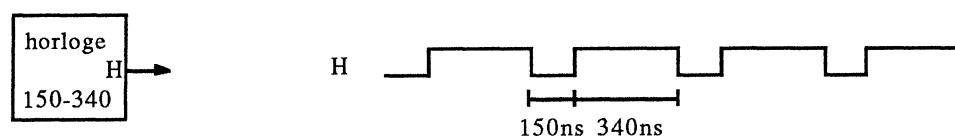
Le schéma ci-dessous illustre le comportement d'un compteur. Ce circuit possède une entrée, et 4 sorties représentant un nombre en binaire. Chaque fois que l'entrée passe de 0 à 1, les sorties changent et prennent pour valeur l'ancienne valeur incrémentée d'une unité (modulo 16).

A un instant donné, les sorties de ce circuit ne sont pas seulement fonction de la valeur de l'entrée courante, elles dépendent de l'histoire de l'entrée : combien de fois est elle passée de 0 à 1. Un tel circuit n'est pas un circuit combinatoire, il a de la mémoire : un tel circuit s'appelle un *circuit séquentiel*.



#### Générateurs de signaux

Le schéma suivant illustre le comportement d'un générateur d'horloge : c'est un circuit sans entrée, qui délivre sur sa sortie un signal périodique qui vaut alternativement 0 pendant 150 ns et 1 pendant 340 ns.



Ce circuit n'est pas un circuit combinatoire, il a une activité propre : un tel circuit s'appelle un *générateur de signaux*.

## 2 - Etude de quelques circuits combinatoires

### 2.1 - Notations

On a l'habitude de coder les valeurs de vérité d'une proposition (*vrai* et *faux*) respectivement par 1 et 0. Dans ce codage,  $X$  étant une variable booléenne (de valeur 0 ou 1), on se permet de confondre les expressions suivantes, moyennant un abus de langage qui assimile la valeur booléenne de  $X$  avec la valeur de vérité d'une proposition (d'une relation) :

$$(X=1) \equiv X \quad (X=0) \equiv /X$$

Pour exprimer l'aspect conditionnel de la valeur d'une variable (si une certaine condition est réalisée, la variable prend la valeur  $v_1$ , sinon elle prend la valeur  $v_2$ ), on utilise la *sélection* :

SI (condition) ALORS ( $v_1$ ) SINON ( $v_2$ ).

Ceci est une fonction à trois arguments :

- la condition (deux valeurs possibles : vrai ou faux).
- les valeurs  $v_1$  et  $v_2$  (dans le même domaine).

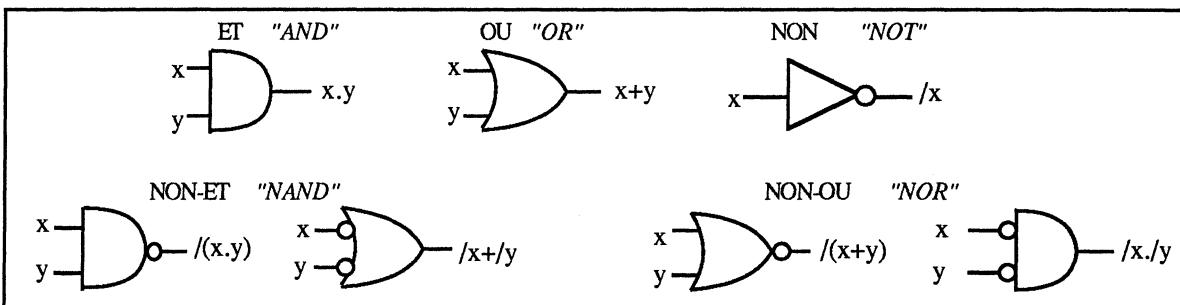
La valeur de cette fonction est dans le même domaine que  $v_1$  et  $v_2$ .

On connaît les propriétés suivantes de la sélection, faciles à vérifier :

SI (vrai) ALORS ( $v_1$ ) SINON ( $v_2$ )	$\equiv$	$v_1$
SI (faux) ALORS ( $v_1$ ) SINON ( $v_2$ )	$\equiv$	$v_2$
SI (cond) ALORS ( $v$ ) SINON ( $v$ )	$\equiv$	$v$
SI ( $v=v_1$ ) ALORS ( $v_1$ ) SINON ( $v$ )	$\equiv$	$v$
SI ( $X$ ) ALORS (1) SINON (0)	$\equiv$	$X$
$X \cdot Y \equiv$ SI ( $X$ ) ALORS ( $Y$ ) SINON (0)	$\equiv$	SI ( $X$ ) ALORS ( $Y$ ) SINON (0)

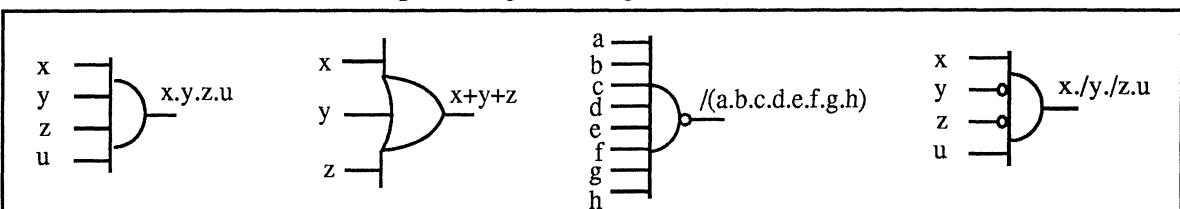
### 2.2 - Les portes

Les portes réalisent les opérations courantes de l'algèbre de Boole.



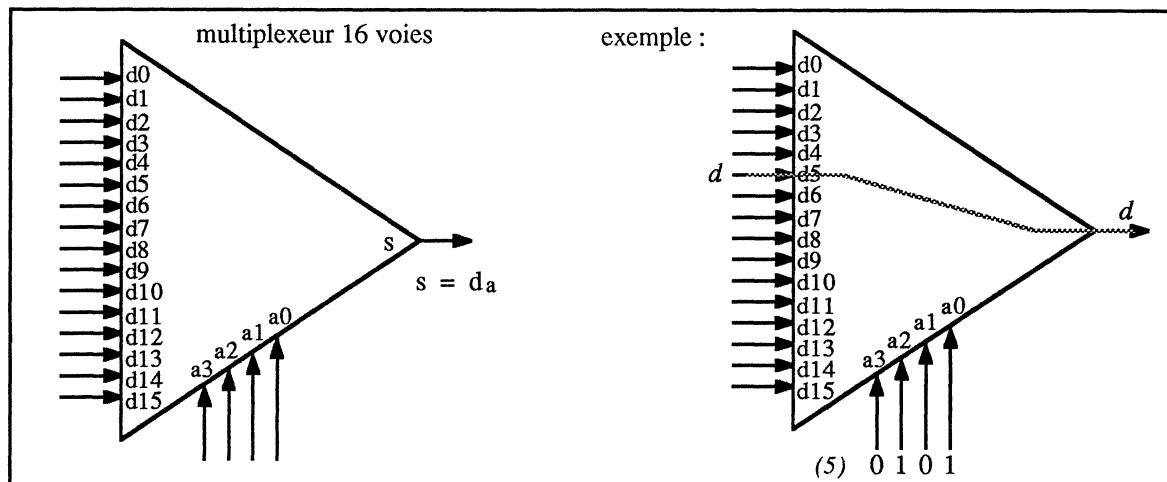
Dans symboles traditionnels utilisés pour dessiner ces circuits, les petits ronds symbolisent la négation logique, et grâce à De Morgan,  $\text{non}(x.y) = \text{non}(x) + \text{non}(y)$ , il y a deux symboles pour dessiner le NON-ET ; de même pour le NON-OU. On utilise l'un ou l'autre des symboles selon ce qui est le plus clair pour la compréhension d'un schéma.

Des circuits analogues existent (ou on peut les faire facilement) pour un nombre quelconque d'entrées, et même avec divers panachages de négations sur les entrées :



## 2.3 - Les multiplexeurs (sélecteurs) :

Un multiplexeur permet de sélectionner une donnée parmi plusieurs : on lui présente des données (1 bit) sur un certain nombre d'entrées, ainsi qu'un numéro (une adresse), et il affiche en sortie la donnée présentée sur l'entrée correspondant à ce numéro.

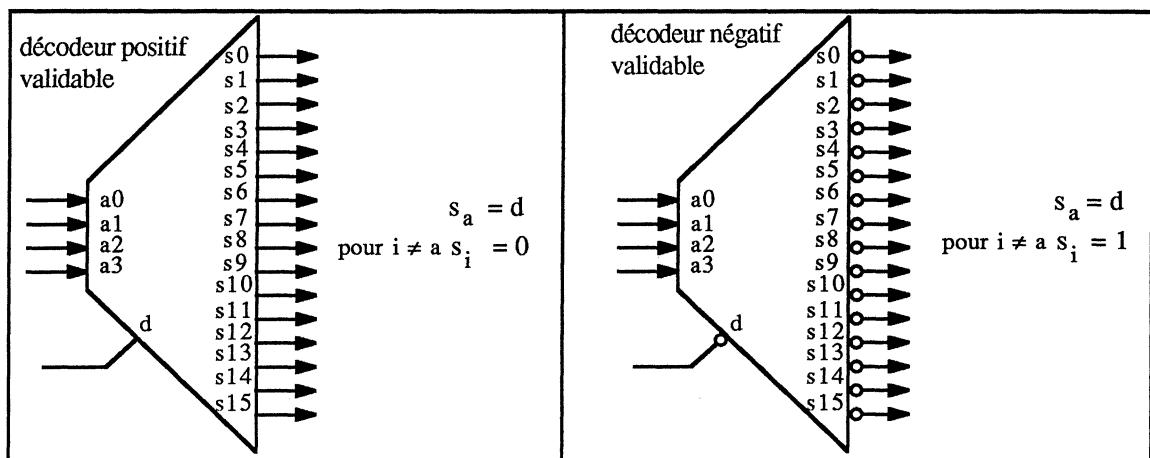


Pour  $N$  (deux, trois ou quatre) entrées d'adresses, les multiplexeurs ont  $2^N$  (quatre, huit ou seize) entrées de données (voies).

## 2.4 - Les décodeurs

Un décodeur permet d'activer une possibilité parmi plusieurs : on lui donne en entrée  $a$  un numéro (une adresse) et il active la sortie qui correspond à ce numéro, laissant inactives toutes les autres.

L'état "activé" peut être matérialisé par 1 (décodeur positif) ou 0 (décodeur négatif). Sur la plupart des décodeurs, une entrée de validation  $d$  permet de contrôler le fonctionnement global : inactivée, elle rend toutes les sorties inactives.



Les décodeurs ont  $N$  (deux, trois ou quatre) entrées binaires pour le code, et  $2^N$  (quatre, huit ou seize) sorties. On parle de décodeurs deux vers quatre, trois vers huit, etc...

Le décodeur affirmatif (validable) peut être décrit par la formule :

$$s_k = \text{SI } (d \text{ ET } (k=a)) \text{ ALORS } 1 \text{ SINON } 0 \quad \text{ou encore} \quad s_k = d \text{ ET } (k=a)$$

Ceci peut également s'exprimer sous les deux formes équivalentes suivantes :

- forme 1)  $s_k = \text{SI } (d) \text{ ALORS } (k=a) \text{ SINON } 0.$   
 forme 2)  $s_k = \text{SI } (k=a) \text{ ALORS } d \text{ SINON } 0.$

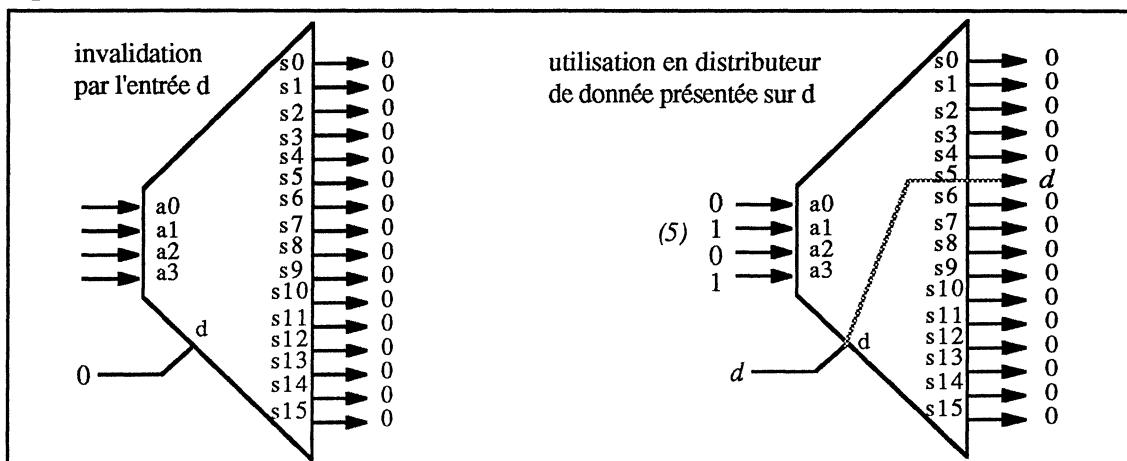
La forme 1) fait apparaître le rôle de *validation* de l'entrée  $d$  :

Si  $d=1$ , le décodeur active une de ses sorties, sinon il les maintient toutes inactives.

La forme 2) montre le décodeur comme un *distributeur de donnée* :

L'entrée  $d$  est considérée comme une donnée de un bit, transmise sur la sortie désignée par l'adresse  $a$  ; les autres sorties sont forcées à 0 ; c'est pour cela que le décodeur validable est encore appelé "démultiplexeur".

Exemples :



## 2.5 - L'additionneur binaire

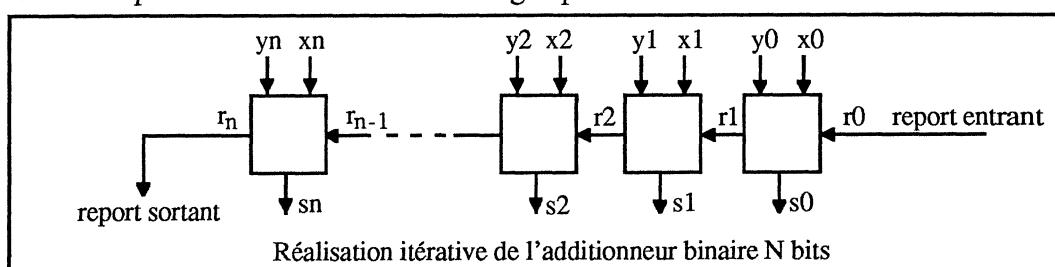
### Report entrant, report sortant

Comme nous l'avons vu au chapitre sur les représentations, l'addition binaire peut être définie par l'algorithme itératif suivant :

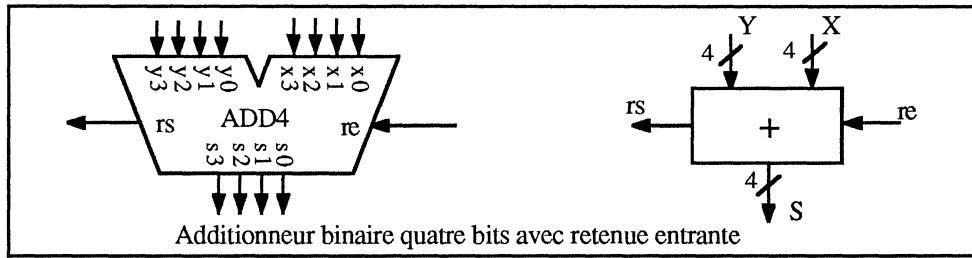
$r_i$	report de rang $i$
$x_i$	chiffre de rang $i$ du 1er opérande
$y_i$	chiffre de rang $i$ du 2ème opérande
$r_{i+1}$	report de rang $i+1$
$s_i$	chiffre de rang $i$ de la somme

$x_i$	$y_i$	$r_i$	$r_{i+1}$	$s_i$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
0	1	1	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Une réalisation possible de l'additionneur binaire  $N$  bits consiste à construire des modules pour un bit, et à les associer en cascade de manière à réaliser l'itération : le report sortant d'un module est le report entrant du module de rang supérieur .



Le circuit d'addition binaire est plus intéressant si  $r_0$  n'est pas fixé à priori, mais est laissé disponible à l'utilisateur grâce à une entrée supplémentaire appelée *report entrant* (re). Le report généré par le dernier rang,  $r_n$ , est appelé *report sortant* (rs).



La fonction précise réalisée par ce circuit est donc l'addition binaire de  $x$ ,  $y$  et du report entrant  $re$  considéré comme un chiffre de poids faible :

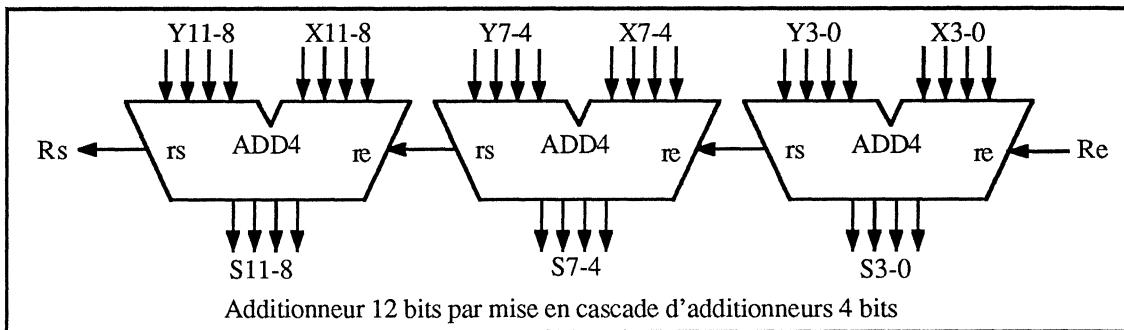
$$rs, s = {}_2[ [x]_2 + [y]_2 + [re]_2 ]$$

On peut également considérer  $re$  comme une entrée de commande conditionnant l'addition de 1 à la somme de  $x$  et  $y$ , ce qui conduit à la formulation suivante :

$$rs, s = \text{SI } (re) \text{ ALORS } {}_2[ [x]_2 + [y]_2 + 1 ] \text{ SINON } {}_2[ [x]_2 + [y]_2 ]$$

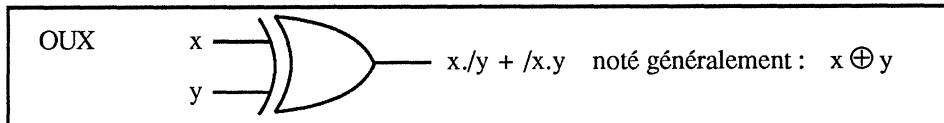
### Mise en cascade

Le report entrant permet la réalisation d'additionneurs pour des données de grande taille à partir d'additionneurs plus petits : il suffit de connecter plusieurs additionneurs en cascade, le report sortant de chacun étant connecté au report entrant du suivant.



## 2.6 - Le ou exclusif

Le ou exclusif (OUX) peut être défini par l'expression booléenne suivante :



Bien que très simple, le ou-exclusif peut être formulé de nombreuses façons :

- sa sortie vaut 1 si exactement une des entrées vaut 1 :

$$x \oplus y = x./y + /x.y$$

- on peut le considérer comme un *comparateur d'égalité* pour 1 bit :

$$x \oplus y = \text{SI } (x=y) \text{ ALORS } 0 \text{ SINON } 1$$

- on peut le considérer comme un *inverseur conditionnel* :

$$x \oplus y = \text{SI } (y) \text{ ALORS } /x \text{ SINON } x$$

- on peut enfin le considérer comme un *additionneur modulo 2* :

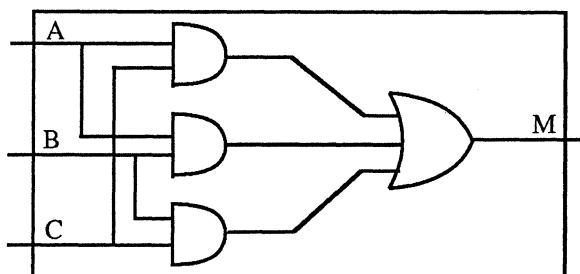
$$x \oplus y = [{}_2[x] + {}_2[y]]_2 \text{ modulo } 2$$

C'est là un bel exemple de multiples formulations pour une même opération.

**Exercice 1**

A l'aide de portes NON-ET à 2 entrées, réaliser les portes suivantes :

NON      OU à 2 entrées      ET à 2 entrées      NON-OU à 2 entrées      ET à 4 entrées      OU à 4 entrées

**Exercice 2**

- 1- Réaliser le circuit ci-dessus en n'utilisant que des portes NON-ET.
- 2- Réaliser ce circuit en n'utilisant que des portes NON-OU.
- 3- déterminer le temps de calcul de ce circuit, sachant que les portes NON-ET et NON-OU utilisées ont un temps de calcul de 10ns.
- 4- Essayer de dire simplement ce que fait ce circuit.

**Exercice 3**

- Réaliser une porte ET à 2 entrées en utilisant un décodeur 4 voies.

**Exercice 4**

En utilisant toutes sortes de portes, réaliser :

- 1- Un décodeur 2 voies (sans entrée de validation).
- 2- Un décodeur 4 voies (sans entrée de validation).
- 3- Un décodeur 4 voies validable.

**Exercice 5**

Réaliser un décodeur 16 voies validable à l'aide de décodeurs 4 voies validables et de portes.

**Exercice 6**

Réaliser un multiplexeur 2 voies

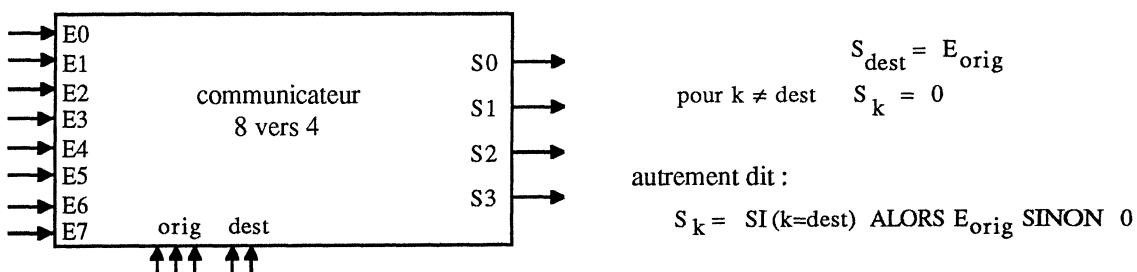
- 1- à l'aide de portes,
- 2- à l'aide d'adaptateurs 3-états.

**Exercice 7**

Réaliser un multiplexeur 8 voies à l'aide de adaptateurs 3-états et d'un décodeur.

**Exercice 8**

- Réaliser le circuit suivant, "communicateur 8 vers 4". Ce circuit à 8 entrées de données (1bit), 4 sorties de données (1bit également), 3 entrées pour désigner une entrée de données (l'origine) et 2 entrées pour désigner une sortie (la destination). Ce circuit transmet la donnée de l'entrée désignée sur la sortie désignée, et force les autres sorties à 0.



**Exercice 9**

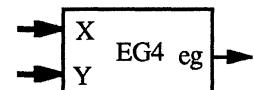
Toute fonction booléenne peut être donnée par une table, la “table de vérité” de la fonction. On peut “cabler la table de la fonction” grâce à un multiplexeur : on place sur chaque entrée de donnée la valeur (0 ou 1) de la fonction correspondant à chaque configuration de valeurs pour les variables de la fonction; il suffit alors de présenter les variables de la fonction sur les entrées d’adresse du multiplexeur : le multiplexeur va adresser la table ainsi cablée et afficher en sortie la valeur de la fonction en ce point.

- Réaliser selon ce principe la fonction booléenne de 3 variables x, y et z :

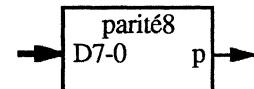
$$(/x.y + x.y).z + (x./y + /x.y)./z$$

**Exercice 10**

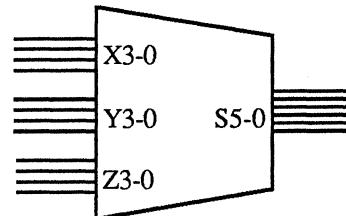
- Réaliser, à l'aide de circuits OUX, un comparateur d'égalité pour des données de 4 bits. La sortie doit valoir 1 si les deux opérandes sont égaux, 0 sinon.

**Exercice 11**

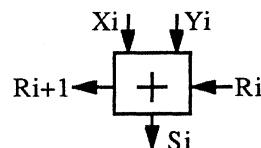
- Réaliser le circuit suivant, appelé circuit de *calcul de parité*, dont la sortie vaut 0 si le nombre de ses entrées à 1 est pair, et 1 si le nombre d'entrées à 1 est impair :
- On utilise des ou-exclusifs dont le temps de calcul vaut 10ns ; trouver la façon la plus performante de réaliser le circuit de parité pour 8 bits, et indiquer son temps de calcul.

**Exercice 12**

- Réaliser un additionneur binaire de 3 opérandes de 4 bits, en utilisant exclusivement des additionneurs binaires 4 bits. Le résultat doit être “exact”, sur 6 bits donc :
- Indiquer le temps de calcul de votre circuit, sachant que les additionneurs utilisés ont un temps de calcul de 30ns.

**Exercice 13**

- On considère l'additionneur 1 bit :



On pose  $Gi = Xi \cdot Yi$  et  $Pi = Xi \oplus Yi$  ;

$Gi$  signifie que le couple  $Xi, Yi$  “génère” un report ( $Ri+1$  vaut 1, indépendamment de  $Ri$ ) ;

$Pi$  signifie que le couple  $Xi, Yi$  “propage” un report ( $Ri+1$  vaut ce que vaut  $Ri$ ) ;

- Exprimer  $Ri+1$  et  $Si$  en fonction de  $Pi$ ,  $Gi$  et  $Ri$ .
- En déduire un schéma économique de l'additionneur 1 bit à l'aide de portes NAND.
- Déterminer le temps de calcul d'un additionneur 16 bits construit par mise en cascade de cet additionneur 1 bit (temps de calcul du NAND = 10ns).

On s'intéresse à la réalisation plus directe, et plus rapide, d'un additionneur 4 bits.

- Exprimer  $R4$  en fonction de  $Pi$ ,  $Gi$  ( $i=0 \dots 3$ ) et  $R0$ .
- En déduire un circuit qui calcule  $R4$  de façon performante.
- Déterminer le temps de calcul d'un additionneurs 16 bits obtenu par mise en cascade de 4 additionneurs 4 bits ainsi construits.

## 3 - Quelques méthodes de réalisation

### 3.1 - Critères de qualité d'une réalisation

Pour ce qui nous concerne, la réalisation de circuits logiques consistera toujours à fabriquer un circuit à partir d'autres circuits :

- soit de circuits qui figurent au catalogue d'un constructeur.
- soit conçus précédemment par nous-mêmes, à partir de circuits existants.

1<sup>er</sup> critère : *Il doit être correct et réalisable*

Le circuit doit évidemment assurer correctement sa fonction !

De plus, il faut pouvoir le réaliser, c'est-à-dire qu'il doit être composé de circuits existants.

Quels sont les circuits existants ? que font-ils ? il en existe des milliers à l'heure actuelle, et nul ne les connaît tous. Leur description est donnée dans les catalogues de constructeurs, et leur compréhension demande de quelques secondes (pour les portes par exemple) à quelques mois (pour certaines unités centrales d'ordinateur).

2<sup>ème</sup> critère : *Il doit offrir certaines performances*

Les performances interviennent après le premier critère (à quoi servirait un appareil performant qui ne fonctionnerait pas correctement ?). Pourtant, elles sont importantes, plus que dans d'autres domaines (le logiciel par exemple) parce que les circuits sont des objets matériels coûteux à l'achat comme au fonctionnement.

Quels sont les critères de performances ? ils sont malheureusement nombreux et difficiles à cerner. Citons entre autres : prix, encombrement, consommation, vitesse, fiabilité ...

Un bon circuit est issu d'un compromis entre ces divers paramètres.

En règle générale on cherche à utiliser le moins possible de composants et à utiliser les composants les plus simples. La notion de simplicité n'est pas très nette ; un critère impartial est le nombre de broches (les entrées et les sorties), c'est souvent ce qui fixe le coût d'un composant à l'heure actuelle ; cependant, il est admis que certains circuits sont plus simples que d'autres, par exemple les portes sont jugées plus simples que les additionneurs, pour la seule raison qu'elles sont plus faciles à réaliser en électronique.

Lorsque l'on conçoit un circuit, il est bon de procéder en deux phases :

- 1) Recherche d'une solution correcte, c'est la *phase de recherche (ou d'analyse)*,
- 2) Simplification de cette solution, c'est la *phase d'optimisation (ou de synthèse)*.

Les méthodes et techniques de programmation s'appliquent bien à la conception de circuits. La démarche est très similaire à celle de l'écriture d'un programme, à quelques différences près :

- L'absence d'un langage de programmation bien défini ; c'est même une première difficulté. Un langage de programmation offre un nombre limité de primitives, quelques types de base pour les valeurs, quelques opérations, quelques schémas de construction ; le programme est une combinaison de ces éléments. Pour faire du circuit, on n'a a priori aucune primitive, si ce n'est l'immense catalogue de tous les composants existants, et le savoir faire de la communauté à un moment donné qui, à la différence d'un bon langage, est en perpétuelle évolution.

- Les réalisations en circuits sont habituellement beaucoup plus simples que celles possibles par logiciel (tout simplement parce qu'on n'y arriverait pas ou que ce serait trop coûteux) ... En d'autres termes, les algorithmes mis en œuvre sous forme de circuits sont généralement triviaux (ce qui ne signifie pas qu'ils soient toujours faciles à trouver). Cela compense en partie la difficulté précédente.

- Enfin, les performances ont un rôle souvent prépondérant dans le choix entre plusieurs solutions correctes.

## 3.2 - Méthode basée sur l'algèbre de Boole

### 3.2.1 - Exposé de la méthode

Cette méthode est fondée sur la remarque suivante : quelle que soit la fonction à réaliser, quel que soit le problème donné au départ, les entrées, comme les sorties, du circuit sont des bits. Chaque sortie, considérée individuellement, est donc une simple fonction booléenne des entrées.

#### Phase de recherche :

On cherche à exprimer chaque sortie sous forme d'une fonction booléenne des entrées :

- Dans certains cas, on peut deviner directement une expression de la fonction.
- Sinon, on envisage tous les cas pour les valeurs d'entrée et on construit la table de la fonction. On obtient alors une expression booléenne pour chaque sortie.

Quand on a la table de vérité, la fonction booléenne s'en déduit simplement :

- A chaque ligne de la table où la fonction vaut 1 correspond un monôme complet, c'est-à-dire un ET de toutes les variables, chaque variable étant complémentée ou non selon que la variable vaut 0 ou 1 sur cette ligne
- La fonction est le OU de tous ces monômes complets.

La phase d'analyse est terminée. On pourrait s'arrêter là : on a une solution. Il suffirait de cabler un circuit à base de portes (ET, OU, NON) de structure analogue aux expressions trouvées.

Exemple : soit à réaliser un additionneur binaire modulo 4 (opérandes 2 bits, résultat 2 bits).

additionneur binaire modulo 4		x1	x0	y1	y0	s1	s0
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	1	0	0	1	1	0
0	0	1	1	1	1	1	1
0	1	0	0	0	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	1	1
0	1	1	1	0	0	0	0
1	0	0	0	0	1	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	0	0	0	0
1	1	1	0	0	1	0	1
1	1	1	1	1	0	1	0

$s1 =$

$$\begin{aligned} &/x1./x0.y1./y0 \\ &+ /x1./x0.y1.y0 \\ &+ /x1.x0./y1.y0 \\ &+ /x1.x0.y1./y0 \\ &+ x1./x0./y1.y0 \\ &+ x1./x0.y1./y0 \\ &+ x1.x0./y1./y0 \\ &+ x1.x0.y1.y0 \end{aligned}$$

$s0 =$

$$\begin{aligned} &/x1./x0./y1.y0 \\ &+ /x1./x0.y1.y0 \\ &+ /x1.x0./y1./y0 \\ &+ /x1.x0.y1./y0 \\ &+ x1./x0./y1.y0 \\ &+ x1./x0.y1.y0 \\ &+ x1.x0./y1./y0 \\ &+ x1.x0.y1./y0 \end{aligned}$$

#### Phase d'optimisation :

Cette phase consiste à appliquer les règles usuelles de simplification d'expressions booléennes de manière à diminuer la quantité de portes nécessaires. Pour un petit nombre de variables, on utilise généralement des tableaux de Karnaugh.

Exemple : pour l'additionneur binaire modulo 4, nous obtenons :

table de s1

x1	x0	00	01	11	10
y1	y0	00	01	11	10
00	00			11	11
01	01	11		11	
11	11	11		11	
10	10	11	11		

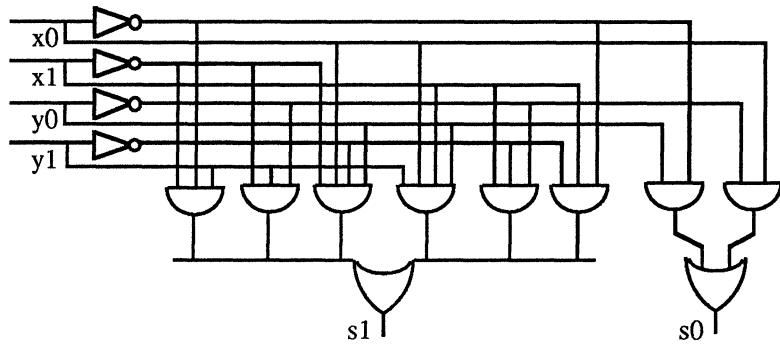
table de s0

x1	x0	00	01	11	10
y1	y0	00	01	11	10
00	00			11	11
01	01	11		11	
11	11	11		11	
10	10	11	11		

$$s1 = /x1./x0.y1 + /x1.y1./y0 + /x1.x0./y1.y0 + x1.x0.y1.y0 + x1./y1./y0 + x1./x0.+y1$$

$$s0 = /x0.y0 + x0./y0$$

Ce qui conduit à la réalisation suivante :



### 3.2.2 - Bon usage et limitations de la méthode booléenne

La méthode booléenne est une méthode “bête”, et doit donc être employée avec “intelligence”. Elle est “bête” en ce sens qu’elle ne demande aucune réflexion : une machine est parfaitement capable de l’appliquer ; ce n’est pas une méthode d’approche pour un problème compliqué : elle ne doit être appliquée que pour des sous-ensembles simples que l’on aura au préalable dégagé par une analyse plus globale et abstraite du problème. Par contre, pour ces sous-ensembles simples (par exemple, moins de 6 variables), il ne faut hésiter à l’appliquer. Nous indiquons ci-dessous les principaux inconvénients et limitations de la méthode booléenne :

*1) Elle conduit toujours à une réalisation exclusivement à base de portes.*

C'est normal, puisque dès le départ on cherche à exprimer la fonction à réaliser sous forme de fonctions booléennes, à base de ET, OU, NON. On a peu de chances de voir apparaître au sein des formules un additionneur, un décodeur, un adaptateur trois états, ni aucun autre circuit existant où que l'on sait déjà réaliser.

*2) Elle est impraticable pour un nombre important de variables d'entrées.*

Cette méthode n'est praticable, à la main, que jusqu'à six variables d'entrées, et il faut déjà une bonne dose de patience pour remplir les 64 cases d'une table... Avec des moyens automatiques, à l'aide d'un ordinateur, on peut pousser un peu plus loin, disons jusqu'à une vingtaine de variables (environ 1000 000 cases pour la table de la fonction). Le nombre de cases croît exponentiellement avec le nombre de variables ( $2^n$ ).

*3) Elle conduit souvent à des solutions irréalistes.*

Il faut également considérer le réalisme de ce que délivre cette méthode : la taille du circuit obtenu. En nombre de portes, elle est en général proportionnelle à l'exponentielle du nombre de variables ; parfois (on a de la chance), les expressions se simplifient beaucoup, mais c'est rare et purement fortuit, puisque la méthode ignore la structure propre du problème.

*4) Elle conduit à une réalisation sans structure.*

C'est peut-être là le plus grave défaut, la solution est dépourvue de structure ; plus exactement, les solutions de tous les problèmes ont la même structure, que l'on peut résumer en :

- Une couche de NON, pour avoir les variables sous forme directes et inversées.
- Une couche de ET pour calculer les monômes.
- Une couche de OU (un par sortie) pour terminer.

Les variantes s'obtiennent en faisant appel aux lois de De Morgan qui permettent de remplacer les couches de ET/OU par deux couches de NON-ET ou par deux couches de NON-OU.

Chaque partie de la réalisation est sans intérêt ; si on l'isole, on obtient en général un ensemble de portes sans fonctionnalité connue. L'utilisateur du circuit peut être satisfait de la réalisation, il n'est pas concerné par les l'intérieur du circuit. Par contre, le concepteur n'est pas satisfait car il n'a rien appris. Si un problème voisin se présente, il ne pourra pas profiter de l'expérience acquise par la résolution du premier.

## Quand utiliser la méthode booléenne ?

1) *Lorsque la fonction a peu de variables et que le circuit à réaliser n'existe pas déjà.*

Jusque quatre ou cinq variables (moins de six en tous cas), et si on ne voit pas de solution évidente pour réaliser le circuit avec ce dont on dispose, on peut recourir à la méthode booléenne.

2) *Lorsque la fonction à réaliser est "définie par le hasard", c'est-à-dire sans structure naturelle.*

Par exemple le circuit de contrôle d'un afficheur sept segments : quels rapports peuvent exister entre le code binaire d'un chiffre hexadécimal et le dessin de ce chiffre ? Une telle fonction ne présente aucune structure directement exploitable ; elle est le fruit du hasard, telle qu'elle est définie par sa table. Il est donc naturel de l'aborder par sa table, c'est à dire d'utiliser la méthode booléenne.

3) *En vue d'une utilisation de circuits programmables.*

La méthode booléenne conduit à des circuits dont la structure est indépendante de la nature du problème. Cela conduit naturellement à préfabriquer, selon cette structure, des "circuits à tout faire" auxquels il ne reste plus qu'à fixer quelques caractéristiques propres à chaque réalisation : ce sont ce qu'on appelle les "circuits programmables". Dans de nombreux cas, la structure combinatoire sera trop complexe pour être étudiée à la main, et il faudra recourir à des outils de C.A.O.

Il existe essentiellement deux sortes de circuits combinatoires programmables :

- Les mémoires mortes (ROM : "Read Only Memory") : matérialisent entièrement une table de vérité.
- Les tableaux logiques programmables (PLA : "Programmable Logic Array") : réalisent des sommes de monômes en nombres plus ou moins restreints.

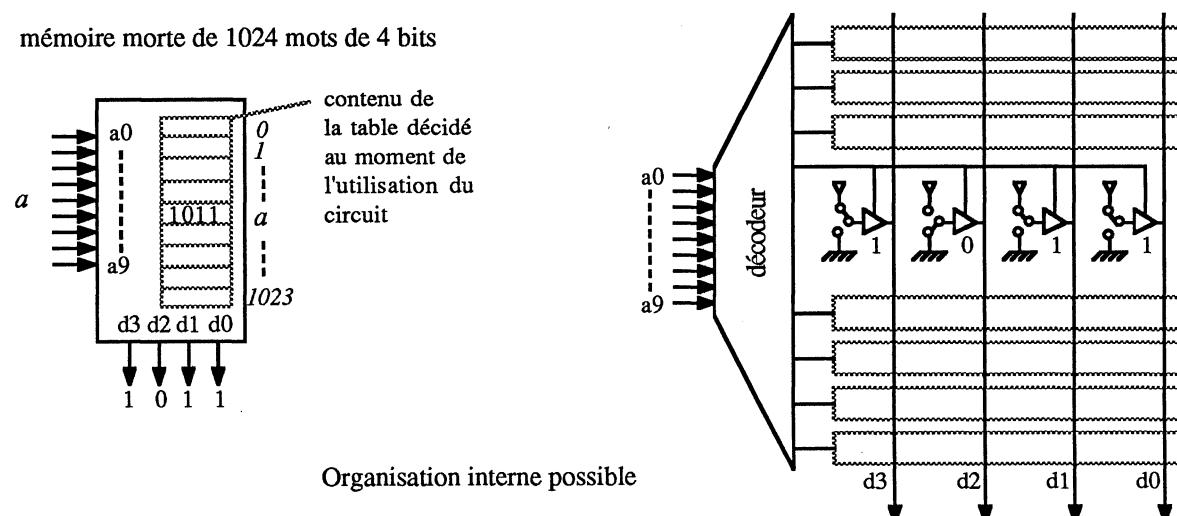
### 3.2.3 - ROM et PLA

#### Les ROM :

Les mémoires mortes ont N entrées d'adresse (généralement de 8 à 16) et P sorties de données (généralement 4 ou 8 bits). A l'intérieur, il y a  $2^N$  mots de P bits, enregistrés de façon plus ou moins définitive, et le circuit délivre en sortie la valeur du mot spécifié par l'adresse.

On réalise n'importe quelle fonction de N variables en inscrivant sa table dans la mémoire.

L'inscription des mots dans la mémoire morte est décidée au moment de la conception d'un circuit. Le procédé d'inscription dépend de la technologie, ce peut être par exemple des fusibles que l'on détruit comme l'illustre la figure.

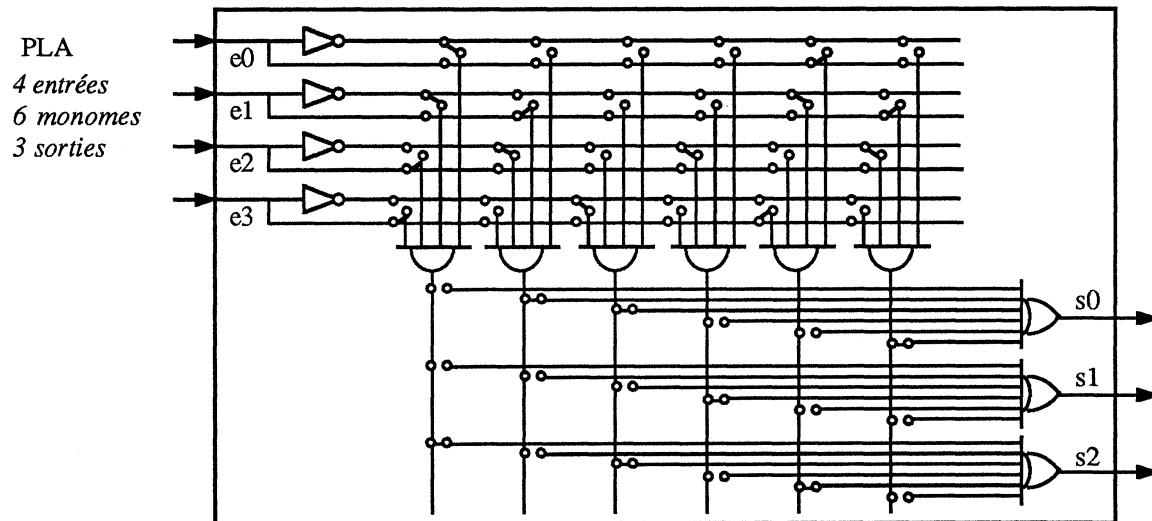


### Les PLA :

Les tableaux logiques programmables offrent un certain nombre d'entrées (N) et de sorties (P). A l'intérieur, M portes ET réalisent des monômes, et P portes OU fournissent en sortie des sommes de ces monômes.

Le choix des monômes et le choix des sommes est réalisé par la destruction de certaines liaisons.

On ne peut pas réaliser n'importe quelle fonction des entrées, car on peut être limité dans le tableau de ET ou celui de OU.



### 3.3 - Méthode basée sur la décomposition fonctionnelle

#### 3.3.1 - Le découpage fonctionnel

Lorsqu'on veut réaliser un circuit, c'est pour résoudre un problème donné, qui possède en général sa propre structure, et on a intérêt à en profiter dans la recherche d'une solution.

Cette méthode utilise les connaissances et l'expérience du concepteur, qui doit connaître :

- Les composants disponibles, les circuits qu'il sait déjà réaliser, les avoir vus dans divers contextes d'utilisation.
- Les propriétés des données traitées et des codages utilisés : par exemple, pour des nombres, il faut connaître l'arithmétique et les codages habituels des nombres.

#### 3.3.2 - Phase d'analyse

Elle se traite en trois parties :

- 1) Structurer les entrées et les sorties.
- 2) Décomposer les fonctions à réalisées.
- 3) Analyser séparément les sous-fonctions, en recherchant les relations entre les entrées et les sorties (on aura souvent recours à la méthode booléenne, en dernier lieu).

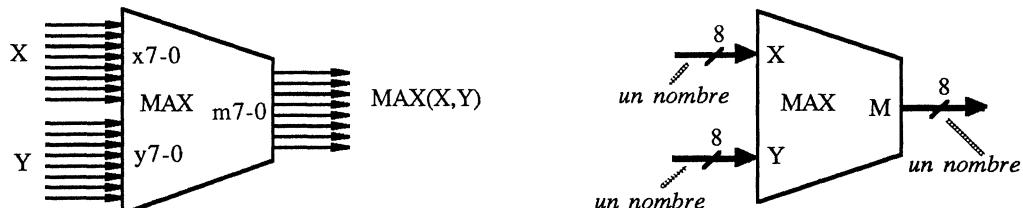
Lorsque l'on entreprend la réalisation des circuits composants, il faut oublier les rôles particuliers qu'ils ont dans le circuit total, et les étudier isolément. Ce procédé d'analyse descendante est une bonne façon de remplacer un problème complexe par plusieurs problèmes plus simples.

Nous l'illustrerons sur deux exemples.

#### Premier exemple : calcul du MAX de deux nombres

Ce circuit doit calculer le maximum de deux nombres représentés en binaire sur 8 bits.

On cherche d'abord à structurer les entrées et les sorties : on les regroupe en paquets, auxquels on donne une signification, une interprétation. La nature et le rôle qu'elles jouent dans le problème est généralement un bon guide (un nombre, un booléen, un caractère, un code ...). Ici, il est naturel de considérer les entrées et sorties comme des nombres.

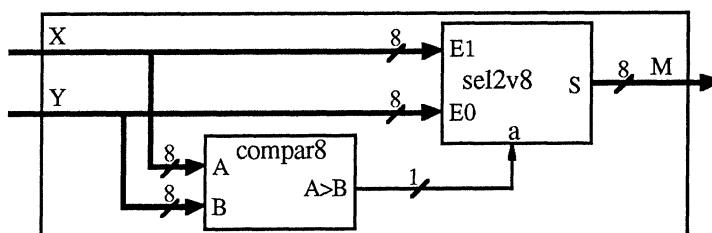


On cherche ensuite à exprimer chacun des paquets de sortie en fonction des paquets d'entrée, en restant le plus possible dans le cadre des interprétations qu'on leur attribue.

Connaissant un peu l'arithmétique, on trouve facilement une expression pour la sortie M (dans tout problème, il est nécessaire de connaître un minimum de propriétés) :

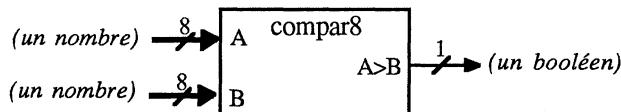
$$M = SI (X > Y) ALORS X SINON Y$$

Nous avons une première ébauche du circuit à réaliser :

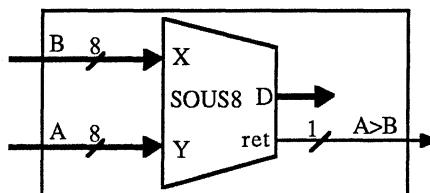


Dans ce schéma, on voit apparaître de nouveaux circuits à réaliser. On espère qu'ils sont plus simples à réaliser que le circuit donné au départ.

La comparaison  $X > Y$  nécessite un circuit de comparaison de deux nombres représentés en binaire sur huit bits (compar8) : recevant deux nombres en entrée, il délivre un booléen. En cherchant bien, on le trouverait certainement dans un catalogue de circuits ; à défaut, il faut le réaliser.

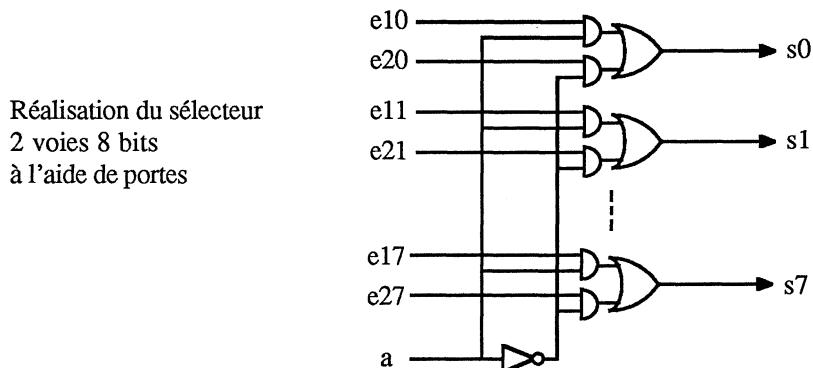


On peut penser à utiliser la retenue de la soustraction binaire, qui est significative d'une différence qui "aurait du être négative" ; d'où la réalisation suivante :



Si nous connaissons le soustracteur binaire huit bits SOUS8 (par exemple dans les catalogues), l'élément COMPAR8 est terminé, sinon il faut réaliser le soustracteur, etc...

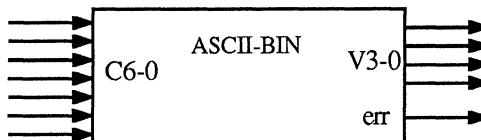
La sélection pourra être réalisée par un sélecteur à deux voies de huit bits (sel2v8) que nous savons réaliser, soit à l'aide d'adaptateurs trois-états, soit à l'aide de portes.



L'analyse du circuit MAX est ainsi terminée ; il ne reste plus qu'à rassembler les morceaux : remplacer dans le schéma global chaque élément par sa réalisation, en respectant les connexions entre blocs.

### Deuxième exemple : transformation de codes

Soit à réaliser un circuit ASCII-BIN qui transforme un code ASCII de caractère héxadécimal, c'est-à-dire dans l'ensemble {"0", ..., "9", "A", ..., "F"}, en le vecteur de 4 bits qui représente en binaire la valeur associée à ce chiffre. De plus, ce circuit doit délivrer un indicateur d'erreur *err*, à 1 si le caractère présenté en entrée n'est pas un chiffre héxadécimal.



Un rapide examen des codes ASCII nous montre que deux cas se présentent :

- pour {"0", ..., "9"}, les interprétations binaires des codes ASCII sont {30h,...39h}
- pour {"A", ..., "F"}, les interprétations binaires des codes ASCII sont {41h,...46h}

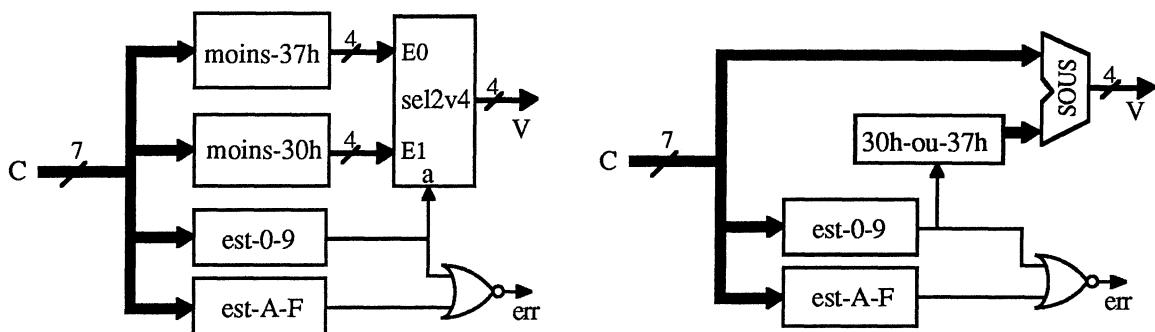
Il faut donc discerner ces deux cas, et soustraire 30h dans le premier cas, et soustraire 37h dans le second. Nous considérons donc C et V comme des nombres :

$$V = \text{SI } (\text{est-0-9}(C)) \text{ ALORS } C - 30h \text{ SINON } C - 37h$$

$$\text{err} = \text{NON } (\text{est-0-9}(C)) \text{ OU est-A-F}(C)$$

On voit apparaître de nouvelles fonctions,  $\text{est-0-9}(C)$ , qui indique que  $C$  est un élément de  $\{"0", \dots, "9"\}$ , et  $\text{est-A-F}(C)$ , qui indique que  $C$  est un élément de  $\{"A", \dots, "F"\}$ .

Cette formulation nous conduit au premier schéma général, partie gauche de la figure suivante.



On peut éviter l'emploi de deux soustracteurs, en faisant porter la sélection sur la valeur soustraite, plutôt que sur le choix du résultat ; ceci conduit à la solution de droite de la figure, qui correspond à la formulation :

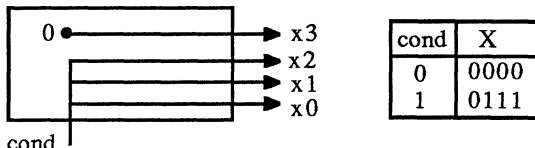
$$V = C - (\text{SI}(\text{est-0-9}(C)) \text{ ALORS } 30h \text{ SINON } 37h)$$

On peut poursuivre un peu plus l'analyse et remarquer que le résultat fait seulement 4 bits, et donc ne faire porter la soustraction que sur les 4 bits de poids faibles de  $C$  (les bits de poids faibles du résultat sont indépendants des bits de rangs plus élevés des opérandes) :

$$V = C_3 \cdot 0 - (\text{SI}(\text{est-0-9}(C)) \text{ ALORS } 0h \text{ SINON } 7h)$$

Le choix entre  $0h$  et  $7h$  peut être fait par un sélecteur à deux voies de 4 bits ; mais dans ce cas particulier, choix entre  $0$  et une valeur, il existe une solution encore plus simple :

Choix entre  $0$  et une valeur quelconque



Pour  $\text{est-0-9}$  et  $\text{est-A-F}$ , on peut les exprimer par :

$$\text{est-0-9}(C) = (C_6 \cdot 4 = 011) \text{ ET } (C_3 \cdot 0 \in \{0000, \dots, 1001\})$$

$$\text{est-A-F}(C) = (C_6 \cdot 4 = 100) \text{ ET } (C_3 \cdot 0 \in \{0001, \dots, 0110\})$$

Ces sous fonctions sont suffisamment simples pour envisager la méthode booléenne :

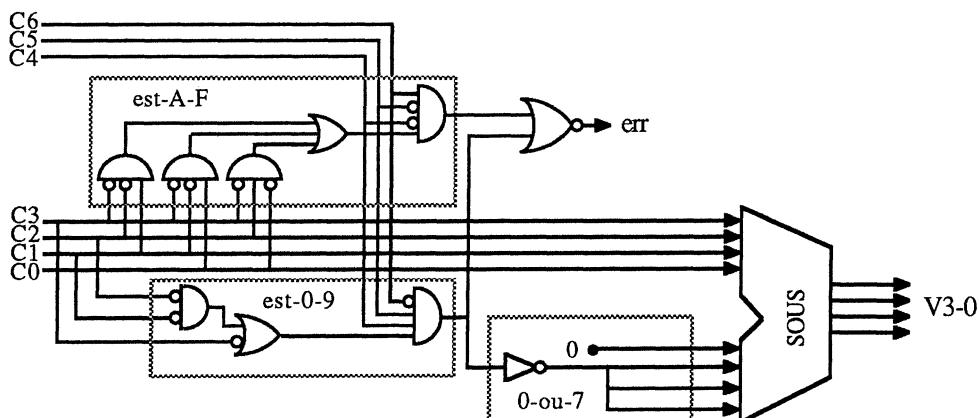
$C_3 \cdot 0 \in \{0000, \dots, 1001\}$				
$C_1 \cdot 0$				
$C_3 \cdot 2$	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11				
10	1	1		

$$/C_3 + /C_2 \cdot C_1$$

$C_3 \cdot 0 \in \{0001, \dots, 0110\}$				
$C_1 \cdot 0$				
$C_3 \cdot 2$	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11				
10				

$$/C_3 \cdot /C_2 \cdot C_1 + /C_3 \cdot C_1 \cdot C_0 + /C_3 \cdot C_2 \cdot C_0$$

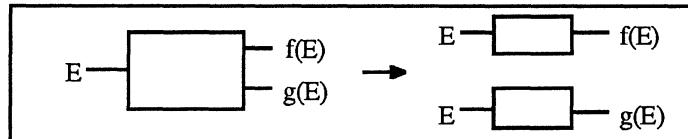
Il ne reste plus qu'à rassembler les morceaux, pour former le schéma définitif ; sur la figure suivante, nous avons laissé le découpage fonctionnel résultatnt de l'analyse qui vient d'être faite :



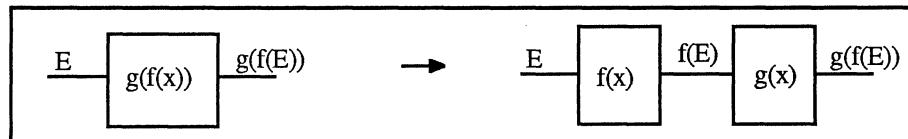
### 3.3.3 Quelques schémas de décomposition

Nous présentons maintenant les schémas de décomposition les plus usuels.

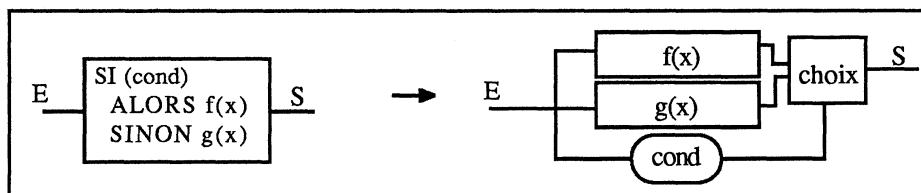
**Dissociation des sorties** : des sorties indépendantes sont analysées séparément.



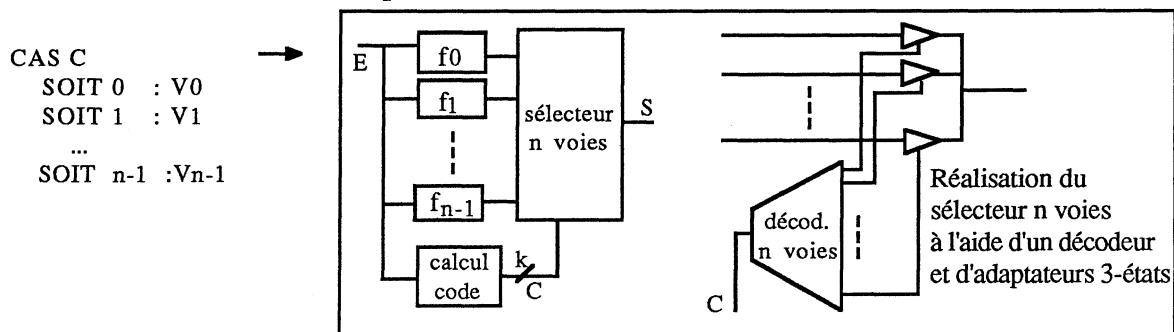
**Composition des fonctions** : (correspond à la notion de séquentialité). Les résultats délivrés par un bloc fonctionnel servent d'entrée à un autre bloc.



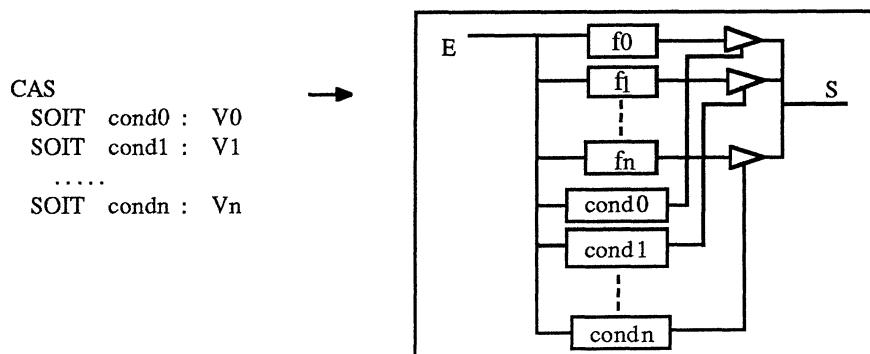
**Sélection** (notion de conditionnelle) : le résultat est l'une ou l'autre de deux valeurs, suivant le résultat de l'évaluation d'une condition.



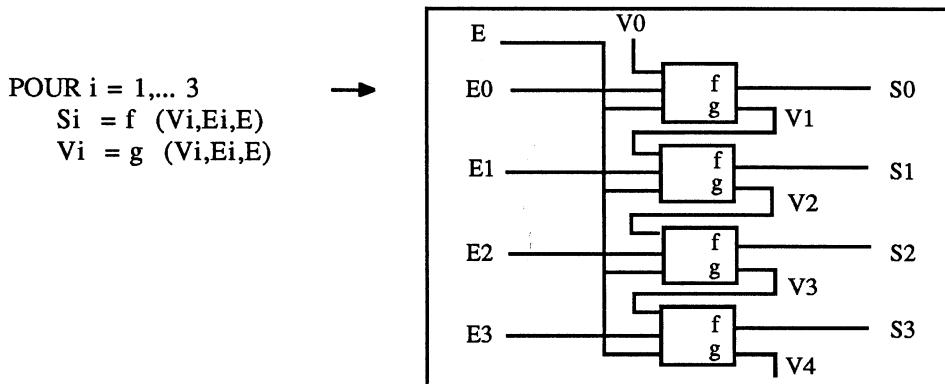
**Sélection par code** : La sélection peut être généralisée à plus de deux possibilités : selon la valeur d'un code C dans un intervalle  $\{0, 1, \dots, n-1\}$ , le résultat est respectivement une des valeurs  $V_0, V_1, \dots$  ou  $V_{n-1}$ . On peut utiliser la notation suivante :



**Sélection par conditions exclusives** : On dispose de conditions booléennes  $C_0, C_1, \dots, C_n$ , mutuellement exclusives (au plus une d'entre elles est vraie). Le résultat est l'une des valeurs  $V_0, V_1, \dots$  ou  $V_n$ , selon celle des conditions qui est vraie.



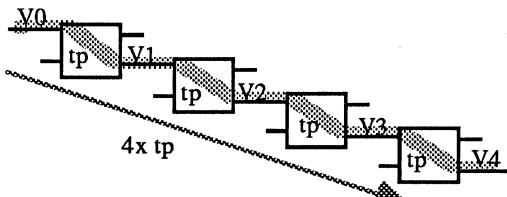
**Récurrence :** On cherche à exprimer les sorties par une formule de récurrence faisant intervenir les entrées et une variable d'itération  $v_i$ .



On remarquera que chaque étage d'itération nécessite une réplication du même circuit de base. Un exemple typique est la réalisation itérative de l'additionneur  $n$  bits à l'aide d'étages additionneurs 1 bit.

Généralement lorsqu'on réalise un circuit selon ce schéma, on en prévoit l'extension possible : on ne fixe pas la valeur de la variable d'itération du premier étage de l'itération ; elle est décidable par l'extérieur sous forme d'une entrée ( $V_0$ ) ; on prévoit également une sortie pour la valeur de la variable d'itération générée par le dernier étage ( $V_4$ ). Ainsi l'utilisateur du circuit peut facilement étendre l'itération par simple mise en cascade.

Pour un tel circuit itératif, le temps de calcul est généralement proportionnellement à la taille des données : il faut  $n \times t_p$  pour être certain d'un résultat correct de la variable d'itération générée par le dernier étage,  $n$  étant le nombre d'étages et  $t_p$  le temps de calcul pour un étage (temps de calcul pour obtenir  $V_{i+1}$  à partir de  $V_i$  et des autres entrées de l'étage).



Souvent il faudra mettre en oeuvre des techniques particulières, adaptées à chaque cas, pour améliorer ce temps : ceci consiste en général à calculer la variable d'itération par un moyen plus direct (et plus coûteux), pour plusieurs étages à la fois.

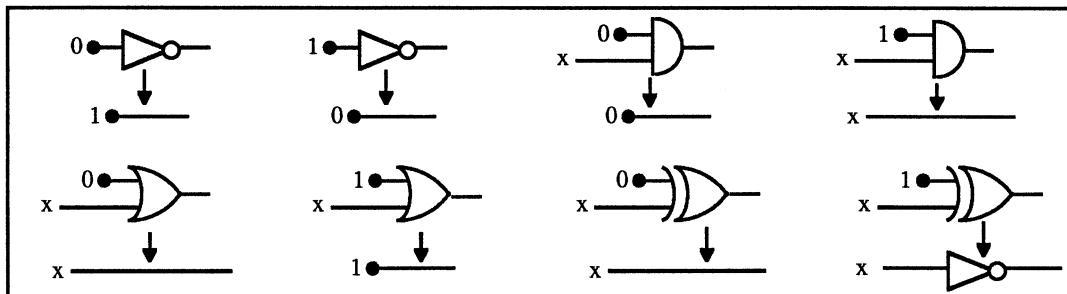
### 3.3.4 - Phase d'optimisation

La phase d'analyse conduit souvent à des circuits relativement volumineux. Dans la phase de synthèse on va s'intéresser à l'usage particulier des circuits composants du circuit total, chose dont on n'avait pratiquement pas tenu compte dans l'analyse. On va alors pouvoir pratiquer certaines simplifications dues à ces particularités ; ce sont essentiellement :

- des propriétés des entrées du circuit : entrées constantes, entrées dupliquées, etc ...
- de la non utilisation de certaines sorties d'un circuit.
- des factorisations rendues possibles par la distributivité de certaines opérations.

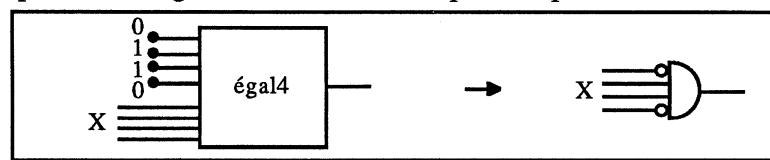
#### Propriétés connues sur les entrées

Les cas d'entrées constantes (0 ou 1) donnent lieu à des simplifications triviales :

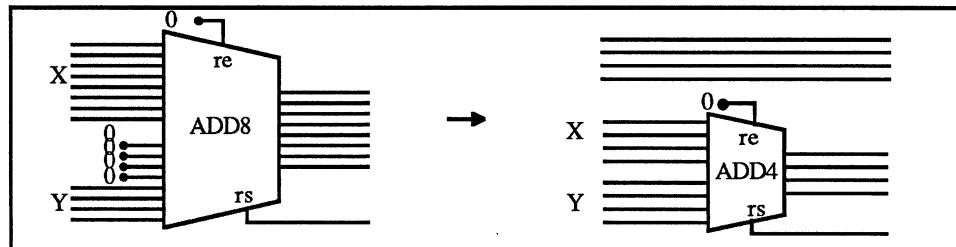


Il en est d'autres, tels les exemples suivants :

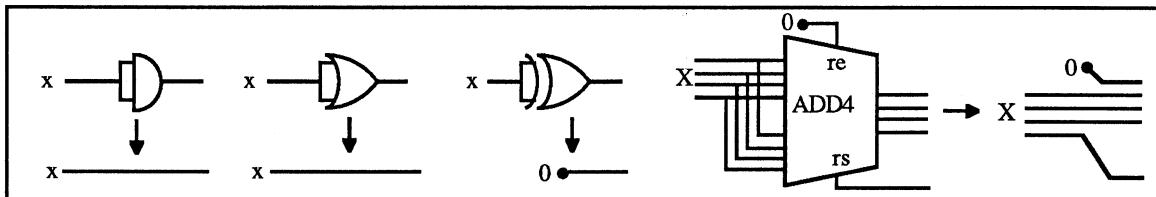
- Remplacer la comparaison d'égalité à une constante par une porte :



- L'addition à zéro se transforme en identité ; de même, l'addition binaire avec un opérande dont les bits de poids faibles valent 0 permet l'emploi d'un additionneur plus petit :



- L'égalité entre certaines entrées est aussi source de simplifications importantes :



Bien évidemment, on peut encore essayer de tirer profit de toute corrélation connue entre des entrées, par exemple : telle entrée est la négation de telle autre, etc...

### Non utilisation de certaines sorties

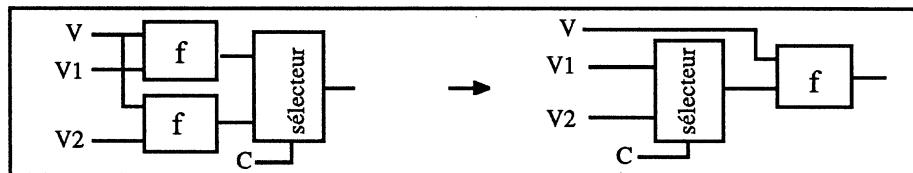
On utilise un circuit C dont certaines sorties ne sont pas utilisées ; on peut simplifier alors la réalisation de C en supprimant tous les composants qui ne servent qu'à élaborer les sorties inutilisées.

### Les factorisations

On cherche à profiter au maximum d'un même calcul intermédiaire. Une des sources les plus fréquentes de telles simplifications proviennent de la propriété suivante de la sélection :

$$\text{SI } (C) \text{ ALORS } f(V, V_1) \text{ SINON } f(V, V_2) \equiv f(V, \text{SI } C \text{ ALORS } V_1 \text{ SINON } V_2)$$

Ceci nous permet de simplifier le circuit comme indiqué ci-dessous, par remplacement des deux occurrences du calcul de  $f$  par une seule :



Ce principe se généralise facilement pour les sélections multiples.

### Exemple 1 :

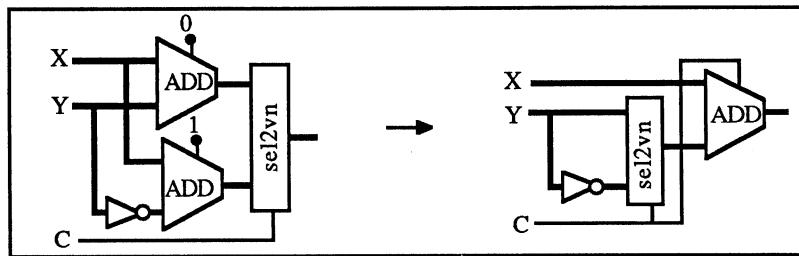
Il s'agit d'un circuit d'addition/soustraction doté d'une commande C ; pour C=0, le circuit réalise l'addition et pour C=1, la soustraction :

$$\text{SI } C \text{ ALORS différence}(X, Y) \text{ SINON somme}(X, Y),$$

En réalisant la soustraction par "addition du complément plus 1", ceci conduit à :

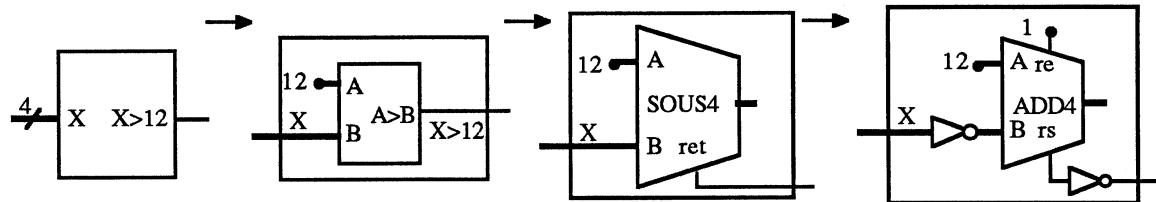
qui se factorise :  
 et s'arrange en :

SI C ALORS SOM(1,X,non(Y)) SINON SOM(0,X,Y),  
 SOM(SI C ALORS 1 SINON 0, X, SI C ALORS non(Y) SINON Y)  
 SOM(C, X, SI C ALORS non(Y) SINON Y)

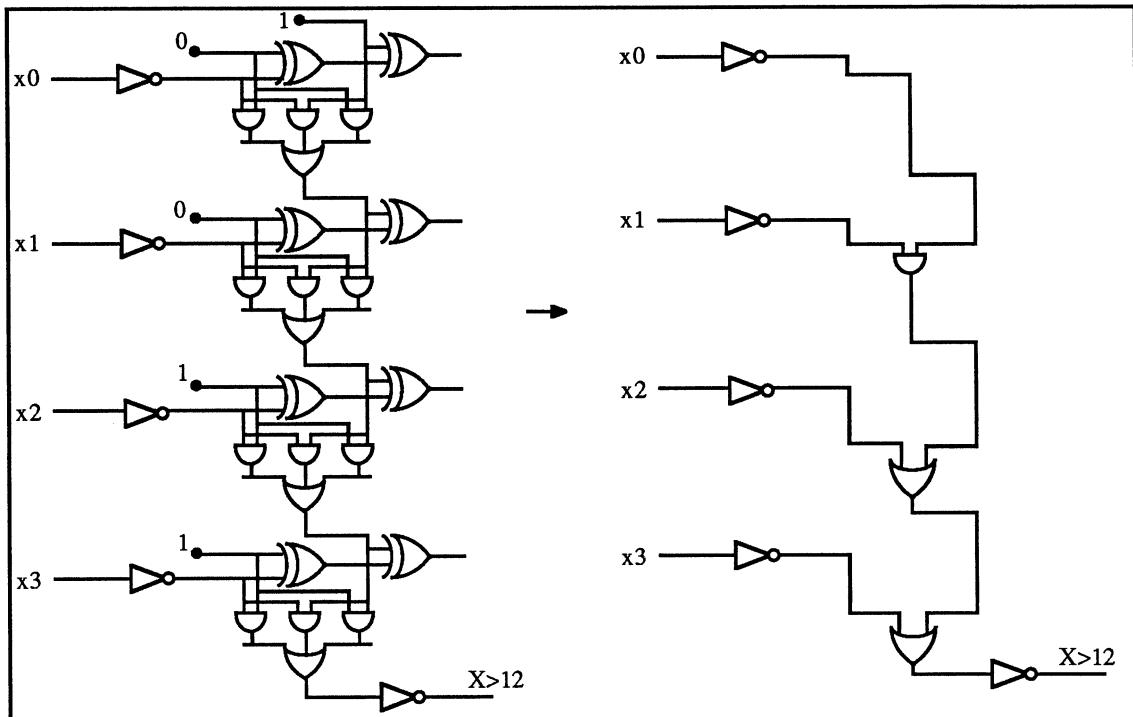


### Exemple 2 :

Soit à réaliser un comparateur de supériorité entre un nombre en binaire sur 4 bits et l'entier 12. L'analyse de ce circuit est illustrée ci-dessous : la comparaison-à-12 est réalisée par comparaison avec la constante 12, la comparaison est réalisée par la retenue d'une soustraction, et la soustraction binaire est réalisée par “addition du complément plus 1”.

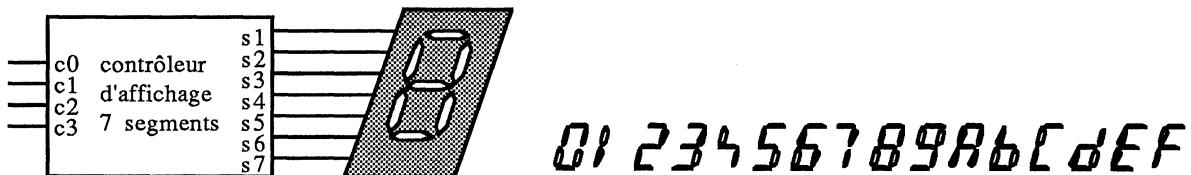


On connaît une réalisation du circuit d’addition, à base de portes et de ou-exclusifs, comme illustré sur la partie gauche de la figure suivante. En appliquant les simplifications mentionnées, on obtient la solution illustrée à droite :



**Exercice 14**

- Réaliser un circuit “contrôleur d’affichage 7 segments”; ce circuit reçoit en entrée 4 bits et délivre en sortie 7 signaux permettant de dessiner sur 7 segments lumineux l’image du chiffre hexadécimal correspondant. Chaque segment est allumé pour un 1 et éteint pour un 0.

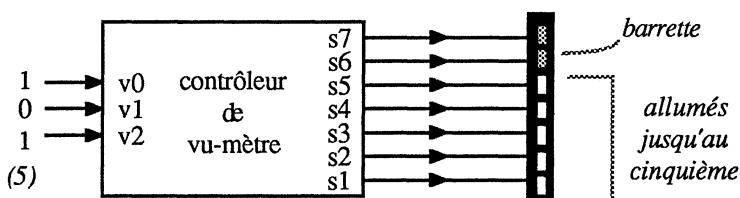
**Exercice 15**

- Réaliser un additionneur binaire modulo 3. C’est un circuit qui a pour entrées la représentation binaire de deux nombres de l’intervalle  $\{0,1,2\}$  et donne en résultat la représentation binaire de la somme modulo 3 de ces deux nombres.

**Exercice 16**

On se propose de réaliser un “contrôleur de vu-mètre” : c’est un dispositif qui permet de visualiser une grandeur numérique (un volume sonore par exemple) au moyen d’une barrette lumineuse qui s’étire plus ou moins en fonction de la grandeur.

- 1) Réaliser, par la méthode booléenne, le contrôleur de vu-mètre illustré ci-dessous. La grandeur à visualisée est représentée en binaire sur 3 bits ; la barette est constituée de 7 segments qui s’allument pour un 1. Pour la grandeur 0, la barette doit être entièrement éteinte.



- 2) Réaliser ce même circuit en utilisant judicieusement un décodeur (et des portes).
- 3) Réaliser un circuit similaire pour une entrée 4 bits, avec une barrette de 15 segments.
- 4) Réaliser un circuit similaire pour une grandeur relative (positive ou négative) représentée en complément sur 4 bits, la barette étant divisée en les 7 segments du haut pour les grandeurs positives et les 8 segments du bas pour les grandeurs négatives.

**Exercice 17**

On désire réaliser un circuit capable de décaler 16 bits de  $n$  positions à gauche avec introduction de 0 dans les positions libérées. Le nombre de positions,  $n$  (entre 0 et 15), est fourni en binaire sur une entrée du circuit.

- Envisager une solution basée sur une sélection du résultat parmi les 16 décalés possibles ;
- Envisager une solution basée sur une décomposition du décalage de  $n$  positions en décalages de 1, 2, 4 et 8 positions.
- Comparer les deux solutions, en quantité de matériel et en temps de calcul.

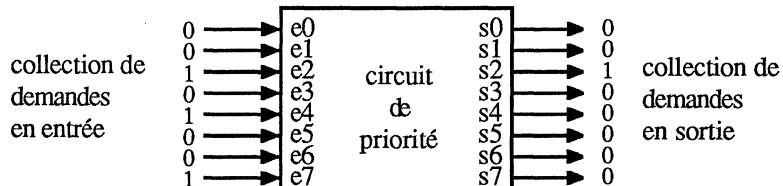
**Exercice 18**

Le circuit suivant est un *circuit de priorité*. Ce circuit reçoit en entrée une collection de demandes en provenance de 8 sources (par exemple en provenance de diverses personnes qui sont susceptibles de demander l’usage d’un appareil unique). Le numéro de chaque source

définit un ordre de priorité entre elles (0 plus prioritaire que 1 ... plus prioritaire que 7). Le circuit doit délivrer en sortie une collection de demandes, soit vide si la collection de demandes en entrée est vide, soit la collection réduite à une seule demande, celle de la plus prioritaire des sources demandeuses.

Une collection de demande est représentée par un vecteur de 8 bits, chacun associé à une source : ce bit vaut 1 s'il y a demande de la part de cette source, 0 sinon.

Exemple :



- Réaliser ce circuit, en prévoyant de pouvoir l'étendre facilement pour des collections de demandes plus grosses.

### Exercice 19

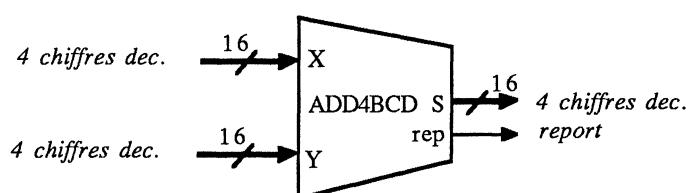
Réaliser un incrémenteur 4 bits, INC4, en particulierant le circuit additionneur réalisé à base de portes et de ou-exclusifs. Ce circuit reçoit 4 bits en entrée et délivre 5 bits en sortie qui sont le résultat de l'addition binaire de l'entrée et de 1.

### Exercice 20

Dans le même esprit que l'exercice 19, réaliser un comparateur arithmétique 4 bits qui, étant donnés deux opérandes A et B indique si A>B (interprétation binaire).

### Exercice 21

Réaliser un opérateur d'addition pour la représentation décimale DCB, par exemple pour des opérandes de 4 chiffres décimaux (16 bits donc).



- 1) On utilisera, dans un premier temps, des additionneurs binaires 4 bits : on cherchera une solution qui consiste à corriger, à l'aide d'un additionneur binaire auxiliaire, le résultat fourni par l'addition binaire de chaque paire de chiffres décimaux.
- 2) On cherchera ensuite à remplacer l'additionneur de correction par un montage à base de portes obtenu en particulierant cet additionneur auxiliaire.

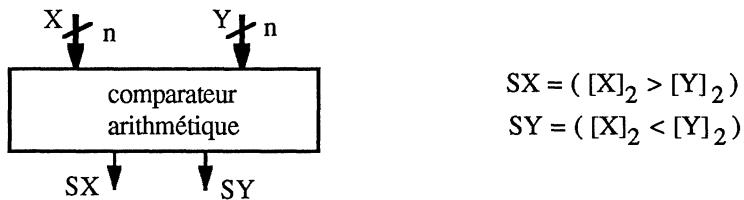
### Exercice 22

Réaliser un multiplicateur binaire pour opérandes de 4 bits. Le résultat doit être exact sur 8 bits donc. On appliquera la formule suivante :

$$_2[x_3x_2x_1x_0] \times _2[y_3y_2y_1y_0] = ([x_3]2^3 + [x_2]2^2 + [x_1]2 + [x_0]) \times _2[y_3y_2y_1y_0]$$

**Exercice 23**

On désire étudier diverses réalisations d'un comparateur arithmétique binaire  $n$  bits :



## 1) Recherche d'un schéma itératif par bit :

- 1.1 - On envisage un algorithme de comparaison qui compare bit par bit, à partir des poids forts.

On pose :  $SXi = ([X_{n-1}...X_i]_2 > [Y_{n-1}...Y_i]_2)$   
 $SYi = ([X_{n-1}...X_i]_2 < [Y_{n-1}...Y_i]_2)$

Trouver les formules de recurrence :

$$\begin{aligned} SXi &= f(SXi+1, SYi+1, Xi, Yi) \\ SYi &= g(SXi+1, SYi+1, Xi, Yi) \end{aligned}$$

- 1.2 - En déduire un schéma pour un comparateur 32 bits.

- 1.3 - On dispose de portes NAND (à 1, 2 et 3 entrées) dont le temps de calcul est  $\Delta$ . Déterminer le temps de calcul du comparateur 32 bits ainsi réalisé. Donner la formule générale pour  $n$  bits.

1) Schéma itératif par paquets de  $k$  bits :

Pour améliorer les performances, on décompose les opérandes en paquets de  $k$  bits (par exemple,  $k=4$  bits). Chaque paire de paquets peut être considéré comme une paire de chiffres  $CXi, CYi$  en base  $2^k$ , et on applique l'algorithme précédent sur ces chiffres.

On pose (comparaison des chiffres) :  $SCXi = ([CXi] > [CYi])$   
 $SCYi = ([CXi] < [CYi])$

On pose également :  $SXi = ([CX_{n-1}...CX_i]_{2^k} > [CY_{n-1}...CY_i]_{2^k})$   
 $SYi = ([CX_{n-1}...CX_i]_{2^k} < [CY_{n-1}...CY_i]_{2^k})$

- 2.1 - Trouver les formules de recurrence :

$$\begin{aligned} SXi &= f(SXi+1, SYi+1, SCXi, SCYi) \\ SYi &= g(SXi+1, SYi+1, SCXi, SCYi) \end{aligned}$$

- 2.2 - En déduire un schéma pour un comparateur 32 bits, selon une décomposition en :

- 4 paquets de 8 bits.
- 8 paquets de 4 bits.

Déterminer le temps de calcul dans chacun de ces deux cas.



# CIRCUITS SEQUENTIELS

## 1 - Notion de système séquentiel

### 1.1 - Etat interne

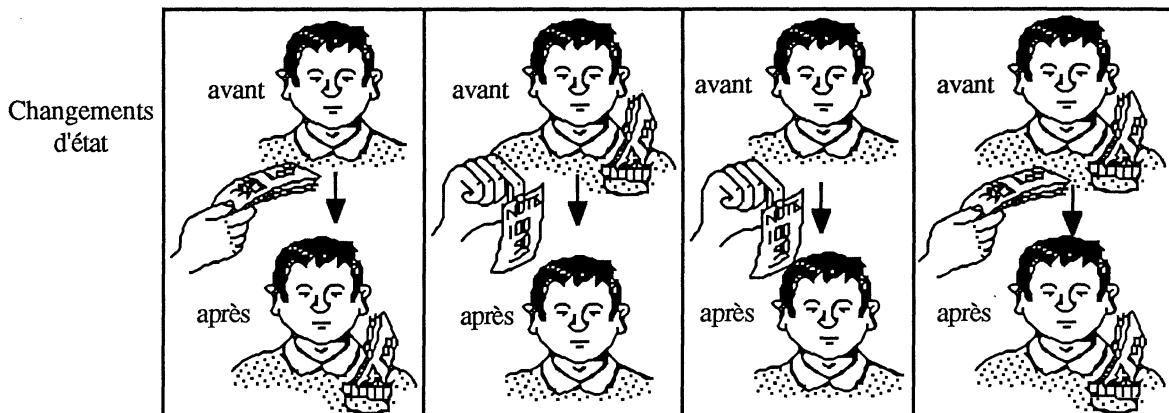
Le comportement d'un système combinatoire ne dépend que les variables d'entrée ; le personnage ci-dessous a un comportement combinatoire : selon qu'on lui montre de l'argent ou une facture, il affiche un sourire ou des pleurs.



Par contre le comportement d'un système séquentiel fait intervenir quelque chose de l'intérieur du système, que l'on appelle son *état interne* (ou plus simplement son état).

Par exemple, le personnage ci-dessous a un état qui peut avoir une des deux valeurs, "pauvre" ou "riche". Certaines actions sur les entrées provoquent un changement de l'état du système :

- on peut lui proposer de l'argent, et si le personnage est pauvre il devient riche ;
- on peut lui demander de payer une facture, et s'il est riche, il devient pauvre.



Les sorties dépendent, en plus des entrées, de l'état interne du système.

Par exemple : quand il est pauvre, le personnage affiche un sourire à la vue de l'argent et un visage flegmatique à la vue d'une facture ; quand il est riche, sa réaction est différente, il reste flegmatique à la vue de l'argent et pleure à la vue d'une facture.

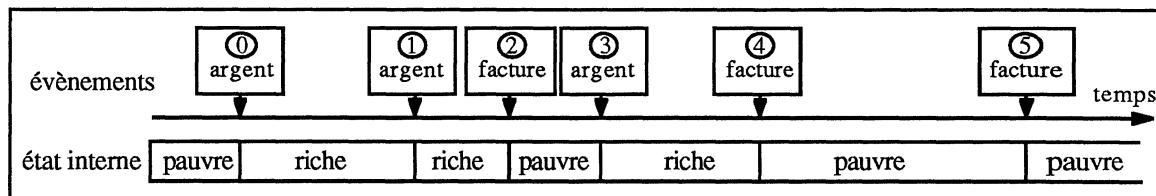


## 1.2 - Notion d'évènement

Un *évenement* est quelque-chose capable de provoquer le changement d'état interne d'un système séquentiel ; dans l'exemple ci-dessus, chaque proposition d'argent et chaque présentation de facture est un évènement.

Chaque évènement que reçoit le système a un numéro d'ordre au cours du temps : 0, 1, 2 ...

Chaque évènement a une valeur : présentation de facture ou proposition d'argent notre exemple.



### Évènements purs - évènements valués

On peut imaginer des évènements dont l'ensemble des valeurs possibles est réduit à un seul élément ; il n'y a qu'une seule possibilité, il n'y a pas d'information accompagnatrice. De tels évènements sont appelés des *événements purs*, il sont simplement une séquence d'occurrences.

Une telle séquence d'évènements purs s'appelle une *horloge* (Angl. "clock").

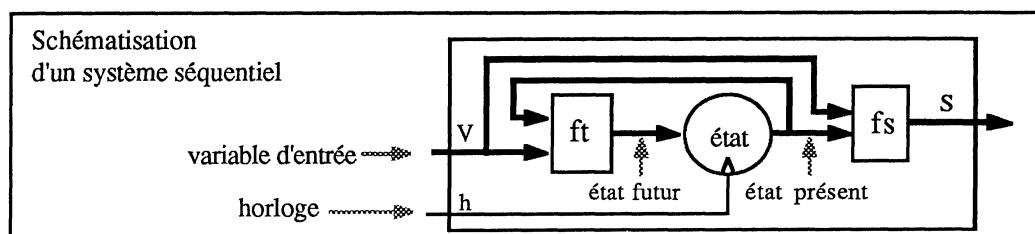
Si on dispose en plus d'une variable qui a des valeurs bien définies aux moments où se produisent des évènements purs, on peut considérer les évènements qui se produisent aux mêmes moments que ces évènements purs accompagnés de la valeur de la variable à ces moments : ceci est une technique fréquemment utilisée pour définir une séquence d'évènements valués à partir d'une séquence d'évènements purs.

Par exemple les "arrivées du facteur" sont des évènements purs ; on pourra prendre pour variable la "main du facteur", avec deux valeurs possibles, porteuse d'argent ou porteuse de facture, et considérer que chaque arrivée du facteur accompagnée de la valeur de sa main à ce moment constitue un évènement valué dont la valeur est soit une proposition d'argent, soit une présentation de facture.

## 1.3 - Description d'un système séquentiel

Un système séquentiel peut être décrit par :

- des entrées de valeur (V),
- une entrée d'horloge (h) qui fournit une séquence d'évènements purs (des tops),
- un état interne, qui peut prendre un nombre fini de valeurs,
- des sorties de valeurs (S),
- une fonction de transition (ft), qui définit la valeur future de l'état à partir de la valeur présente de l'état et des entrées,
- une fonction de sortie (fs), qui définit la valeur présente des sorties à partir la valeur présente de l'état et des entrées.



Dans le schéma ci-dessus, il faut imaginer l'état interne comme le contenu d'une mémoire, et chaque top d'horloge comme l'ordre d'écriture de cette mémoire ; ainsi, la machine étant dans un certain état, on peut changer à volonté les valeurs V, cela n'a aucune influence sur l'état interne ; pour changer d'état, la machine doit recevoir un top sur h, et le nouvel état

après le top est fonction de l'état avant le top et des entrées V au moment du top. Dans cette image, il faut considérer chaque top comme comme infiniment bref, c'est-à-dire comme séparant le temps de façon infiniment précise entre avant et après le top, et l'écriture du nouvel état comme infiniment rapide ; nous verrons comment cela peut être réalisé, au paragraphe sur les horloges et les dispositifs de mémorisation.

Un tel système est également appelé "machine d'état fini" (Angl. "*finite state machine*") ; on dit aussi "automate d'état fini" ou plus simplement "automate".

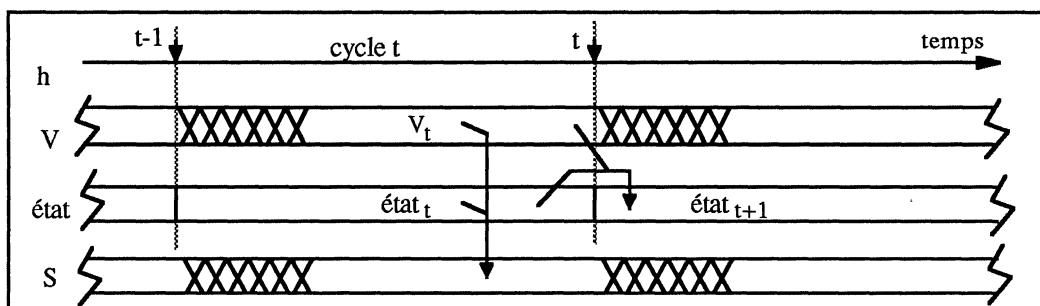
On a coutume de noter un tel système de la façon suivante :

$$\begin{aligned} S &= fs(V, \text{état}) \\ \text{état}_{t+1} &= ft(V_t, \text{état}_t) \end{aligned}$$

Dans une telle formulation,  $t$  est un nombre entier, numéro d'ordre d'un évènement ; l'intervalle entre deux évènements consécutifs s'appelle un cycle ; "état<sub>t</sub>" est la valeur de l'état pendant le cycle  $t$  ; enfin,  $V_t$  est la valeur de la variable d'entrée au moment de l'évènement  $t$ .

Une autre notation souvent employée (et qui signifie la même chose) est la suivante :

$$\begin{aligned} S &= fs(V, \text{état}) \\ \text{état} &:= ft(V, \text{état}) \end{aligned}$$



### Table des états - Diagramme des états

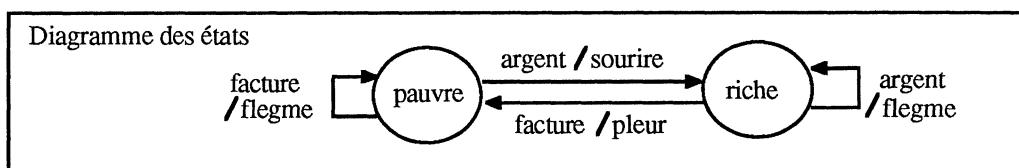
Les deux fonctions  $ft$  et  $fs$  peuvent être énoncées par n'importe quelle technique ; dans les cas simples, elles peuvent être données par une table ; on l'appelle "table des états", par exemple :

Table des états :

V	état	$ft(V, \text{état})$	$fs(V, \text{état})$
argent	pauvre	riche	sourire
facture	pauvre	pauvre	flegme
argent	riche	riche	flegme
facture	riche	pauvre	pleur

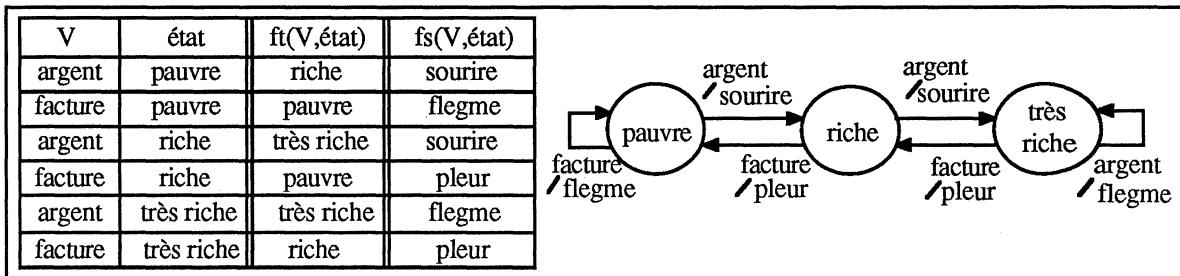
On utilise également une notation graphique appelée "diagramme des états" :

- chaque valeur possible de l'état est inscrite dans un cercle,
- chaque transition  $e_2 = ft(x, e_1)$  est représentée par une flèche de  $e_1$  vers  $e_2$  porteur de chaque valeur  $x$  de la variable d'entrée qui provoque cette transition,
- chaque transition est également accompagnée de la valeur correspondante de la sortie,  $fs(x, e_1)$ .



Pour lire un tel diagramme, il faut imaginer le système séquentiel comme un jeton quelque part sur un des cercles ; l'état présent du système est là où se trouve le jeton ; si on veut savoir la sortie pour telle valeur de l'entrée, on cherche la flèche porteuse de cette valeur et on lit ce qu'il y a sous le “/” qui l'accompagne ; si on veut changer d'état, on suit la flèche porteuse de la valeur d'entrée et on déplace le jeton vers le cercle d'arrivée. Ainsi, un fonctionnement particulier du système correspond à une promenade dans le diagramme.

Le système pris en exemple est un peu trop simple : l'état après un évènement ne dépend pas (vraiment) de l'état avant cet évènement, il ne dépend que de V au moment de l'évènement. L'exemple suivant reflète mieux le cas général :

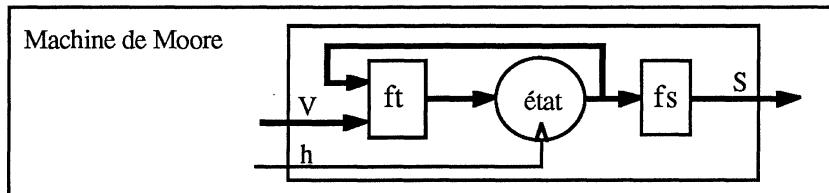


### Machines de Mealy - machines de Moore

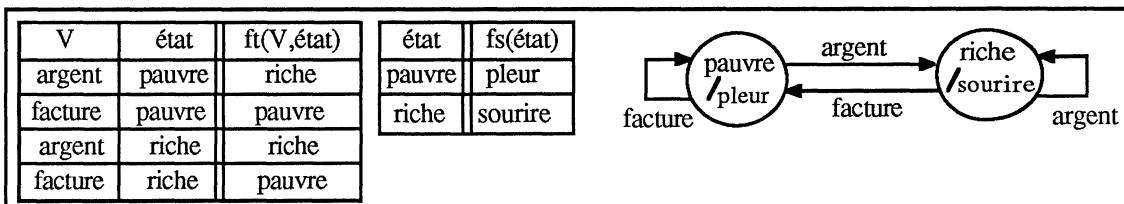
Un système séquentiel général, dans lequel les sorties dépendent combinatoirement de l'état et des entrées s'appelle une “machine de Mealy”. Les deux systèmes précédents sont des machines de Mealy.

Il existe des cas particuliers, dans lesquels les sorties ne dépendent que de l'état ; on les appelle des “machines de Moore”. Une machine de Moore a une description légèrement plus simple :

$$\begin{aligned} S &= fs(\text{état}) \\ \text{état} &:= ft(V, \text{état}) \end{aligned}$$



Exemple : un personnage qui sourit quand il est riche et pleure quand il est pauvre.



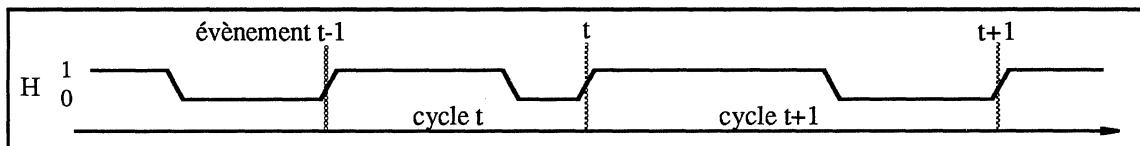
Dans le diagramme des états, on peut noter les valeurs de sortie directement sur les états, puisque la valeur de sortie ne dépend que de l'état. La sortie ne change pas pendant tout un cycle puisqu'elle ne dépend combinatoirement que de l'état qui lui-même ne change pas pendant un cycle ; c'est cette propriété qui conduit souvent à concevoir une machine de Moore plutôt qu'une machine de Mealy pour certaines applications.

## 2 - Circuits synchrones

Un circuit synchrone est un système séquentiel qui comporte :

- des entrées de valeurs (V),
- des sorties (S),
- une entrée horloge (H) qui sert à provoquer les changements d'état.

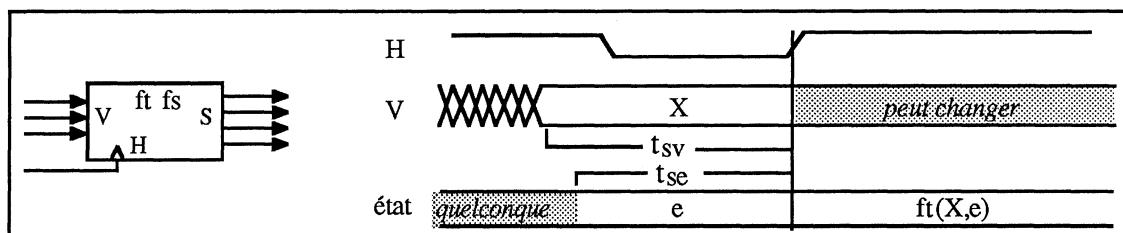
L'horloge H est un signal binaire qui passe alternativement à 0 et à 1. Les événements sont les passages de 0 à 1 de l'entrée H ; on dit également "fronts montants" ou "transitions positives".



### Comportement d'un circuit synchrone

#### En ce qui concerne le changement d'état :

Une valeur logique (X) doit être maintenue fixe pendant au moins une certaine durée  $t_{sv}$  et l'état doit avoir une valeur inchangée (e) depuis au moins une certaine durée  $t_{se}$  avant le front montant de H. L'état du circuit prend la valeur donnée par la fonction de transition, c'est-à-dire  $f_t(X,e)$ .

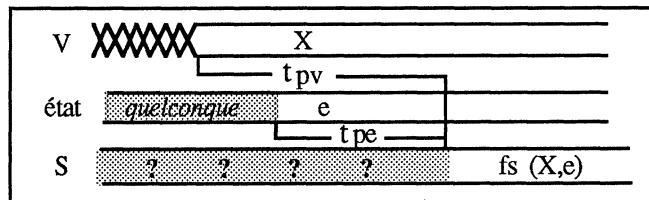


$t_{sv}$  et  $t_{se}$  sont les *temps de prépositionnement* (Angl. "set-up-time") des entrées et de l'état ; ce sont les durées nécessaires pour calculer le futur état.

Généralement, on peut changer V aussitôt après le front de H. Cependant, certains circuits exigent que V soit maintenu un peu après le front de H, pendant une durée  $t_h$  appelée *temps de maintien* (Angl. "hold-time") ; ces circuits sont de mauvais circuits, difficiles à utiliser en certains cas. Certains circuits, par contre, permettent que V soit changé un peu avant le front de H, ils sont alors caractérisés par un  $t_h$  négatif.

#### En ce qui concerne les sorties :

Les sorties se comportent exactement comme un circuit combinatoire qui aurait V et l'état pour entrées : si V est maintenu à une valeur logique (X) depuis une certaine durée  $t_{pv}$  et si l'état a une valeur inchangée (e) depuis une certaine durée  $t_{pe}$ , alors la sortie affiche  $fs(X,e)$ .



Ces circuits sont appelés synchrones, ce qui signifie que :

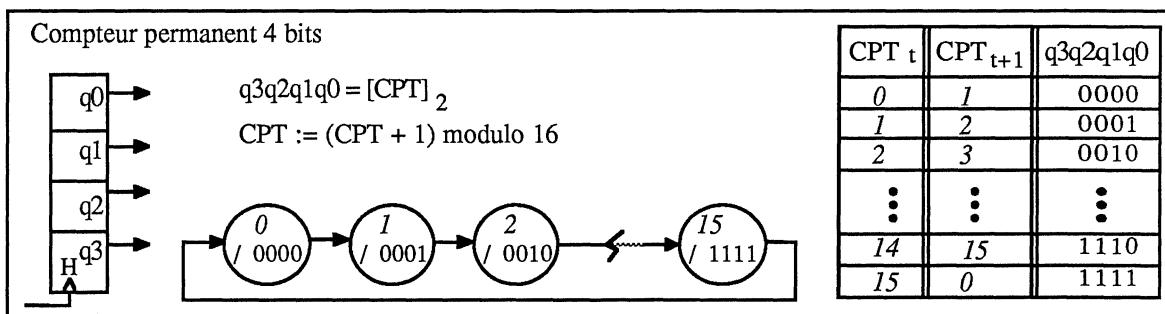
- le moment du changement d'état est sous la seule dépendance de l'entrée d'horloge,
- les autres entrées n'agissent pas par elles-mêmes ; pour déterminer le nouvel état, seule compte leur valeur au moment du coup d'horloge (et un peu avant).

### 3 - Quelques exemples de circuits séquentiels

#### 3.1 - Compteurs

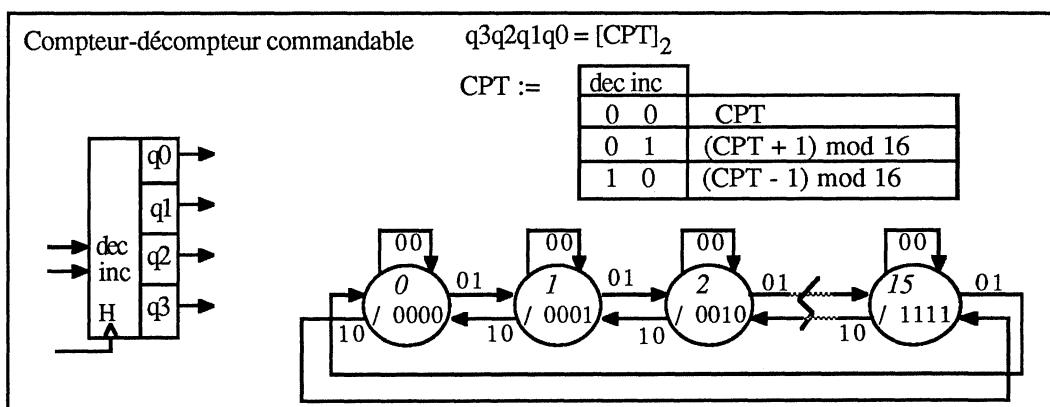
##### Compteur permanent

La figure suivante montre un compteur 4 bits ; c'est un circuit qui n'a que l'entrée d'horloge H. Son état a 16 valeurs possibles, 0, 1 ... 15, et à chaque coup d'horloge son état s'incrémente, repassant à 0 à la suite de 15. Il a 4 sorties qui affichent l'état en binaire sur 4 bits.



##### Compteur-décompteur commandable

Ce circuit dispose de deux entrées *inc* et *dec* qui permettent de contrôler le changement d'état : lors d'un front montant de H, selon les valeurs de *inc* et *dec*, ou bien l'état ne change pas, ou bien il est incrémenté, ou bien il est décrémenté. Dans ce circuit, comme dans tous les circuits qui nous intéressent ici, seule l'horloge a une action sur l'état, c'est la seule source d'événements ; si par exemple on s'excite sur les deux entrées *inc* ou *dec*, en les maintenant ou en les faisant changer de toutes les façons possibles, rien ne se passe si H ne transite pas de 0 à 1.



Dans cet exemple, la fonction de transition n'est pas définie pour *inc,dec*=11.

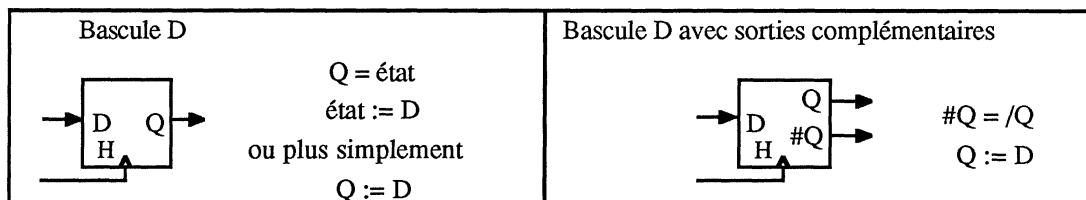
Pour de tels compteurs, réalisés en technologie courante, les durées précédemment mentionnées ont pour ordre de grandeur : tse = 30ns, tsv = 15ns, tpe = 10ns, th = 0ns . Ceci signifie que la fréquence d'horloge maximum est fHmax= 33MHz (1/tse), et pour chaque cycle les commandes *inc* et *dec* doivent avoir une valeur correcte et stable 15ns avant la fin du cycle. De plus, la valeur des sorties est garantie correcte 10ns après chaque coup d'horloge.

### 3.2 - Bascules

Les bascules sont des circuits dont l'état a deux valeurs possibles, 0 et 1. L'état contient donc exactement 1 bit d'information.

#### Bascule D

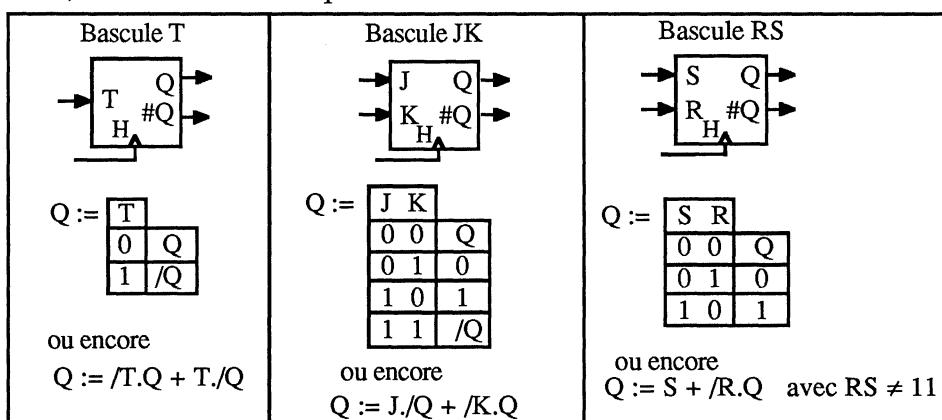
La bascule D est la plus simple et la plus utile, des bascules. C'est une mémoire élémentaire de 1 bit, qui enregistre la donnée présente sur l'entrée D au moment du front d'horloge. Elle affiche son état en sortie Q. Certaines bascules offrent en plus une sortie  $\#Q$  qui est le complément de l'état.



Cette bascule est parfois appelée "délais", car sa sortie Q affiche l'entrée D retardée d'un cycle (plus exactement, la sortie Q vaut ce que vaut D à la fin du cycle précédent).

#### Bascules T, JK et RS

Les bascules T, JK et RS ont le comportement suivant :



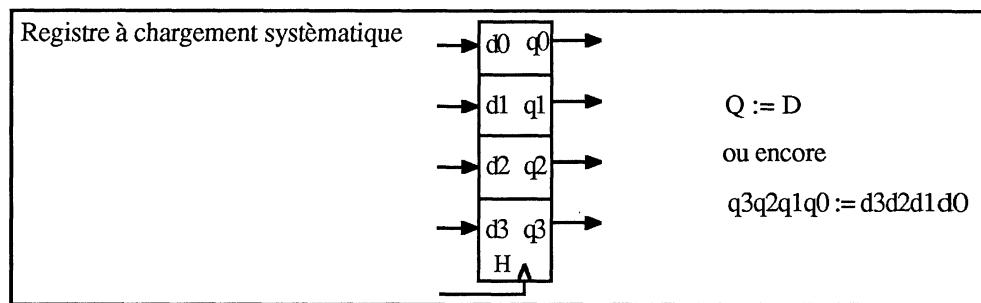
- La bascule T complémente son état si son entrée T est à 1, sinon l'état est inchangé.
- La bascule JK permet, selon les valeurs de J et K, de laisser l'état inchangé, de le mettre à 1, de le mettre à 0 ou de le complémenter.
- La bascule RS (Angl. "set, reset") ressemble à la JK, mais elle n'a pas la possibilité d'inverser l'état. Le comportement n'est pas défini pour RS=11, c'est-à-dire que si on l'utilise avec RS=11, on ne peut pas affirmer quel est l'état après le coup d'horloge.

### 3.3 - Registres

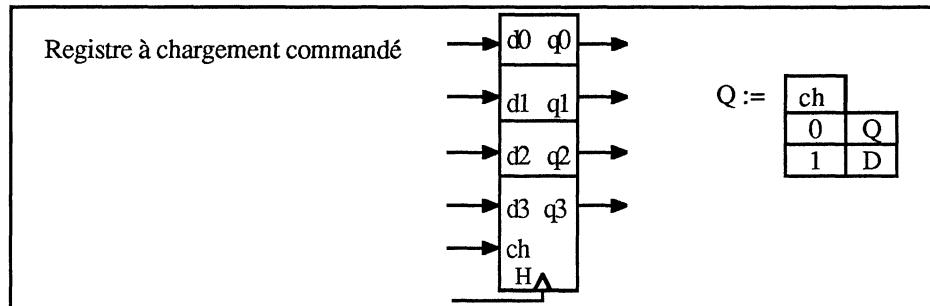
Les registres ont un état composé de  $n$  bits (généralement  $n = 4$  ou  $8$ ) ; l'état est directement disponible en sortie et n'importe quelle valeur présentée en entrée peut être enregistrée au moment du front d'horloge. Les registres sont en fait la juxtaposition de plusieurs bascules.

#### Registre à chargement systématique

Ce circuit enregistre, à chaque front de H, les données présentées en entrée. Il fonctionne exactement comme plusieurs bascules D mises en parallèle :

**Registre à chargement commandé**

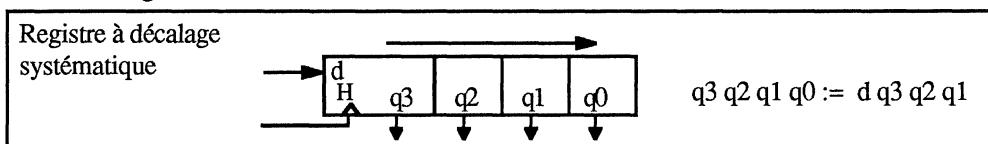
Ce circuit enregistre la donnée présentée en entrée si l'entrée de commande de chargement (ch) est à 1, sinon l'état reste inchangé.

**3.4 - Registres à décalage**

Les registres à décalage sont des registres dont l'état, formé de n bits, peut être décalé d'une position, vers la droite ou vers la gauche.

**Registre à décalage systématique**

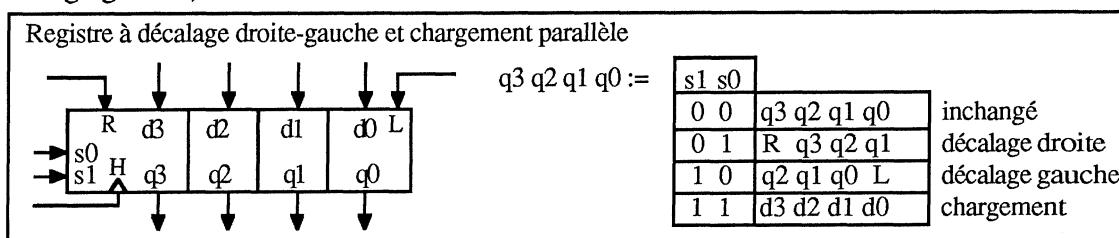
A chaque coup d'horloge, ce circuit décale son état d'une position de bit vers la droite (vers les indices faibles) et enregistre l'entrée de donnée  $d$  (1 bit) dans la position de bit libérée par le décalage.



Un tel registre sert souvent à convertir des données "séries", c'est-à-dire présentées bit par bit sur une ligne au cours du temps, en des données "parallèles", présentées en une fois sur plusieurs lignes ; c'est pourquoi l'entrée  $d$  est appelée "entrée série", et les sorties  $q$  sont appelées "sorties parallèles".

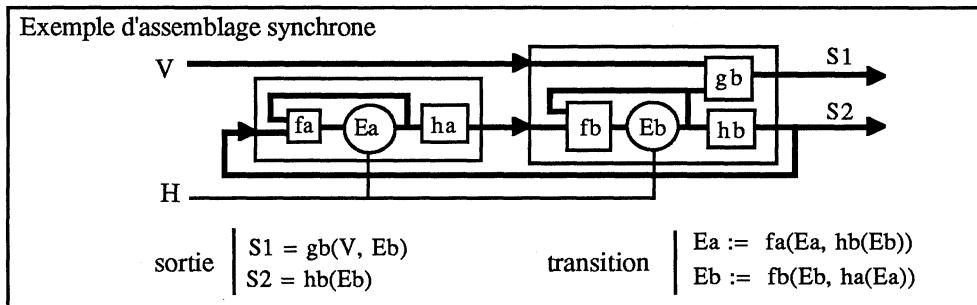
**Registre à décalage droite-gauche et chargement parallèle**

Ce circuit permet, selon ce qui est présenté sur les entrées de commande  $s_1, s_0$ , soit de laisser l'état inchangé, soit de le décaler vers la droite, soit de le décaler vers la gauche, soit d'enregistrer une donnée présentée sur les entrées parallèles  $d_3-0$ . Lors des décalages, la position de bit libérée prend la valeur présente sur l'entrée  $R$  (décalage droite) ou  $L$  (décalage gauche).



## 4 - Assemblages synchrones

On peut interconnecter plusieurs circuits synchrones et relier toutes les entrées d'horloge à une horloge unique : on obtient ainsi un nouveau circuit synchrone, dont l'état est formé de la composition des états des composants (le produit cartésien) et dont la fonction de transition et la fonction de sortie dépendent des interconnections réalisées.



On ne doit pas créer de “boucles de dépendance combinatoire” : partant d'une entrée de circuit et cheminant le long des connexions des sorties qui en dépendent combinatoirement, on ne doit pas retomber pas sur cette entrée.

Par contre on peut créer des boucles, à condition que ce soit par l'intermédiaire de l'état (dépendance séquentielle) ; sur l'exemple, c'est le cas de la boucle “fa Ea ha fb Eb hb fa”.

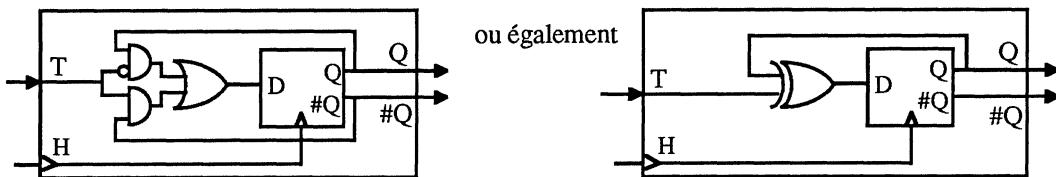
### Exemples d'assemblages synchrones

Les exercices qui suivent ont pour but de s'entrainer à combiner des circuits séquentiels ; les exemples sont suffisamment simples pour être abordés intuitivement, sans méthode particulière : il suffit de trouver quelle fonction (de l'état et des entrées du circuit total) appliquer sur les entrées des composants utilisés pour que le circuit total ait le comportement désiré.

#### Exercice 1

Réaliser les bascules T, JK et RS à l'aide d'une bascule D et de portes.

Pour exemple, nous traitons ici la réalisation de la bascule T ; pour avoir le comportement d'une bascule T, il faut appliquer sur l'entrée D de la bascule utilisée une certaine fonction de T (entrée du circuit total) et de l'état Q de la bascule utilisée ; pour T=0, l'état doit rester inchangé : il suffit donc dans ce cas d'appliquer Q sur D ; pour T=1, l'état doit être complémenté : il suffit donc dans ce cas d'appliquer /Q sur D. Ceci conduit à la réalisation suivante :



#### Exercice 2

Réaliser la bascule D à l'aide de la bascule T , la bascule D à l'aide de la bascule JK, la bascule T à l'aide de la bascule JK.

#### Exercice 3

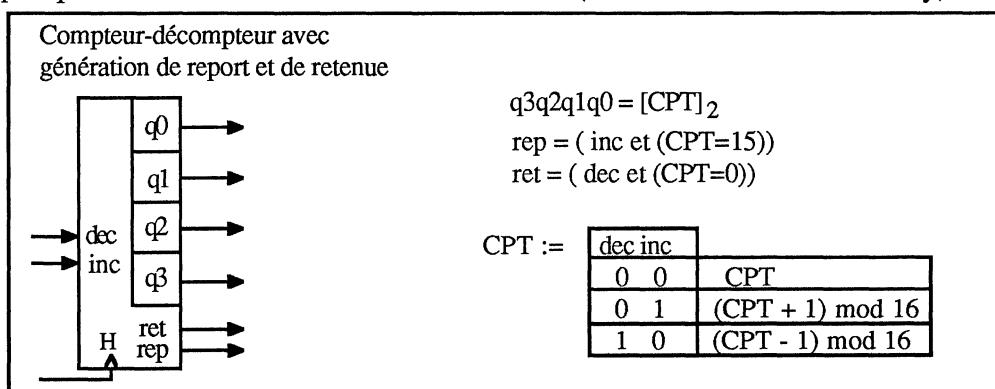
Réaliser un registre à chargement commandé en utilisant un registre à chargement systématique.

**Exercice 4**

- Réaliser un registre à décalage droite-gauche et à chargement parallèle, en utilisant un registre à chargement systématique.
- Réaliser un registre 8 bits à décalage droite-gauche et à chargement parallèle à l'aide de registres 4 bits à décalage droite-gauche et à chargement parallèle.

**Exercice 5**

- Réaliser un compteur-décompteur commandable en utilisant un registre à chargement commandé et un additionneur binaire 4 bits.
- Réaliser un compteur-décompteur 4 bits qui signale sur deux sorties, *ret* et *rep*, une retenue lorsque l'on a l'intention de décrémenter alors que le compteur est à 0 et un report lorsque l'on a l'intention d'incrémenter alors que le compteur est à 15. Dans cet exemple, *ret* et *rep* dépendent de l'état et des entrées *inc* et *dec* (c'est une machine de Mealy).



- A l'aide de tels circuits, réaliser un compteur-décompteur 8 bits (modulo 256).

**Exercice 6**

Réaliser un compteur-décompteur 4 bits à l'aide de bascules T et de portes ; On utilisera le principe suivant :

l'incrémation est réalisée en complémentant les bits de poids faibles jusqu'à la rencontre du premier 0 :  $xxx..xx0111 + 1 = xxx..xx1000$

la décrémation est réalisée en complémentant les bits de poids faibles jusqu'à la rencontre du premier 1 :  $xxx..xx1000 - 1 = xxx..xx0111$

Commencer par étudier indépendamment la réalisation d'un compteur, et d'un décompteur, en cherchant une solution modulaire par bit, puis étendre ces solutions au cas du compteur décompteur.

## 5 - Synthèse à l'aide de bascules

### 5.1 - Principes généraux

Un système séquentiel “simple” peut être réalisé à l'aide de bascules ; un système est simple lorsqu'on peut le décrire totalement à l'aide :

- soit de son diagramme d'états,
- soit de tables pour sa fonction de transition et sa fonction de sortie.

Ceci suppose que le nombre d'état du système soit petit, moins de quelques dizaines d'états.

L'analyse du problème conduit à définir

- le nombre des valeurs possibles pour son état interne  $e$  :  $e \in \{A, B, C \dots\}$
- sa fonction de transition  $ft(V, e)$
- sa fonction de sortie  $fs(V, e)$

L'état interne  $e$  doit être codé sur un nombre suffisant de bits (au moins  $n$  bits pour  $2^n$  états)

$$e \leftrightarrow q_1 q_2 \dots q_n$$

N'importe quel codage est acceptable, car l'état interne ne se voit pas depuis l'extérieur du circuit, seules les sorties sont visibles (en ce sens l'état est une entité abstraite, un pur intermédiaire de définition du comportement du système ; on peut le concrétiser comme bon nous semble)

Ayant choisi un codage de l'état, on transforme les fonctions  $ft$  et  $fs$  en leur analogue,  $FT$  et  $FT$ , sur les codes de l'état :

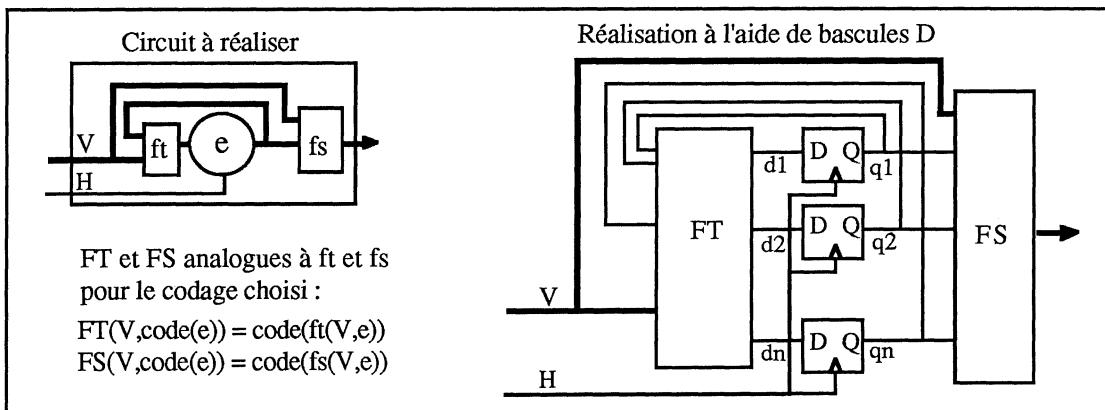
$$\begin{aligned} FT(V, q_1 q_2 \dots q_n) &\leftrightarrow ft(V, e) \\ FS(V, q_1 q_2 \dots q_n) &\leftrightarrow fs(V, e) \end{aligned}$$

On utilise une bascule pour chaque bit du code de l'état : généralement on prend des bascules D, car les autres types de bascules n'apportent pas grand chose de plus.

### Utilisation de bascules D

L'utilisation de la bascule D est simple et naturelle : il suffit d'appliquer  $FT(V, q_1 q_2 \dots q_n)$  sur les entrées  $d_1 d_2 \dots d_n$  des bascules, ce qui réalise le comportement désiré :

$$q_1 q_2 \dots q_n := FT(V, q_1 q_2 \dots q_n)$$



Exemple :

On désire réaliser un circuit d'arbitrage entre deux clients susceptibles d'utiliser une ressource qui n'admet qu'un seul occupant à la fois (par exemple deux enfants qui veulent jouer au vélo, et il n'y a qu'un vélo) ; le but du circuit est d'accorder correctement la ressource à un seul client à la fois.

Les clients manifestent respectivement sur  $e_1$  et  $e_2$  leurs désirs d'utiliser la ressource :

$e_i = 1$  si le client  $i$  désire la ressource,  $e_i = 0$  s'il ne la désire pas.

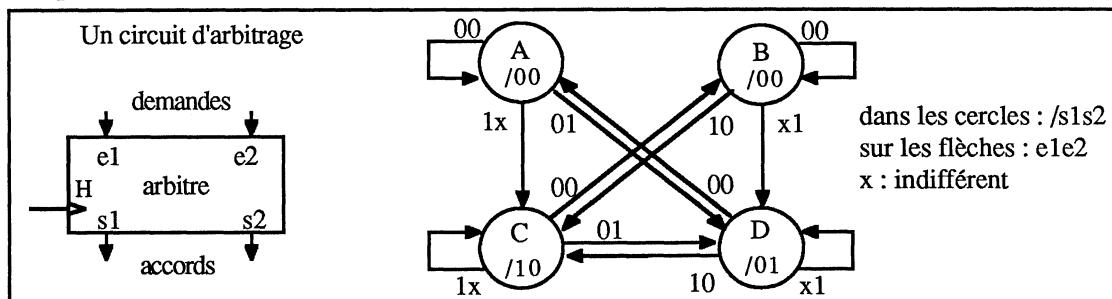
Le circuit génère sur  $s_1$  et  $s_2$  l'attribution de la ressource à chaque client :

$s_i = 1$  si la ressource est attribuée au client  $i$ ,  $s_i = 0$  sinon.

Lorsque la ressource est attribuée à un client, elle le demeure tant que ce client la désire.

Lorsque la ressource est libre, elle est attribuée au premier qui la demande ; en cas de demandes simultanées, elle est attribuée selon une certaine priorité, mais pour éviter les injustices, la priorité entre les clients change à chaque attribution : le client qui obtient la ressource devient le moins prioritaire.

L'analyse du problème conduit à l'automate suivant : il y a quatre états, deux états A et B dans lesquels la ressource est libre, et deux états C et D dans lesquels la ressource est occupée.



Codage de l'état : nous prendrons un codage dense, sur 2 bits  $q_1$  et  $q_2$ .

codage de l'état

état	$q_1 q_2$
A	0 0
B	0 1
C	1 0
D	1 1

fonction de transition

$$q_1 q_2 := FT(e_1, e_2, q_1, q_2)$$

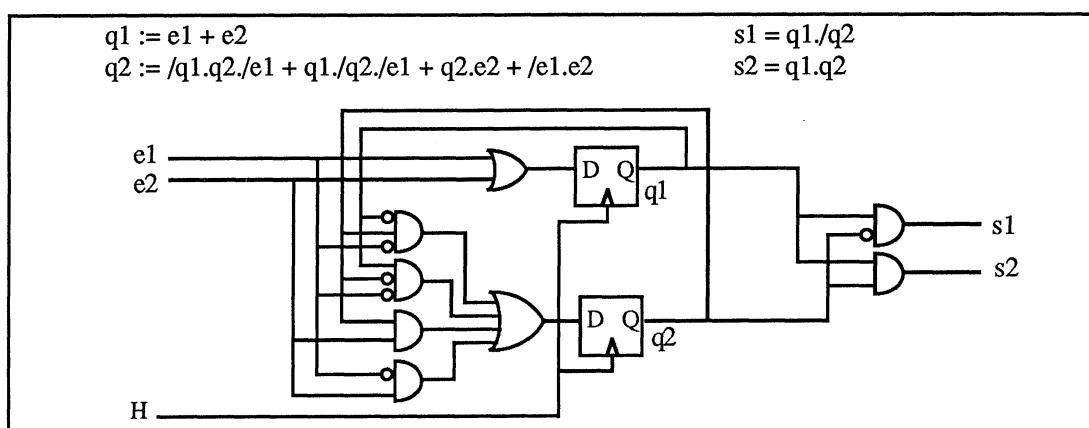
		e <sub>1</sub> e <sub>2</sub>	00	01	11	10
		q <sub>1</sub> q <sub>2</sub>	00	11	10	10
		00	00	11	10	10
		01	01	11	11	10
		11	00	11	11	10
		10	01	11	10	10

fonction de sortie

$$s_1 s_2 = FS(q_1, q_2)$$

$q_1 q_2$	$s_1 s_2$
0 0	0 0
0 1	0 0
1 1	0 1
1 0	1 0

Ce qui conduit au circuit :



## Utilisation de bascules JK

L'utilisation de bascules JK est moins naturelle : pour chaque bascule  $q_i$  il faut déterminer deux fonctions à appliquer sur  $J_i$  et  $K_i$  ; la méthode pratique est la suivante : on s'intéresse aux 4 cas de transition de la bascule  $q_i$  :

- |                            |   |
|----------------------------|---|
| $FT_{Ti}(\dots0\dots) = 0$ | $\rightarrow J_i=0 K_i=x$ ( $K_i$ peut être quelconque) |
| $FT_{Ti}(\dots1\dots) = 1$ | $\rightarrow J_i=x K_i=0$ ( $J_i$ peut être quelconque) |
| $FT_{Ti}(\dots0\dots) = 1$ | $\rightarrow J_i=1 K_i=x$ ( $K_i$ peut être quelconque) |
| $FT_{Ti}(\dots1\dots) = 0$ | $\rightarrow J_i=x K_i=1$ ( $J_i$ peut être quelconque) |

Ceci conduit pour  $J_i$  et  $K_i$  à des fonctions incomplètes que l'on peut optimiser.

Avec l'exemple précédent :

q1	e1e2	00	01	11	10
q1q2	00	0	1	1	1
00	0	1	1	1	1
01	0	1	1	1	1
11	0	1	1	1	1
10	0	1	1	1	1

J1	e1e2	00	01	11	10
q1q2	00	0	1	1	1
00	0	1	1	1	1
01	0	1	1	1	1
11	x	x	x	x	x
10	x	x	x	x	x

K1	e1e2	00	01	11	10
q1q2	00	x	x	x	x
00	x	x	x	x	x
01	x	x	x	x	x
11	1	0	0	0	0
10	1	0	0	0	0

q2	e1e2	00	01	11	10
q1q2	00	0	1	0	0
00	0	1	0	0	0
01	1	1	1	0	0
11	0	1	1	0	0
10	1	1	0	0	0

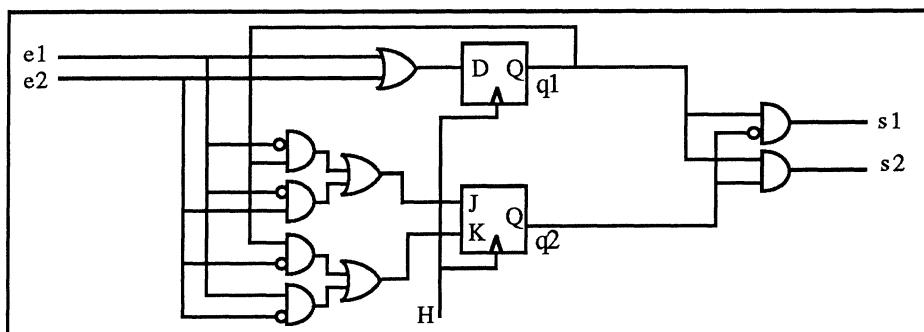
J2	e1e2	00	01	11	10
q1q2	00	0	1	0	0
00	0	1	0	0	0
01	x	x	x	x	x
11	x	x	x	x	x
10	1	1	0	0	0

K2	e1e2	00	01	11	10
q1q2	00	x	x	x	x
00	x	x	x	x	x
01	0	0	0	0	1
11	1	0	0	0	1
10	x	x	x	x	x

Ce qui donne :

$$\begin{aligned} J1 &= e1 + e2 \\ K1 &= /e1.e2 \\ J2 &= /e1.q1 + /e1.e2 \\ K2 &= q1/e2 + e1./e2 \end{aligned}$$

On a intérêt ici à conserver une bascule D pour  $q_1$ , ce qui donne le schéma suivant :



Le codage de l'état est laissé libre au concepteur du circuit ; pour un problème donné, un codage peut s'avérer meilleur qu'un autre, c'est-à-dire simplifier la réalisation de FT et FS, mais c'est souvent perdre son temps que d'entamer une telle recherche.

Il y a cependant deux sortes de codages qu'il faut connaître, car ils donnent souvent d'assez bons résultats : ce sont le codage à un bit par état le codage "incorporant les sorties".

## 5.2 - Codage à un bit par état : machine “à jeton”

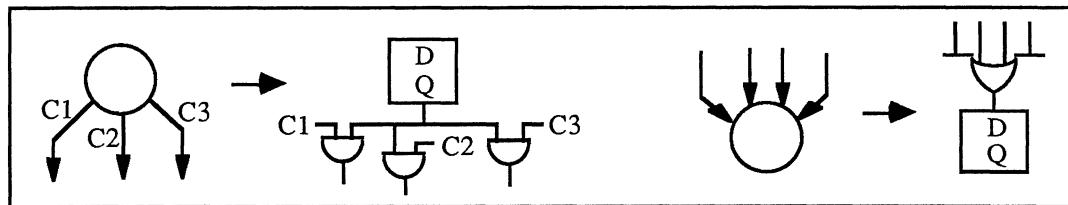
On associe chaque bascule à un état, et à tout moment une seule des bascules est à 1, celle qui est associée à l'état courant :

état	A	B	C	D	...
codage	00001	00010	00100	01000	

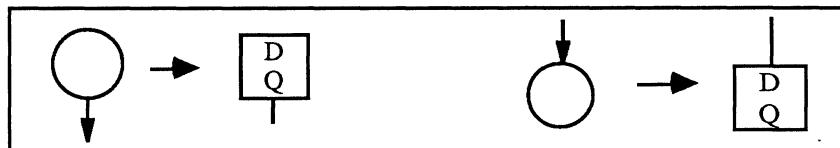
Le “1” se comporte comme un jeton qui circule de bascule en bascule, de façon analogue à l'état qui circule de cercle en cercle dans le graphe du diagramme des états.

Le circuit d'interconnexion des bascules D est directement calqué sur le diagramme des états :

- chaque sortie de flèche d'un état correspond à une porte ET entre la sortie de la bascule et la condition portée par la flèche,
- chaque entrée de bascule est un OU de toutes les flèches incidentes sur l'état correspondant.

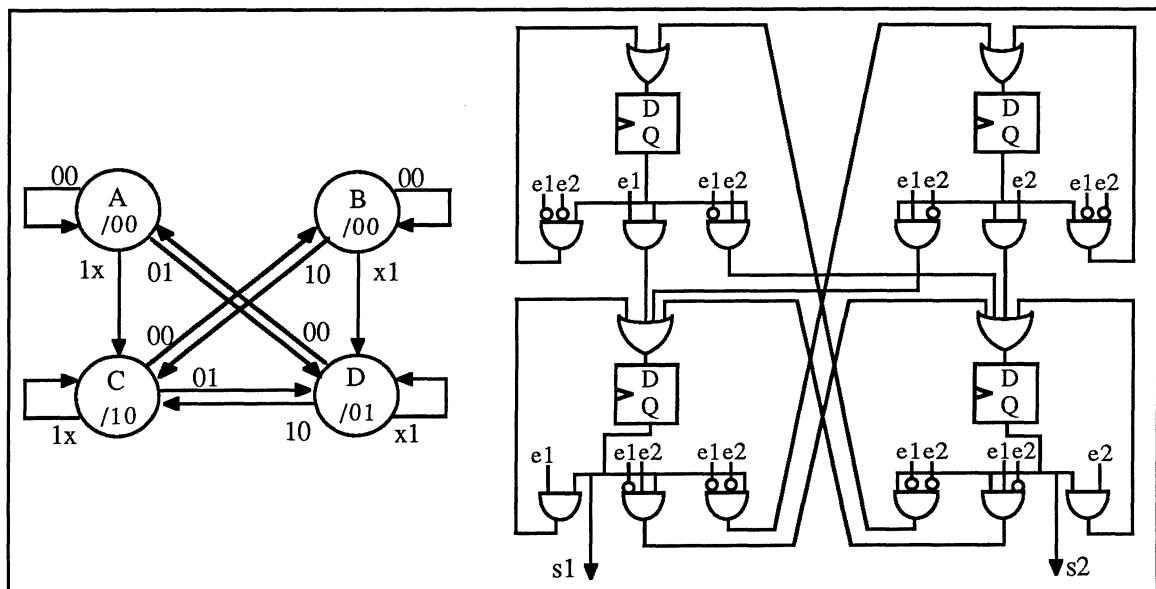


Bien évidemment, ce schéma général se simplifie en supprimant les ET pour les transitions inconditionnelles et les OU à une seule entrée :



La génération des sorties est également simple à concevoir : chaque bit de sortie est simplement le OU des sorties de bascules correspondant aux états où cette sortie vaut 1.

Le schéma suivant montre la réalisation du circuit d'arbitrage selon cette méthode :



### 5.3 - Codage incorporant les sorties

Lorsque les sorties (ou certaines sorties) ne sont fonction que de l'état (Moore), on peut prendre ces bits de sortie comme bits d'état. Bien évidemment, dans le cas général les sorties ne caractérisent pas l'état à elles seules (des états différents affichent les mêmes sorties) : on complète alors le codage de l'état par un nombre suffisant de bits.

L'avantage ici est que la fonction de sortie est toute réalisée : les sorties sont directement des sorties de bascules. On est souvent amené à un tel codage lorsqu'on utilise des composants programmables (PAL) qui offrent un nombre limité de sorties dont une grande partie sont des sorties de bascules. On est amené à ceci également lorsque l'on doit générer des sorties "sans aléas de commutation" : les sorties étant prélevées en sorties de bascules, on est certain qu'elles restent stables ou transitent proprement.

Avec l'exemple précédent, on prendra  $q_1=s_1$ ,  $q_2=s_2$ , et pour compléter l'état, une bascule  $q_3$  :

	$q_1$	$q_2$	$q_3$
A	0	0	0
B	0	0	1
C	1	0	x
D	0	1	x

Nous laissons au lecteur le soin de réaliser le schéma avec ce codage.

### Exercice 7

On considère les deux systèmes séquentiels présentés au début du chapitre, le personnage qui peut être pauvre ou riche, et le personnage qui peut être pauvre, riche ou très riche .

Réaliser un circuit séquentiel qui simule chacun de ces systèmes, avec les correspondances suivantes :

entrée : E	sortie : S1S2
0 $\leftrightarrow$ facture	0 0 $\leftrightarrow$ flegme
1 $\leftrightarrow$ argent	0 1 $\leftrightarrow$ sourire
	1 0 $\leftrightarrow$ pleur

### Exercice 8

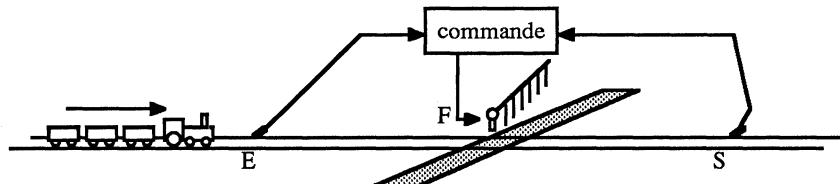
On désire réaliser un "détecteur de transitions positives" : une entrée  $e$  varie parmi 0 et 1, la durée des 0 et des 1 étant un nombre entier de cycles quelconque ; la sortie  $s$  doit valoir 1 uniquement pendant le cycle qui suit le passage de 0 à 1 de l'entrée.



- Concevoir un diagramme des états de ce circuit.
- Réaliser ce circuit à l'aide de bascules D.
- Réaliser ce circuit à l'aide de bascules JK.

**Exercice 9**

Le circuit suivant doit commander un passage à niveau :

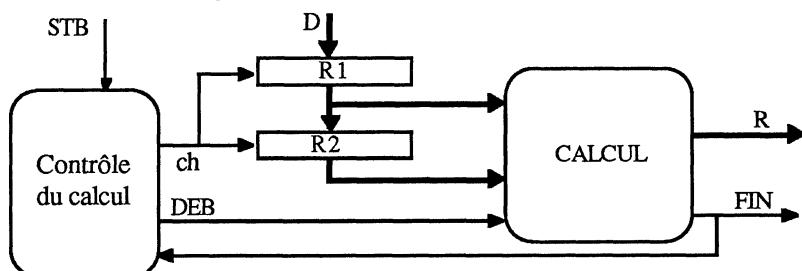


Deux capteurs E (entrée) et S (sortie) signalent l'entrée et la sortie d'un train ; les signaux E et S durent exactement 1 cycle pour le passage d'un train ; il peut y avoir au maximum 2 trains entre les points E et S ; un train peut entrer alors que simultanément un autre sort ; le passage à niveau doit être fermé s'il y a au moins un train entre E et S.

- Donner un diagramme d'état du circuit de commande.
- Réaliser ce circuit à l'aide de bascules D.

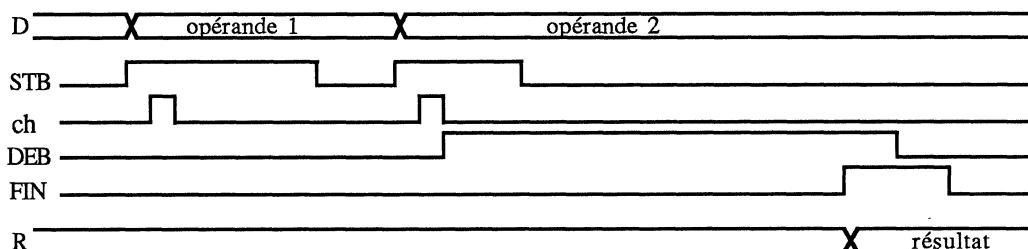
**Exercice 10**

Une machine à calculer est organisée comme suit :



Deux opérandes sont introduits séquentiellement par l'entrée D, et enregistrés dans R1 et R2. Chaque opérande est accompagné d'un signal STB à 1.

Une fois les opérandes rangés, la partie contrôle lance le calcul en affichant le signal DEB=1 ; le calcul dure un nombre de cycle quelconque et variable ; la fin du calcul est signalée par FIN=1. Le résultat est affiché sur R, et un nouveau calcul peut être alors lancé.



On se propose de réaliser la partie "contrôle du calcul".

- Donner un diagramme d'états de ce circuit.
- Donner son schéma à base de bascules D
  - avec un codage dense de l'état
  - avec un codage "machine à jeton".

**Exercice 11**

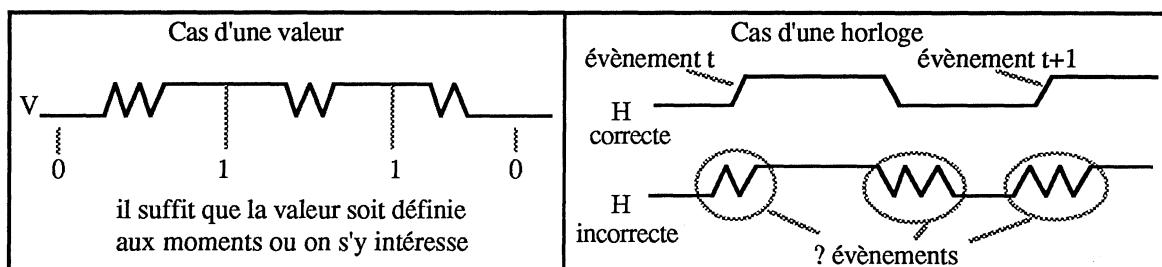
On désire réaliser un circuit d'arbitrage, analogue à celui donné en exemple, mais pour trois clients ; on conservera le même principe de priorité tournante : le client qui obtient l'accord devient le moins prioritaire pour les demandes ultérieures.

- 1) Donner les aspects externes de ce circuits (entrées et sorties).
- 2) Dessiner son diagramme d'états.
- 3) Envisager la réalisation du circuit ; chercher si possible une réalisation modulaire par client qui soit facilement généralisable à  $n$  clients.

## 6 - Manipulation de signaux

### 6.1 - Filtrage et fusion d'horloges

Une horloge est un signal qui prend alternativement les valeurs 0 et 1, et de ce point de vue elle est de même nature physique que les autres variables. Cependant pour manipuler des horloges il faut prendre certaines précautions, car le rôle d'une horloge est différent du rôle d'une variable. Pour une variable, tout ce qui compte c'est sa valeur au moment où on s'y intéresse, et peu importe si elle prend cette valeur en passant par un nombre quelconque de transitions intermédiaires. Par contre, pour une horloge, ce qui compte c'est justement la transition : lorsque l'horloge passe de 0 à 1 ou de 1 à 0, elle doit le faire en une fois, sans hésitation.

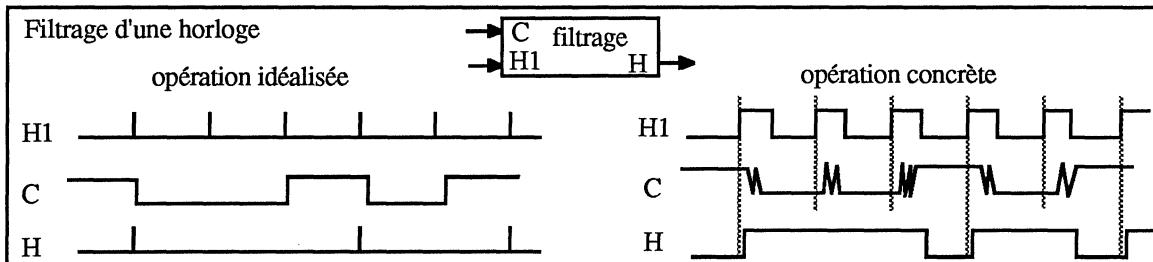


Nous appellerons "signal" une variable dont l'évolution au cours du temps continu est importante pour le fonctionnement du circuit, contrairement à une variable qualifiée de "valeur" pour laquelle seulement la valeur pendant certaines périodes compte ; l'horloge est un signal en ce sens.

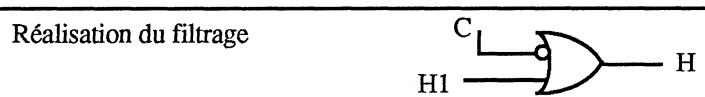
Il y a deux opérations que l'on peut définir sur les horloges : le filtrage et la fusion.

#### Filtrage d'une horloge par une condition

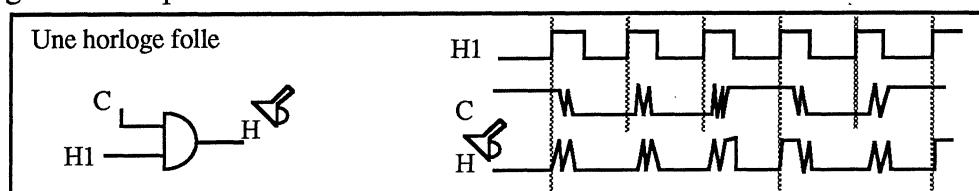
Cette opération consiste, à partir d'une horloge H1 et d'une condition C, à fabriquer une horloge H formée uniquement des événements de H1 qui se produisent lorsque C est vraie.



La condition est supposée changer de façon correcte vis-à-vis de H1 : C peut prendre des valeurs quelconques pendant le début du cycle ( $H1=1$ ) ; on demande par contre que C ait pris sa valeur correcte (0 ou 1 stable) à la fin du cycle, plus précisément pendant toute la période  $H=0$ . Sous ces hypothèses, le filtrage ci-dessus peut être réalisé par une porte OU :



Remarque : un raisonnement naïf aurait pu conduire à «si C vaut 1, l'horloge doit passer, si C vaut 0 elle ne doit pas passer : donc  $H = C \cdot H1$ », ce qui ne marche pas ici, comme le montre le diagramme temporel suivant :

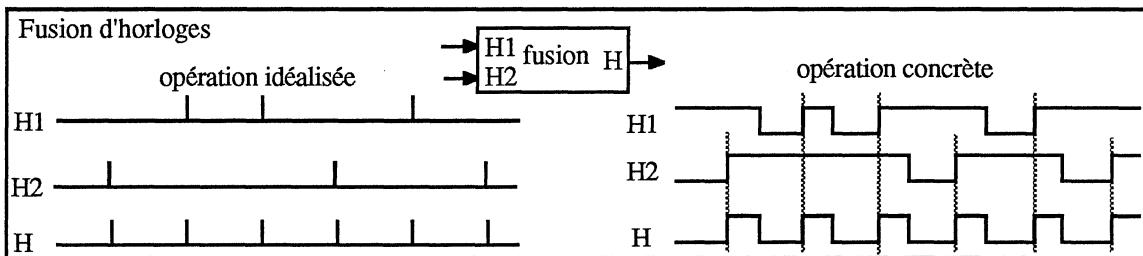


Ceci montre qu'il faut être prudent avec les manipulations d'horloges. De façon pratique, il faut procéder ainsi :

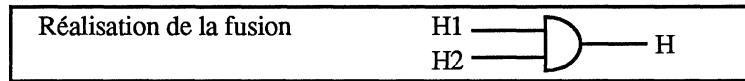
- Dessiner un diagramme temporel montrant la forme du signal désiré en fonction des signaux d'entrée.
- Tirer de ce diagramme une table donnant la sortie en fonction de la valeur des entrées, en tenant compte que la sortie doit avoir une valeur définie, même si certaines entrées n'ont pas de valeur définies (cas de C en début de cycle dans l'exemple).
- Utiliser des portes, en considérant que :
  - un 1 en entrée de OU détermine un 1 à sa sortie, indépendamment des autres entrées.
  - un 0 en entrée de ET détermine un 0 à sa sortie, indépendamment des autres entrées.

### Fusion d'horloges mutuellement exclusives

A partir de deux horloges H1 et H2 mutuellement exclusives, c'est-à-dire dont les événements ne sont pas simultanés (sont suffisamment espacés), on fabrique l'horloge qui contient les événements de ces deux horloges.



Nous nous plaçons dans le cas concret où les deux signaux H1 et H2 ne sont jamais à 0 en même temps : dans ce cas, la fusion peut être réalisée par une porte ET :

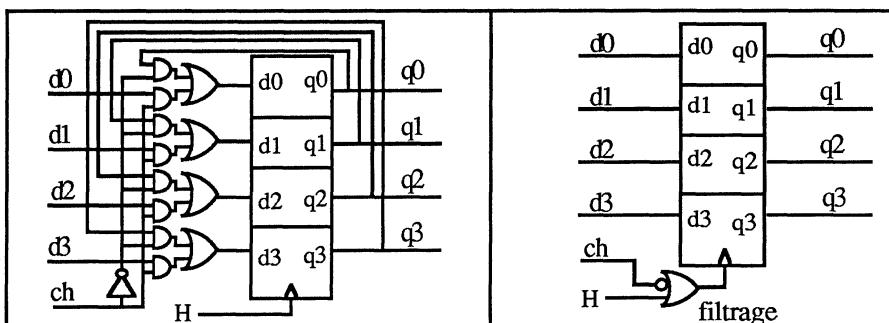


### Applications du filtrage et de la fusion d'horloge

#### Chargement conditionnel

On a souvent besoin de réaliser un changement d'état conditionnel, c'est-à-dire changer d'état si une condition est vraie et rester dans le même état si elle est fausse. Il y a deux façons de rester dans le même état :

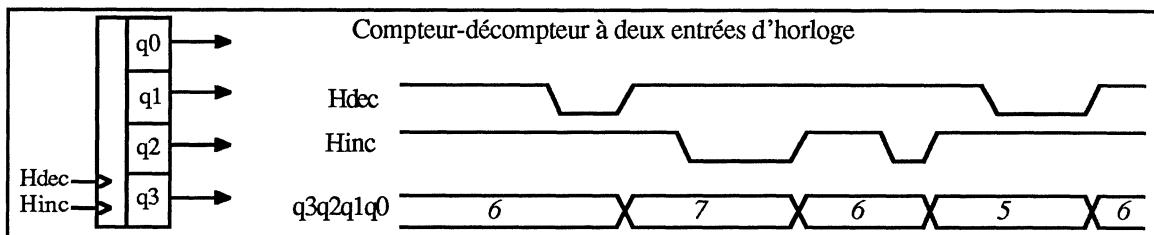
- La première façon consiste à changer d'état pour le même état : il suffit de recycler l'état en entrée des bascules .
- La seconde façon consiste à ne pas changer d'état en filtrant l'horloge par la condition.



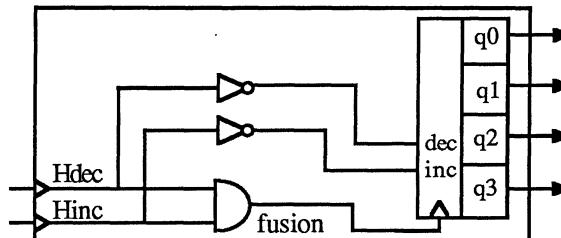
#### Circuits à plusieurs entrées d'horloge

Certains circuits ont plusieurs entrées d'horloge, et selon qu'une transition se produit sur l'une ou l'autre des entrées d'horloge, cela provoque tel changement d'état ou tel autre.

Prenons pour exemple un compteur-décompteur muni de deux entrées d'horloge, l'une (Hinc) provocant l'incrémentation du compteur et l'autre (Hdec) la décrémentation.



La figure suivante montre comment réaliser un tel compteur-décompteur à partir d'un compteur-décompteur à une seule entrée d'horloge :



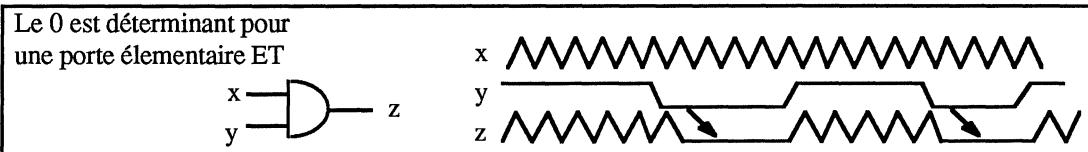
On remarquera que Hdec et Hinc ont été complémentées pour générer les commandes *inc* et *dec*, car ces horloges sont à 0 lorsqu'elles sont actives, avant leur front montant.

Remarque : dans le cas d'un filtrage ou d'une fusion, l'horloge résultante n'est pas exactement synchrone avec les horloges d'entrée, mais est légèrement retardée du temps de traversée d'une porte (de l'ordre de 5ns) ; on admettra qu'un tel décalage entre les horloges n'est pas gênant (c'est vrai si les circuits ont un  $t_h$  légèrement négatif) ; cela interdit cependant de pratiquer plusieurs opérations en cascade sur les horloges, à cause des retards cumulés.

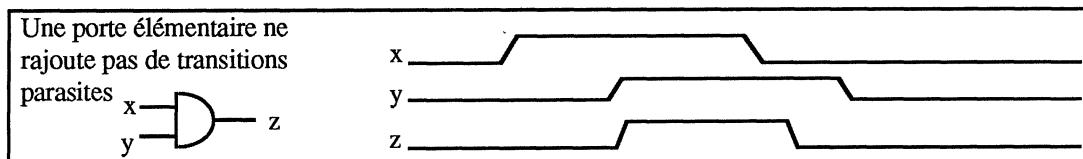
## 6.2 - Propriétés dynamiques des portes - Aléas de commutation

Nous présentons ici les propriétés, simples mais essentielles, des portes offertes par les constructeurs. Mais d'abord une petite définition : on dit qu'une valeur est *déterminante* pour un argument d'une fonction si cette valeur pour cet argument détermine à lui seul la valeur de la fonction, indépendamment de la valeur des autres arguments. Ainsi, le 0 est déterminant pour un argument du ET, et le 1 est déterminant pour un argument du OU.

**Détermination** : si une entrée de porte a une valeur déterminante depuis au moins  $t_p$ , la sortie a la valeur donnée par la fonction indiquée pour la porte. Ceci est garanti, même si les autres entrées changent, ou valent des tensions sans signification logique. Cette propriété permet d'assurer qu'un signal a une valeur déterminée, alors que certaines variables dont il dépend sont indéterminées.



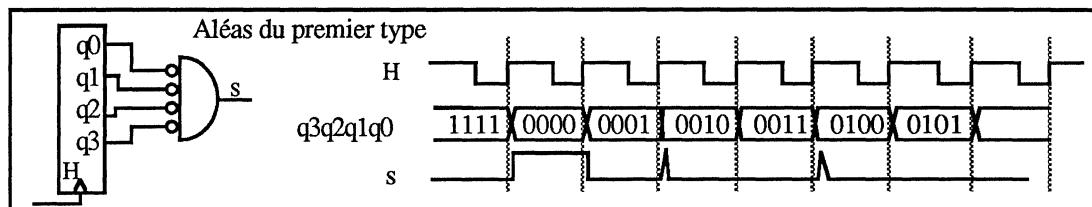
**Monotonie** : Si une entrée varie de façon monotone (sa tension qui ne fait que croître ou décroître quand elle passe dans le domaine sans signification logique) et que les autres entrées ont des valeurs logiques fixes depuis au moins une durée  $t_p$  et y sont maintenues, la sortie varie de façon monotone, en accord avec la fonction du circuit. Cette propriété garantit que le circuit ne rajoute pas des transitions parasites, sous réserve de respecter certains délais entre les transitions des entrées.



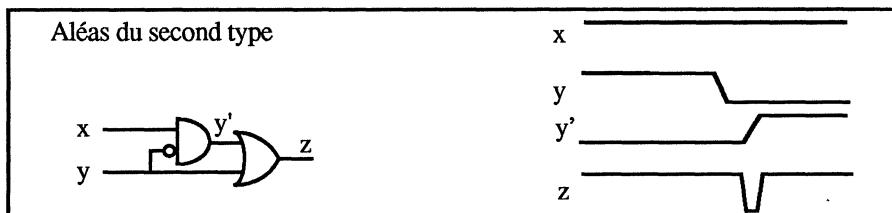
**Aléas de commutation :** On appelle “aléa de commutation” toute valeur transitoire parasite des sorties d’un circuit combinatoire qui apparaissent lorsque les entrées changent. Les aléas ne sont gênant que pour les signaux susceptibles d’avoir un effet, par exemple une horloge ; les aléas sont sans importance pour les autres variables.

On distingue deux types d’aléas :

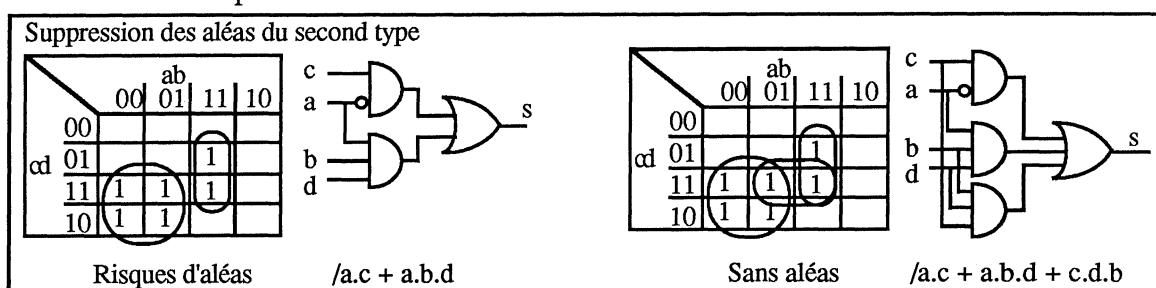
Les aléas du **premier type** se produisent suite aux changements simultanés de plusieurs entrées. De tels aléas sont inévitables, car les entrées changent en réalité dans une fourchette de temps qui n’est pas nulle et les délais d’influences des entrées sur les sorties sont différents. La figure suivante illustre de tels aléas : il s’agit d’un compteur suivit d’un détecteur de la valeur 0 ; logiquement, la sortie est à 1 pendant 1 cycle et à 0 pendant les 15 cycles suivants ; cependant on risque d’observer des impulsions parasites en début de certains cycles, par exemple quand le compteur passe de 0011 à 0100, le détecteur peut voir la valeur transitoire 0000 (formée des 0 de tête de l’état précédent, et les 0 de queue du nouvel état).



Les aléas du **second type** se produisent alors qu’une seule entrée change. La figure suivante en est un exemple ; la fonction réalisée est  $y + /y \cdot x$ , équivalente à  $x + y$  ; pour  $x = 1$ , la valeur de la fonction est constante, égale à 1 ; cependant si  $y$  transite de 1 à 0, on peut observer sur la sortie  $z$  un passage transitoire à 0, comme le suggère le diagramme temporel.



Les aléas du second type peuvent être évités lorsqu’on génère le signal de sortie sous forme de somme de monomes réalisés avec des portes élémentaires ; la technique est la suivante : à partir du tableau de Karnaugh de la fonction, si deux 1 sont adjacents il faut que la réalisation utilise un monome qui recouvre ces deux 1.



En fait, les problèmes d’aléas ne sont pas importants lorsqu’on réalise des circuits synchrones dans lesquels les changements d’état interne sont uniquement provoqués par l’horloge du système : seule compte alors la valorisation correcte du futur état au moment de la transition d’horloge, en fin de cycle.

Nous avons présenté ces problèmes pour sensibiliser le lecteur sur le fait que si on est amené à manipuler des signaux, tels que des horloges, il faut le faire avec la plus grande prudence : “calculer” des instants est plus compliqué que calculer des valeurs.

## 7 - Changement d'état asynchrone - Verrous

### 7.1 - Verrous (latches)

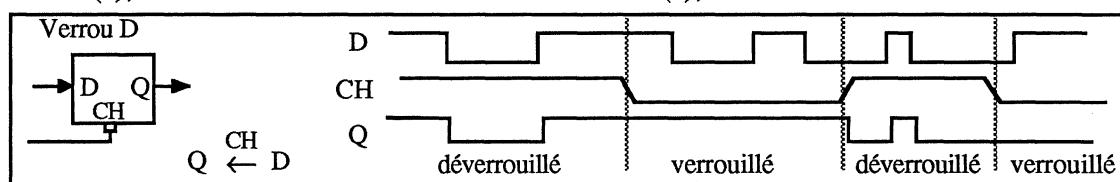
Il y existe deux sortes de dispositifs mémorisateurs : les verrous (Angl. "latch") et les registres (Angl. "register")

#### Verrous :

Un verrou dispose d'une commande CH qui permet de changer l'état mémorisé :

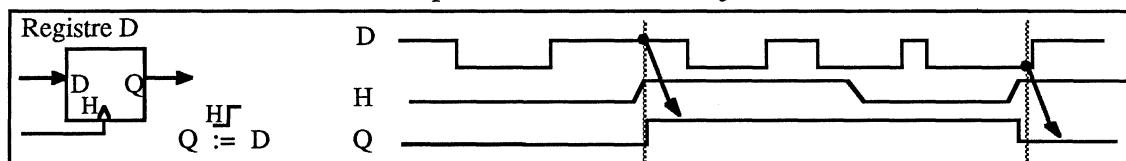
- (1) quand  $CH=1$  : le verrou change continuellement d'état ; l'état Q suit les évolutions de l'entrée D (on dit que le verrou est "transparent") ;
- (2) quand  $CH=0$  : le verrou conserve son état ; l'état Q est fixe et indépendant de D.

Une commande telle que CH s'appelle une commande de déverrouillage de l'état. Dans la situation (1), l'état est déverrouillé et dans la situation (2), l'état est verrouillé.

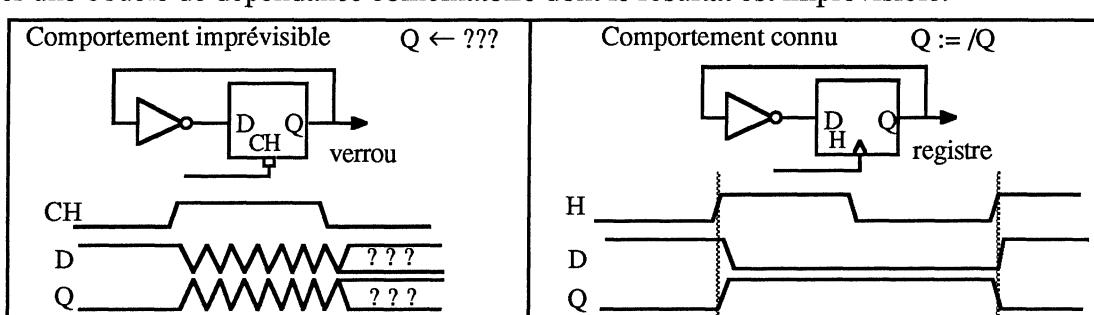


#### Registres :

Les registres ont une entrée d'horloge H, et lorsque H transite de 0 à 1 le registre change d'état et affiche en sortie Q la valeur présente sur l'entrée D juste avant la transition de H.



Il n'est pas possible de construire simplement un système séquentiel en conservant l'état dans un verrou et en appliquant sur les entrées du verrou le résultat du calcul du nouvel état, car sitôt que l'état commence à changer (sitôt que CH passe à 1), les sorties du verrou changent et ces changements sont répercutés sur le calcul du nouvel état, ce qui a pour effet de créer une boucle de dépendance combinatoire dont le résultat est imprévisible.



#### En résumé, les différences sont :

Verrou : la sortie est *continulement* sensible aux variations des entrées tant que  $CH=1$ .

Registre : la sortie change de valeur *une fois* lors de chaque transition de H.

C'est cette différence que nous notons ici par les deux notations :

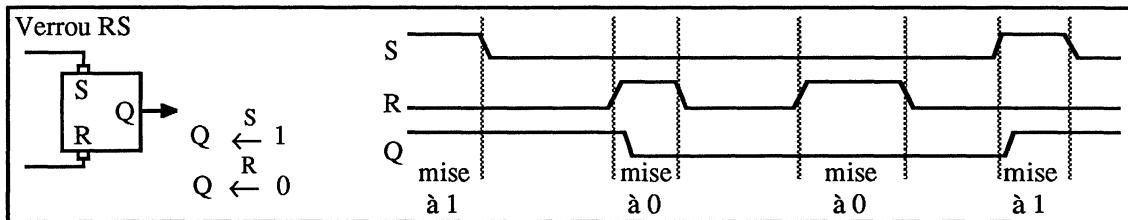
" $:=$ ", pour indiquer l'enregistrement synchrone sur un front d'horloge, à la manière d'un registre.

" $\leftarrow$ ", pour indiquer le changement d'état par déverrouillage.

## 7.2 - Exemples de verrous

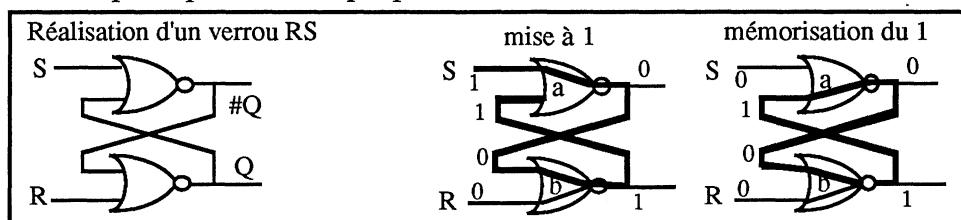
### Verrou RS

Le verrou RS est une mémoire de 1 bit, commandée par deux entrées, R et S ; un niveau 1 sur R met l'état à 0, et un niveau 1 sur S met l'état à 1 ; R et S tous deux à 0 verrouillent l'état à sa valeur présente.



On peut réaliser un verrou RS à l'aide de portes :

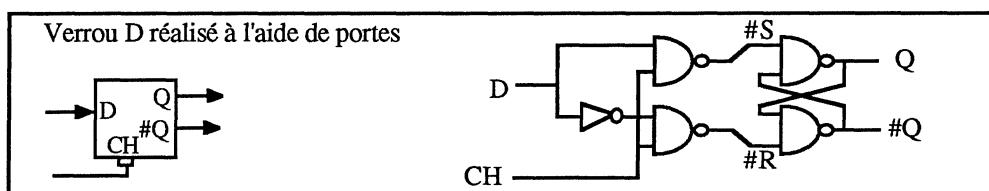
La mise à 1 se passe ainsi : le 1 appliqué sur l'entrée S de la porte (a) détermine un 0 sur la sortie  $\#Q$  (et ce quelque soit la valeur de l'autre entrée) ; les deux entrées de la porte (b) sont alors à 0, et sa sortie Q prend donc la valeur 1 ; à partir de ce moment, la seconde entrée de la porte (a) étant à 1, elle détermine à elle seule le 0 sur  $\#Q$  : le 1 sur S peut disparaître, l'état Q=1 est conservé. Le principe est identique pour la mise à 0.



### Verrou D

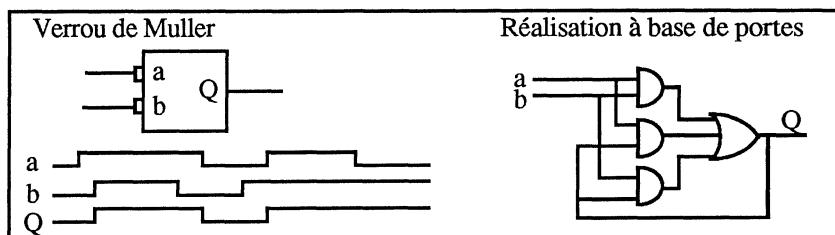
Le schéma suivant montre une réalisation possible du verrou D à l'aide de portes : c'est un verrou RS commandé selon D et CH :

- pour  $CH=0$ ,  $\#R\#S=11$ , le verrou conserve sa valeur
- pour  $CH=1$ ,  $D=0$ ,  $\#R\#S=01$ , le verrou prend la valeur 0
- $D=1$ ,  $\#R\#S=10$ , le verrou prend la valeur 1



### Verrou de Muller

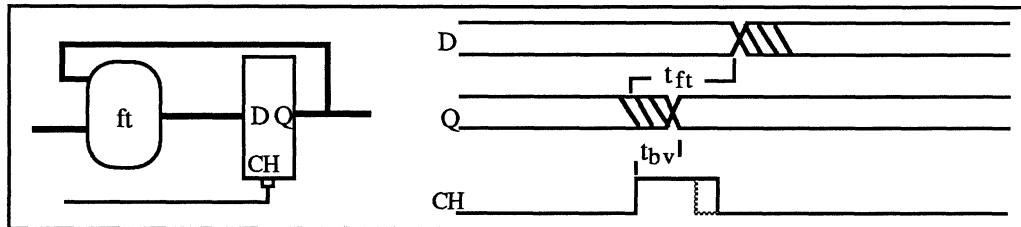
Le verrou de Muller réalise ce qu'on appelle un "rendez-vous" entre plusieurs partenaires ; pour deux partenaires, il y a deux entrées  $a$  et  $b$  : l'état Q passe à 1 lorsque les deux partenaires sont présents ( $a=b=1$ ) et passe à 0 lorsque les deux partenaires ont quitté le rendez-vous ( $a=b=0$ ).



Nous laissons au lecteur le soin de justifier le fonctionnement de la réalisation proposée.

### 7.3 - Horloge à deux phases - Registre maître-esclave

Comme nous l'avons vu précédemment, il est difficile d'utiliser les verrous pour conserver l'état interne d'un système séquentiel ; en fait pour avoir un changement d'état correct, il faudrait ajuster précisément la durée du créneau de la commande de chargement ( $CH=1$ ) :



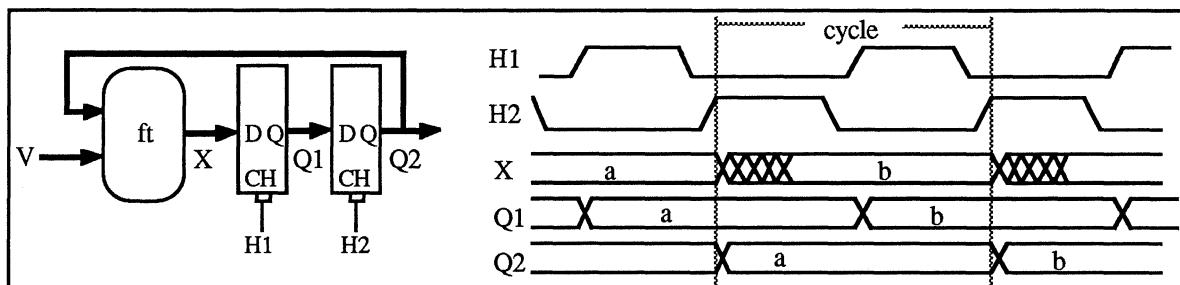
Cette durée  $t_{ch}$  doit être comprise dans un intervalle :

$$tbv < t_{ch} < tft$$

où  $tbv$  est le temps maximum de basculement du verrou et  $tft$  le temps minimum de réaction du circuit de calcul  $ft$  vis à vis de ses entrées, ceci afin de garantir que le verrou change bien d'état et que ses entrées ne changent pas pendant le chargement. Cette technique est très difficilement réalisable, d'autant qu'il est souvent impossible de connaître avec quelque assurance le temps *minimum* de réaction d'un circuit de calcul ( $tft$ ) ; elle n'est pratiquement employée que pour certaines parties des "super calculateurs", pour des raisons de performance ; le réglage de la largeur des créneaux est délicat, à cause des dispersions des caractéristiques des composants, et elle amène souvent à rallonger artificiellement  $tft$  par des circuits retardateurs.

#### Horloge à deux phases

La technique la plus utilisée, car élégante et pratiquement sans contrainte sur la durée des créneaux, consiste à enregistrer l'état dans deux verrous commandés par une horloge à deux phases. L'horloge à deux phases est formée de deux signaux  $H1$  et  $H2$  sans recouvrement (non à 1 simultanément) :



En début de cycle,  $H1=0$  et  $H2=1$ , le verrou  $Q1$  est verrouillé (il contient l'état de ce cycle) et le verrou  $Q2$  est transparent (il affiche l'état en sortie) ; en fin du cycle,  $H1=1$  et  $H2=0$ ,  $Q1$  prend la valeur de  $X$  (le futur état), mais  $Q2$  (l'état) ne change pas car il est alors verrouillé ; le futur état n'apparaît en sortie qu'au début du cycle suivant ; ainsi l'état du système est parfaitement stable pendant tout le cycle et ne change qu'aux frontières de cycles, lors de la transition montante de  $H2$ .

L'ensemble des deux verrous  $Q1, Q2$  forment un *registre* : on peut appliquer sur ses entrées une fonction quelconque de son état et lorsque  $H2$  transite, il réalise correctement :

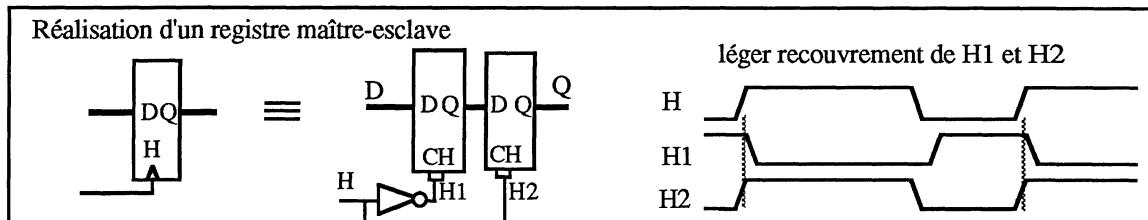
$$Q2 := ft(V, Q2)$$

Cette réalisation d'un registre à l'aide de deux verrous s'appelle un registre maître-esclave (Angl. "*Master-Slave*") ; le verrou  $Q1$  est le verrou maître, et  $Q2$  le verrou esclave.

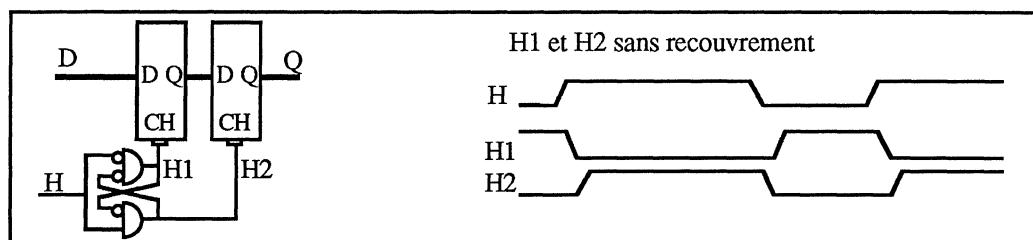
Les auteurs Mead et Conway donnent une image amusante de cette technique : on peut assimiler un verrou à une vanne et un registre à une écluse ; il est bien connu qu'il faut deux vannes pour faire une écluse.

## Registre maître-esclave à une seule entrée d'horloge

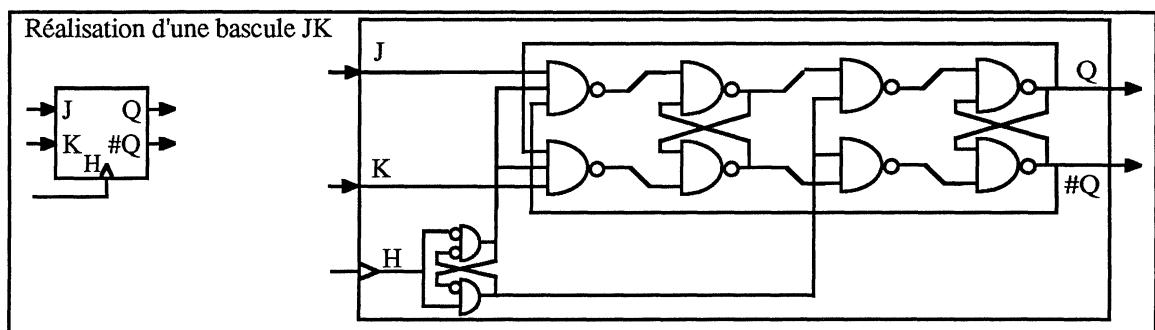
On peut aller un peu plus loin et fabriquer internement H1 et H2 à partir d'un signal d'horloge unique H. Pour cela on peut prendre  $H1 = /H$  et  $H2 = H$ ; il y a cependant risque d'un léger recouvrement si H1 est réalisé par un inverseur ; cela fonctionne tout de même correctement avec la plupart des circuits usuels, car les verrous ont souvent un temps de maintien légèrement négatif : quand H2 ouvre le verrou Q2, H1 n'a pas encore fermé le verrou Q1, mais Q1 reste stable même si D change quelques nanosecondes avant que H1 ne le verrouille.



Le schéma suivant montre une technique "purement logique" pour obtenir les deux signaux H1 et H2 sans recouvrement :



A titre d'exemple, nous montrons ici une réalisation d'une bascule JK; nous laissons au lecteur le soin d'y reconnaître un montage maître-esclave de deux verrous RS :



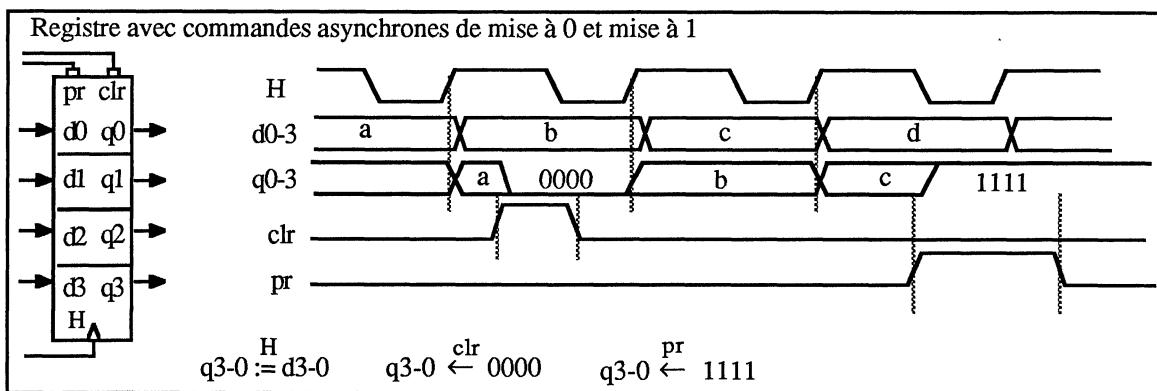
## 7.4 - Circuits à commandes mixtes : horloge et commandes de déverrouillage

La plupart des circuits que l'on trouve dans le commerce sont mixtes ; ils ont :

- Une entrée d'horloge qui provoque le changement d'état synchrone (:=),
- Des entrées dites "asynchrones" qui provoquent un déverrouillage de l'état (←),

L'exemple suivant est un registre à chargement systématique, doté de commandes asynchrones *clr* (*clear*) de mise à 0, et *pr* (*preset*) de mise à 1 :

- *clr*=1 déverrouille l'état à 0000
- *pr*=1 déverrouille l'état à 1111.



Si l'horloge transite alors qu'une commande asynchrone est active, la transition d'horloge n'a aucun effet : on a coutume de dire que les commandes asynchrones sont "prioritaires".

### Usage des commandes asynchrones

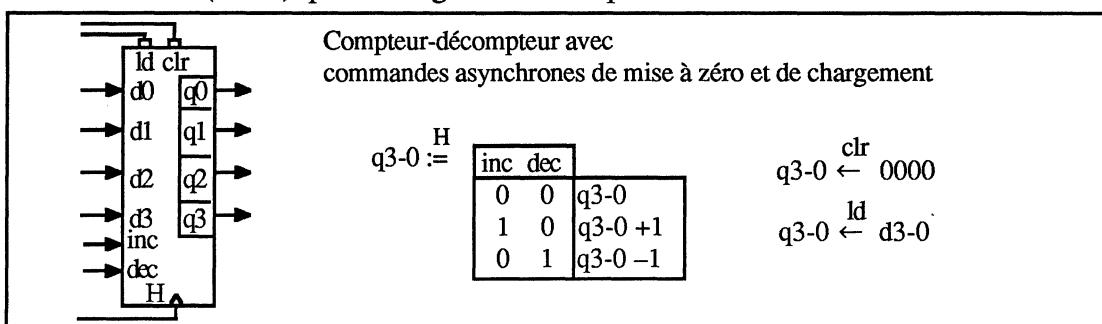
Les commandes asynchrones servent essentiellement à initialiser l'état des divers composants d'un système séquentiel synchrone, et sont peu utilisées en cours de fonctionnement (parce que c'est délicat). Si on se limite à cet usage, les commandes asynchrones ne posent aucun problème particulier et simplifient grandement la circuiterie d'initialisation.

Par contre, si on désire les utiliser en cours de fonctionnement du circuit, il faut en assumer les aspects difficiles :

- Une commande asynchrone doit être considérée comme un **signal** : il faut donc les générer sans aléas de commutation ; un parasite transitoire à 1 sur une commande *clr* d'un registre provoque un désastre irréversible : le registre prend la valeur 0 et son état est perdu.
- Une commande asynchrone prend effet aussitôt : si par une commande asynchrone C on déverrouille une variable d'état X à la valeur V, c'est dans le cycle présent que X prend la valeur V, et non pour le cycle suivant.
- La commande C et la valeur V ne doivent pas dépendre de l'état de X, sinon le résultat du déverrouillage est imprévisible.

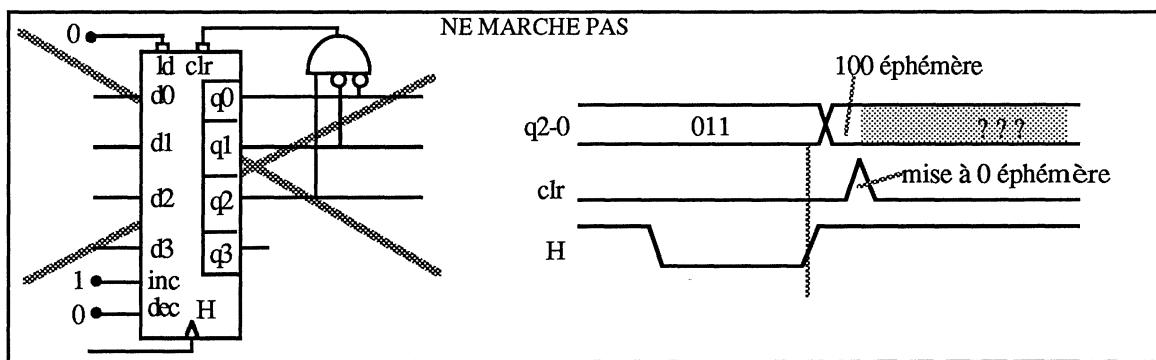
### exemple :

Le circuit utilisé est un compteur-décompteur, doté de deux commandes asynchrones, *clr* qui met l'état à 0 et *ld* ("load") qui le charge à la valeur présentée sur les entrées *d3-0*.

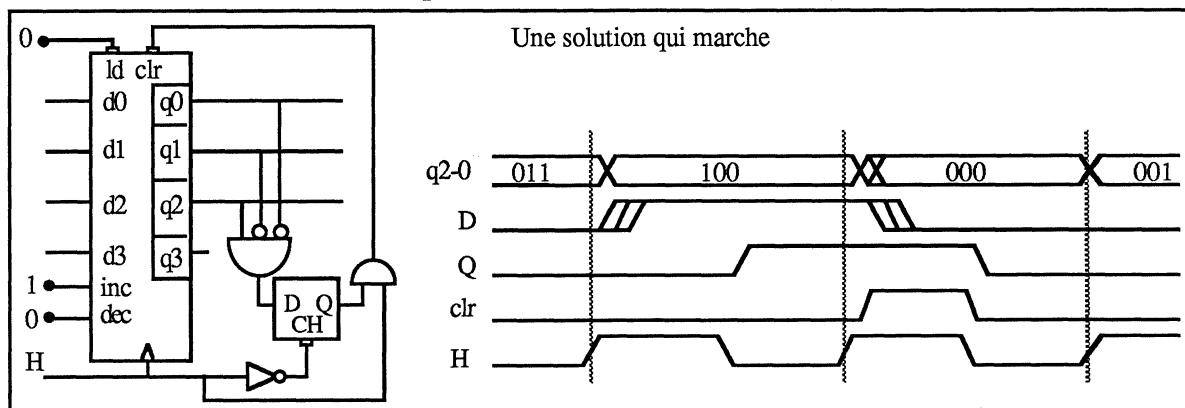


On désire réaliser un compteur modulo 5, c'est-à-dire qui passe à 0 après 4.

La première idée qui vient à l'esprit est d'utiliser la commande de mise à zéro en générant *clr*=1 lorsque l'on est dans l'état 4 ; ceci ne marche pas, comme le montre le graphique temporel : sitôt que le compteur passe dans l'état 4, *clr* passe à 1 provoquant le déverrouillage à 0000 ; déjà ce n'est pas bon, car ce n'est pas à ce moment que l'on voulait passer à 0, mais au front suivant de H (l'état 4 n'a qu'une existence éphémère) ; on n'est même pas certain que le compteur passe à 0, il peut bien prendre n'importe quelle valeur, car sitôt qu'il commence à changer d'état il n'est pas encore à 0 mais il n'est plus à 4 et *clr* devient inactive : la commande *clr* peut avoir une durée insuffisante pour garantir le passage à 0 ; le compteur change d'état, mais ni au bon moment, ni pour la bonne valeur.



Une technique pour utiliser correctement les commandes de déverrouillage consiste à utiliser des verrous auxiliaires pour reconstituer le fonctionnement d'un registre synchrone selon le principe maître-esclave ; pour le problème ci-dessus, il faut déverrouiller l'état à 0 au début du cycle après celui où l'état vaut 4 ; pour cela, on peut utiliser un verrou D qui mémorise la condition ( $q_2=0$ ) à la fin de chaque cycle ( $H=0$ ) ; en début de cycle ( $H=1$ ) on utilise cette condition mémorisée pour commander le déverrouillage à 0 du compteur :



**Exercice 12**

Réaliser le filtrage et la fusion pour des horloges à front descendant.

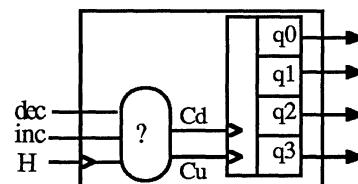
**Exercice 13**

Réaliser un registre à décalage droite conditionnel à l'aide d'un registre à chargement systématique, par la technique du filtrage d'horloge.

**Exercice 14**

On dispose d'un compteur-décompteur muni de deux entrées d'horloge, l'une (Cu) provoquant l'incrémentation et l'autre (Cd) la décrémentation.

- Réaliser, à l'aide de ce circuit, un compteur-décompteur doté d'une seule entrée d'horloge et de deux commandes *inc* et *dec*.

**Exercice 15**

Réaliser un compteur à l'aide de bascules D avec sorties complémentaires, en utilisant le filtrage d'horloge. On utilisera le principe : "complémenter jusqu'au premier 0".

**Exercice 16**

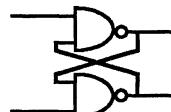
Justifier la méthode de suppression des aléas du second type, en utilisant les propriétés dynamiques des portes élémentaires. On décomposera ainsi la réalisation de la sortie *s* :

$$s = A + B.y + C./y$$

*A*, *B* et *C* étant des sommes de monomes indépendants de *y*, et on analysera les transitions de *y*.

**Exercice 17**

- On peut réaliser un verrou RS à l'aide de portes NON-ET :

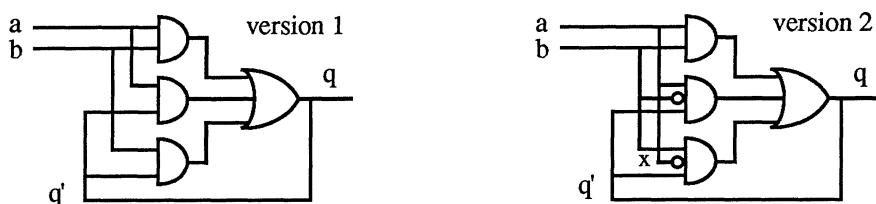


Déterminer, pour ce schéma, les entrées R, S et les sorties Q et #Q ; indiquer les valeurs d'entrées qui provoquent la mise à 0, la mise à 1 et celles qui maintiennent la mémorisation.

- Réaliser un verrou RS uniquement à l'aide de portes ET et OU (sans inverseurs).

**Exercice 18**

On donne ci-dessous deux réalisations d'un verrou de Muller :



La version 1 est correcte : le verrou se comporte bien comme spécifié. Par contre la version 2 est incorrecte ; cela peut sembler curieux, car si on ouvre la boucle qui relie q à q', la fonction de a,b et q' réalisée en q est la même dans les deux cas.

- Expliquer pourquoi la version 2 ne marche pas, et essayer de justifier que la version 1 marche.

**Exercice 19**

En s'inspirant de la solution présentée au §7.4, réaliser un compteur-décompteur modulo 10 à l'aide d'un compteur décompteur doté de commandes asynchrone *clr* et *ld*.

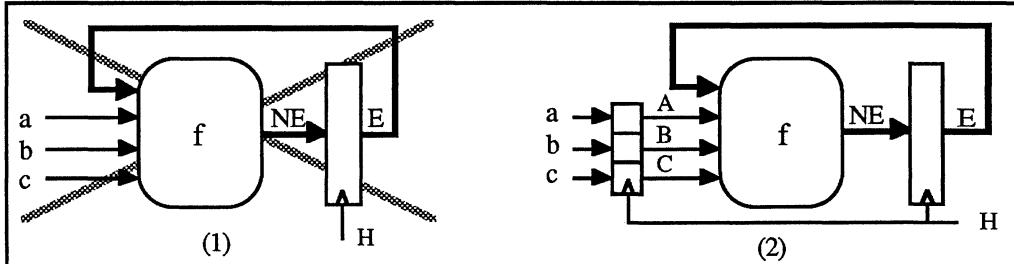
En comptage, ce compteur doit passer de 9 à 0, et en décomptage il doit passer de 0 à 9.

## 8 - Prise en compte d'entrées asynchrones

Un machine synchrone peut avoir des entrées asynchrones avec son horloge H, c'est-à-dire des signaux qui peuvent changer à des moments sans aucune relation avec H : c'est nécessairement le cas pour les communications avec l'extérieur réel, et parfois à l'intérieur d'un système, lorsqu'il est composé de parties fonctionnant sur des horloges différentes.

### 8.1 Nécessité de synchroniser les entrées asynchrones

Les entrées asynchrones doivent être synchronisés, par enregistrement dans des bascules dès l'entrée, avant de pouvoir participer à la logique interne de la machine :



Le comportement de la machine (1) est totalement imprévisible : puisque  $a,b,c$  changent à des moments totalement indépendants de H, le nouvel état, c'est-à-dire la valeur de NE au moment de H a peu de chance d'être égal à  $f(E,abc)$  ; il a même peu de chance d'être égal à  $f(E,xyz)$   $xyz$  étant des valeurs quelconques ; en fait il y a peu de chances pour que cela ait le moindre rapport avec  $f$  !

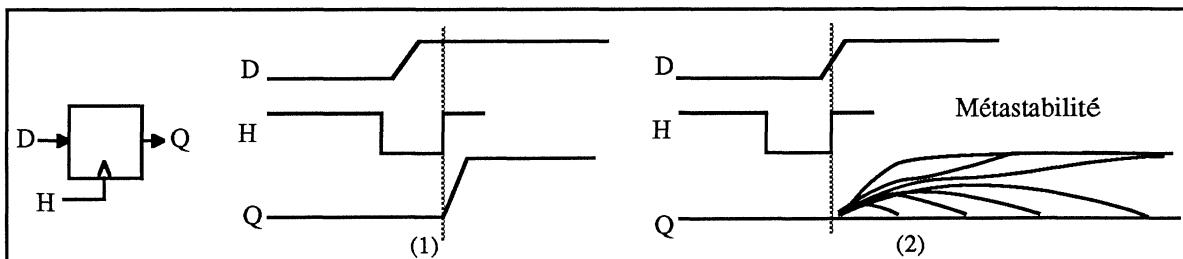
Le comportement de la machine (2) est plus prévisible : si on fait l'hypothèse ( $h$ ), fausse mais acceptable sous certaines conditions, que les sorties de bascules valent 0 ou 1 après le coup d'horloge, le nouvel état vaut  $f(E,ABC)$  puisque ABC ont une valeur maintenue stable depuis le début du cycle. Les valeurs de ABC sont les valeurs de  $abc$  au voisinage du coup d'horloge. Les bascules ABC ont un rôle purement synchronisateur des entrées  $abc$ .

### 8.2 Le problème de la métastabilité

Le principe précédent est basé sur l'hypothèse ( $h$ ) : au bout d'un temps borné (connu) après un coup d'horloge, la sortie d'une bascule (plus généralement d'un système physique doté d'un nombre fini d'états stables) est dans l'état 0 ou dans l'état 1 ; cette hypothèse est fondamentalement fausse (fondamentalement en ce sens qu'elle est contraire à certaines théories physiques, thermodynamique notamment, et non pas simplement due à une difficulté technologique).

Le diagramme suivant illustre le comportement d'une bascule qui échantillonne une entrée asynchrone : si l'entrée change assez tôt avant le front d'horloge (1), respectant le temps de prépositionnement indiqué pour la bascule, la sortie prend avec certitude la valeur de l'entrée ; par contre si l'entrée change dans une fenêtre de temps critique autour du front de H (2), la bascule peut entrer dans un état intermédiaire, qui n'est ni 0 ni 1.

Cet état n'est pas stable, la bascule va nécessairement se décider pour un 0 ou pour un 1, mais on ne peut affirmer avec certitude au bout de combien de temps : un tel état est qualifié de "métastable" ; la moindre perturbation d'origine thermique provoque le basculement vers 0 ou vers 1, mais le temps nécessaire pour sortir de cet état suit une loi *statistique*.



La prise en compte d'une telle variable peut avoir des conséquences catastrophiques : elle peut être comprise comme un 0 par une partie du système, et comme un 1 par une autre partie, ou encore être comprise comme un 1 au début du cycle, puis comme un 0 vers la fin du cycle.

Il n'y a pas de solution logique à ce problème, mais heureusement c'est un faux problème, car il suffit d'attendre suffisamment après le coup d'horloge pour que la probabilité que la bascule n'ait pas décidé entre 0 et 1 soit infinitésimale : quand cette condition est respectée, on considère que l'hypothèse ( $h$ ) est vraie ; les mises en défaut de l'hypothèse sont comptabilisées au titre des pannes matérielles qui sont de toutes façons inévitables dans un système. Pour quantifier les effets néfastes dues à la métastabilité, on utilise la formule suivante, vérifiée expérimentalement et plus ou moins justifiée théoriquement :

$$\text{Taux de défaillance : } TP = K_1 \cdot F_h \cdot F_d \cdot e^{-\frac{t-t_0}{K_2}}$$

Le taux de défaillance  $TP$  est le nombre de fois par seconde où une valeur métastable est utilisée par le système ;  $t$  est le délai après le front d'horloge à partir duquel le système considère que la sortie de la bascule a une valeur (un 0 ou un 1) ;  $F_h$  est la fréquence de l'horloge, et  $F_d$  la fréquence de changement de la donnée ;  $K_1$ ,  $K_2$  et  $t_0$  sont des paramètres qui caractérisent la bascule utilisée ; l'ordre de grandeur de ces paramètres pour les bascules usuelles est :

$$K_1 \approx 10^{-7} \text{ s} = 100 \text{ ns} \quad K_2 \approx 10^{-9} \text{ s} = 1 \text{ ns} \quad t_0 \approx 30 \text{ ns}$$

Prenons des exemples numériques ; considérons  $F_h = 10^7 \text{ s}$  (cycle de 100 ns), et  $F_d = 10^7$  (donnée changeant toutes les 100 ns en moyenne).

Si on s'autorise à prendre en compte la sortie de la bascule au bout de  $t = 50 \text{ ns}$  après le coup d'horloge on obtient :

$$TP = 10^{-7} \cdot 10^7 \cdot 10^7 \cdot e^{-20} \approx 1/100$$

c'est-à-dire environ une défaillance toutes les deux minutes, ce qui sera sans doute considéré comme intolérable pour un système informatique sérieux.

Si par contre on attend un peu plus, par exemple  $t = 70 \text{ ns}$ , on obtient :

$$TP = 10^{-7} \cdot 10^7 \cdot 10^7 \cdot e^{-40} \approx 10^{-11}$$

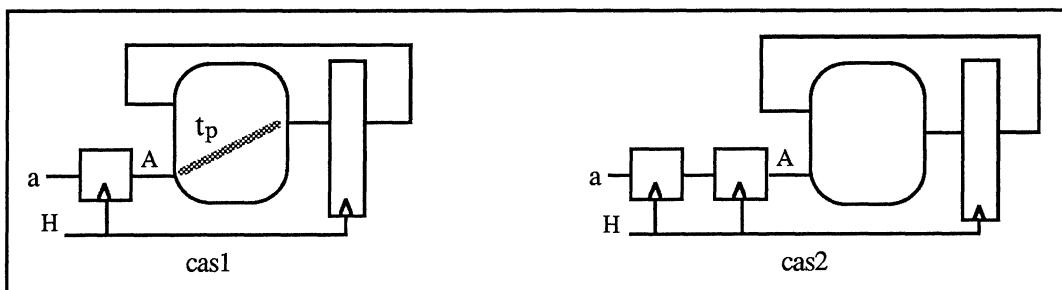
soit moins d'une défaillance tous les 3 ans, ce qui est déjà plus acceptable.

Pratiquement, on tient compte de ce problème de la façon suivante :

On se fixe un taux de défaillance à atteindre (par exemple une par jour, une par an, une par siècle...) ; cela nous donne le délai  $t$  à respecter. Puis on prend ce délai pour temps de basculement de la bascule, c'est-à-dire que l'on considère que sa sortie n'est correctement valorisée qu'au terme de ce délai après le début du cycle.

Ici deux cas se présentent :

- si ce délai  $t$  augmenté du temps de calcul  $t_p$  des circuits combinatoires dans lesquels intervient la donnée est inférieur au cycle de la machine, cela marche bien (cas 1) ;
- sinon on rajoute une bascule en entrée dont le rôle est simplement destiné à rajouter un cycle complet pour permettre la résolution de la métastabilité (cas 2).





# VI - UNITES DE CONTROLE

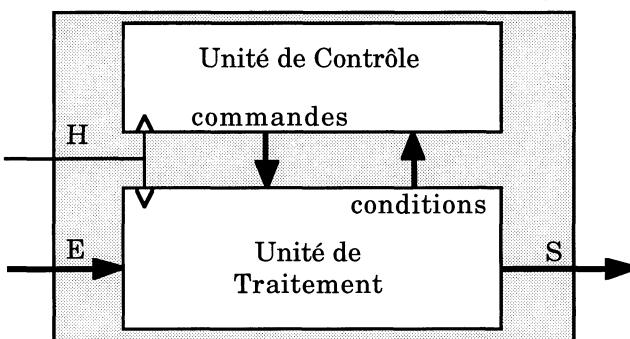
---

## 1 - Unité de contrôle et unité de traitement

### 1.1 - Principes généraux

Une bonne méthode pour réaliser un système complexe consiste à le décomposer en deux parties :

- Une unité de traitement qui réalise les traitements élémentaires sur les données au moyen d'opérateurs, de registres et de chemins de données.
- Une unité de contrôle qui réalise le séquencement des opérations de façon à réaliser la fonctionnalité désirée.



Cette décomposition permet de résoudre un problème complexe à l'aide de deux modules simples : une unité de contrôle agissant sur une unité de traitement, à la façon d'un programme qui agit sur des données.

#### Commandes et conditions

L'unités de contrôle l'unité de traitement sont interconnectées par les commandes et les conditions.

- Les commandes, générées par l'unité de contrôle, définissent l'opération réalisée à chaque cycle.
- Les conditions, issues de l'unité de traitement, influencent le séquencement des commandes.

#### Unités de traitement

Les unités de traitement comportent trois types d'éléments.

- Des mémoires (registres), pour la conservation des données.

- Des opérateurs, qui doivent permettre d'effectuer les calculs nécessaires.

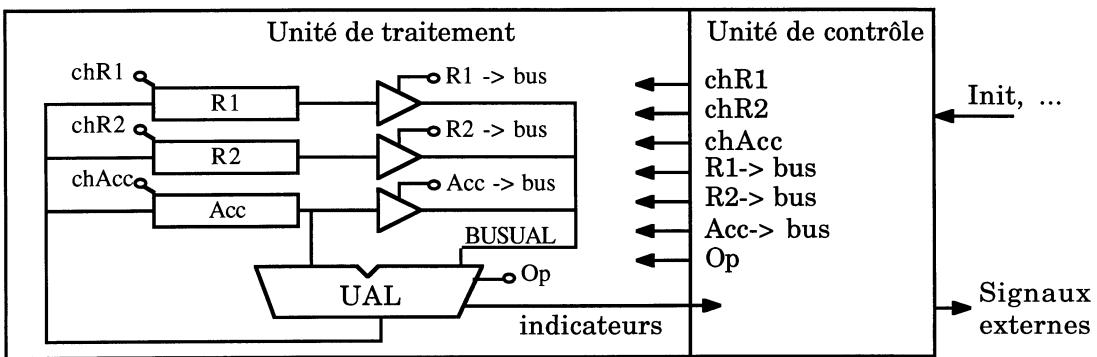
On les regroupe souvent dans un ou plusieurs blocs fonctionnels appelés “unité arithmétique et logique” (UAL).

- Des liaisons entre mémoires et opérateurs, ou chemins de données. Ces chemins de données sont des connexions simples (une seule source, un ou plusieurs destinataires) ou des *bus* (plusieurs sources, généralement en technologie trois-états, dont une seule peut être validée à un instant donné).

#### Unités de contrôle

Une unité de contrôle est un système séquentiel. Ses entrées sont les conditions provenant de l'unité de traitement, ses sorties les commandes envoyées vers l'unité de traitement : chargement de registres, commande d'opérateurs et contrôle de chemins de données. Elle peut également échanger des signaux avec l'extérieur (initialisation, commandes et conditions externes).

## Exemple



R1, R2 et Acc sont des registres : les commandes chR1, chR2 et chAcc sont activées en fonction de l'algorithme à exécuter.

Le bus BUSUAL présente l'opérande droit à l'UAL : les commandes exclusives R1 -> bus, R2 -> bus et Acc -> bus déterminent sa valeur.

L'opération effectuée par l'UAL est spécifiée par la commande Op (addition, soustraction, transfert ...). L'opérande gauche est le contenu du registre Acc. Le résultat peut être chargé dans R1, R2 ou Acc.

Les indicateurs de l'UAL peuvent servir de conditions : ils interviennent dans les transitions de l'unité de contrôle.

Par exemple, pour réaliser l'opération “Acc:= R1 + R2” avec cette unité de traitement, l'unité de contrôle pourra générer une séquence de commandes sur deux cycles :

- cycle 1 : R1 -> bus, Op= transfert, chAcc
- cycle 2 : R2 -> bus, Op= add, chAcc

## 1.2 - Méthode de décomposition

### Algorithme de résolution

Il faut commencer par rédiger un algorithme décrivant le fonctionnement de la machine. Il comprend des actions séquentielles (à effectuer les unes après les autres), mais aussi, et on cherchera à l'exhiber autant que possible, des actions parallèles (que l'on peut effectuer simultanément).

### Choix d'une unité de traitement

Il faut déterminer les composantes (mémoires, opérateurs et chemins de données) utiles pour réaliser l'algorithme : toutes les variables de l'algorithme doivent être conservées dans des mémoires, et toutes les opérations doivent correspondre à un opérateur. Les chemins de données doivent permettre toutes les affectations et tous les calculs de l'algorithme.

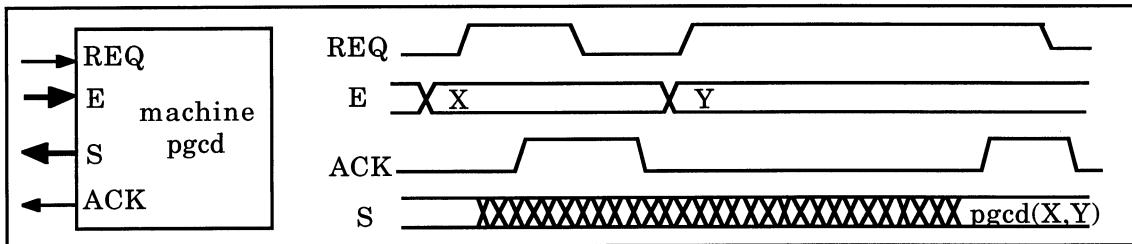
On peut s'orienter vers une structure très spécialisée correspondant exactement aux besoins de l'algorithme, ou vers une structure plus générale pouvant s'adapter à d'autres algorithmes. Une structure spécialisée est plus performante, mais se prête mal à des modifications de l'algorithme.

### Diagramme des états de l'unité de contrôle

L'étape suivante consiste à traduire l'algorithme en un diagramme d'états. Chaque état correspond à une instruction, ensemble de commandes à générer dans un même cycle pour effectuer une action de l'algorithme, compte tenu des possibilités de l'unité de traitement. L'unité de contrôle est généralement une machine de Moore.

### 1.3 - Exemple de réalisation

Nous allons illustrer la démarche en réalisant une machine à calculer des pgcd. Les opérandes X et Y sont fournis successivement sur une même entrée E, suivant un protocole de demande-réponse : l'entrée REQ ("Request") permet de valider les opérandes et de lancer le calcul, la sortie ACK ("Acknowledge") signale successivement la prise en compte de X puis la fin du calcul.



### Algorithme de fonctionnement de la machine

On calcule le pgcd par la méthode classique des différences successives.

```

faire % boucle éternelle de fonctionnement de la machine %
  jusque MontéeReq faire % attend opérande X %
    ACK:=0, RX:=E, quand REQ=1 sortir % voir remarque 2 %
    fait;
  jusque RetombéeReq faire % acquitte X et attend retombée de REQ %
    ACK:=1, quand REQ=0 sortir
    fait;
  jusque MontéeReq faire % attend opérande Y %
    ACK:=0, RY:=E, quand REQ=1 sortir % voir remarque 2 %
    fait;
  jusque FinCalculPgcd faire % calcul du pgcd %
    quand RX=RY sortir,
      si RX>RY alors RX:=RX-RY
      sinon RY:=RY-RX
    finsi
    fait;
  jusque RetombéeReq faire % affiche pgcd %
    S=RX, ACK:=1, quand REQ=0 sortir
    fait
fait
```

#### Remarques

1 - Nous utilisons ici la virgule pour indiquer l'exécution simultanée de plusieurs actions, et le point-virgule pour indiquer l'exécution séquentielle.

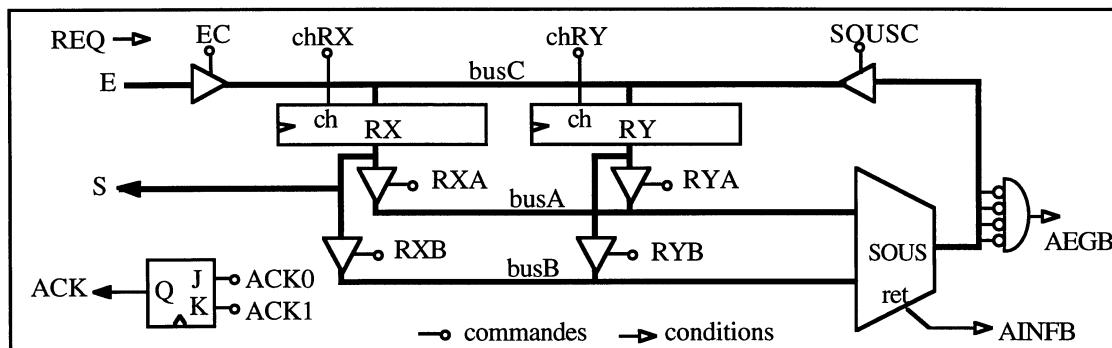
2 - L'affectation `ACK:=0` est faite à chaque itération, et non une seule fois avant la boucle, de manière à être fait en même temps que le test `quand REQ=1 sortir`, ce qui diminue le nombre d'états du contrôle ; pour la même raison, E est enregistré à chaque itération plutôt qu'une seule fois après la boucle.

#### Choix d'une unité de traitement

L'algorithme utilise les opérations de soustraction et de comparaison arithmétique ( $<$ ,  $=$ ,  $>$ ). Ces quatre opérations peuvent être réalisées avec un seul soustracteur, en lui adjoignant un test de nullité du résultat pour la comparaison " $=$ ", et en utilisant la retenue pour la comparaison " $<$ " ; la

comparaison “>” est simplement obtenue par “(non<) et (non=)”.

Nous choisissons ici une structure moyennement spécialisée, organisée autour de trois bus : les bus A et B permettent de sélectionner les opérandes du soustracteur (réaliser X-Y ou Y-X), et le bus C sert à sélectionner la valeur en entrée des registres RX et RY.



#### Commandes :

chRX : chargement de RX  
 RXA : envoi de RX sur le bus A  
 RXB : envoi de RX sur le bus B  
 EC : envoi de E sur le bus C  
 ACK0 : mise à 0 de ACK

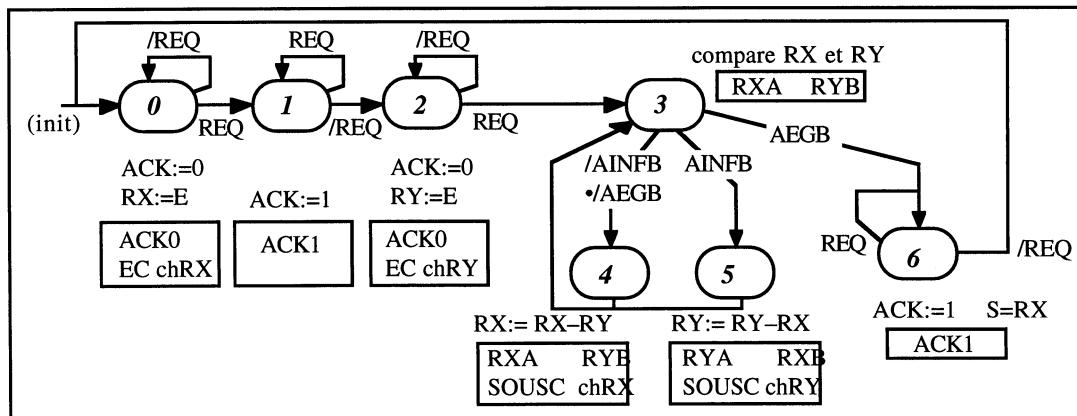
chRY : chargement de RY  
 RYA : envoi de RY sur le bus A  
 RYB : envoi de RY sur le bus B  
 SOUSC : envoi de SOUS sur le bus C  
 ACK1 : mise à 1 de ACK

#### Conditions :

AEGB : la donnée du bus A est égale à celle du bus B.  
 AINF : la donnée du bus A est inférieure à celle du bus B.  
 REQ : valeur de l'entrée REQ.

## Diagramme des états de l'unité de contrôle

Chaque instruction du programme est réalisée par une combinaison des commandes offertes par l'unité de traitement ; on fait figurer pour chaque état les commandes qu'il faut générer dans cet état.



Il reste enfin à réaliser effectivement cette unité de contrôle, ce qui est l'objet du paragraphe suivant.

## 2 - Réalisation des unités de contrôle

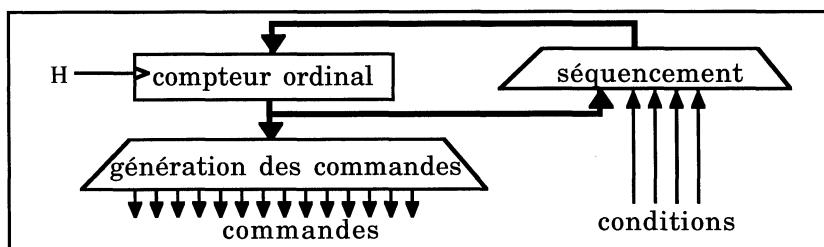
### 2.1 - Structure générale d'une unité de contrôle

Une unité de contrôle est un système séquentiel standard. On utilise cependant une terminologie particulière :

L'état est appelé "compteur ordinal" : comme celui d'un ordinateur, il indique où en est l'exécution au sein du programme.

La fonction de sortie est appelée "génération des commandes" : elle indique les commandes à exécuter en fonction de la valeur du compteur ordinal.

La fonction de transition est le "séquencement" : elle détermine la nouvelle valeur du compteur ordinal en fonction de sa valeur présente et des conditions.



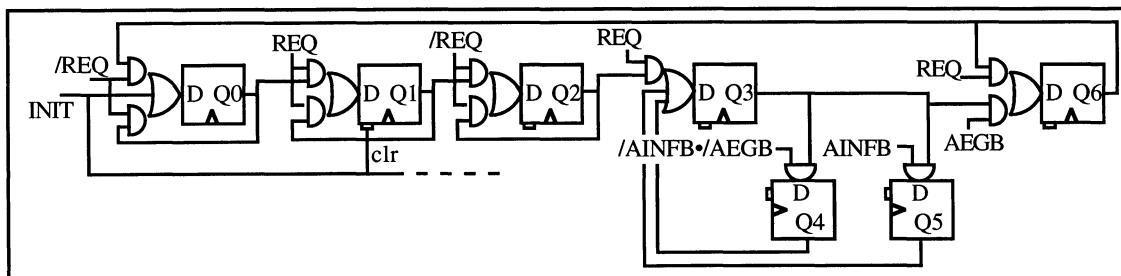
Il existe de nombreuses façons de réaliser une unité de contrôle. Nous considérons ici trois techniques classiques :

- unité de contrôle à bascules,
- unité de contrôle à compteur chargeable,
- unité de contrôle microprogrammée.

### 2.2 - Unité de contrôle à bascules

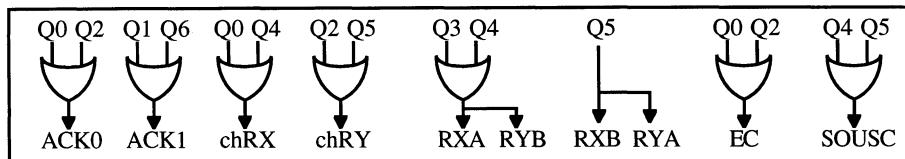
On peut réaliser l'état de l'automate de contrôle à l'aide de bascules ; on appelle cela une "unité de contrôle câblée". Le codage de l'état est à priori sans importance. On choisit souvent un "automate à jeton" (une bascule par état). La structure du circuit est alors similaire au diagramme des états.

Dans l'exemple de la machine pgcd, le circuit de séquencement est :



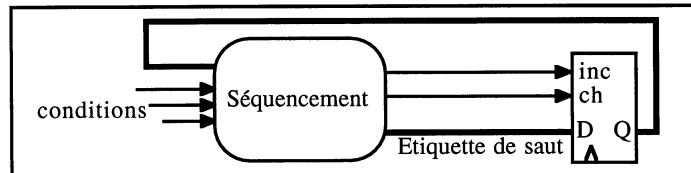
La commande INIT assure le démarrage initial dans l'état 0.

Génération des commandes : chaque commande est l'union des états où cette commande vaut "1" :



## 2.3 - Unité de contrôle à compteur chargeable

Cette méthode tire profit du fait que le diagramme des états est souvent très “linéaire”. L’idée est d’utiliser l’incrémentation d’un compteur pour faire la plus grosse partie de la fonction de transition, le reste (les “sauts”) étant réalisé par chargement du compteur. On profite également de la possibilité de ne pas changer d’état pour réaliser les boucles sur un seul état (en général, ce sont des attentes).



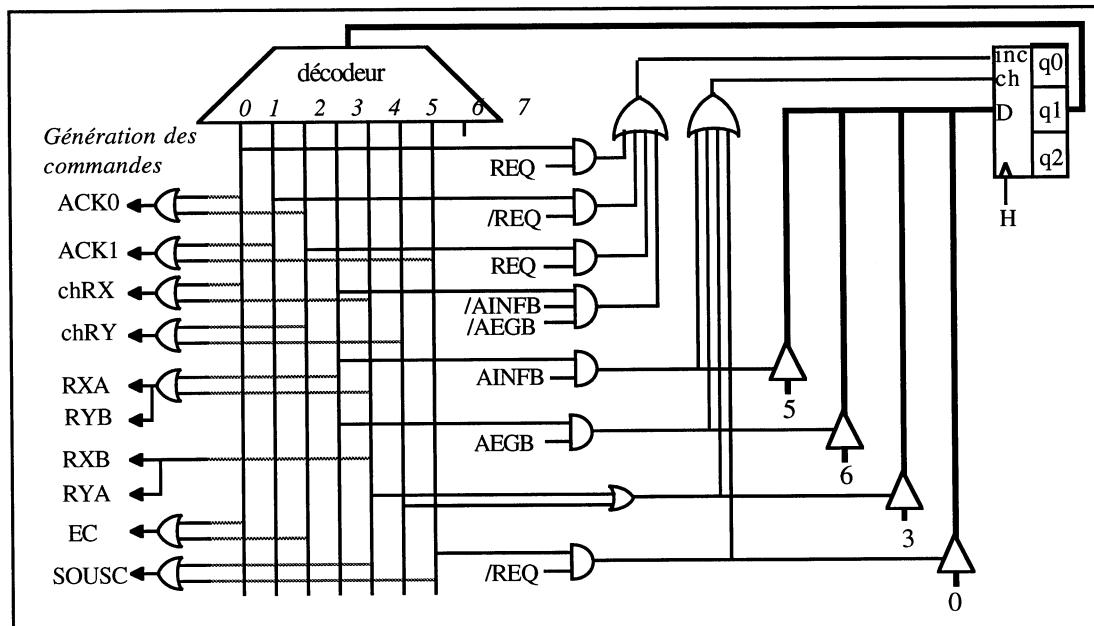
A chaque cycle, selon l’état et les conditions, le compteur est incrémenté, chargé, ou laissé inchangé. On établit une table des cas de chargement et d’incrémantation qui détermine les commandes inc et ch du compteur, ainsi que la valeur à charger pour les cas de chargement.

Dans l’exemple de la machine pgcd, cela donne :

état	condition	inc	ch	valeur chargée
0	REQ	1	0	
1	/REQ	1	0	
2	REQ	1	0	
3	/AINFB./AEGB	1	0	
3	AINFB	0	1	5
3	AEGB	0	1	6
4		0	1	3
5		0	1	3
6	/REQ	0	1	0

Les cas où inc et ch valent tous les deux “0” ne figurent pas dans la table ; ils correspondent aux boucles sur une seule instruction. La valeur chargée n’est spécifiée que si on charge : elle peut être quelconque dans les autres cas.

Avec cette table, on élabore les commandes inc et ch du compteur, et la valeur à charger.

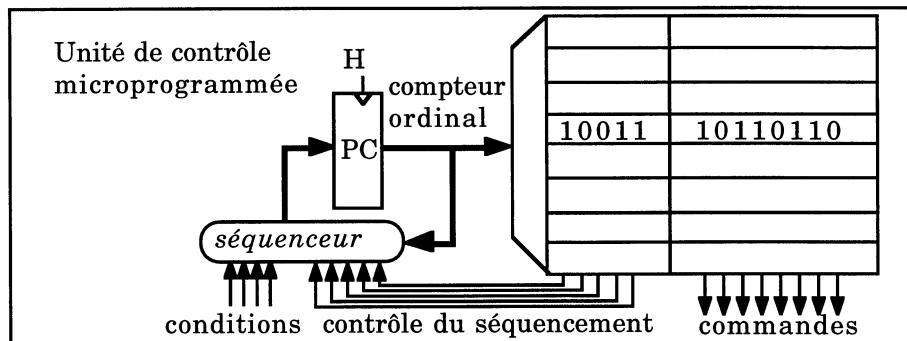


Dans le schéma proposé, la valeur chargée résulte d'une sélection parmi plusieurs valeurs possibles (les "étiquettes" de saut) selon l'état et les conditions. On aurait pu calculer une expression logique pour chaque bit, en espérant des simplifications.

La génération des commandes est faite à partir du décodage du compteur.

## 2.4 - Unité de contrôle microprogrammée

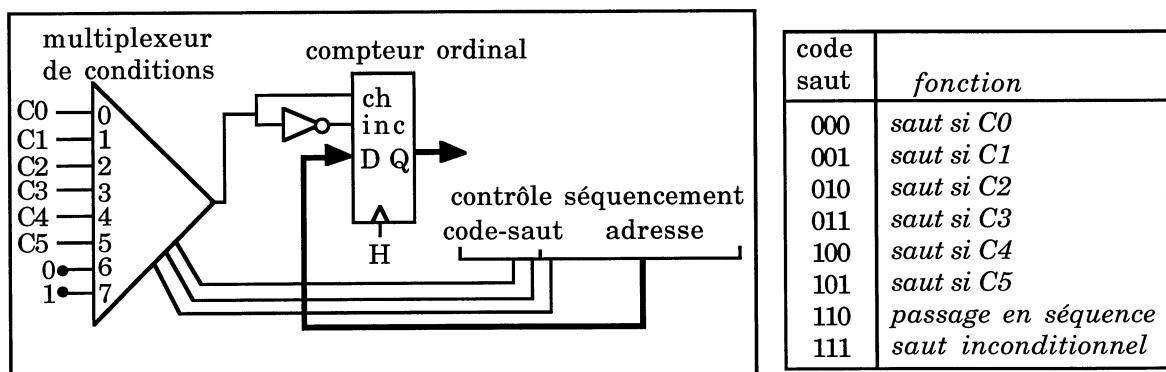
Dans une unité de contrôle microprogrammée, on assure les fonctions de séquencement et de génération des commandes à l'aide d'une mémoire adressable (généralement morte), adressée par le compteur ordinal. L'algorithme est alors plus facile à modifier : il suffit de changer le contenu de la mémoire. Le contenu de la mémoire est le "microprogramme", et chacun de ses mots s'appelle une micro-instruction.



### Séquenceur :

Le séquencement est assuré en grande partie par la mémoire adressable, mais il n'est pas envisageable d'y enregistrer n'importe quelle fonction de séquencement car cela conduirait à une mémoire trop large. Un nombre limité de primitives de séquencement sont réalisées par un circuit appelé "séquenceur", lui-même commandé par des codes inscrits dans la mémoire de microprogramme. Les séquenceurs les plus simples offrent uniquement les sauts conditionnels : le compteur ordinal est soit incrémenté, soit chargé avec une adresse donnée dans le champ "contrôle du séquencement".

La figure suivante illustre la réalisation d'un séquenceur simple :

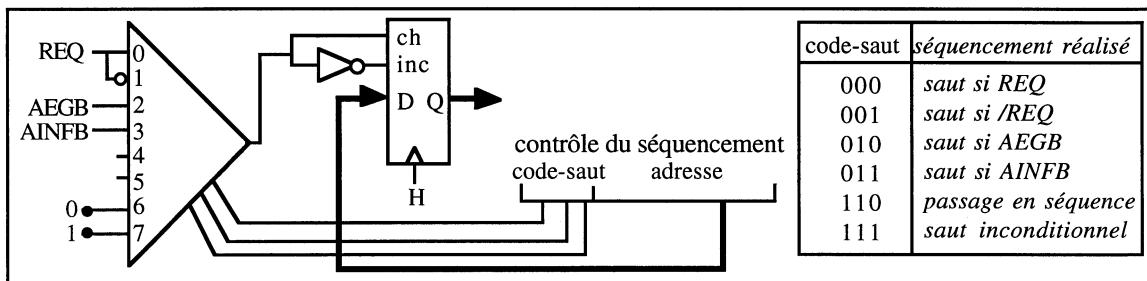


Le champ contrôle de séquencement est constitué de deux parties :

- champ “code-saut” : indique un saut conditionnel selon une condition parmi 6, un saut inconditionnel, ou le passage en séquence
- champ “adresse de saut” : étiquette en cas de saut.

Dans cet exemple, la commande de chargement est délivrée par un multiplexeur qui, selon le code du saut, sélectionne la condition testée. Ainsi, si la condition est vraie, le compteur ordinal est chargé, ce qui réalise un saut à l’étiquette indiquée dans le champ “adresse de saut”. Le saut inconditionnel est réalisé comme un “saut selon une condition toujours vraie” et le passage en séquence comme un “saut selon une condition toujours fausse”.

Pour la machine pgcd, on peut utiliser le séquenceur comme suit :



### Micropogramme :

Le micropogramme correspondant est indiqué ci-dessous, sous deux formes : une forme symbolique, telle qu’elle pourrait être reconnue par un assembleur, et le code binaire, qui est le contenu exact de la mémoire.

adresses	ACK1	ACK0	chRX	chRY	RXA	RYA	RXB	RYB	EC	SOUSC	Code-saut	adres
0	ACK0	EC	chRX		saut si <i>/REQ</i>	0			01	10 00 00 10	001	0000
1	ACK1				saut si <i>REQ</i>	1			10	00 00 00 00	000	0001
2	ACK0	EC	chRY		saut si <i>/REQ</i>	2			01	01 00 00 10	001	0010
3	RXA	RYB			saut si <i>AEGB</i>	7			00	00 10 01 00	010	0111
4	RXA	RYB			saut si <i>AINFB</i>	6			00	00 10 01 00	011	0110
5	RXA	RYB	SOUSC	chRX	saut	3			00	10 10 01 01	111	0011
6	RXB	RYA	SOUSC	chRY	saut	3			00	01 01 10 01	111	0011
7	ACK1				saut si <i>REQ</i>	7			10	00 00 00 00	000	0111
8					saut	0			10	00 00 00 00	111	0000
	<i>programme symbolique</i>											
	<i>programme binaire</i>											

La pauvreté de ces primitives de séquencement (une seule étiquette de saut conditionnel) oblige à introduire des états parasites qui ne figurent pas dans le diagramme initial. Un état a au plus deux successeurs directs : l'état incrémenté et l'étiquette de saut. Cela oblige à réaliser les aiguillages multiples par des cascades de sauts. Ainsi l'état trois du diagramme est réalisé par les micro-instructions 3 et 4, qui testent “RX=RY” et “RX<RY” en deux étapes.

## Aiguillages multiples

Un séquenceur simple, avec une seule étiquette pour chaque saut conditionnel, oblige à introduire des états parasites qui ne figurent pas dans le diagramme initial.

Des séquenceurs plus évolués permettent les aiguillages multiples. On pourrait indiquer plusieurs adresses de saut dans la micro-instruction, chacune associée à une condition, mais cela serait trop coûteux en mémoire. On préfère associer un index (un petit entier) à chaque combinaison intéressante de conditions et l'utiliser pour modifier l'adresse de saut. En général, l'index fournit directement les bits de poids faible de l'adresse, les autres bits provenant du champ adresse de saut.

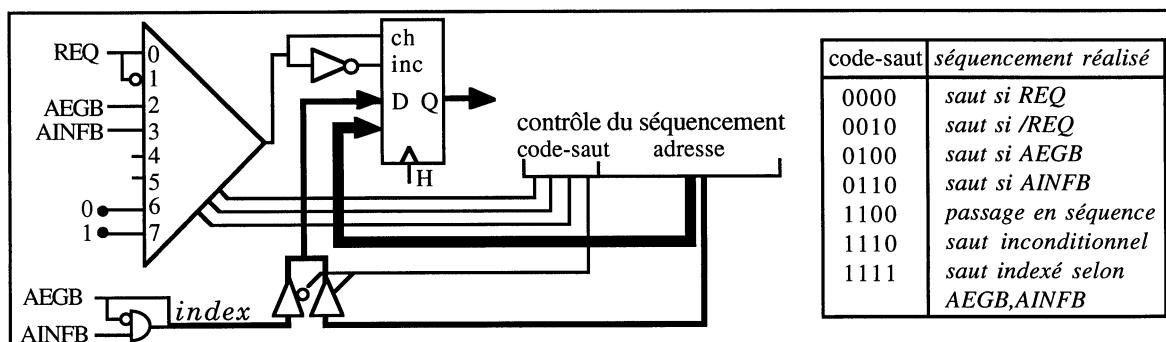
Dans notre exemple, pour réaliser l'aiguillage multiple selon toutes les combinaisons possibles de AEGB et AINFB, on associe un index à chacune. Le code de saut, par exemple "1111", indique le "saut indexé selon AEGB,AINFB".

AEGB	AINFB	index	code-saut	adresse-saut	saut indexé à l'adresse :
0	0	0			
0	1	1	1111	etiq	etiq + index
1	x	2			

Cette forme de saut aura pour effet de sauter à l'étiquette :

étiq      si /AEGB•/AINFB    (A>B)  
 étiq+1    si /AEGB•AINFB    (A<B)  
 étiq+2    si AEGB            (A=B)

Une réalisation possible est illustrée sur le schéma suivant :



Remarque : les destinations du saut étant des adresses consécutives, ceci oblige à placer à ces adresses des sauts inconditionnels vers la suite logique du programme.

# Exercices

## Exercice 1 Multiplieur séquentiel

Pour multiplier  $X$  par  $Y$ , une méthode consiste à cummuler séquentiellement les produits partiels :

$$P = Y \times X = \sum Y_i X x 2^i$$

Selon que  $Y_i$  vaut 0 ou 1, le terme  $Y_i X x 2^i$  est soit 0, soit  $X$  décalé de  $i$  positions vers les poids forts.

Exemple, avec opérandes de 4 bits et résultat sur 8 bits :

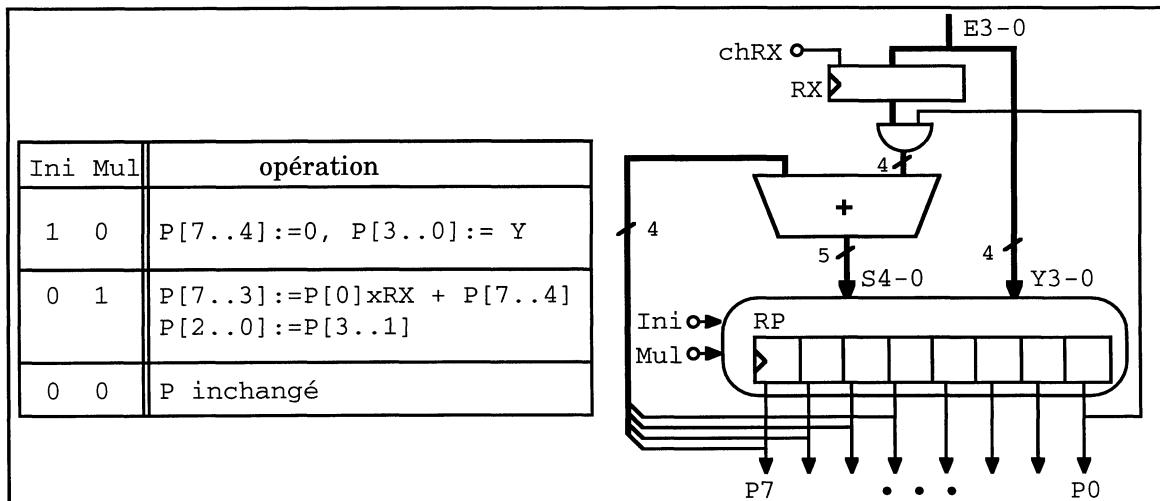
$Y = 0101 \quad (5)$ $X = 1101 \quad (13)$	$\begin{array}{r} 0 \times X \times 2^3 \\ 1 \times X \times 2^2 \\ 0 \times X \times 2^1 \\ 1 \times X \times 2^0 \end{array}$ $\begin{array}{r} + \\ + \\ + \\ + \end{array}$ $\begin{array}{r} 0 \ 0 \ 0 \ 0 \\ 1 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \\ 1 \ 1 \ 0 \ 1 \end{array}$ <hr style="width: 100%;"/> $0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \quad (65)$
---	---

Pour éviter d'avoir à faire les additions sur 8 bits, on préfère commencer par  $Y_0$  et cummuler les produits partiels  $Y_i X$  en poids forts d'un accumulateur RP, que l'on divise par deux à chaque étape. Ceci revient à procéder à la transformation de formule suivante :

$$\begin{aligned} P &= Y_0 \cdot X + Y_1 \cdot X \cdot 2 + Y_2 \cdot X \cdot 2^2 + Y_3 \cdot X \cdot 2^3 \\ &= [[[[Y_0 \cdot X \cdot 2^4] \div 2] + Y_1 \cdot X \cdot 2^4] \div 2 + Y_2 \cdot X \cdot 2^4] \div 2 + Y_3 \cdot X \cdot 2^4 \end{aligned}$$

### Unité de traitement

Le schéma suivant indique la structure d'une unité de traitement pour effectuer des multiplications selon ce principe :



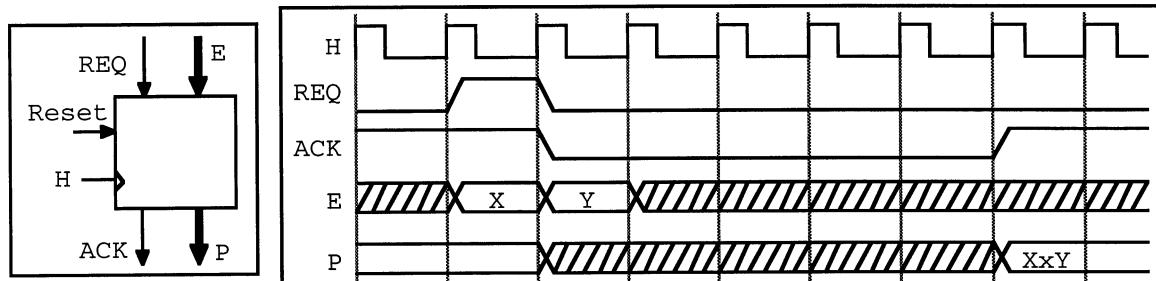
La multiplication est effectuée en un cycle d'initialisation (Ini) suivi de quatre cycles d'étapes de multiplication (Mul).

1 - Indiquer les valeurs successives du registre RP pour les 5 cycles de la multiplication de 5 par 13 (afin de vérifier que cela fonctionne).

2 - Donner le schéma du registre RP à l'aide de bascules et de portes.

## Unité de contrôle

Le multiplicateur complet sera utilisé comme indiqué ci-dessous :



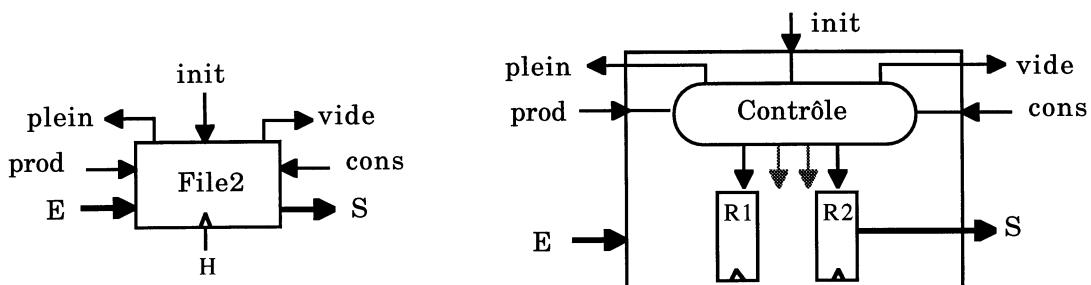
Les opérandes **X** et **Y** sont présentés successivement sur l'entrée **E**. L'opérande **X** est affiché pendant juste un cycle, accompagné du signal **REQ**, et l'opérande **Y** est affiché le cycle suivant. En fin de calcul, la machine maintient le résultat sur **P**, accompagné du signal **ACK**.

3 - Définir l'unité de contrôle par un diagramme d'états.

4 - Réaliser l'unité de contrôle à l'aide de bascules, en utilisant un codage "un bit par état". Prévoir l'initialisation par la commande **Reset**.

## Exercice 2

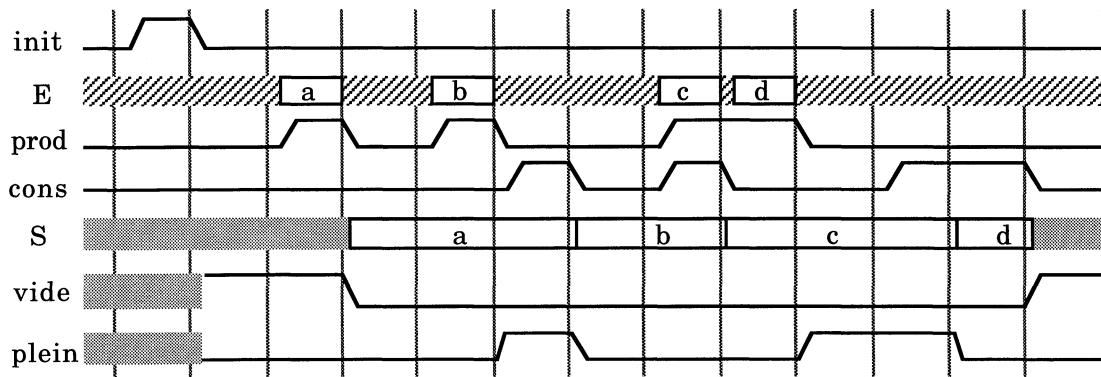
On désire réaliser une file de deux mots de quatre bits, avec deux registres **R1** et **R2**. Les données sont retirées par la commande **cons** dans l'ordre où elles sont introduites par **prod**.



**R2**, tête de la file, contient la valeur prochainement consommée. La sortie **S** affiche **R2** en permanence (la commande **cons** n'a pour effet que de signaler la consommation).

- **init** initialise la file à vide.
- On peut avoir simultanément **prod=1** et **cons=1** dans un cycle, lorsque la file n'est ni vide ni pleine : cela doit faire passer directement la donnée produite en tête).
- deux indicateurs **vide** et **plein** indiquent respectivement que la file contient zéro ou deux éléments.

Le diagramme suivant précise le comportement attendu.



1 - Donner un schéma de la partie concernant la mémorisation des données, en faisant apparaître les commandes qui permettent son contrôle.

2 - Donner le diagramme des états de la partie contrôle. Le comportement pourra être quelconque en cas d'opération illicite (prod dans une file pleine ou cons dans une file vide).

3 - Réaliser l'unité de contrôle à l'aide de bascules, en utilisant un codage de l'état qui incorpore les sorties vide et plein.

### Exercice 3

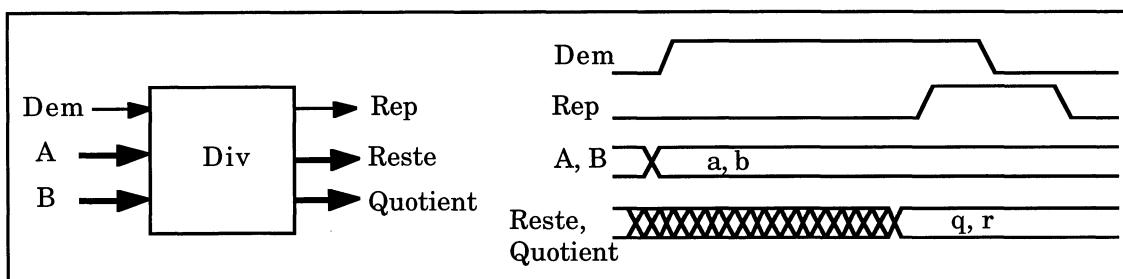
Nous voulons concevoir une machine qui calcule le quotient et le reste de la division de deux nombres fournis sur les entrées A et B.

L'algorithme proposé compte combien de fois on peut retrancher B de A. De plus, pour synchroniser la machine et son utilisateur, on utilise deux signaux classiques Dem (demande de calcul) et Rep (réponse de la machine) :

```

faire
  jusque DemandeCalcul faire % attente opérandes %
    Rep:=0, R:=A, D:=B, Q:=0, quand Dem=1 sortir
    fait;
    tantque R≥D faire
      R:= R-D, Q:= Q+1
    fait;
    jusque RésultatAcquitté faire % attente retombée de Dem %
      Rep:=1, afficher(Reste=R,Quotient=Q),
      quand Dem=0 sortir
    fait
  fait

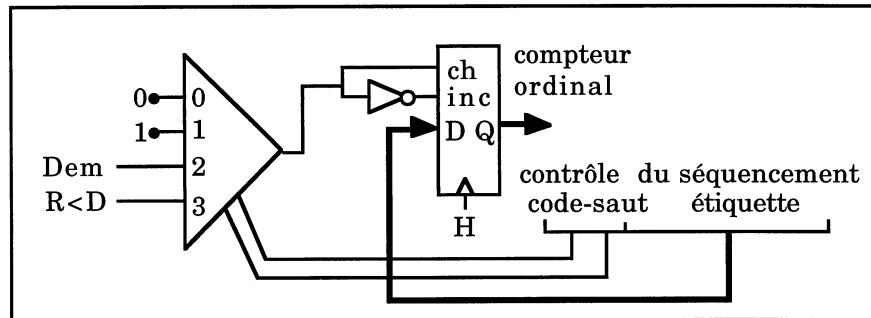
```



1- Concevoir une unité de traitement qui permette d'exécuter toutes les opérations élémentaires de cet algorithme. Définir précisément ses commandes et ses conditions.

- 2- Spécifier l'unité de contrôle correspondant à cet algorithme par un diagramme d'états.
- 3- Réaliser cet automate à l'aide de bascules, en utilisant un codage dense de l'état (indiquer seulement les équations).

On décide de réaliser une unité de contrôle microprogrammée pour cette machine, avec le séquenceur suivant :



Remarque : ce séquenceur offre peu de possibilités, mais vous devez l'utiliser tel qu'il est.

- 4- Préciser le format des micro-instructions.

Donner le microprogramme sous forme symbolique et sous forme binaire

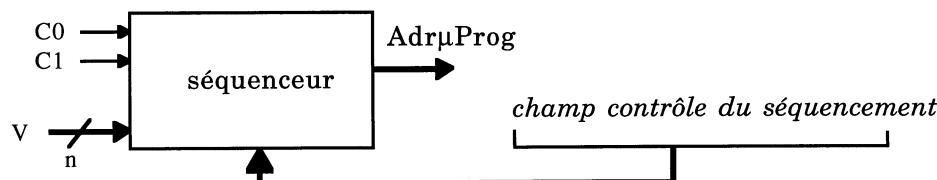
#### Exercice 4

Réécrire le microprogramme de la machine de calcul du pgcd avec un séquenceur permettant les aiguillages à trois branches.

#### Exercice 5

On veut réaliser un séquenceur qui permette :

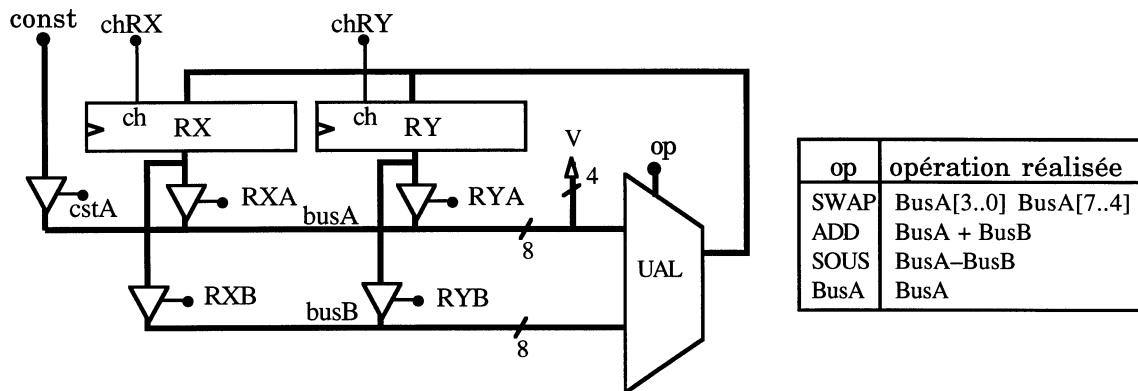
- le passage en séquence inconditionnel,
- quatre sauts conditionnels, selon les conditions C0, C1 et leurs négations,
- les aiguillages à n branches selon la valeur d'une variable entière v.



La valeur v est prélevée sur l'unité de traitement. Les branches d'un aiguillage seront implantées à des adresses successives (les micro-instructions situées à ces adresses contiennent un saut inconditionnel).

Définir le format du champ "contrôle du séquencement" pour des valeurs v de quatre bits, et donner le schéma complet du séquenceur.

On considère l'unité de traitement suivante :



L'UAL réalise l'opération indiquée par la commande `op`. L'opération SWAP permute les quatre bits de poids faibles et de poids forts. L'argument `v` des aiguillages est la valeur des bits de poids faible du bus A.

La valeur de `const` est spécifiée dans un champ de la micro-instruction.

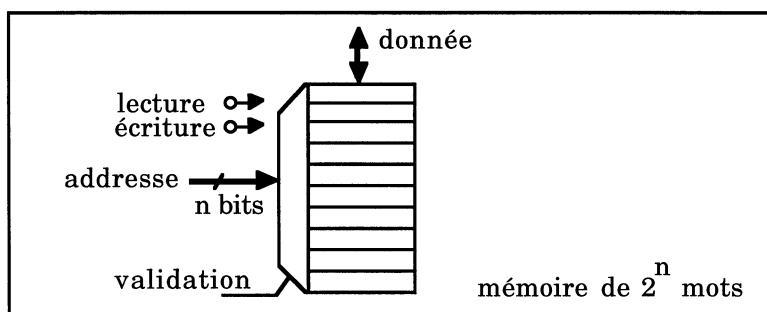
Rédiger le microprogramme qui calcule dans `RY` le nombre de bits à “1” du registre `RX` (par exemple : si `RX` contient 01101011, on doit obtenir 5 dans `RY`).

# MEMOIRES VIVES

## 1 - Mémoires vives adressables

### 1.1 - Généralités

Une mémoire vive permet d'enregistrer et de restituer à la demande des informations. Dans les mémoires adressables, ces informations sont organisées en mots de taille fixe, désignés par une adresse.



Les mémoires adressables sont couramment appelées RAM, de l'anglais random access memory : mémoire à accès aléatoire, ce qui signifie que l'on peut avoir accès immédiatement à n'importe quel mot, indépendamment de ceux auxquels on a accédé précédemment.

Une mémoire de  $2^n$  mots nécessite  $n$  bits d'adresse. Les échanges de données se font par des lignes généralement bidirectionnelles, de largeur égale à la taille du mot. L'accès se fait en spécifiant l'adresse du mot et la nature de l'échange (lecture ou écriture), et en activant la validation.

Il existe deux types de composants mémoire intégrés, aux domaines d'application assez spécifiques, les mémoires statiques et les mémoires dynamiques :

- Les **mémoires statiques** (SRAM : Static RAM) conservent l'information enregistrée pendant une durée illimitée, tant que le circuit est sous tension. Simples d'emploi, elles peuvent offrir des temps d'accès très courts (de 10ns à 50ns). La consommation en énergie dépend de la technologie, mais elle est assez importante et d'autant plus élevée que la mémoire est rapide. On les utilise pour réaliser des mémoires spécifiques (communication entre

processeurs, tampons d'entrée/sortie), de petites mémoires de travail pour processeurs rapides, des mémoires "cache" (destinées à améliorer le temps moyen d'accès d'un ordinateur à sa mémoire principale), ou encore les mémoires centrales de petits systèmes (pour des raisons de simplicité).

- Les **mémoires dynamiques** (DRAM : Dynamic RAM) perdent leur contenu au bout d'un temps limité, de l'ordre de quelques millisecondes. Un mécanisme de rafraîchissement est nécessaire pour régénérer leur contenu périodiquement, ce qui complique nettement leur utilisation. Leur temps d'accès est plus long (de 50ns à 200ns), mais elles sont beaucoup plus intégrées que les mémoires statiques (quatre à huit fois plus que les SRAM). On les utilise comme mémoires centrales d'ordinateurs et pour les mémoires de visualisation des écrans graphiques.

## Evolution

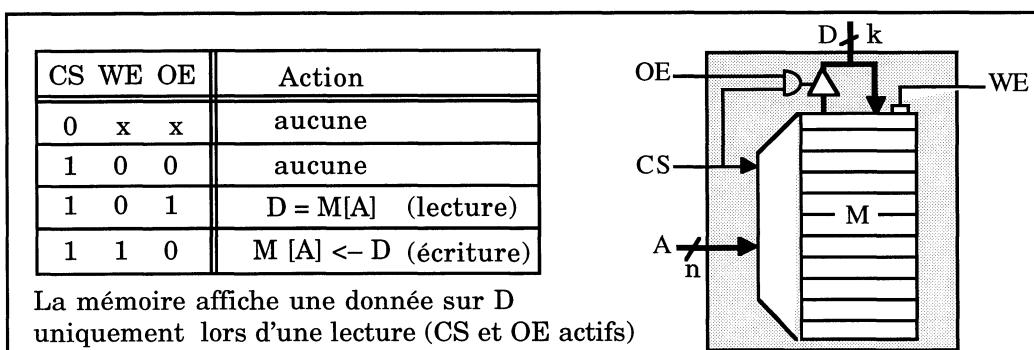
Le tableau suivant indique les capacités par boîtier de mémoires dynamiques pour ces quinze dernières années. On constate que le taux d'intégration des mémoires a pratiquement quadruplé tous les trois ans. Parallèlement, le coût des mémoires a fortement baissé.

1976	1978	1980	1983	1986	1990	1993
4k bits	16k bits	64k bits	256k bits	1M bits	4M bits	16M bits

## 1.2 - Mémoires statiques

### 1.2.1 - Aspects externes

Nous présentons ici un exemple typique d'organisation. Les signaux de contrôle des mémoires statiques sont toujours sensiblement les mêmes.



- CS : validation du boîtier (anglais : Chip Select). Pour CS inactif, WE et OE sont sans effet.
- WE : commande d'écriture (anglais : Write Enable). Si CS est actif, elle provoque l'écriture de l'entrée D dans le mot adressé.
- OE : validation de la sortie (anglais : Output Enable). Si CS et OE sont actifs, le contenu du mot adressé est affichée sur D, sinon la sortie D est laissée libre. On utilise généralement OE comme commande de lecture, ce qui permet d'éviter les conflits sur les lignes de données lors d'une écriture.
- A : entrées d'adresse, n bits d'adresse pour  $2^n$  mots.
- D : lignes d'entrée-sortie de données, k lignes pour des mots de k bits. Les mots font généralement quatre ou huit bits.

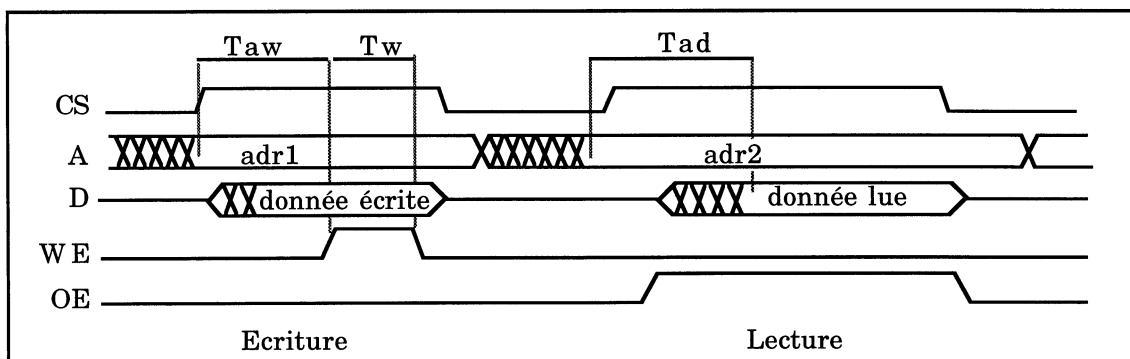
La mémoire présentée ici possède une ligne commune pour les entrées et les sorties de données. Il en existe d'autres types, avec lignes distinctes.

### 1.2.2 - Fonctionnement temporel

Pour une écriture, l'extérieur active CS, affiche l'adresse et la donnée puis active le signal WE. Pendant l'écriture, la commande OE doit être maintenue inactive pour que le circuit laisse libre les lignes de données.

Pour une lecture, l'extérieur active CS, positionne l'adresse et active OE. La mémoire place alors la donnée adressée sur D.

Le diagramme suivant illustre le déroulement d'une écriture et d'une lecture. Pour l'exemple, nous considérons des signaux de commande actifs à 1, mais il faut noter qu'en général les signaux de commande sont actifs à 0 (ils s'appellent alors /CS, /WE et /OE).



Les principaux paramètres temporels sont :

- Tad : délai entre le positionnement de l'adresse et la validité de D, encore appelé "temps d'accès en lecture".
- Taw : temps à respecter entre le positionnement de l'adresse et le signal

d'écriture WE, de manière à ne pas écrire n'importe où.

$T_w$  : durée minimum de maintient de WE.

$T_{aw} + T_w$  est le temps d'accès en écriture ; il est souvent égal au temps d'accès en lecture (10 à 100 ns pour les composants usuels).

### 1.2.3 - Constitution de plans mémoire

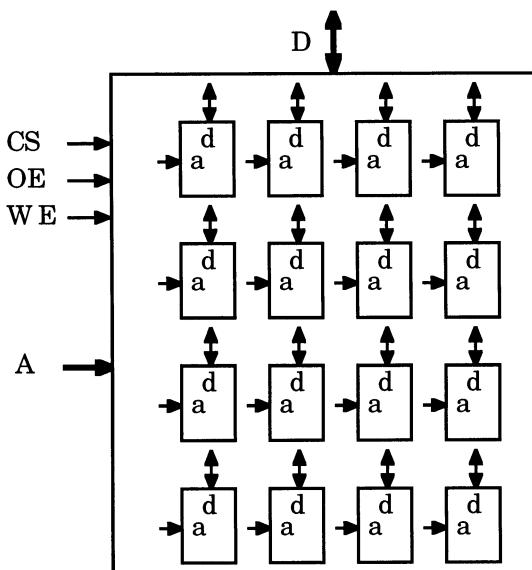
Un "plan mémoire" est l'assemblage de plusieurs composants pour former une mémoire plus grande.

L'assemblage peut se faire selon deux dimensions :

- L'assemblage horizontal (ou en largeur) permet de réaliser des mots plus larges. Les boîtiers sont en parallèle, avec les mêmes signaux d'adresse et de contrôle.

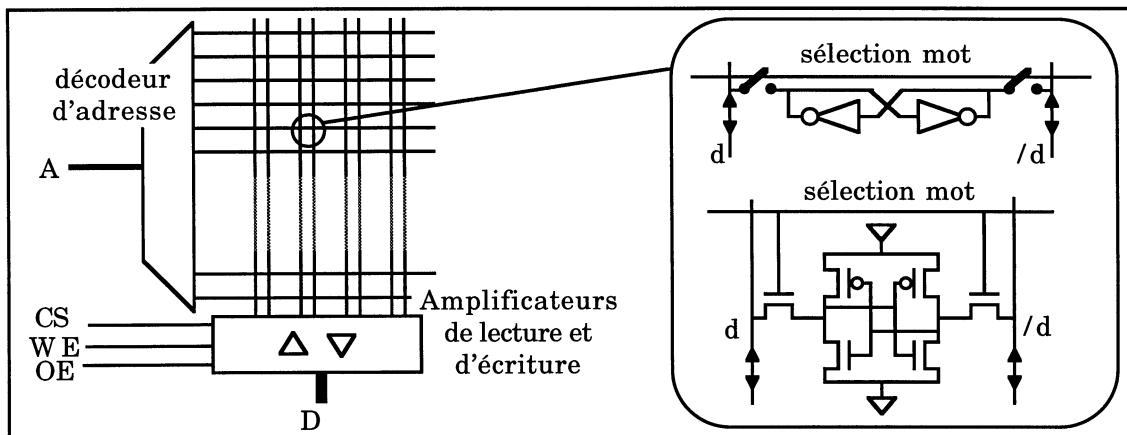
- L'assemblage vertical (ou en profondeur) permet d'augmenter le nombre de mots. On interconnecte les broches de données. Un seul boîtier est validé, grâce au CS et au décodage des bits d'adresse supplémentaires de la mémoire totale.

Ces deux modes d'expansion sont généralement associés.



### 1.2.4 - Structure interne d'une mémoire statique

Chaque bit de mémoire est physiquement constitué de deux amplificateurs inverseurs bouclés, ce qui offre les deux états stables. Les bits du mot sélectionné par le décodeur d'adresse sont affichés sur les colonnes d et /d (la donnée et son complément, pour une meilleure fiabilité de la lecture).



La lecture consiste à amplifier ces signaux et à les afficher en sortie.

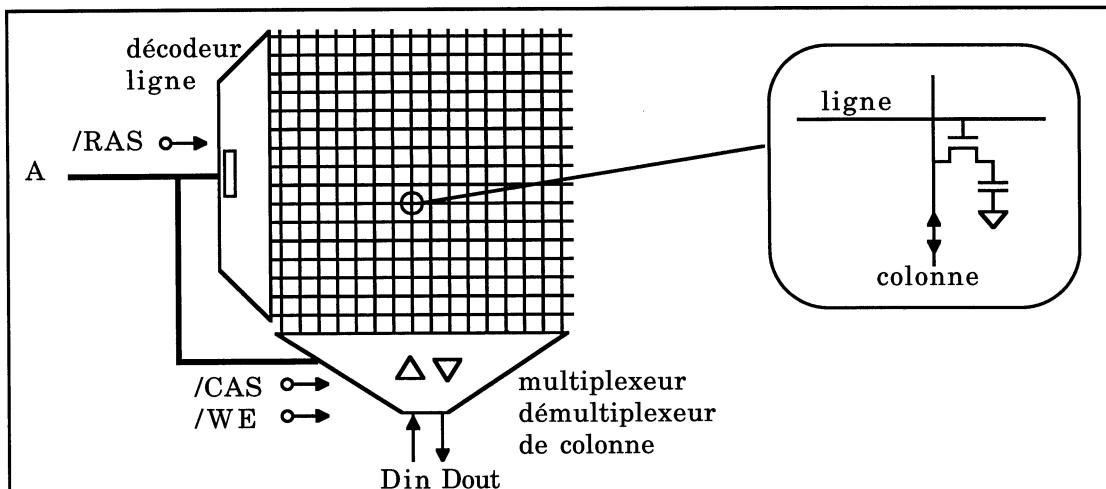
L'écriture est faite en forçant la valeur à écrire sur les colonnes  $d$  et  $/d$ . Ce forçage est réalisé par des amplificateurs d'écriture, assez puissants pour faire basculer la valeur enregistrée. Ce montage ressemble à celui d'un verrou RS, mais ici le bon fonctionnement doit être assuré par les caractéristiques physiques particulières des transistors et des amplificateurs d'écriture.

On en retiendra que la mémorisation est de durée illimitée car la valeur mémorisée est entretenue par le montage bouclé, et qu'il faut six transistors par bit.

### 1.3 - Mémoires dynamiques

#### 1.3.1 - Principe de fonctionnement

Les caractéristiques originales des mémoires dynamiques sont la durée limitée de l'information enregistrée et le multiplexage des bits d'adresse.



## Durée limitée de l'information enregistrée

Chaque bit de mémoire est réalisé par la capacité parasite d'un transistor, chargée ou déchargée à travers ce transistor lors de l'écriture. L'information se maintient pendant un temps limité car la charge de cette capacité se dégrade au cours du temps. Le taux d'intégration est très élevé car il n'y a qu'un seul transistor par bit, alors qu'il en faut six pour une mémoire statique.

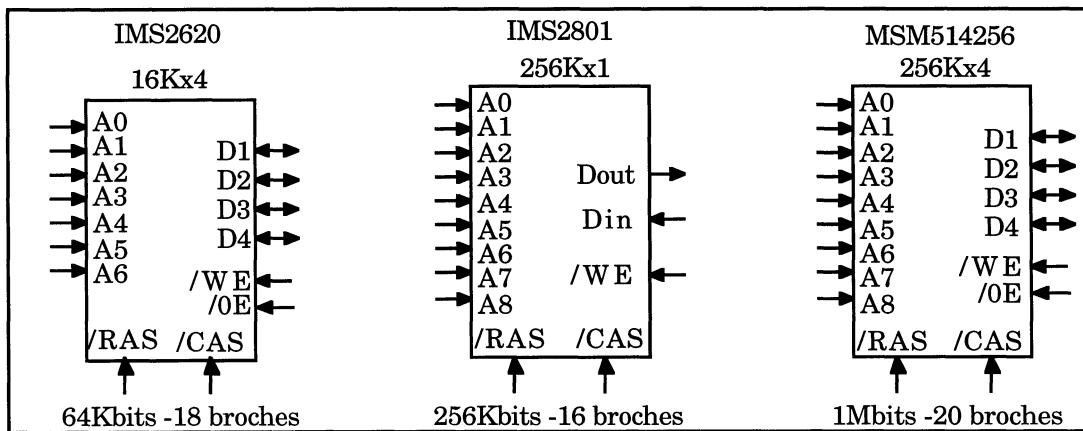
## Multiplexage des bits d'adresse

La mémoire est formée de N lignes de N colonnes de bits. A cause cette structure interne matricielle, les bits d'adresse sont multiplexés en deux parties au cours du temps :

- adresse de ligne, validée par /RAS (“row address strobe”),
- puis adresse de colonne, validée par /CAS (“column address strobe”).

Tout accès commence par la lecture d'une ligne entière. Cette lecture est destructrice, car les capacités sont déchargées sur les fils de colonnes. Dans un deuxième temps, la ligne est entièrement réécrite, après amplification, et modification du bit concerné en cas d'écriture. La mémoire utilise donc les adresses de ligne et de colonne successivement.

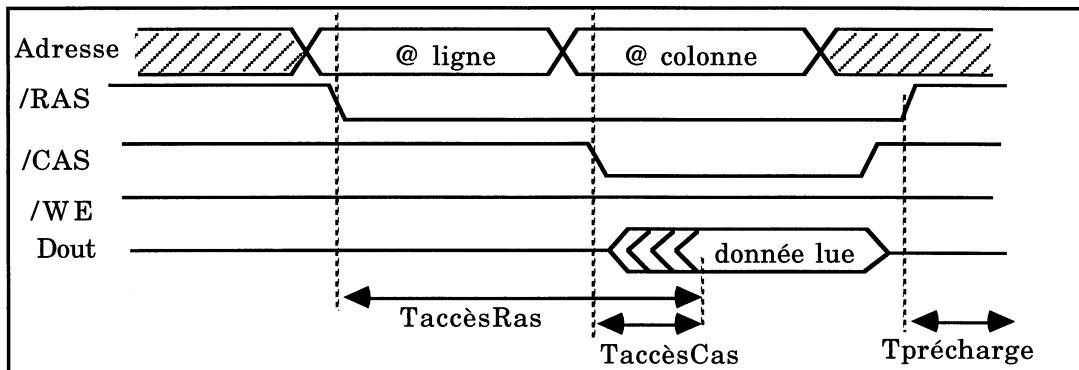
Un intérêt du multiplexage est de diviser par deux le nombre de broches d'adresse. Ainsi les boîtiers de mémoire sont petits, comme le montre le brochage des composants suivants :



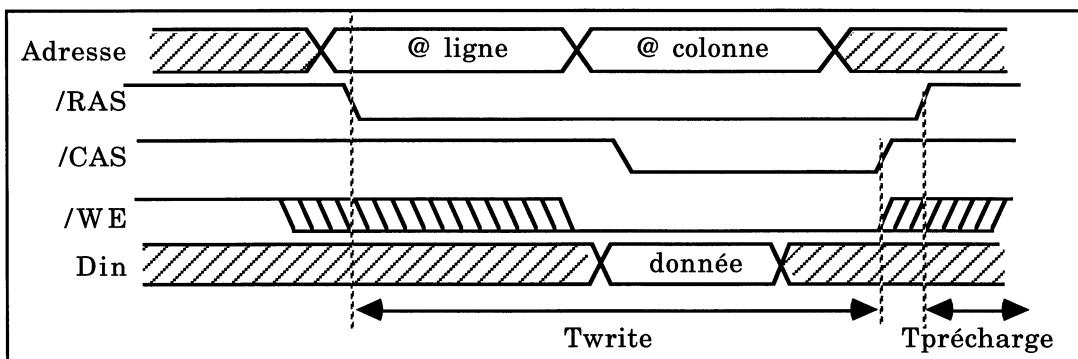
### 1.3.2 - Chronogrammes des accès

Les trois accès de base sont la lecture, l'écriture et le rafraîchissement.

#### Lecture

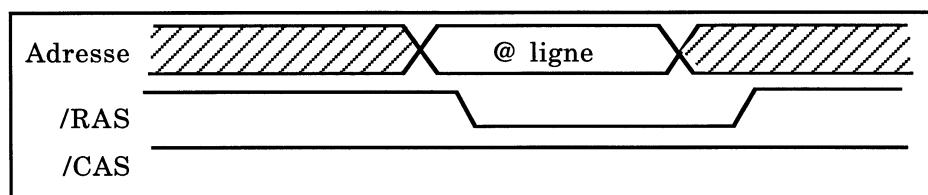


#### Écriture



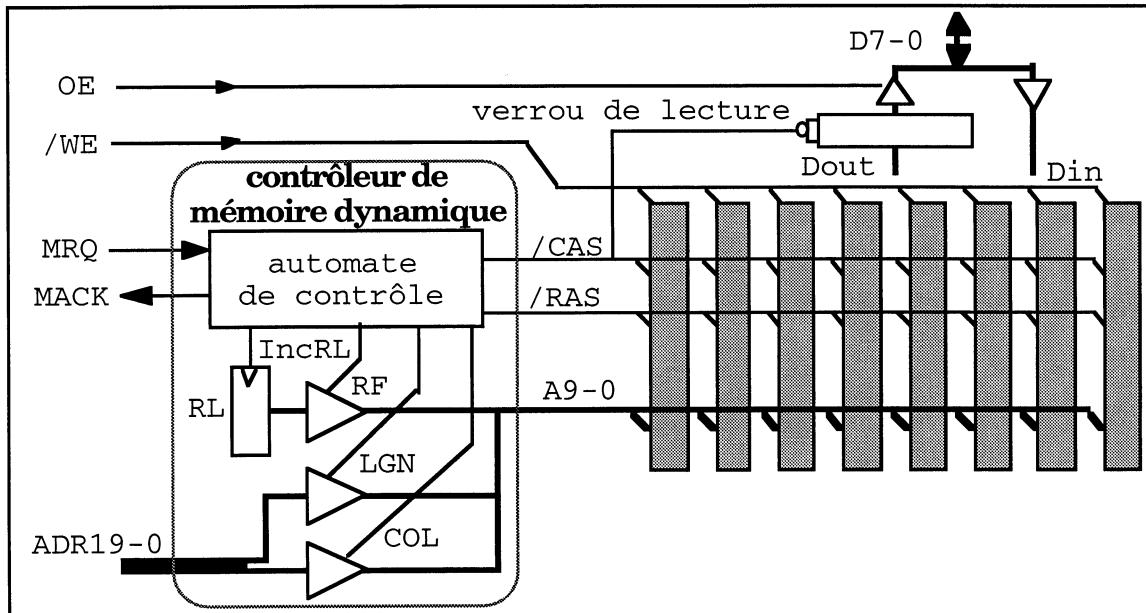
#### Rafraîchissement

Le rafraîchissement se fait par ligne entière à l'intérieur du boîtier, en activant simplement **/RAS** accompagné du numéro de ligne à rafraîchir.



### 1.3.3 - Constitution d'un plan de mémoire dynamique

Le schéma suivant illustre une mémoire dynamique de 1Méga-octet, réalisée à l'aide de composants 1Méga-bit. Cette mémoire incorpore un contrôleur de mémoire dynamique qui réalise le rafraîchissement.



#### Rôle du contrôleur de mémoire

Le contrôleur partage l'utilisation de la mémoire entre les accès de l'utilisateur et les accès de rafraîchissement. Ceux-ci arrivent à des moments quelconques par rapport aux accès utilisateur, et la mémoire est indisponible pendant ce temps. L'accès utilisateur doit donc être mis en attente. Pour cela, une paire de signaux **MRQ** et **MACK** assure la synchronisation : l'utilisateur demande un accès par **MRQ**, et lorsque l'accès est terminé le contrôleur répond par **MACK**.

Le contrôleur fonctionne avec une horloge personnelle de période rapide, par exemple 50 ns, qui sert de base de temps pour la chronologie des divers signaux (**/RAS**, **/CAS**, **RF**, **LGN**, **COL** ...).

Le multiplexage de l'adresse lors des accès de l'utilisateur est commandé par les signaux **LGN** (numéro de ligne : **ADR10-19**) et **COL** (numéro de colonne : **ADR0-9**). En lecture, la donnée est enregistrée dans un verrou chargé. Une fois la donnée dans ce verrou, l'utilisateur dispose ainsi d'un temps arbitraire pour prélever la donnée et supprimer sa demande **MRQ**.

Pour réaliser un cycle de rafraîchissement, le contrôleur valide (par la commande **RF**) l'adresse de la ligne à rafraîchir contenue dans le registre **RL** et génère le signal **/RAS** vers tous les composants. Au même moment, la commande **IncRL** incrémente le compteur **RL** de manière à rafraîchir régulièrement toutes les lignes.

## 2 - Mémoires à accès particulier

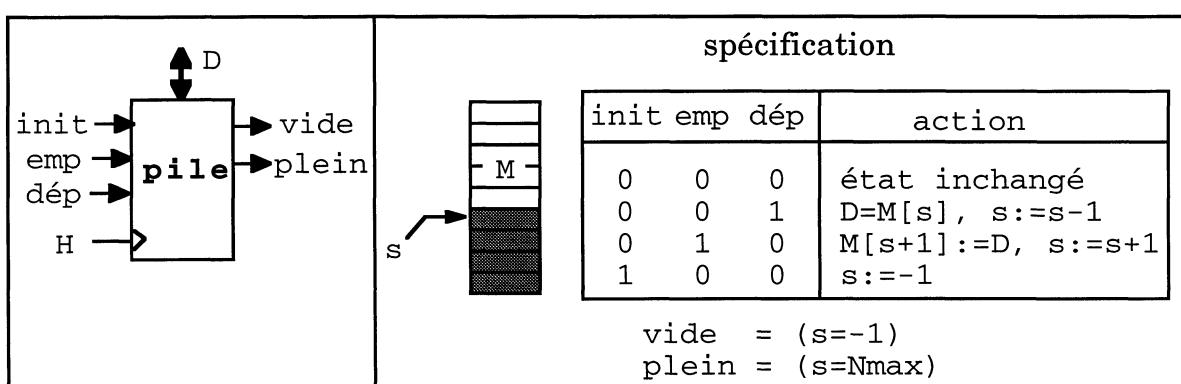
### 2.1 - Mémoires à adressage implicite - piles et files

Dans certaines applications, l'ordre dans lequel on lit les données est directement lié à celui dans lequel elles ont été enregistrées. Les deux principaux mécanismes sont les piles (accès LIFO : last in first out) et les files (accès FIFO : first in first out). Dans ces deux cas, l'emplacement de la donnée que l'on lit n'est pas indiqué par une adresse, mais découle de l'historique des lectures et des écritures.

#### 2.1.1 - Pile

Dans une pile, les données sont prélevées dans l'ordre inverse de leur enregistrement. L'enregistrement s'appelle l'empilement et le prélèvement le dépilement. Lors du dépilement, la donnée lue est "consommée", c'est-à-dire retirée de la pile.

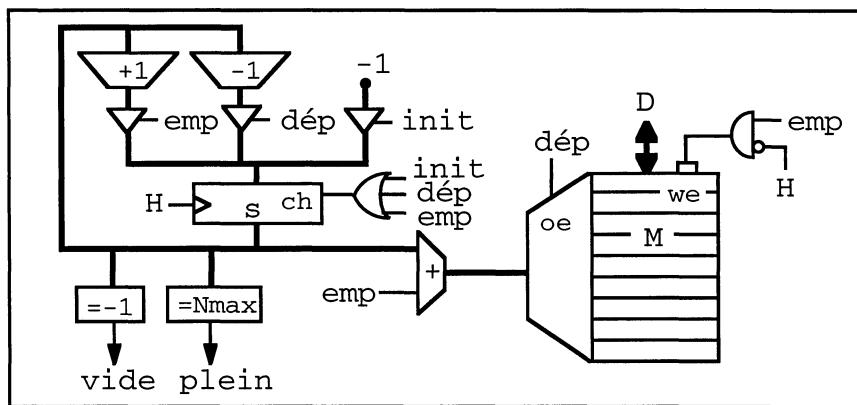
La commande **init** permet d'initialiser la pile à vide (aucune donnée enregistrée). Les commandes **emp** et **dép** assurent l'empilement et le dépilement. L'indicateur **vide** signale que la pile est vide. La capacité d'une pile est nécessairement finie : un indicateur **plein** signale que la capacité maximum est atteinte.



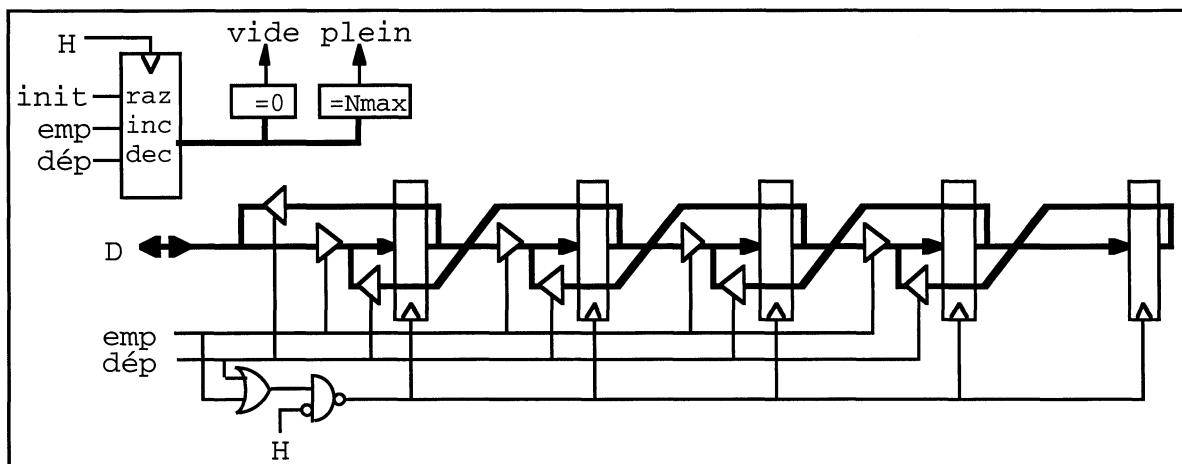
Le fonctionnement de la pile peut être spécifié au moyen de deux variables, un tableau **M** et un indice **s**. Le tableau sert à conserver les données, dans leur ordre d'enregistrement, et l'indice (le sommet de la pile) est le rang de la donnée la plus récente qui n'a pas encore été prélevée.

### Réalisations d'une pile

On peut réaliser une pile de multiples façons. Une méthode consiste à calquer les formules de spécifications précédentes, en utilisant une mémoire adressable pour le tableau **M** et un registre pour l'index **s**, accompagné d'opérateurs pour le calcul de **s+1** et **s-1**.

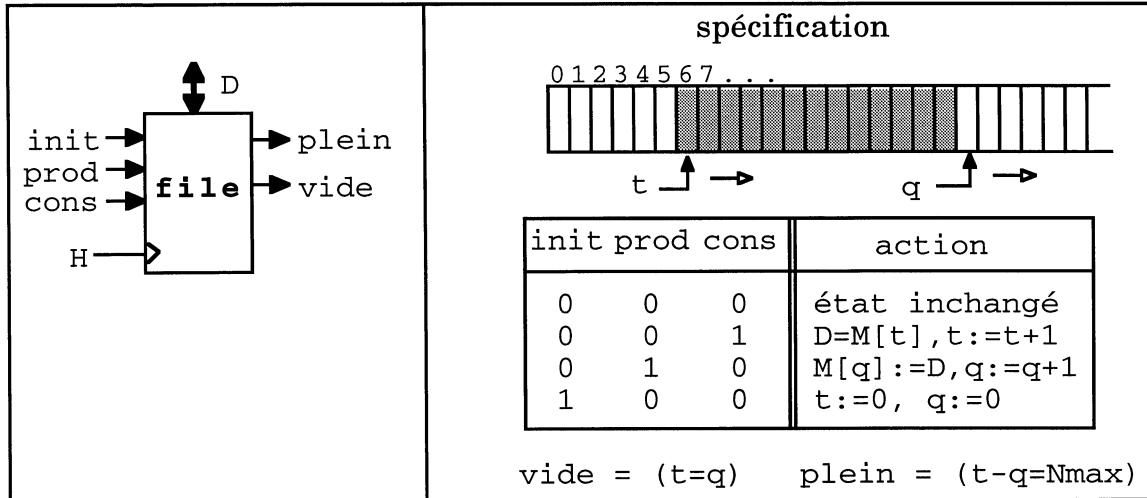


Une autre méthode, mieux adaptée dans le cas d'une réalisation de circuit intégré, consiste à utiliser des registres montés en décalage. L'empilement “pousse” les données à droite, en enregistrant la nouvelle donnée, et le dépilement les décale à gauche pour faire apparaître le nouveau sommet.



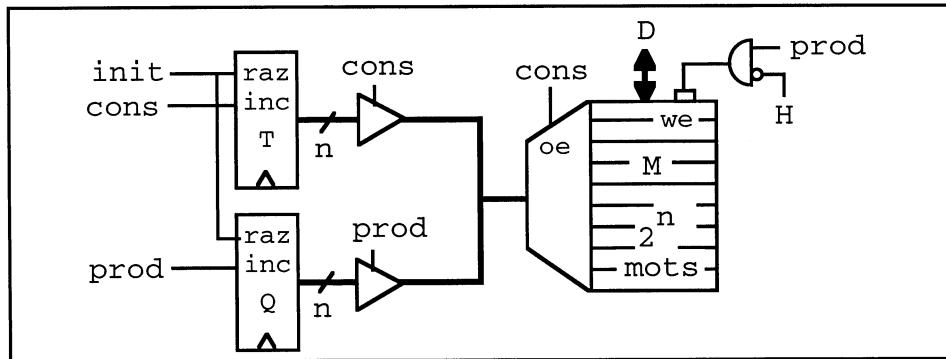
### 2.1.2 - File

Dans une file, les données sont prélevées dans l'ordre de leur enregistrement. L'enregistrement s'appelle une production et le prélèvement une consommation. Le fonctionnement peut être spécifié par un tableau **M**, potentiellement infini, indicé par deux entiers : **t**, la tête, est l'indice de la plus ancienne donnée non encore consommée et **q**, la queue, est l'indice de la prochaine donnée à enregistrer.



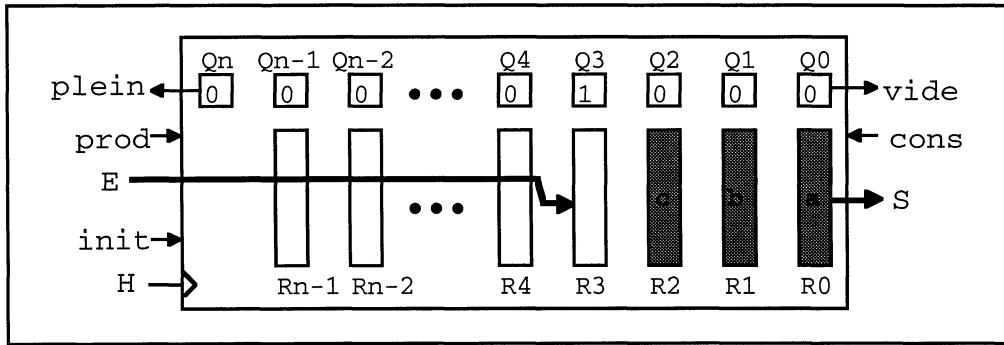
### Réalisations d'une file

Une file peut être réalisée à l'aide d'une mémoire adressable et de registres pour représenter les indices de tête et de queue. Les mémoires étant de taille finie, on simule le tableau infini en gérant circulairement un espace fini : les indices **t** et **q** sont représentés par des pointeurs **T** et **Q** incrémentés modulo la taille de la file.



Un inconvénient de cette réalisation est que la mémoire adressable ne permet qu'un seul accès à la fois, car il y a un seul décodeur d'adresse et les lignes de données sont bidirectionnelles. Ceci n'est pas gênant dans le cas d'une pile qui n'a d'ordinaire qu'un seul utilisateur. En revanche, dans la plupart des applications d'une file il y a deux utilisateurs "indépendants", un producteur qui s'autorise à produire si la file n'est pas pleine, et un consommateur qui s'autorise à consommer si elle n'est pas vide. Il est alors préférable d'avoir une file qui permette une simultanéité de production et de consommation.

On peut également réaliser une file à l'aide de registres, selon le principe suivant :



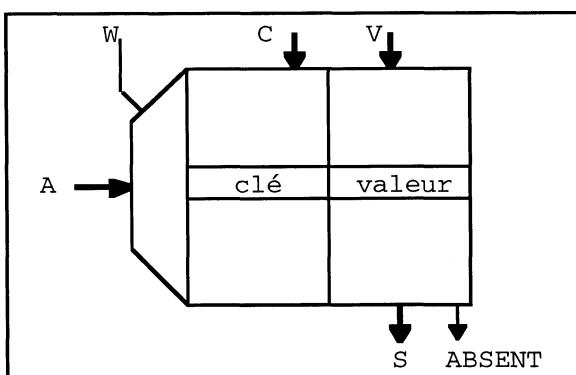
Il y a ici des lignes séparées pour l'entrée de données, **E**, et la sortie, **S**, ce qui permet une production et une consommation simultanées.

La donnée de tête est contenue dans l'emplacement fixe **R0**. Elle est affichée en permanence sur **S**. A chaque registre est associée une mémoire un bit **qi** pour repérer la queue de file par **qi=1**. Le bit **q0** indique la condition **vide** et un bit supplémentaire **qn** indique la condition **plein**. **q0** est initialisé à 1 et les autres **qi** sont initialisés à 0. Lors d'une production, la donnée **E** est enregistrée dans le registre de queue et les **qi** sont décalés vers la gauche. Lors d'une consommation, les données et les **qi** sont décalés vers la droite.

## 2.2 - Mémoires associatives

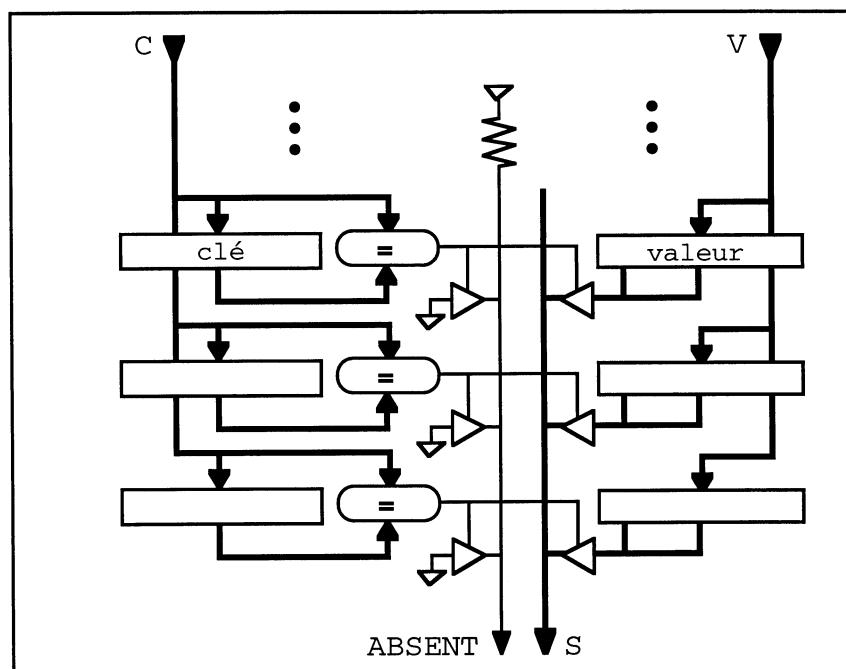
Avec une mémoire associative on accède à un mot en lecture non pas en donnant son adresse mais une partie de son contenu, la "clé".

Chaque mot de la mémoire est composé d'un champ clé et d'un champ valeur. L'écriture se fait par adressage : lorsque la commande **w** est activée, le mot complet fourni sur les entrées **c** (clé) et **v** (valeur) est écrit à l'adresse indiquée par **A**. La lecture se fait en donnant une clé sur l'entrée **c**. S'il existe, le champ valeur du mot dont la clé vaut **c** est affiché sur la sortie **s**. Le résultat de la lecture n'est défini que si les clés enregistrées sont différentes. De plus, un indicateur **ABSENT** indique qu'aucun mot ne contient la clé.



Ces mémoires sont utilisées pour réaliser des tables de correspondance à accès rapide, par exemple les tables de conversion des adresses virtuelles en adresses réelles dans les processeurs doté d'un mécanisme de mémmoire virtuelle : le temps d'accès clé->valeur est du même ordre de grandeur que pour un accès à une mémoire adressable rapide, 10ns à 50ns.

Pour cela, chaque mot possède son propre comparateur d'égalité avec la clé. Le résultat de ce comparateur valide l'affichage du champ valeur sur le bus de donnée.



L'indicateur **ABSENT** est généré par une ligne “collecteur ouvert” : si une clé est égale à **c**, le comparateur correspondant force un 0 sur la ligne, sinon la ligne est maintenue à 1 par une résistance de rappel.

## Exercices

### Exercice 1

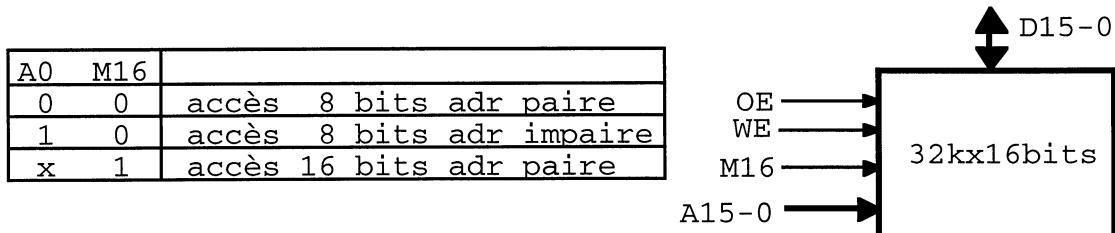
Réaliser une mémoire adressable de seize mots de quatre bits en utilisant des verrous D (un verrou de  $k$  bits par mot), sélectionnés par un décodeur.

## Exercice 2

Dessiner le schéma d'interconnexion de composants mémoire de 2k mots de 8 bits pour réaliser une mémoire de 8k mots de 16 bits.

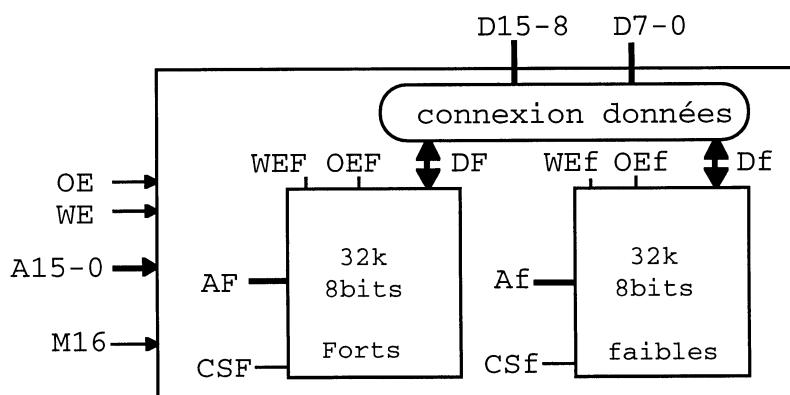
## Exercice 3

On dispose d'un processeur qui travaille sur des données de 8 bits et 16 bits. La mémoire de ce processeur est organisée selon le format le plus large, 16 bits, mais permet également de lire ou d'écrire un octet :



La mémoire est adressée par octets (seize bits d'adresse : **A15-0**), et le signal **M16** indique le type d'accès : mot de 16 bits (**M16 = 1**) ou octet (**M16 = 0**). Les mots de 16 bits sont toujours alignés sur des adresses paires. Pour un accès 8 bits, la donnée est présentée sur les lignes **D7-0**.

On réalise cette mémoire avec des composants 32k x 8bits, comme l'illustre le schéma suivant. Un des composants mémorise les octets d'adresse paire (poids faibles d'un mot de 16 bits), et l'autre mémorise les octets d'adresse impaire (poids forts) :



1. Déterminer le plus simplement possible les signaux de commande de chacun des composant mémoire :
- CSF, CSf, WEF, WEf, OEF, OEf, AF, Af.**

On s'intéresse maintenant aux connexions de données.

## 2- Indiquer comment connecter très simplement DF.

On les désigne comme suit les diverses possibilités de connexion entre **DF**, **D15-8** et **D7-0** :

**E1** : "D7-0 → DF"    **E2** : "D15-8 → DF"

**L1** : "DF → D7-0"    **L2** : "DF → D15-8"

## 3- Déterminer ces variables et en déduire un schéma complet de la mémoire.

### Exercice 4

Indiquer un brochage possible pour des boîtiers mémoires de seize Mégabits (nombre et rôle des broches)

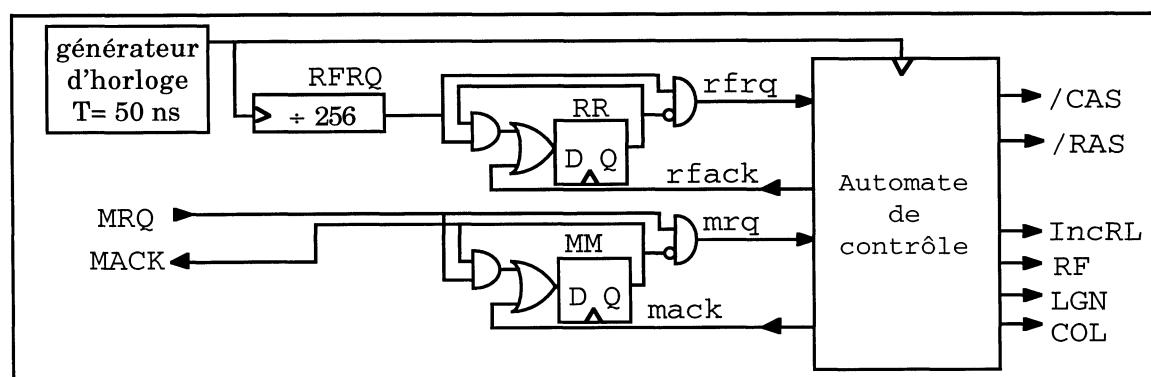
- organisés en mots de 1 bit, entrées et sorties de données séparées
- organisés en mots de 4 bits, entrées et sorties communes.

### Exercice 5 – contrôleur de mémoires dynamiques –

On dispose de composants mémoires dynamiques 1Mbits organisés en 256 lignes de 4096 bits. Chaque ligne doit être rafraîchie toutes les 4.4 ms. Les caractéristiques de ces mémoires sont :

**TaccèsRas = Tw = 120 ns   TaccèsCas = 20 ns   Tprécharge = 75 ns**

Avec 256 lignes à rafraîchir toutes les 4.4 ms, la période maximum de rafraîchissement est de 17.8 µs ( $\approx 4.4 \text{ ms} / 256$ ). Nous utilisons une horloge de période 50ns. La demande de rafraîchissement **RFRQ**, issue d'un diviseur de fréquence par 256, passe à 1 toutes les 12.8 µs, ce qui respecte la période maximale imposée.



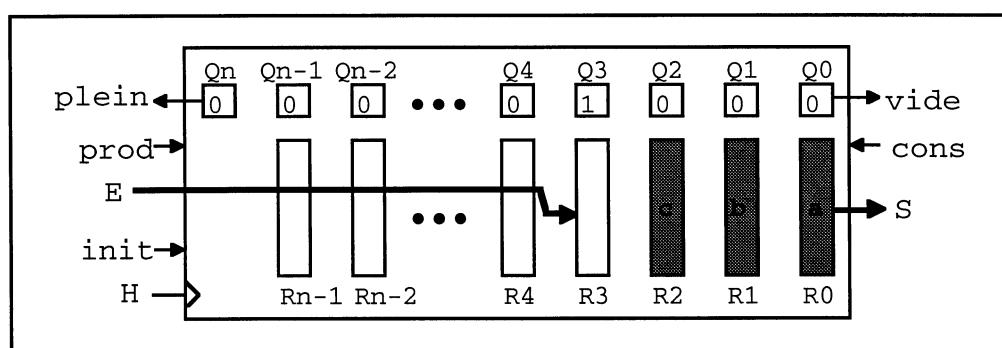
Chaque demande d'accès ou de rafraîchissement doit être traitée une fois et une seule. Ceci est assuré grâce aux bascules **MM** et **RR** :

La bascule **MM** passe à 1 quand l'automate de contrôle signale la fin de l'accès par **mack**, et à 0 quand l'utilisateur supprime sa demande. Ainsi, le signal **mrq** signifie "demande non encore acquittée". Le fonctionnement de la bascule **RR** est similaire, avec les signaux **rfack** et **rfrq** à la place de **mack** et **mrq**.

**Question** : dessiner le diagramme des états de l'automate de contrôle du contrôleur de mémoire.

### Exercice 6      Réalisation d'une file

On considère la réalisation d'une file à l'aide de registres, selon le principe évoqué par le schéma suivant :



On accepte une production et une consommation simultanées. On suppose que l'utilisateur ne consomme pas si la file est vide et ne produit pas si la file est pleine.

- 1 - Exprimer le fonctionnement au moyen de tables et de formules d'affectation.
- 2 - Donner le schéma d'une position *i* de la file (**Qi** et un bit du registre **Ri**).

# BUS DE PROCESSEURS

---

## 1 - Interconnexion de composants séquentiels

Tout assemblage logique vise à faire travailler des circuits en coopération, ce qui nécessite des interactions entre les différents composants.

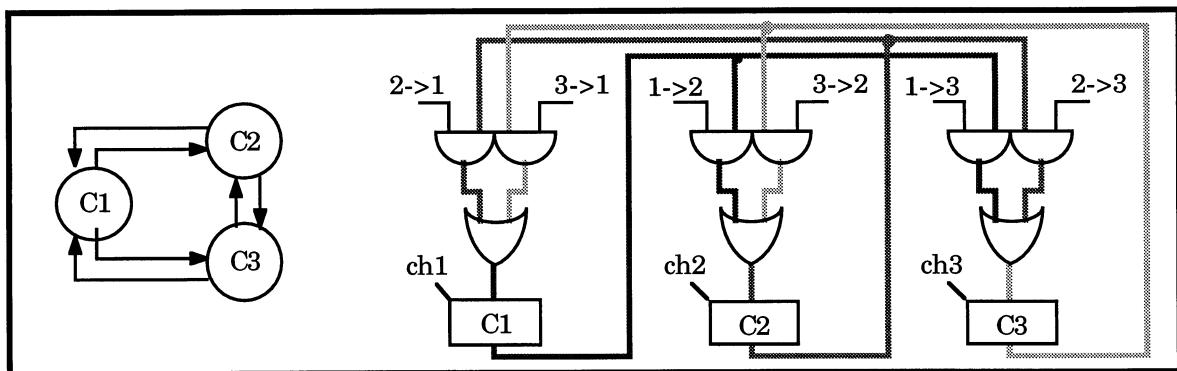
On se place dans un contexte synchrone : les composants sont modélisés par des registres à chargement contrôlé (les entrées sont chargées si la commande de chargement est active) ; les sorties sont toujours disponibles.

Pour établir une liaison cohérente entre des composants, il faut gérer pour chacun la commande de chargement et la source des données incidentes.

Diverses techniques d'interconnexion offrent des possibilités plus ou moins riches, à des coûts (complexité des circuits annexes, nombre de fils) variables.

### 1.1 - Liaison directe point à point

Chaque composant peut recevoir son information de tout autre composant, et émettre sa valeur vers tout autre composant.



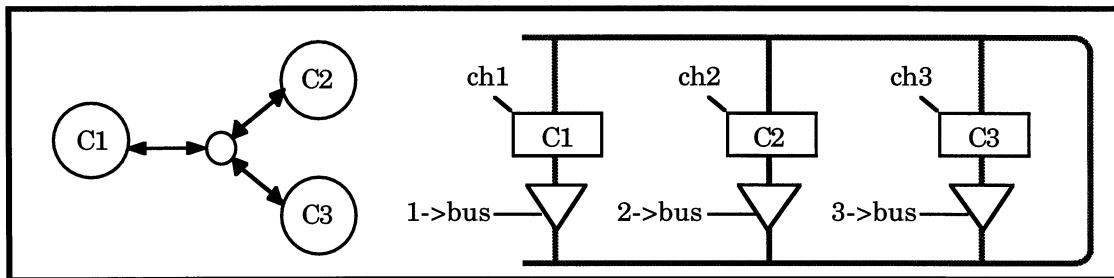
- Solution puissante : toute combinaison peut se réaliser en un cycle.
- Coût élevé : pour  $N$  composants,  $N$  logiques de sélection,  $N^2$  commandes, et  $N$  paquets de fils de liaison.

De plus, l'extension (ajout d'un composant) est délicate, et conduit à modifier les circuits de sélection de source déjà réalisés “ET” supplémentaires, “OU” plus larges ...).

## 1.2 - Liaison par canal centralisé : le bus

Toutes les informations passent par le même canal : chacun peut y émettre sa valeur, et tous peuvent la charger.

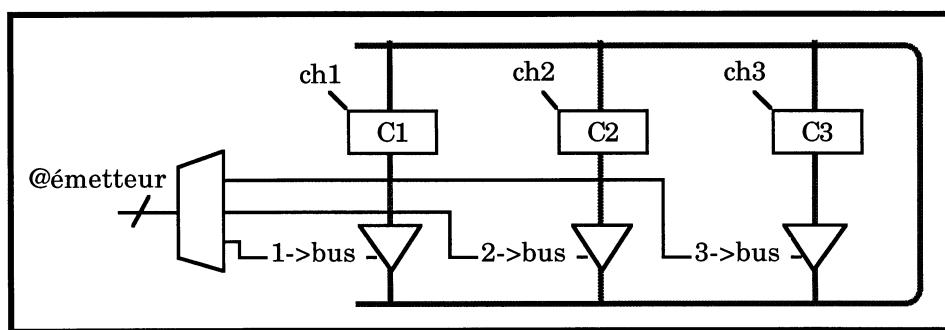
Le canal est matérialisé par un “bus” en technologie trois-états.



- Solution moins riche : un seul émetteur possible à un instant donné (mais plusieurs récepteurs potentiels).
- Plus économique : pour N composants,  $2 * N$  commandes, N adaptateurs et un seul paquet de fils.

### 1.2.1 - Le bus adressable

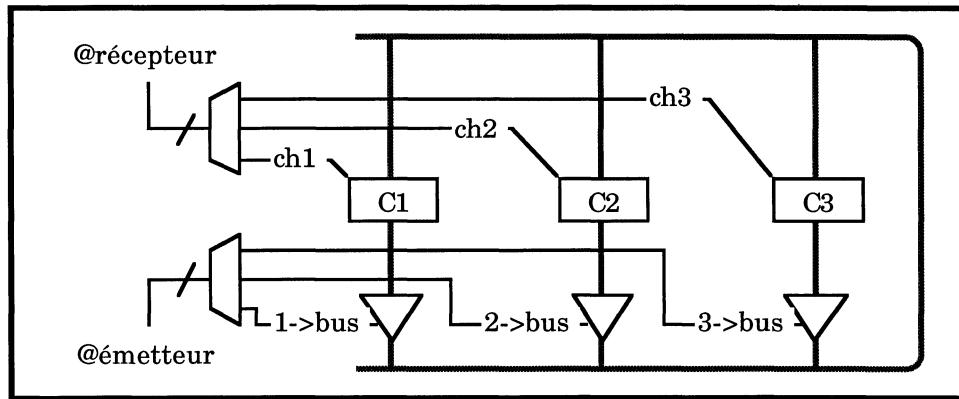
Une solution élégante pour éviter les conflits sur le bus consiste à valider les adaptateurs trois-états à travers un décodeur ; à chaque composant est alors associée une “adresse d’émetteur”.



- Sans contrainte supplémentaire (un émetteur, plusieurs récepteurs), cette approche réduit le nombre de fils : pour N composants,  $\log_2(N)$  fils suffisent pour spécifier l'émetteur.

### 1.2.2 - Le bus restreint

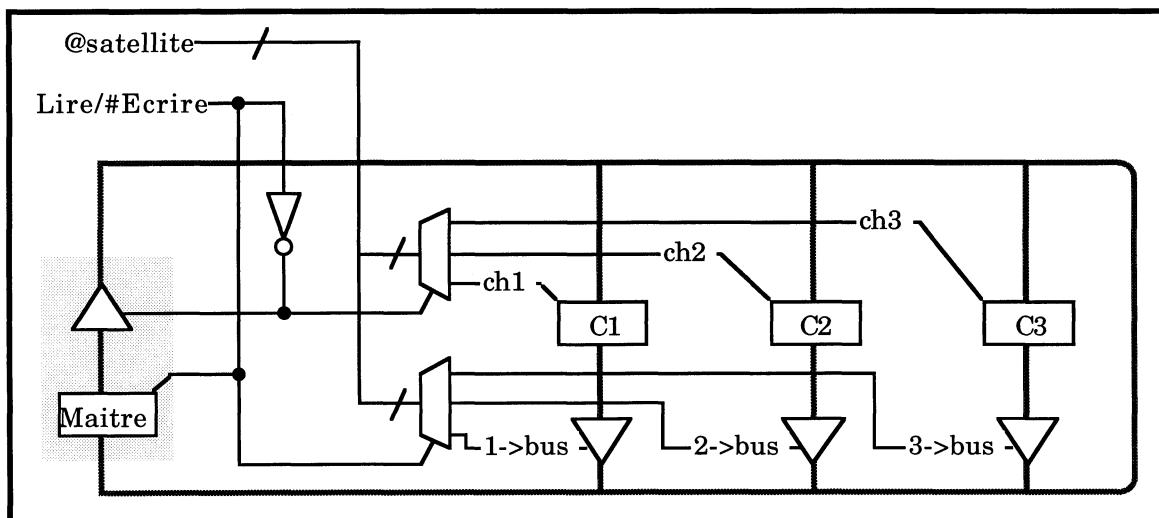
On n'autorise qu'un seul récepteur à la fois, et on associe une "adresse de récepteur" aux composants (généralement la même que l'adresse émetteur).



- Puissance réduite : la diffusion (plusieurs récepteurs simultanés) n'est plus possible.
- Complexité réduite : pour N composants,  $2 * \log_2(N)$  fils de commande.

### 1.2.3 - Le bus à composant maître

Un composant joue un rôle privilégié : il est partenaire de tout échange, soit comme émetteur (unique), soit comme récepteur (unique).

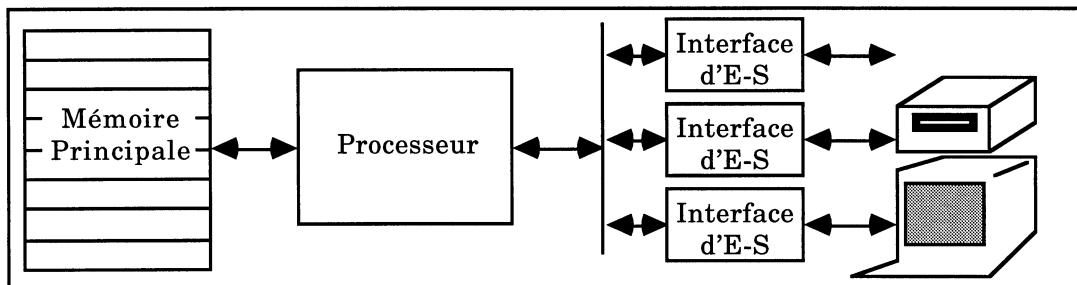


- Puissance encore plus limitée : pas de communication directe entre composants "satellites" (il faut passer par le maître, en deux temps).
- Complexité minimale :  $\log_2(N) + 1$  fil pour  $N + 1$  composants.

## 2 - Architecture générale d'un ordinateur

### 2.1 - Les composantes d'un ordinateur

Un ordinateur classique comporte essentiellement trois types d'organes : le processeur (ou “unité centrale”), la mémoire principale et des dispositifs d'entrée-sortie.



- La mémoire principale contient les programmes exécutables et les données manipulées par ces programmes.
- Les dispositifs d'entrée-sortie permettent de communiquer avec l'extérieur (terminaux, réseaux, appareils divers) ou de conserver des informations de façon plus économique ou plus permanente qu'en mémoire centrale (mémoires de masse, mémoires spéciales). L'accès aux dispositifs externes se fait à travers des “interfaces d'entrée-sortie” qui permettent la standardisation des communications (données binaires de taille fixe : bits, octets, etc...) et la synchronisation entre le monde extérieur et l'exécution des programmes.
- Le processeur exécute les programmes : il “lit” les instructions depuis la mémoire principale, en assure le décodage, et réalise ou fait réaliser les actions correspondantes.

### 2.2 - Trois aspects du processeur

#### 2.2.1 - Interface logicielle : le modèle de programmation

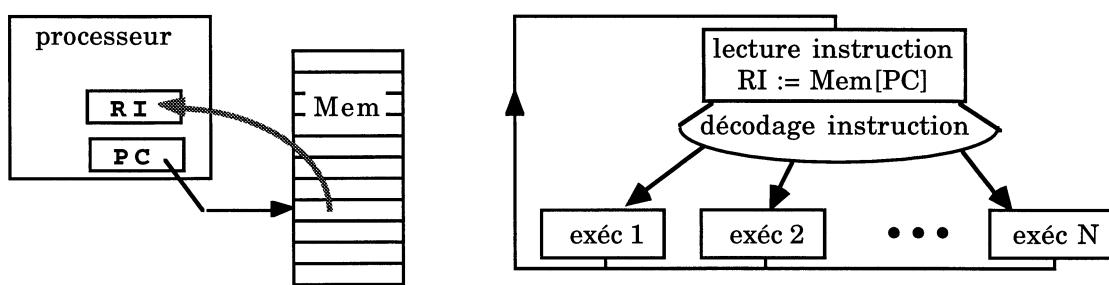
Pour le programmeur, le processeur est défini par un modèle logique :

- une collection de registres et d'opérateurs,
- un répertoire d'instructions (ou langage machine),
- les spécifications de la mémoire principale qu'on doit lui associer (taille, structure ...).

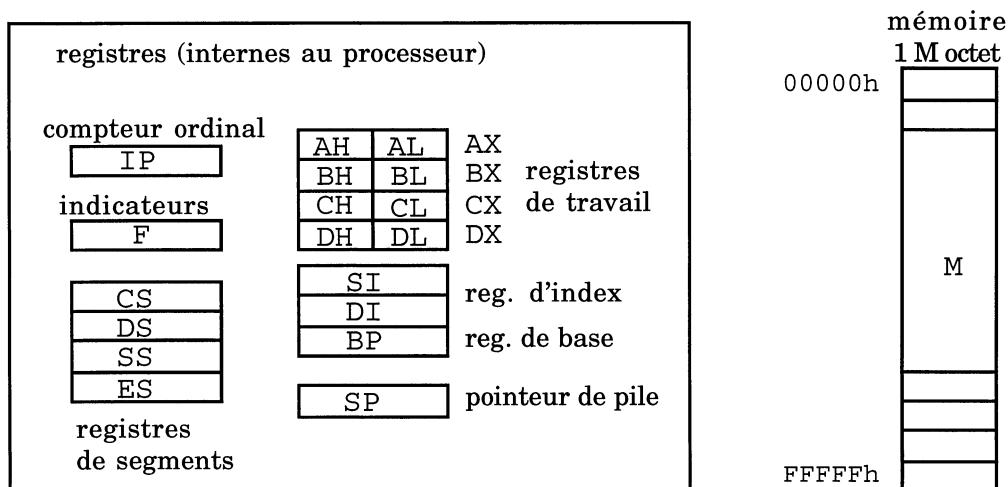
Le modèle décrit le format (codage en bits dans la mémoire) et l'effet de l'exécution de chaque instruction possible.

Le rôle principal du processeur est d'interpréter le langage machine.

Un registre “compteur ordinal” (CO, ou IP “instruction pointer”, ou PC “programm counter”) repère l'instruction à exécuter. Celle ci est placée dans le “registre instruction” (RI), où elle décodée (ce qui engendre la séquence des commandes matérielles qui la réalisent). Le compteur ordinal est alors mis à jour, selon le format de l'instruction ou l'éventuel saut engendré par l'instruction.



Exemple : le modèle logique (simplifié) du processeur i8086.



- Les registres de travail peuvent être utilisés en 8 bits ou en 16 bits. Les autres registres font 16 bits. La mémoire de 1 Moctets est adressée par 20 bits. Pour permettre l'adressage à partir de quantités 16 bits, le processeur utilise un adressage segmenté où chaque segment, limité à 64 koctets, est adressable sur 16 bits.

Le début d'un segment est indiqué dans un des registres de segment :

**CS** : segment de code, contenant le programme

**DS** : segment de donnée, contenant les données usuelles,

**SS** : segment de pile, contenant la pile des appels de procédure,

**ES** : segment auxiliaire, pour usages divers.

Le processeur génère une adresse absolue 20 bits à partir d'un registre de segment **SEG** et d'une adresse relative 16 bits **adr** selon la formule

$$\text{adresse absolue} = 16 * \text{SEG} + \text{adr}$$

Dans la suite, la notation **CS:adr** signifie l'adresse 20 bits **16\*CS+adr**.

- En ce qui concerne le répertoire d'instructions, on peut prendre pour exemple la définition de deux instructions :

<b>ADD AL, v</b>	ajoute la valeur <b>v</b> au registre <b>AL</b>
<b>MOV dep[SI], AL</b>	range <b>AL</b> à l'adresse <b>SI + dep</b>

	format	interprétation
<b>ADD AL, v</b>	<b>04</b> <b>v</b>	$\begin{aligned} AL &:= AL + M[CS:IP+1] \\ F &:= \text{indic. de } AL + M[CS:IP+1] \\ IP &:= IP + 2 \end{aligned}$

Cette instruction est codée sur 2 octets, le code opération (**04h**) suivi de la valeur **v**. Son exécution (interprétation) ajoute à **AL** la valeur **v**, située en mémoire **M[CS:IP+1]**, affecte certains indicateurs du registre **F** (**CY**, **S**, **Z**, ...) et ajoute 2 au compteur ordinal **IP** pour passer à l'instruction suivante.

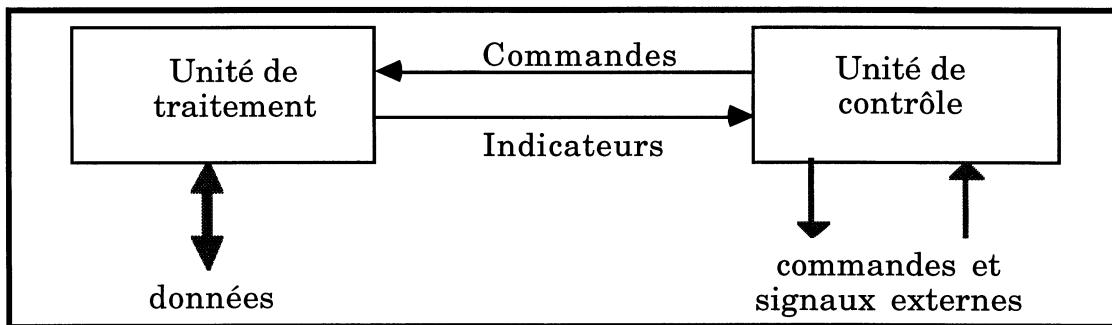
	format	interprétation
<b>MOV dep[SI], AL</b>	<b>88</b> <b>84</b> <b>dep</b>	$\begin{aligned} M[DS:SI+dep] &:= AL \\ IP &:= IP + 4 \end{aligned}$

Cette instruction est codée sur 4 octets, le code opération (**88h 84h**) suivi du déplacement **dep** sur 16 bits. Elle range **AL** en mémoire **M[DS:SI+dep]** et ajoute 4 au compteur ordinal pour passer à la suite.

Pour une description complète du répertoire d'instruction, nous renvoyons le lecteur aux manuels du langage machine du i8086.

### 2.2.2 - Un système séquentiel complexe

Fondamentalement, le processeur est un circuit séquentiel (très) complexe, qui charge et “interprète” des codes binaires qui lui sont donnés en entrée. A ce titre, il est constitué de deux unités synchrones, l’unité de traitement et l’unité de contrôle, rythmées par la même horloge.



L’unité de traitement est constituée de registres, d’opérateurs et de bus de communication internes. Elle peut communiquer avec l’extérieur par un bus de données “externes”. C’est généralement elle (ou un sous-ensemble) qui est présentée comme modèle pour la programmation (“les registres de l’UC ...”).

L’unité de contrôle est un automate qui génère des commandes vers elle-même (chargement du registre instruction, mise à jour du compteur ordinal, etc), vers l’unité de traitement, mais aussi vers les autres composants de la machine. C’est l’ensemble de ces commandes (ou plutôt des actions qu’elles provoquent) qui permet de “réaliser” les instructions (en fait, une instruction du répertoire nécessite généralement un enchaînement de commandes).

Certaines instructions sont internes à l’unité centrale (ne mettent en jeu que des registres ou circuits internes) : elles ne génèrent aucun signal sur des lignes externes.

D’autres font appel à des unités externes (mémoire ou entrée-sortie) ; l’unité de contrôle doit alors générer des commandes “externes” vers ces entités. Elle peut aussi recevoir certains signaux externes.

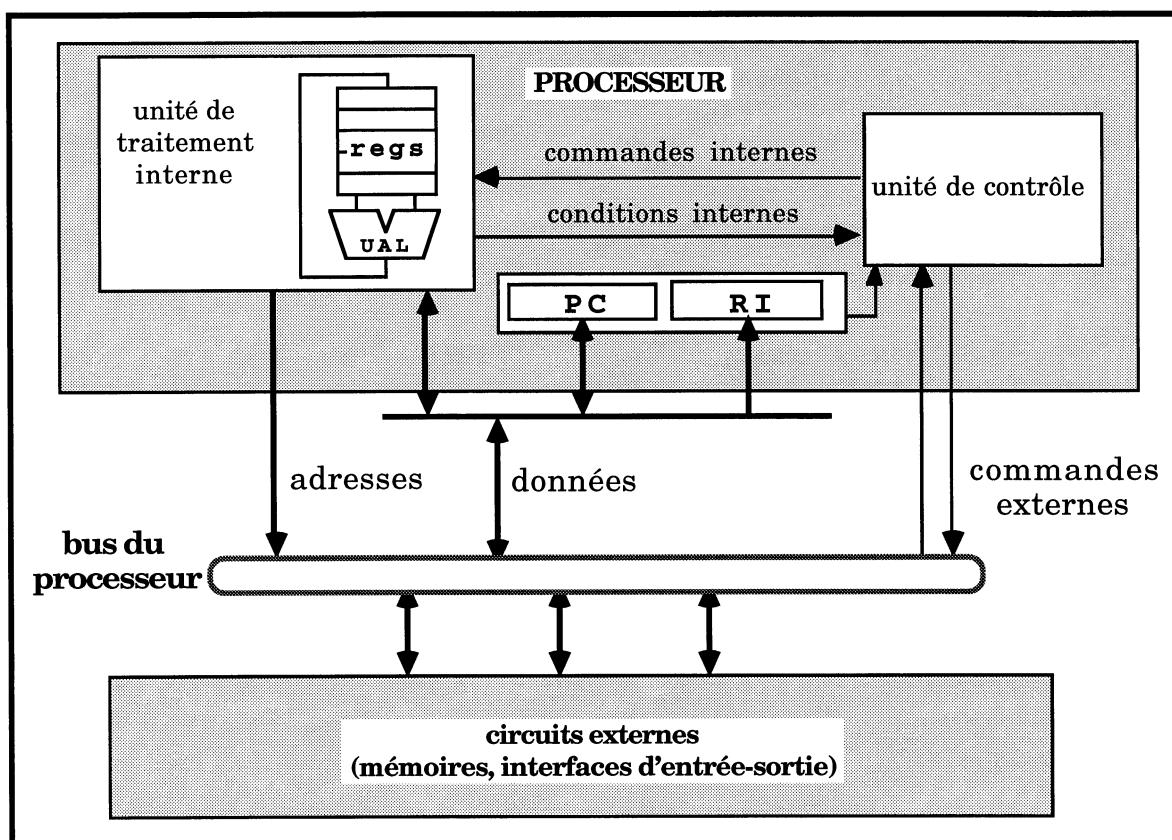
L’ensemble de tous ces signaux, qui s’échangent entre l’unité centrale et les circuits externes (bus de données, lignes de contrôle et signaux externes), est appellé le “bus d’unité centrale”, ou “bus du processeur”.

### 2.2.3 - Interface matérielle : le bus du processeur

L'échange de données entre le processeur et les circuits externes se fait sur le modèle des bus adressables à composant maître, où le processeur est le composant privilégié.

C'est également lui qui contrôle les communications entre composants : adresse (émetteur ou récepteur), sens et validation du transfert.

Le bus du processeur constitue une interface matérielle claire et homogène pour le concepteur d'un ordinateur ("l'intégrateur") : il permet de communiquer avec les organes externes au moyen de lectures et d'écritures de données à certaines adresses.



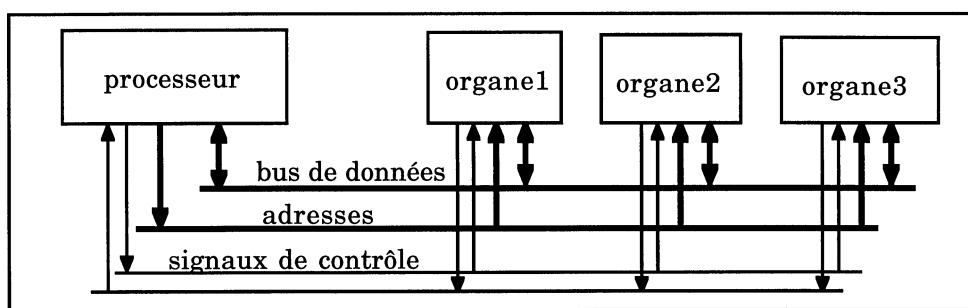
L'ajout de nouveaux organes à l'ordinateur est relativement simple et l'accès à ces organes depuis le logiciel est immédiat, grâce aux instructions de lecture et d'écriture du répertoire, sans qu'il ne soit nécessaire que ces organes aient été prévus au moment de la conception du processeur.

## 3 - Structure et usage d'un bus

### 3.1 - Structure d'un bus de processeur

Tous les organes connectés sur le bus sont désignés par des adresses. Le processeur communique avec un organe par l'écriture d'une donnée (le processeur émet) ou la lecture d'une donnée (le processeur reçoit) à une adresse qui correspond à ce composant.

Les signaux d'un bus de processeur se répartissent en trois groupes : le bus de données, les lignes d'adresses et les signaux de contrôle.



- **Le bus de données** est un bus trois-états sur lequel transitent les informations. Il est bidirectionnel vis-à-vis du processeur : pour une écriture, c'est le processeur qui force une valeur et pour une lecture c'est l'organe désigné par l'adresse. Les largeurs les plus courantes sont 8, 16 ou 32 bits. Le terme "bus de données" peut prêter à confusion : il y circule les données manipulées par les programmes (par exemple lorsque le processeur travaille sur une donnée située en mémoire) mais il y circule également les instructions du programme (pour en permettre le décodage et l'interprétation).
- **Les lignes d'adresses** : le processeur y affiche l'adresse concernée par le transfert de donnée. Ces lignes sont utilisées pour adresser tous les organes, aussi bien pour la mémoire que les interfaces d'entrée-sortie. Le nombre de bits d'adresse détermine le nombre d'éléments adressables ; les tailles usuelles des adresses sont 16 bits (64K adresses), 20 bits (1 Méga adresses), 24 bits (16 Mégas) ou 32 bits (4 Gigas).

- Les **signaux de contrôle** ; ils sont de deux sortes :

- les **commandes**, émises par le processeur, valident un transfert en indiquant sa nature (lecture, écriture ...),  
 - les **signalements externes**, issus des mémoires ou des interfaces, jouent des rôles divers : synchronisation, acquittement, demande d'interruption ...

### 3.2 - L'accès aux composants

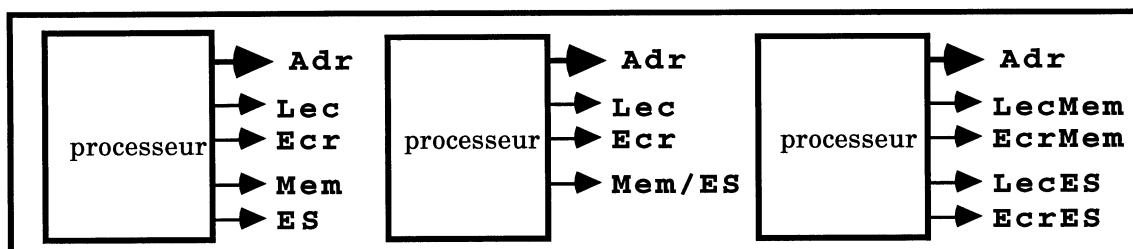
L'accès aux divers composants se fait à travers le bus du processeur : celui-ci positionne des signaux (adresses et commandes) ; le composant désigné doit répondre de façon appropriée, et en temps voulu. Les données elles-mêmes transitent par le bus de données.

#### 3.2.1 Les espaces d'adressage

Le processeur adresse des mémoires et des interfaces d'entrée-sortie. Certaines machines distinguent deux espaces d'adresses : un espace pour la mémoire et un espace pour les entrées-sorties (les "ports d'entrée-sortie").

Le répertoire d'instructions possède alors des instructions différentes pour les accès mémoire et les accès aux ports d'entrée-sortie.

Au niveau matériel, cela se concrétise par des signaux permettant de distinguer les accès à l'un ou l'autre des espaces.



Pour d'autres machines, l'adressage est banalisé : entrées-sorties et mémoire sont dans le même espace. Tous les accès se font avec les mêmes instructions, appelées **LOAD**, **MOVE**, **STORE** ... Dans ce cas, le concepteur de l'ordinateur (l'intégrateur) réserve certaines adresses à des ports d'entrée-sortie, les autres étant disponibles pour la mémoire principale.

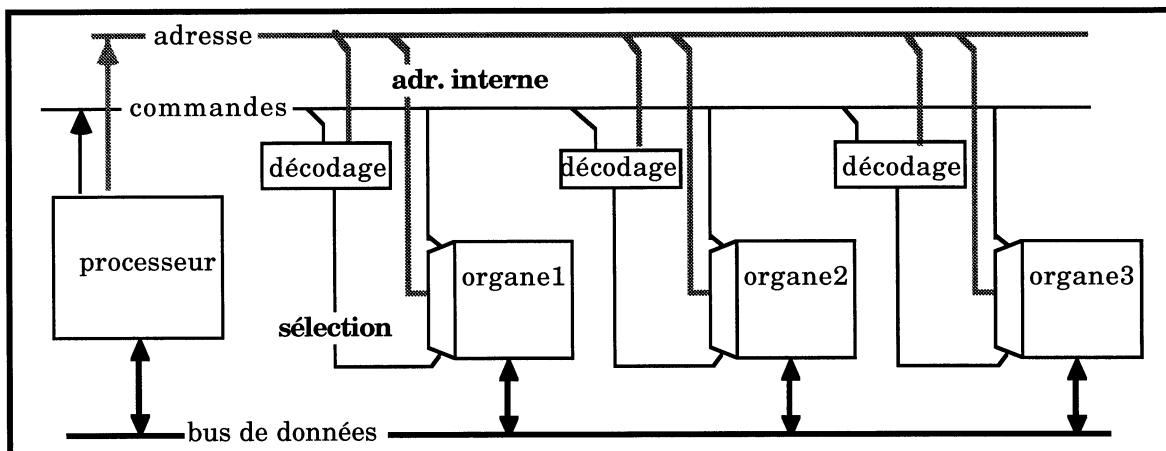
La différence entre ces deux techniques n'est pas fondamentale. L'adressage séparé permet un décodage plus facile au niveau des entrées-sorties. La tendance actuelle est pourtant plutôt pour l'espace banalisé.

### 3.2.2 L'implantation des composants

Les composants comportent souvent plusieurs éléments (éventuellement des millions : mémoires ...), chacun étant désigné par son adresse. L'adresse est constituée de deux parties : l'une spécifie le composant ("adresse du composant"), l'autre l'élément dans le composant ("adresse interne").

A chaque composant est associé un circuit de décodage d'adresse, qui détecte que les commandes signalent un accès, et que l'adresse appartient au domaine attribué au composant. L'implantation d'un organe est donc déterminée par le circuit de décodage d'adresse qui lui est associé.

Certains bits d'adresse (généralement les poids faibles), ainsi que des commandes, sont envoyés directement sur le composant, où ils servent à désigner l'élément et l'action correspondant à l'accès.



On fait en sorte que les adresses des éléments d'un composant soient contigües. Le domaine des adresses du composant est alors caractérisé par l'adresse de début et la taille du composant.

#### Placement “bien aligné”

Le plus souvent, la taille du composant est une puissance de 2, et l'adresse de début est un multiple de la taille. Un tel placement est dit “bien aligné”. Sa sélection et son adressage interne sont très simples.

Pour  $n$  bits d'adresse et un composant de taille  $2^k$  implanté à partir de  $D \times 2^k$  :

- la sélection est effectuée par un monôme qui teste l'égalité entre les  $n-k$  bits d'adresse de poids fort et  $D$ .
- L'adressage interne est réalisé directement par les  $k$  bits de poids faible.

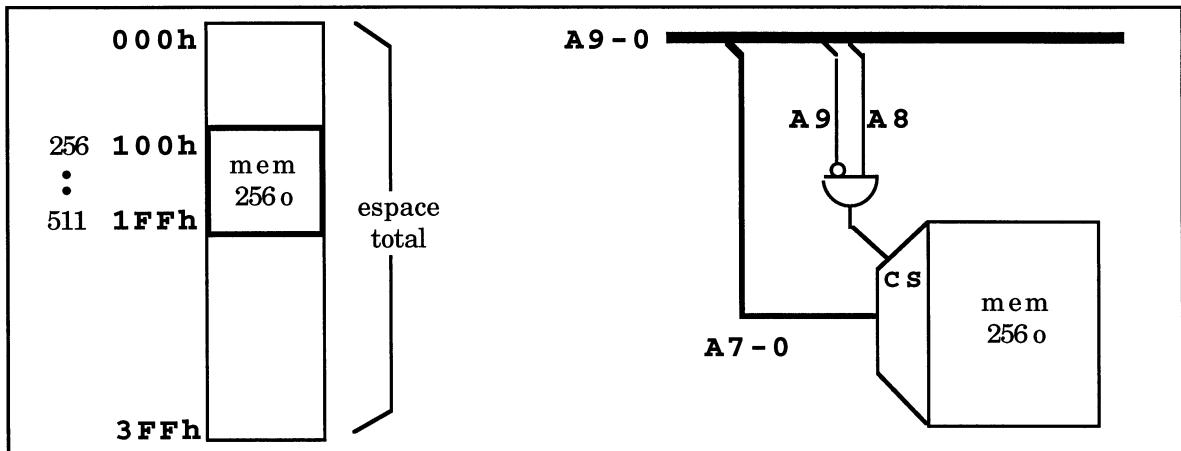
$$\text{sélection} = (A_{n-1} \dots A_k = D) \quad \text{adr. interne} = A_{k-1} \dots A_0$$

Ceci provient du fait que le domaine d'adresses occupé par cet organe est l'intervalle  $[D00..0, D11..1]$ . Il est donc constitué des adresses de la forme  $<D\,XX..X>$ , où  $XX..X$  représente n'importe quelle combinaison de  $k$  bits.

Considérons par exemple une mémoire de 256 mots (8 bits d'adresse interne) et un processeur avec 10 bits d'adresse. On désire placer cette mémoire à partir de l'adresse 256.

Ce placement est bien aligné : on sélectionne la mémoire par le monôme :

$$CS = (A_9 A_8 = 01) = /A_9 \cdot A_8$$



### Placement “mal aligné”

Si le placement n'est pas bien aligné, la sélection (la fonction booléenne qui indique l'appartenance au domaine attribué) est plus complexe. C'est l'union de plusieurs monômes détectant l'appartenance à des sous-domaines.

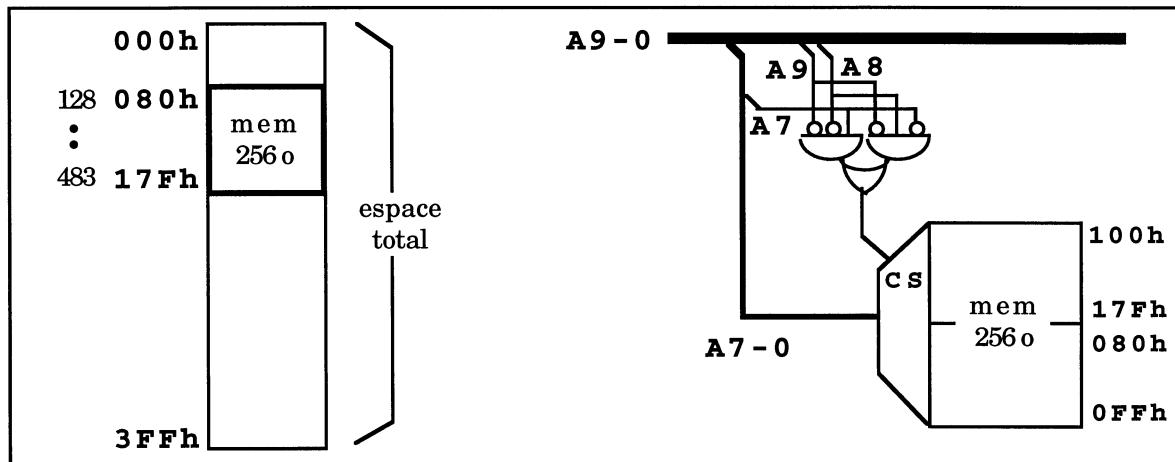
Pour une mémoire vive de taille  $2^k$ , l'adressage interne peut encore se faire directement par les  $k$  bits d'adresse de poids faibles. On ne respecte pas l'ordre des adresses (deux adresses internes successives ne correspondent pas à des adresses externes successives). Cela n'a pas d'importance dans le cas d'une mémoire vive, car tous les mots sont équivalents.

En revanche, il faut se méfier de l'adressage interne si l'ordre des adresses est important (c'est le cas pour des mémoires mortes, dont le contenu est défini par ailleurs, ou pour des organes d'entrée-sortie, dont les adresses internes ont chacune un rôle spécifique).

Ainsi, pour placer la mémoire précédente à l'adresse 128 (**080h**), on peut procéder comme suit :

Le domaine d'adresse est l'intervalle **[080h, 17Fh]**. On peut le décomposer en deux parties : **[080h, OFFh]** et **[100h, 17Fh]**. Ces deux intervalles sont caractérisés par **A<sub>9</sub>A<sub>8</sub>A<sub>7</sub> = 001** et **A<sub>9</sub>A<sub>8</sub>A<sub>7</sub> = 010**.

La formule de sélection est alors :  $CS = /A_9 \cdot /A_8 \cdot A_7 + /A_9 \cdot A_8 \cdot /A_7$



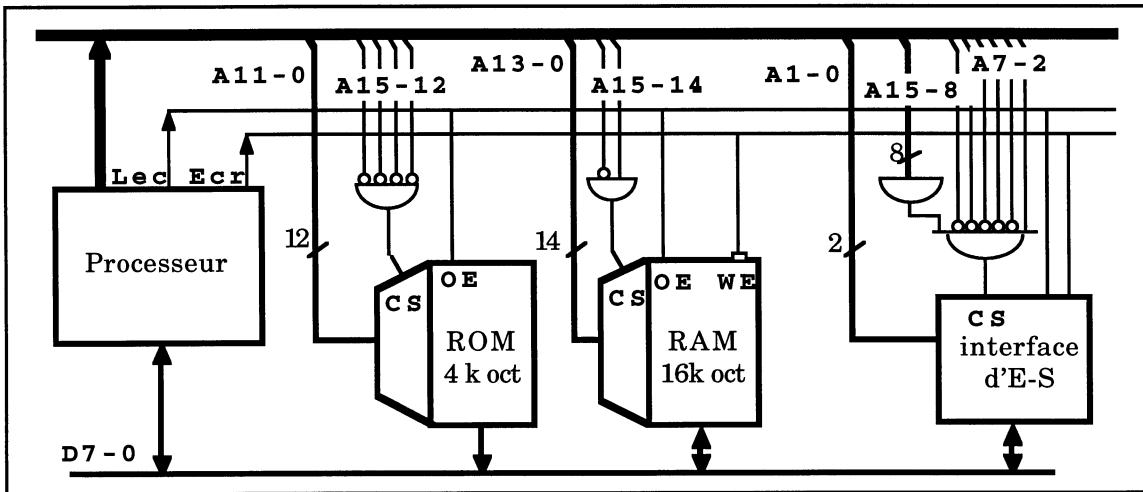
L'adresse interne peut être prise égale à **A7 - 0** : le début (au sens des adresses internes) de la mémoire correspond aux adresses **[100h, 17Fh]** et la fin aux adresses **[080h, OFFh]**, mais c'est sans importance.

### Exemple d'une configuration complète

Nous considérons un processeur avec un bus de données de 8 bits, 16 bits d'adresse, et des commandes **Lec** (lecture) et **Ecr** (écriture).

- La mémoire principale est constituée de deux blocs séparés :
  - un bloc de mémoire morte de 4k octets, implanté de **0000h à OFFFh**,
  - un bloc de mémoire vive de 16k octets, de **4000h à 7FFFh**.
- Les entrées-sorties sont assurées par un circuit d'interface qui occupe quatre adresses : **FF04h, FF05h, FF06h, FF07h**.

Les placements sont bien alignés, et les poids faibles servent directement d'adresse interne aux blocs. Les sélections sont réalisées par le décodage des poids forts de l'adresse : **A15-12=0000** pour la ROM, **A15-14=01** pour la RAM, **A15-2=11111111000001** pour l'interface.



### Remarques

- Les attributions d'adresses doivent assurer qu'au plus un composant est validé à un instant donné.
- On peut parfois simplifier le décodage en ignorant certains bits d'adresse dans le calcul de la validation ; on le paye par un certain gaspillage d'adresses. Ici, la ROM pourrait être sélectionnée par les seuls bits **A15A12=00**. Elle serait vue comme "dupliquée" aux adresses **0000h, 1000h, 2000h, et 3000h**.

### 3.3 - Les cycles bus

Le processeur est un système séquentiel très complexe, et l'on pourrait s'attendre à ce que ses sorties (les signaux qu'il génère) soient difficilement prévisibles. Il n'en est (presque) rien ; en fait, l'interaction du processeur avec son environnement (via son bus) se fait selon quelques schémas temporels bien définis, les "cycles bus".

L'activité de la machine (l'exécution des instructions) se ramène à un enchaînement de ces cycles, qui activent les différents signaux du bus.

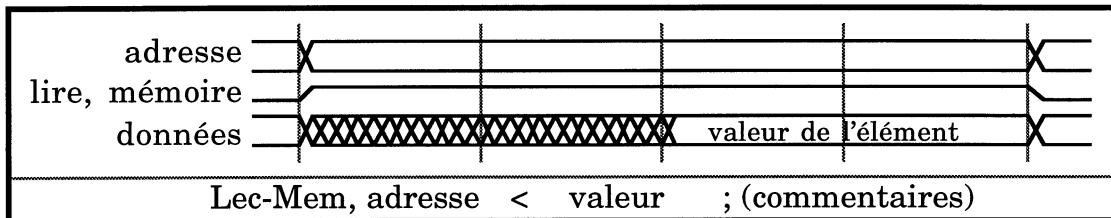
Chacun de ces cycles dure plusieurs cycles de l'horloge de base du processeur (couramment trois ou quatre cycles au moins, qui peuvent se prolonger selon les mécanismes d'attente ou d'acquittement vus au dessus).

#### 3.3.1 - Les cycles bus courants

Ils sont spécifiés par des diagrammes temporels ; on les note parfois par une formule plus "compacte" (une simple notation exprimant le diagramme). Les principaux concernent l'accès mémoire ou entrée-sortie.

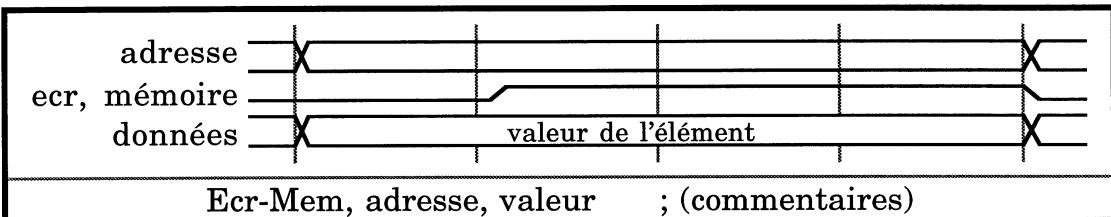
### • Lecture mémoire

Le processeur place l'adresse du mot sur les lignes d'adresses, et active les signaux correspondant à la lecture mémoire. La valeur est placée sur le bus de données par le composant sélectionné ; elle est prise en compte par l'UC.



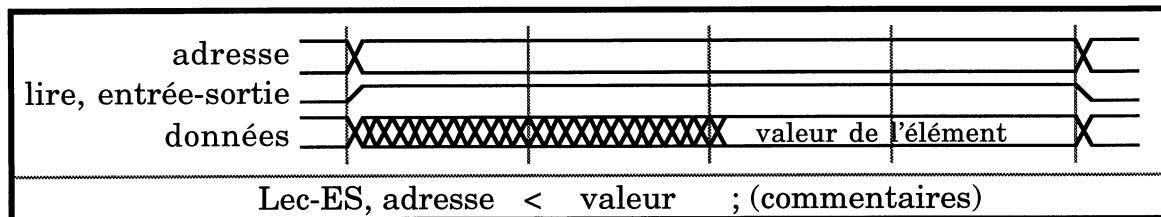
### • Ecriture mémoire

Le processeur place l'adresse du mot concerné sur les lignes d'adresses et la valeur à inscrire, puis active les signaux d'accès mémoire et d'écriture.



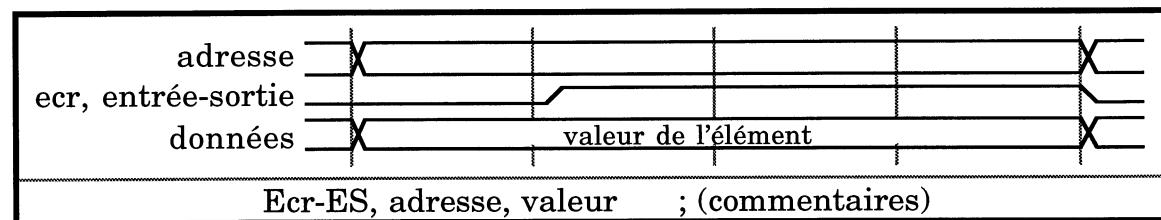
### • Lecture entrée-sortie

Analogue à la lecture mémoire ; l'accès mémoire est remplacé par l'accès entrée-sortie.



### • Ecriture entrée-sortie

Analogue à l'écriture mémoire ; l'accès mémoire est remplacé par l'accès entrée-sortie.



### 3.3.2 - L'exécution des instructions

L'exécution d'une instruction n'est qu'une suite de cycles bus : l'automate du processeur génère ces cycles (vers l'extérieur) et les signaux vers l'unité de traitement qui permettent de les utiliser.

Chaque instruction nécessite un ou plusieurs cycles bus (au moins un pour charger le code instruction, et un pour chaque accès hors du processeur).

Exemples :

- Le cycle de recherche du premier octet de l'instruction courante :  
Lire-Mem, <CO> < code instr. > ; le CO est placé sur les lignes d'adresse ; la valeur obtenue est chargée dans RI
- Ecrire le contenu d'un registre en mémoire :  
Ecr-Mem, < adresse du mot >, < valeur du registre >
- Empiler un registre double (mém. segmentée avec *stack seg* et *stack ptr*) :  
Ecr-Mem, < SS : SP >, < poids faibles >  
Ecr-Mem, < SS : SP - 1 >, < poids forts > ; SP:= SP-2
- Charger un registre dans un autre :  
ne génère aucun cycle d'accès bus (action interne à l'UC).

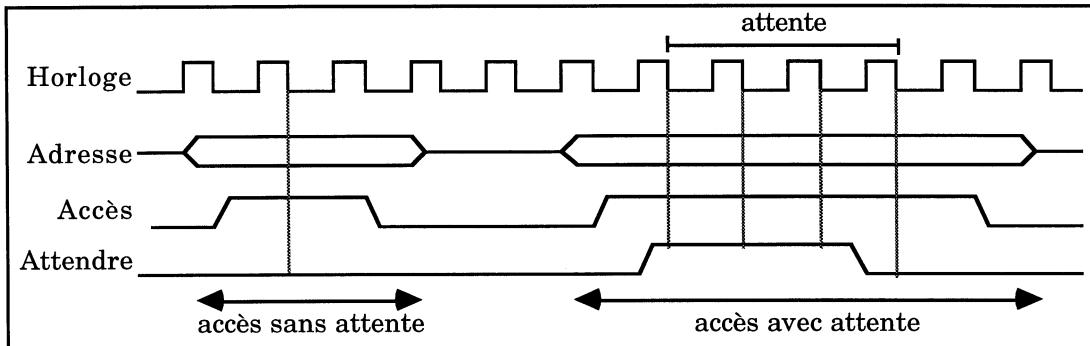
### 3.3.3 - Synchronisation des accès

Pour réaliser un accès, le processeur accorde un certain délai au composant (un ou quelques cycles de son horloge). La plupart des composants, les mémoires rapides notamment, ont des temps d'accès sensiblement de cet ordre. Ce n'est pas toujours le cas, et si le temps d'accès est trop long, il faut faire attendre le processeur.

Deux techniques sont utilisées : le mécanisme de “suspension” (mise en attente du processeur), où les accès ont une durée fixée à priori, éventuellement prolongeable, et le mécanisme d'acquittement, où l'accès se termine sur réponse explicite du composant, avec éventuellement une limite de temps.

#### Mécanisme de mise en attente

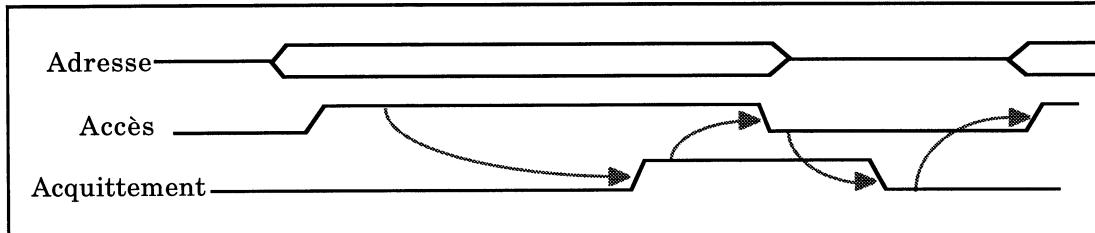
Le composant dispose d'un temps fixé pour réaliser l'accès. S'il ne peut réagir à temps, il envoie, dès le début de l'accès, un signal au processeur. Tant que ce signal est actif, celui ci réalise des “cycles d'attente”, en prolongeant tous ses signaux. Cette méthode est très utilisée, car simple et performante.



### Mécanisme d'acquittement explicite

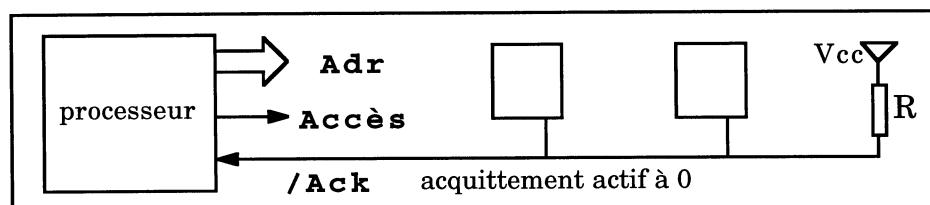
A chaque accès, le processeur attend systématiquement (en maintenant ses signaux) un acquittement provenant du composant.

La demande d'accès et l'acquittement suivent un protocole classique de demande-réponse entrelacées : le processeur lance les commandes pour faire l'accès, et les maintient ; le composant active l'acquittement dès qu'il perçoit la demande d'accès (il le maintient tant que dure l'accès) ; le processeur relâche alors les commandes d'accès, et attend que l'acquittement soit désactivé avant de faire tout autre accès.



### Remarque

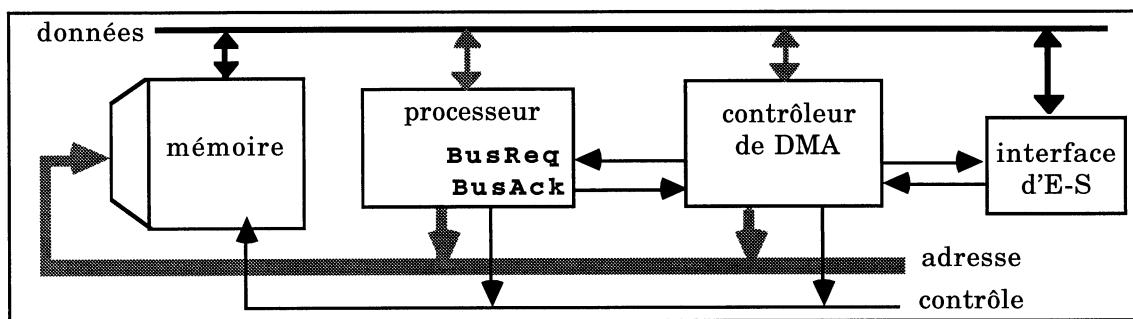
Plusieurs composants sont connectés sur le bus, et chacun doit pouvoir acquitter (ou suspendre) l'accès. Pour des raisons de modularité, on utilise généralement une ligne unique à collecteur ouvert : les signaux sont actifs à zéro, et une résistance de rappel maintient la ligne à "1" en l'absence de signalement (c'est ce qu'on appelle un "OU cablé").



### 3.4 - Autres signaux des processeurs : DMA et interruptions

#### 3.4.1 - Partage du contrôle du bus : DMA

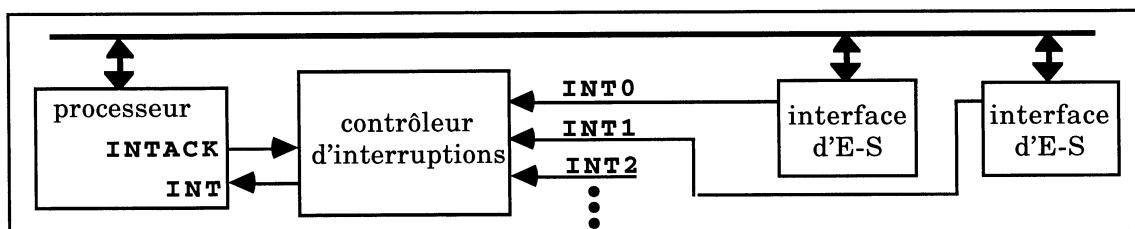
Pour les transferts à grand débit entre la mémoire et certains périphériques (disques, réseaux ...), les processeurs peuvent confier temporairement le contrôle du bus à d'autres composants. Ce mécanisme d'accès direct mémoire (DMA : direct memory access) est assuré par un contrôleur de DMA.



Le contrôleur demande l'accès au bus par un signal **BusReq**, et le processeur lui accorde par un signal **BusAck**. Le contrôleur conserve le bus tant qu'il maintient sa demande **BusReq**. Le contrôleur de DMA communique directement avec l'interface d'entrée-sortie et utilise le bus pour adresser la zone mémoire concernée par le transfert.

#### 3.4.2 - Demandes d'interruptions

Les interruptions sont un mécanisme qui permet de répondre rapidement à des sollicitations externes. Le processeur reçoit des demandes d'interruptions de la part des interfaces d'entrée-sortie et ces signaux provoquent l'exécution de programmes spécifiques destinés à répondre aux événements externes.



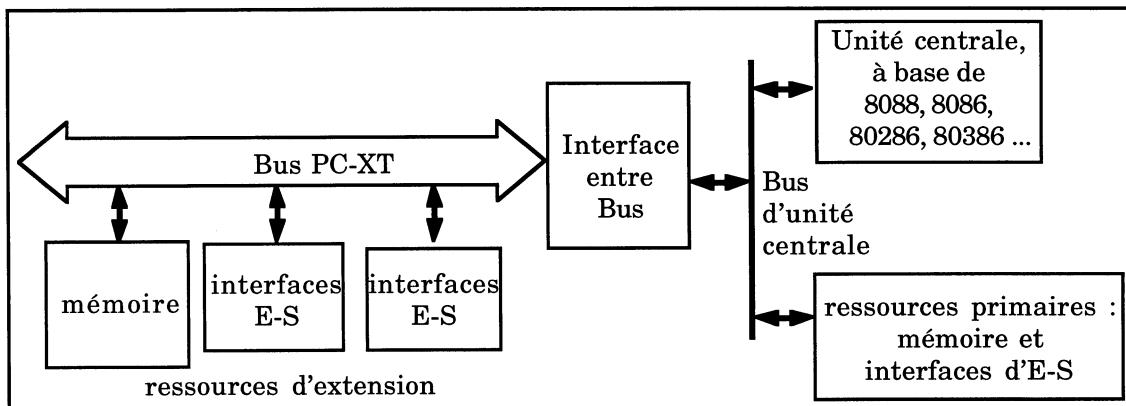
Le processeur offre généralement une seule entrée d'interruption **INT** et elle signale la prise en compte d'une interruption par un signal d'acquittement **INTACK**. Un composant spécial, le contrôleur d'interruption, permet de partager cette entrée de demande unique entre plusieurs sources de demandes.

## 4 - Exemples

### 4.1 - Le Bus PC-XT

Le bus PC-XT n'est pas à proprement parler le bus d'un processeur particulier. C'est un bus d'extension standardisé pour les ordinateurs PC, qui peuvent être dotés de diverses unités centrales du constructeur intel (8088, 8086, 80286, 80386, 80486, etc).

Diverses ressources, dites "primaires" peuvent être directement connectées à l'unité centrale sans passer par ce bus. Evidemment liées à un modèle particulier de processeur, elles peuvent en tirer partie au mieux (optimisation des performances).



Un circuit spécial interface le bus de l'unité centrale et le bus XT qui permet d'ajouter des ressources, mémoires ou interfaces d'entrée-sortie, sous forme de cartes d'extension, reliées au bus par des connecteurs standardisés.

Les spécifications du bus décrivent précisément ses caractéristiques mécaniques (format du connecteur, numérotation des broches ...), logiques (nature et comportement des signaux) et temporelles. Physiquement, les signaux sont distribués sur un connecteur à soixante deux broches.

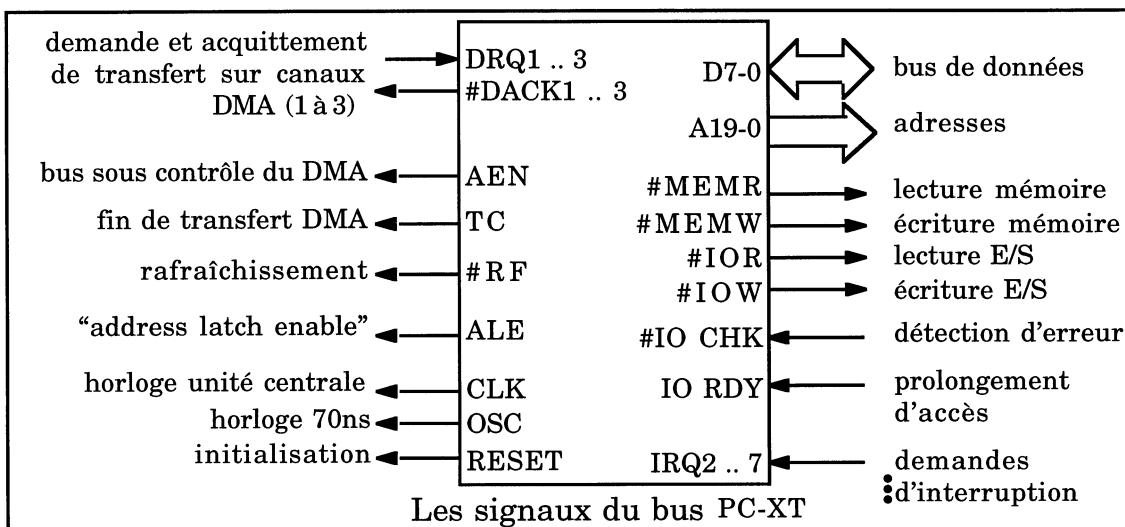
Pour l'intégrateur, ce bus sert de référence. Il peut être considéré comme un vrai bus d'UC.

Le bus PC-XT est peu performant : données de 8 bits, adresses sur 20 bits, cycles d'accès lents (plusieurs centaines de ns). Malgré sa rusticité, il continue à être maintenu sur les PC car un grand nombre de cartes d'interfaces ont été développées pour ce bus.

De nombreux PC sont équipés d'une version élargie et compatible de ce bus, le bus AT, ou ISA, avec 16 bits de données et 24 bits d'adresse.

Les PC modernes (386, 486 et au-delà) disposent souvent de bus d'extension plus performants permettant de profiter de leur puissance. Ces bus sont encore plus ou moins compatibles (donc moins performants : bus EISA), ou au contraire plus performants et non compatibles (bus MCA). Ils sont souvent doublés d'un autre bus plus "local", pour des accès très rapides (Vesa-Local bus, bus PCI ...).

Le bus PC-XT comprend un bus de données (8 bits), des lignes d'adresses (20 bits) et un certain nombre de signaux de contrôle et de service.



### Les données et les adresses

Le bus de données (bidirectionnel) fait huit bits (D7..0), les lignes d'adresses vingt bits (A19..0). L'adressage de la mémoire et des ports d'entrée-sortie sont distingués par des commandes séparées de lecture et d'écriture (actives à "0").

#MEMR ("Memory Read"), #MEMW ("Memory Write") : accès mémoire.

#IOR ("Input Output Read"), #IOW ("Input Output Write") : entrées/sorties.

Une adresse mémoire occupe les vingt lignes A19-A0 : l'espace mémoire est de taille maximale 1M-octets.

Au niveau du i8086, une adresse de port (entrée-sortie) fait seize bits, présentés sur les lignes A15..0. Sur le bus XT, seuls les accès dans l'espace [0200h .. 03FFh] sont transmis.

Le répertoire d'instructions du 8086 (et successeurs) distingue les accès mémoire (“MOV ...”) des accès entrée-sortie (“IN ...” ou “OUT ...”). Le processeur génère donc spécifiquement les signaux #MEMR, #MEMW ou #IOR, #IOW.

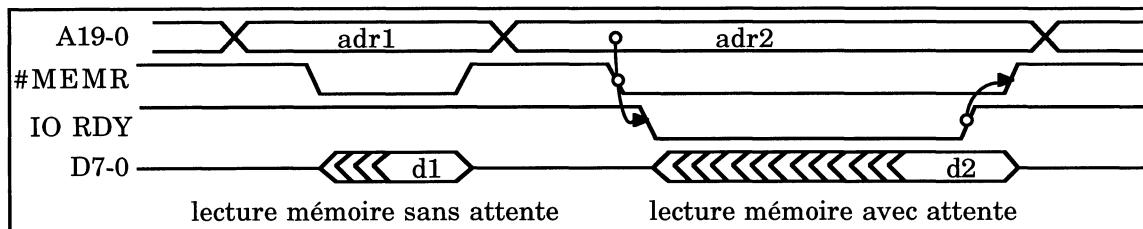
Les instructions d'entrée-sortie, notées IN et OUT, font intervenir le registre AL, l'adresse du port étant dans DX.

IN AL, DX : rangement dans AL de l'octet lu depuis le port indiqué par DX.

OUT DX,AL : écriture du contenu de AL vers le port indiqué par DX.

Elles existent aussi en version 16 bits (deux cycles bus : IN AX,DX, OUT AX,DX), et en version réduite “IN/OUT AL/AX, adrport”, l'adresse étant limitée à 255.

L'entrée IO RDY permet de prolonger les accès ; le composant adressé dispose de quelques nanosecondes (environ 20ns) après la commande de lecture ou d'écriture pour l'activer si nécessaire.



Ce signal ne doit pas être maintenu à “0” plus de quelques microsecondes, car le rafraîchissement des mémoires dynamiques n'est pas effectué pendant l'attente, et leur contenu risque donc d'être perdu .

### Le contrôle du DMA

Les ressources primaires du PC comprennent un contrôleur de DMA. Il offre trois canaux indépendants qui communiquent avec des interfaces d'entrée-sortie par les signaux DRQ1..3 (“Data Request”) et DACK1..3 (“Data Acknowledge”) : un interface indique sur DRQi qu'il est prêt à transférer un octet, et le contrôleur de DMA valide le transfert en lui répondant sur DACKi.

Le signal AEN (“Address enable”) indique que le bus est sous contrôle du DMA (et non de l'unité centrale). Ce signal est nécessaire, car le contrôleur active le signal #IOR ou #IOW pour indiquer le sens du transfert à l'interface (mais l'adresse sur A15-0 est celle de l'octet mémoire avec lequel se fait le transfert) : si AEN est actif, les interfaces d'entrée-sortie doivent ignorer l'adressage.

Le signal TC (“Terminal Count”), issu du contrôleur de DMA, indique la fin d'un transfert sur un canal quelconque.

## Les autres signaux

Le bus PC-XT offre six lignes de demande d'interruption, IRQ2-7. Ce sont des entrées pour un circuit contrôleur d'interruptions (PIC 8259, faisant partie des ressources primaires) qui en offre huit au total (les entrées IRQ0 et IRQ1 sont utilisées par des ressources primaires).

Le signal ALE (“address latch enable”) est activé en même temps que la présentation d'une adresse : il est peu utile dans la pratique.

Le signal d'erreur #IO CHK (“IO Check”) permet aux mémoires et aux interfaces de signaler des erreurs irrécupérables. Ce signal provoque une interruption non masquable sur l'unité centrale.

Le signal RESET permet l'initialisation des circuits raccordés sur le bus. Ce signal est activé lors d'un “reset” matériel du PC.

Le signal #RF (“refresh”) indique un cycle de rafraîchissement pour les mémoires dynamiques. Ce signal est émis périodiquement, toutes les 15 microsecondes environ, accompagné sur A15-0 d'une adresse incrémentée cycliquement. Sur la plupart des PC-XT, les adresses de rafraîchissement et le signal #Refresh sont générés par le canal 0 du contrôleur de DMA.

CLK est l'horloge de l'unité centrale. Ce signal est rarement utilisé.

OSC est une horloge de période 70ns, destinée à servir de base pour la mesure du temps réel.

Diverses alimentations électriques sont également disponibles sur le connecteur XT ( $\pm 5V$ ,  $\pm 12V$  et la masse).

## Les signaux du bus AT

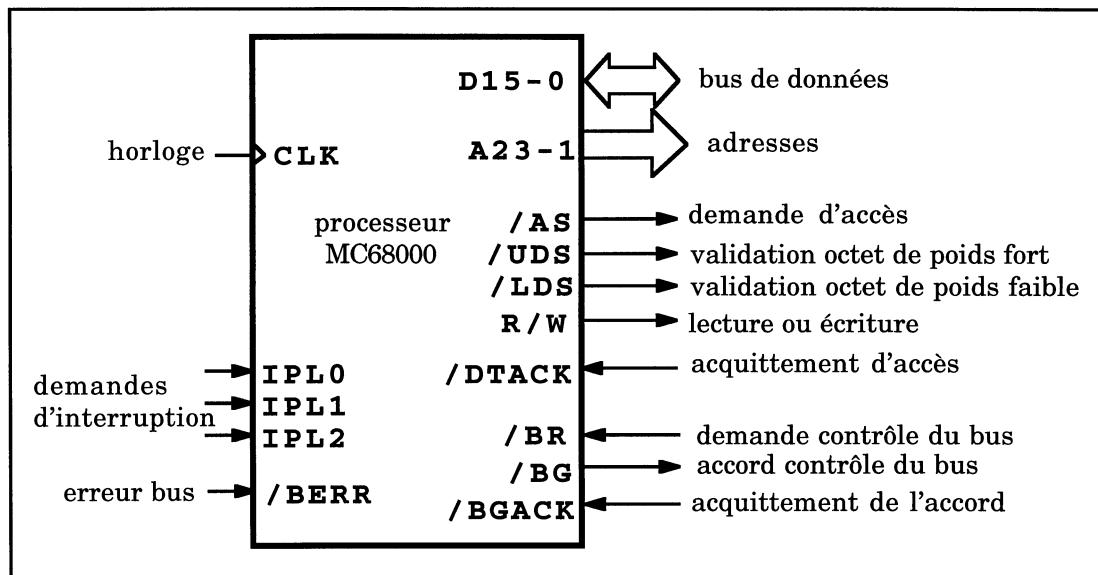
Le bus du PC-AT a été conçu pour tenir compte des possibilités du i80286 tout en restant compatible au maximum avec le bus XT. Il contient à peu près les mêmes signaux, accessibles aux mêmes endroits sur le même connecteur, plus d'autres signaux, sur un connecteur 36 points spécifique du bus AT.

Les principales différences portent sur le bus de données (seize bits) et les lignes d'adresse (vingt quatre bits : espace d'adressage de 16 Moctets). Les signaux associés au DMA sont plus nombreux, de même que les lignes de demande d'interruption.

Quelques autres signaux permettent des accès optimisés un peu exotiques ...

## 4.2 - Le MC68000

Le MC68000 possède un répertoire d'instructions 32 bits. Ses registres font 32 bits et les données manipulées 8, 16 ou 32 bits. Cependant, le bus de données a une largeur de 16 bits seulement, **D15-0**.



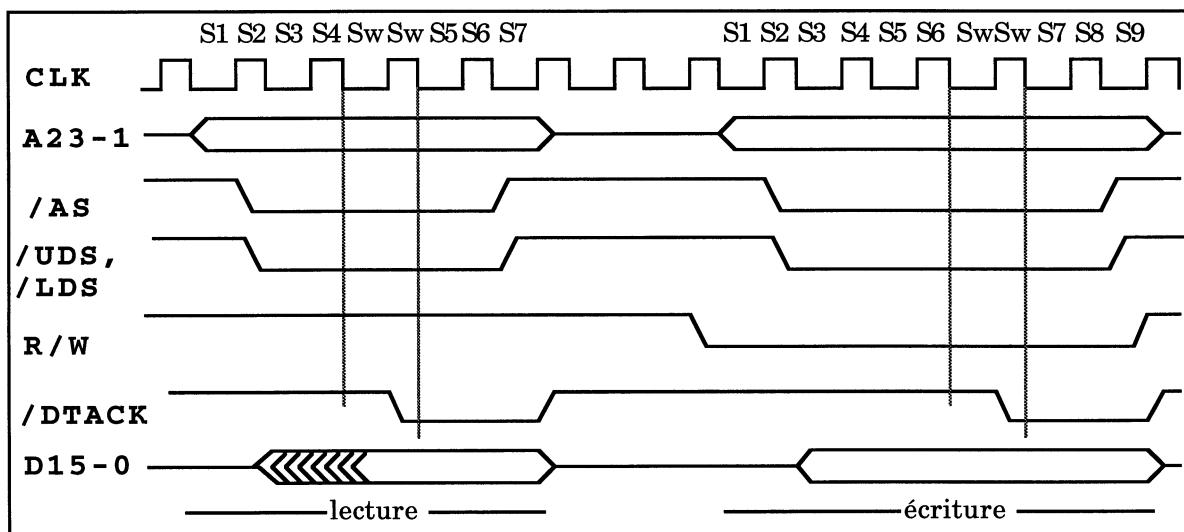
L'espace d'adressage est le même pour la mémoire et les entrées-sorties. Les transferts sont commandés par un signal de demande d'accès **/AS** et une commande **R/W** qui indique s'il s'agit d'une lecture ou d'une écriture.

Le répertoire d'instructions dispose de 24 bits d'adresse **A23-0**, permettant d'adresser 16 Méga octets. Il n'y a pas de ligne physique pour le bit de poids faible d'adresse **A0**. Ce bit est remplacé par deux signaux, **/UDS** (upper data strobe) et **/LDS** (lower data strobe) :

**/LDS** commande le transfert d'un octet d'adresse paire sur **D7-0**,  
**/UDS** commande le transfert d'un octet d'adresse impaire sur **D15-8**,  
les deux signaux **/LDS** et **/UDS** actifs simultanément indiquent un accès 16 bits sur **D15-D0**, l'adresse étant obligatoirement paire dans ce cas.

Le MC68000 utilise un mécanisme d'acquittement explicite des accès, par le signal **/DTACK**.

Le chronogramme ci-dessous illustre le déroulement des accès. L'évolution des signaux est indiquée relativement aux phases d'horloge S1 (phase haute) et S2 (phase basse). Chaque phase dure environ 50ns. L'acquittement est testé en fin de phase S4 pour une lecture, et en fin de phase 6 pour une écriture. Si l'acquittement n'est pas présent à ce moment là, le processeur insère des cycles d'attente.



# Exercices

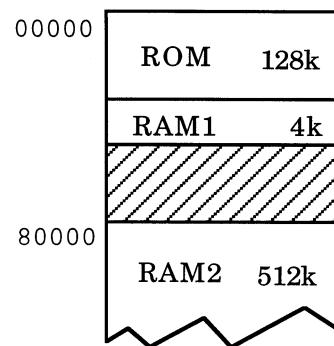
## Exercice 1

Un calculateur possède un bus de données de 8 bits **D7-0**, 20 bits d'adresse **A19-0**, et deux signaux de commande **Ecr** (écriture) et **Lec** (lecture). On désire y connecter les blocs de mémoire suivants :

ROM : mémoire morte de 128k octets,  
à partir de l'adresse 00000h,

RAM1 : mémoire vive de 4k octets,  
à la suite de ROM,

RAM2 : mémoire vive de 512k octets,  
à partir de l'adresse 80000h.



- 1 - Indiquer le nombre de bits d'adresse interne de chacun de ces blocs.
- 2 - Indiquer pour chaque bloc son domaine d'adresses occupé et sa formule de sélection.
- 3 - Dessiner le schéma de connexion des blocs mémoire au bus.
- 4 - On décide de valider RAM1 en n'utilisant que 4 bits d'adresse. Donner une formule de sélection possible. Indiquer combien d'adresses différentes permettent d'accéder à un même mot de RAM1.

Le calculateur dispose des instructions :

**MOV (adr), v** range l'octet **v** à l'adresse **adr**,

**MOV R, (adr)** range l'octet d'adresse **adr** dans le registre **R**.

Expliquer l'effet de la séquence d'instructions suivante :

**MOV (20154h), 35h**

**MOV R1, (25154h)**

## Exercice 2

On désire réaliser une mémoire pour PC, comportant 16k de ROM à partir de l'adresse **A0000h** et 112k octets de RAM à la suite. On dispose de boitiers de ROM de 16k octets, et de boitiers de RAM de 64k octets.

- 1 - Donner le schéma de cette mémoire (les composants utilisés sont suffisamment rapides pour ne pas mettre le processeur en attente).

- 2-** On suppose que la mémoire ROM a un temps d'accès plus long, qui nécessite une mise en attente pendant au moins 250 ns. Donner une solution pour ce problème.

### Exercice 3     Cycles bus engendrés par l'exécution des instructions

Cet exercice fait le lien entre l'exécution d'un programme et ce qui se passe au niveau des signaux du processeur. Nous notons comme suit les quatre sortes d'accès :

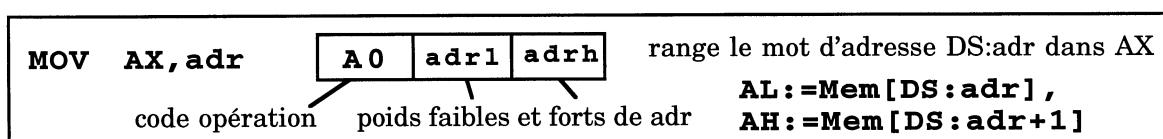
notation :

lecture mémoire	<b>MEMR,</b> <b>adresse</b> > <b>octet-lu</b>
écriture mémoire	<b>MEMW,</b> <b>adresse</b> < <b>octet-écrit</b>
lecture port d'E-S	<b>IOR,</b> <b>adr-port</b> > <b>octet-lu</b>
écriture port d'E-S	<b>IOW,</b> <b>adr-port</b> < <b>octet-écrit</b>

L'exercice consiste à indiquer la séquence des accès bus engendrée par l'exécution des instructions. On considère ici que tous les accès se font à travers le bus du PC (en fait ce n'est le cas que pour les accès aux ressources connectées sur ce bus). On suppose, pour rester simple, que l'unité centrale lit les instructions au fur et à mesure de leur exécution (en réalité, le processeur lit en avance, dans un tampon, les instructions situées à la suite de l'emplacement pointé par le compteur ordinal).

Dans tout ce qui suit on suppose que les registres de segment contiennent les valeurs suivantes : **CS=A000h**, **DS=B000h** et **SS=C000h**.

Considérons par exemple l'instruction :



On suppose que :

- **adr** vaut **2587h**,
- le mot d'adresse **B2587h** contient **5566h**
- l'instruction est placée à l'adresse **A0100h** et **IP** contient **0100h**.

Les accès bus engendrés par l'exécution de cette instruction sont :

(accès bus) (opérations internes au processeur)

<b>MEMR, A0100 &gt; A0</b>	
<b>MEMR, A0101 &gt; 87</b>	
<b>MEMR, A0102 &gt; 25</b>	(PC+:= 3)
<b>MEMR, B2587 &gt; 66</b>	AL:= 66
<b>MEMR, B2788 &gt; 55</b>	AH:= 55

Nous indiquons également dans la trace les affectations de registres réalisées à l'intérieur du processeur, en même temps ou après chaque accès.

Voici la description de quelques instructions du 8086 :

<b>MOV SI,vv</b>	<b>B6 vv1 vvh</b>	range la valeur vv dans SI SI:=vv
<b>MOV AL,v</b>	<b>B0 v</b>	range la valeur v dans AL AL:=v
<b>CMP AL,v</b>	<b>3C v</b>	compare AL et la valeur v S,Z,CY:=indicateurs de (AL-v)
<b>AND AL,v</b>	<b>24 v</b>	ET bit à bit entre AL et v AL := AL ET v
<b>MOV dd[SI],AL</b>	<b>88 84 dd1 ddh</b>	range AL à l'adresse (SI)+dd Mem[DS:(SI)+dd] := AL
<b>JMP d</b>	<b>EB d</b>	saut avec déplacement d PC:=PC+2+d
<b>JZ d</b>	<b>74 d</b>	saut si Z avec déplacement d si Z alors PC:=PC+2+d sinon PC:=PC+2
<b>CALL dd</b>	<b>E8 dd1 ddh</b>	appel avec déplacement dd Mem[SS:SP-1]:= (PC+3)h, Mem[SS:SP-2]:= (PC+3)l, SP:=SP-2, PC:=PC+3+dd
<b>RET</b>	<b>C3</b>	retour de sous programme PC:=Mem[SS:SP+1],Mem[SS:SP], SP:=SP+2
<b>PUSH AX</b>	<b>50</b>	empile AX Mem[SS:SP-1]:=AH, Mem[SS:SP-2]:=AL, SP:=SP-2
<b>POP AX</b>	<b>58</b>	dépile dans AX AL:=Mem[SS:SP], AH:=Mem[SS:SP+1], SP:=SP+2
<b>OUT adrp,AL</b>	<b>E6 adrp</b>	écriture de AL sur le port d'E-S adrp
<b>IN AL,adrp</b>	<b>E4 adrp</b>	lecture dans AL du port d'E-S adrp

on considère les quatre programmes suivants :

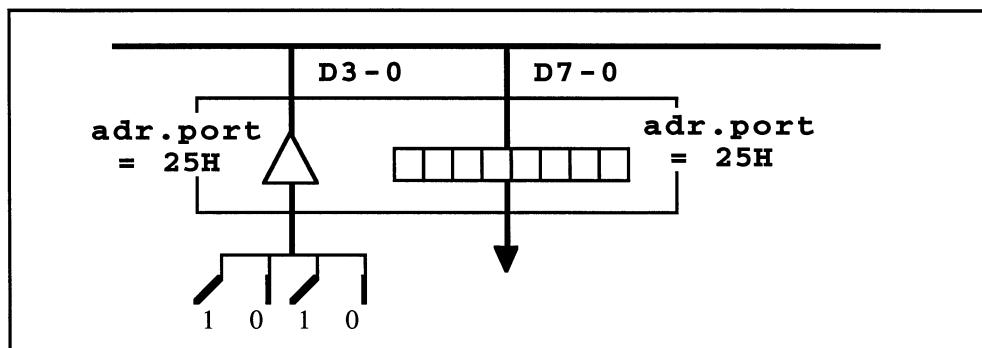
1	2	3	4
MOV SI, 74h MOV AL, 55h MOV TAB[SI], AL	MOV AL, 05h OUT PRT, AL	MOV AL, 05h CMP AL, 05h JZ KKK MOV AL, 00h KKK: MOV AL, 09h	DEB: CALL PPP JMP DEB PPP: PUSH AX POP AX RET

On suppose ces programmes rangés à l'adresse **A0100h**, **SP** contient **0100h**, **AX** contient **8877h**, **TAB=7799h** et **PRT=33h**.

- Donner la séquence des accès bus engendrée par leur exécution.

#### Exercice 4

On considère l'interface d'entrée-sortie suivante :



L'adresse de port **25H** sert à accéder :

- en lecture : à 4 bits affichés sur des clés,
- en écriture : à un registre 8 bits visualisé vers l'extérieur.

- 1 - Préciser le schéma de cette interface, pour un bus PC, réalisé à l'aide d'adaptateurs trois états, de registres et de portes logiques. On considère des registres qui se chargent au front montant d'horloge.
- 2 - Rédiger un programme qui lit les clés et affiche dans le registre la valeur lue complétée par des zéros en poids fort.
- 3 - Donner la séquence des accès bus engendrée par l'exécution de ce programme, supposé rangé en **A0100h**, la valeur affichée sur les clés étant **1010**.

# INTERFACES D'ENTREES-SORTIES

---

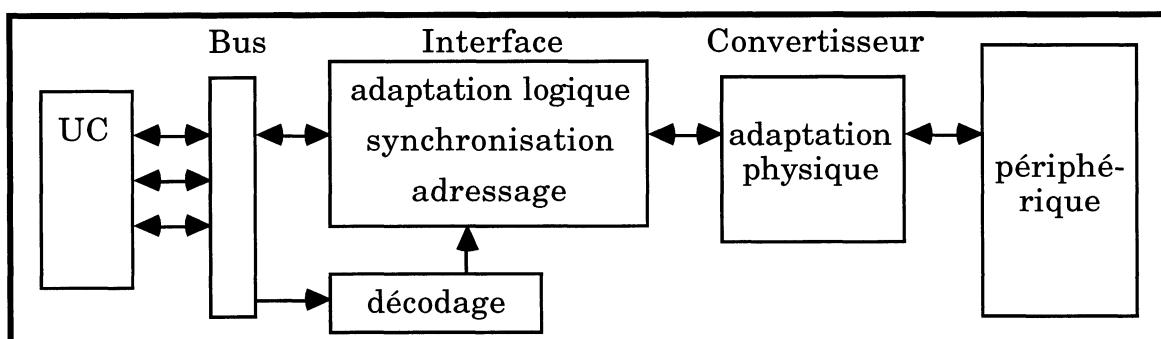
## 1 - Communication avec les périphériques

Les périphériques sont les dispositifs physiques qui permettent les échanges entre la machine et le milieu extérieur ; il en existe de toute sorte : habituels (clavier, écran, imprimante ...), ou plus particuliers (caméra, automate, appareillage tel que lampe, outil de mesure, ou même un autre calculateur).

Il n'y a aucune norme régissant des domaines aussi différents, et leur connexion au calculateur pose divers problèmes.

- Accessibilité : le processeur ne connaît que son bus ; pour spécifier un périphérique, il faut lui associer une adresse.
- Synchronisation : pour la cohérence de la communication (pas de perte, pas de doublons), le processeur et le périphérique doivent être “asservis”.
- Adaptation logique : les données qui passent sur le bus du processeur ont un format déterminé (huit bits, seize bits ...). Chaque périphérique a ses propres formats de données, rarement identiques à ceux du processeur.
- Adaptation physique : le processeur est conçu dans une certaine technologie (signaux logiques dans un standard donné, disons du TTL) ; les signaux du périphérique peuvent être physiquement très différents (tensions différentes, signaux analogiques, signaux optiques ...).

L'adaptation physique est généralement assurée par des circuits spéciaux, qui relèvent de l'électronique. Les autres problèmes sont plus spécifiques de l'informaticien, et sont traités essentiellement au niveau de l'interface.



## 1.1 - Les convertisseurs

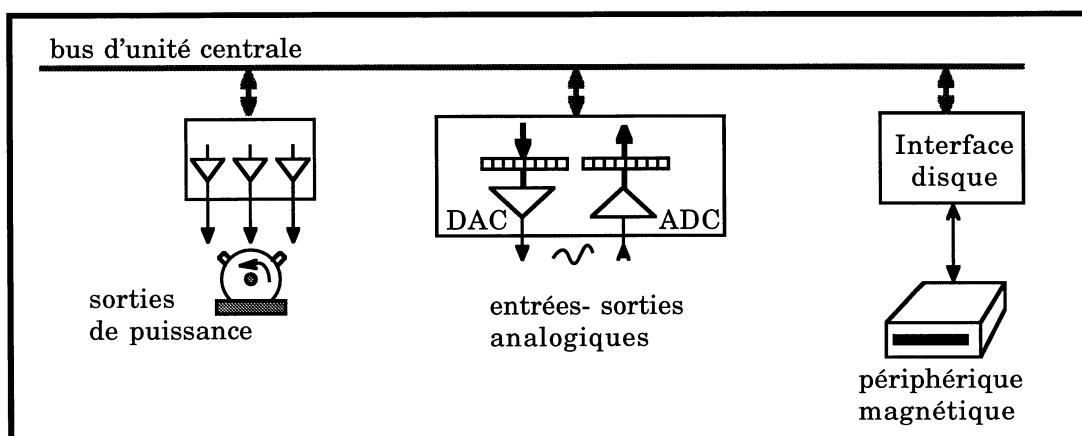
Les données externes se présentent sous des formes physiques diverses.

Pour des signaux logiques, l'adaptation physique consiste en une simple mise en forme : amplification, changement de niveau de tension, changement de mode de codage (courants électriques au lieu de tensions, codage optique ...). Cette transformation s'effectue instantanément, et elle n'a aucune incidence pour l'informaticien (il suffit de mettre le bon circuit d'adaptation ...).

Pour des données externes sous forme analogique, c'est-à-dire appartenant à un intervalle continu de valeurs, on utilise un convertisseur analogique-digital (ADC) pour lire les valeurs, ou un convertisseur digital-analogique (DAC) pour en produire.

Ces conversions provoquent une altération inévitable du signal (l'échantillonnage se fait en des instants discrets, et les valeurs sont représentées avec une précision limitée).

Enfin, les données externes peuvent être sous forme plus spécifiques : états magnétiques, images, sons ... On considère généralement que les transformations de ce type relèvent de dispositifs périphériques spécialisés, l'interface se bornant à connecter le périphérique au bus d'unité centrale.



## 1.2 - L'interface

L'interface traite les autres aspects de la communication : adaptation logique des signaux, adressage des périphériques, et synchronisation.

### 1.2.1 - Adaptation logique

Pour le processeur, une donnée est constituée des  $n$  bits présents simultanément (“en parallèle”) sur le bus de données. Le périphérique, lui, peut exiger des valeurs sur un mode parallèle (de même largeur ou non), ou sur un mode “série” (bit après bit, successivement sur un même “fil”).

La transformation d'un mode parallèle  $n$  bits à un mode série ou à un autre mode parallèle ou s'effectue sous contrôle d'un automate inclus dans l'interface.

De plus, celle ci assure souvent une fonction de “sécurité” des transferts en incorporant des mécanismes de détection d'erreur.

Le procédé de communication série est très utilisé, et a fait l'objet de normalisations. De nombreux circuits d'interface intègrent ces normes : côté calculateur, les chemins de données sont compatibles avec le bus du processeur ; côté extérieur, ils répondent aux spécifications des normes. Ces interfaces sont donc spécifiques d'un calculateur et d'une norme.

Une “norme” série définit l'ordre des bits, leur durée de maintient, les signaux de service ... Les plus communes sont les standards RS232 et RS422.

### 1.2.2 - La synchronisation des échanges

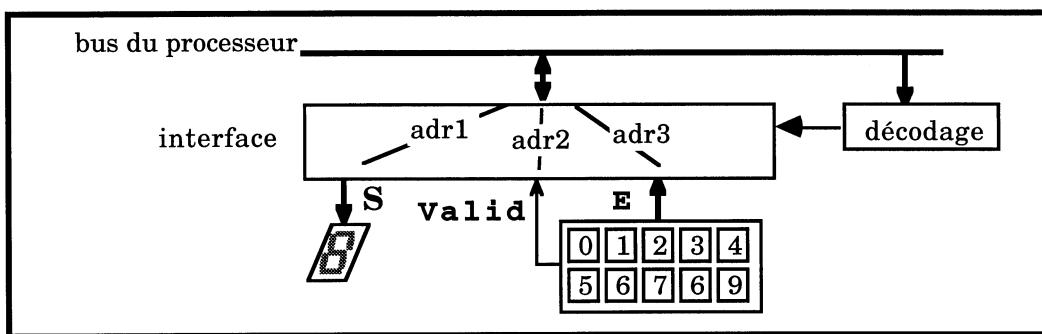
L'unité centrale et le monde extérieur (matérialisé par le périphérique) évoluent de façon indépendante dans le temps. Sans précaution particulière, il n'y a aucune raison pour que le processeur capte la donnée au moment où le périphérique la délivre, ou réciproquement pour que le périphérique s'intéresse à la donnée au moment où le processeur la place sur le bus.

L'une des fonctions essentielles de l'interface est d'assurer la bonne coordination des deux interlocuteurs. Diverses techniques sont utilisées, qui nécessitent le respect d'un protocole de transfert de la part du calculateur et du périphérique. Du côté du calculateur, ce protocole est assuré par une architecture spécifique de l'interface, et une programmation adaptée des échanges.

### 1.2.3 - Adressage de l'environnement

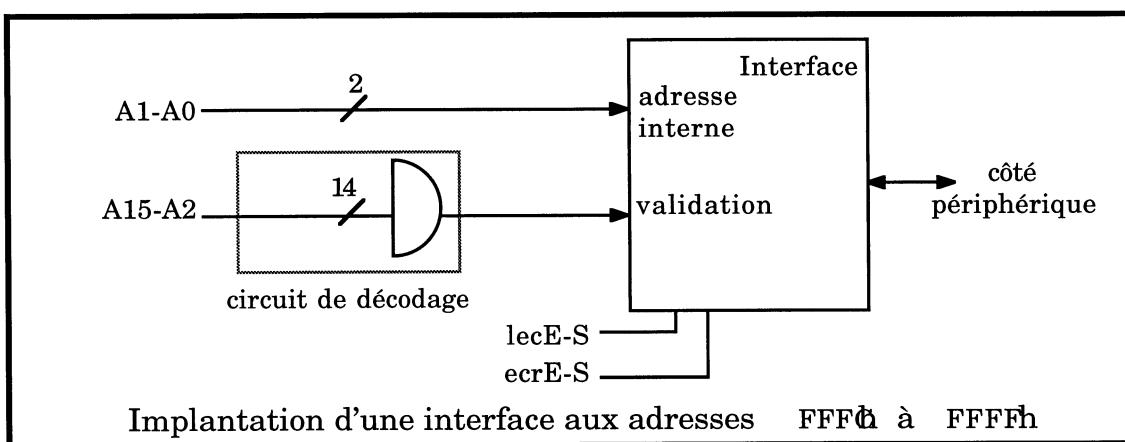
Une interface d'entrée-sortie se présente comme une collection d'adresses de ports sur lesquelles le processeur peut, par programme, envoyer ou recevoir des valeurs. Chacune de ces adresses a un rôle spécifique.

Considérons par exemple un dispositif pour visualiser un chiffre sur un afficheur numérique et recevoir un chiffre frappé sur un clavier. L'interface rend l'afficheur (**S**) accessible par une écriture à l'adresse **adr1**, la valeur de la touche enfoncee (**E**) par une lecture à l'adresse **adr3**, et le signal d'enfoncement d'une touche (**valid**) par lecture à l'adresse **adr2**.



Un circuit d'interface est généralement pourvu d'une entrée de sélection et de quelques lignes d'adresse permettant de désigner un de ses ports internes. Pour le processeur, l'adresse effective du port est la combinaison de l'adresse de l'interface (reconnue par le circuit de décodage) et de l'adresse interne.

Par exemple, avec un bus d'adresses de seize bits et une interface comportant quatre ports d'entrée-sortie que l'on voudrait implanter aux adresses FFFCh à FFFFh, on pourrait avoir le montage suivant.



## 2 - Synchronisation des entrées-sorties

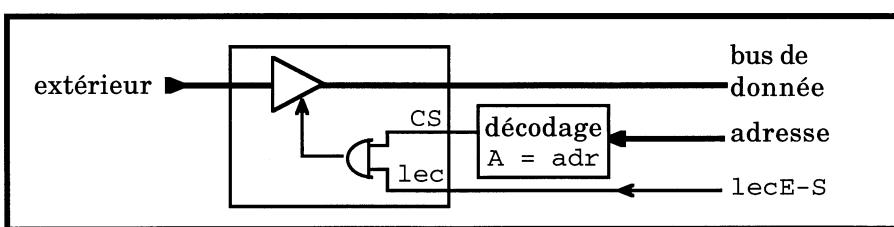
Pour avoir des échanges cohérents avec les périphériques, il faut respecter des protocoles garantissant une certaine synchronisation entre l'émetteur de la donnée et le récepteur : le plus souvent, on veut pouvoir distinguer deux données successives, et n'utiliser qu'une fois chaque donnée.

### 2.1 - Consultation et affichage directs

Le mécanisme de communication le plus simple est l'accès “à la volée” : consultation ou affichage directs de valeurs, sans aucune synchronisation. Malgré l'extrême simplicité du mécanisme, il peut suffire dans certains cas.

#### 2.1.1 - Lecture à la volée

Pour que l'unité centrale puisse consulter des valeurs provenant de l'extérieur, il faut les présenter sur le bus de données aux moments précis où elle exécute des instructions de lecture sur le port d'entrée-sortie correspondant. Le bus étant réalisé en technologie trois-états, il suffit d'intercaler des adaptateurs trois-états entre le périphérique et le bus de données, validés par un circuit de décodage d'adresse, qui détermine l'adresse attribuée au port de lecture.

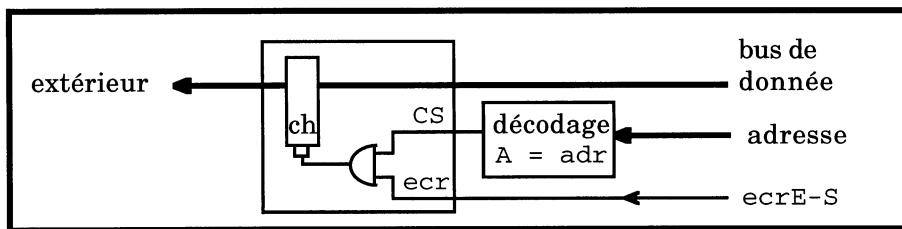


Lorsque l'unité centrale exécute une lecture sur le port correspondant, elle génère un cycle d'accès à cette adresse (*lecE-S*, adresse de port), et le circuit de décodage valide les adaptateurs. La valeur présentée par l'interface pendant le cycle bus est alors disponible pour l'UC.

C'est l'activité programmée sur le calculateur qui décide seule du moment de la lecture. Ce type de lecture n'est utilisable que pour consulter l'état courant d'un système externe (prélever une mesure, tester l'état ouvert/fermé d'une vanne ...).

### 2.1.2 - Affichage à la volée

Pour afficher des valeurs vers l'extérieur, il ne suffit pas de les envoyer sur le bus de données par une instruction d'écriture : la durée de maintien sur le bus n'excède pas un cycle bus. Il faut donc mémoriser la donnée : on connecte le bus de données à l'entrée d'un registre ou d'un verrou, dont le chargement est commandé par l'écriture à l'adresse du port.



La valeur est disponible sur les sorties du registre ou du verrou, et le reste jusqu'à la prochaine écriture sur le même port.

Ce type de sortie peut parfois convenir, pour des commandes de "marche/arrêt", d'ouverture/fermeture de vannes, ou, à l'aide d'un convertisseur digital-analogique, pour contrôler la vitesse d'un moteur électrique, générer de la musique ...

### 2.2 - Entrée-sorties synchronisées par test d'état programmé

Si l'on veut entrer ou sortir des séquences de valeurs, il faut garantir que chaque donnée sera prise en compte une fois et une seule.

Les techniques précédentes ne conviennent pas : rien ne permet au récepteur de savoir quand il y a une nouvelle donnée, ni à l'émetteur de savoir quand le récepteur est prêt à recevoir une nouvelle donnée.

Une technique simple consiste à placer dans l'interface une mémoire de un bit (verrou ou bascule), qui témoigne de l'état de la communication ("donnée disponible"). Chaque émission ("production") met le bit à "1", chaque utilisation ("consommation") le met à "0".

Avant toute utilisation, émetteur et récepteur consultent ce bit, dans une boucle d'attente dont ils ne sortent que lorsque l'état est convenable (en réception : il y a une donnée valide ; en émission : pas de donnée en cours).

Ce petit protocole assure la transmission correcte d'une suite de valeurs, sans perte ni duplication d'éléments.

### 2.2.1 - Synchronisation en entrée

Le bit d'état EP ("entrée présente"), initialement à "0", passe à "1" quand l'extérieur présente une donnée et à "0" quand le programme lit la donnée.

EP = 1 signifie qu'une donnée est présente et n'a pas encore été lue.

EP = 0 signifie qu'il n'y a pas de donnée nouvelle en entrée.

Le programme de lecture d'une donnée est alors :

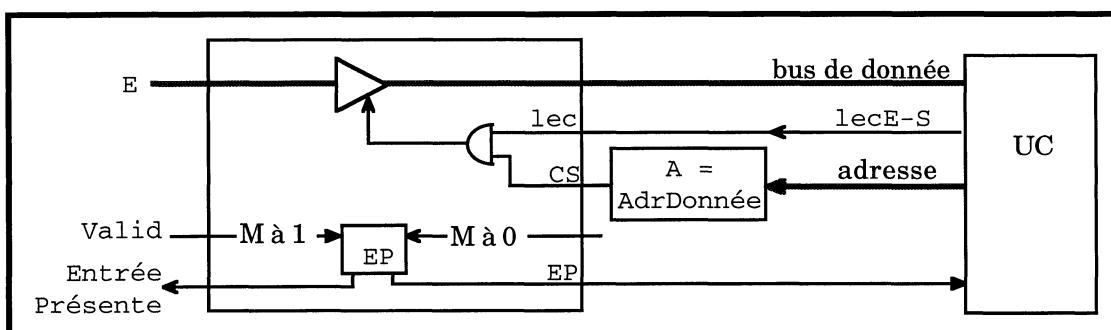
```
tantque EP $\neq$ 1 faire (rien) fait * attente nouvelle donnée *
lire la donnée ;
EP:=0
* consommation donnée *
```

L'attente est réalisée par une boucle qui teste l'état : c'est "l'attente active". Pendant cette attente, l'unité centrale n'est pas disponible pour autre chose.

De son côté, l'extérieur "voit" le bit d'état sur la sortie "Entrée Présente", ce qui permet d'attendre que la donnée courante soit lue avant de fournir la suivante.

Pour chaque nouvelle donnée, l'extérieur active un signal de validation Valid, qui provoque la mise à "1" de EP.

La mise à "0" est faite chaque fois que le processeur vient lire la donnée.



### 2.2.2 - Synchronisation en sortie

Le principe est similaire pour des sorties de données : un bit d'état SP ("sortie présente") est mis à "1" quand le programme écrit une donnée et mis à "0" quand l'extérieur prend en compte cette donnée.

SP = 1 signifie qu'une nouvelle donnée est affichée en sortie.

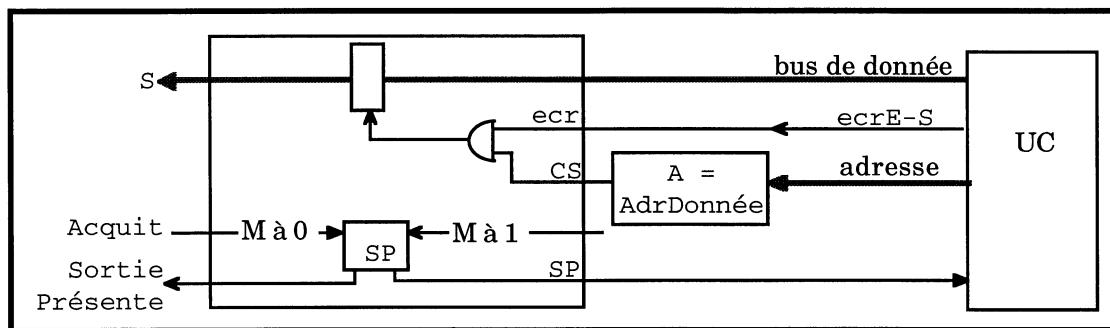
SP = 0 signifie qu'il n'y a pas de donnée nouvelle en sortie, et donc que l'activité programmée peut en placer une.

Le programme d'écriture d'une donnée est alors :

```
tantque SP≠0 faire (rien) fait ; * attente sortie libre *
écrire la donnée;
SP:=1
; * nouvelle donnée *
```

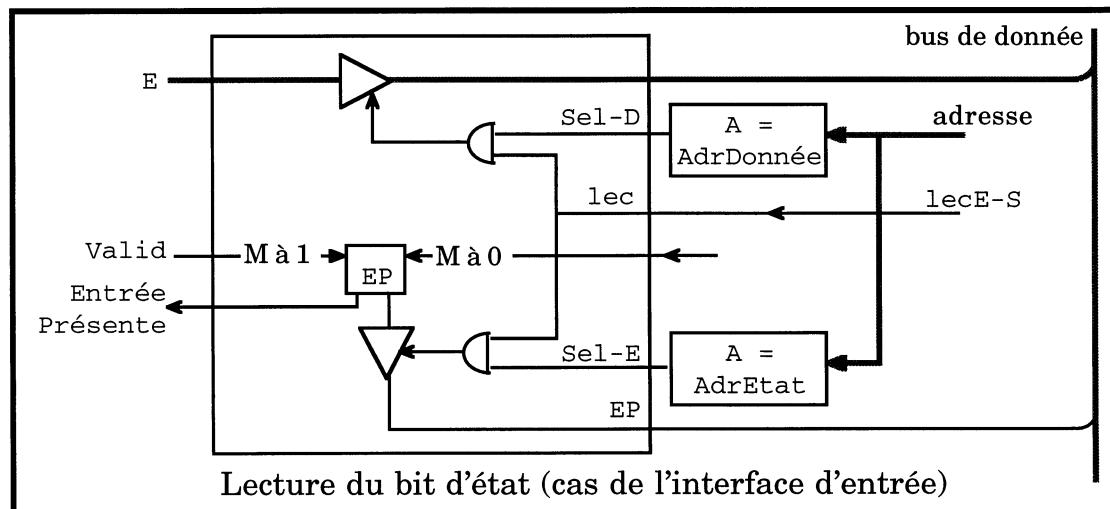
A chaque prise en compte d'une donnée, l'extérieur active un signal d'acquittement Acquit, qui provoque SP:=0.

Le bit d'état est mis à "1" chaque fois que le processeur écrit une donnée.



### 2.2.3 - Lecture du bit d'état

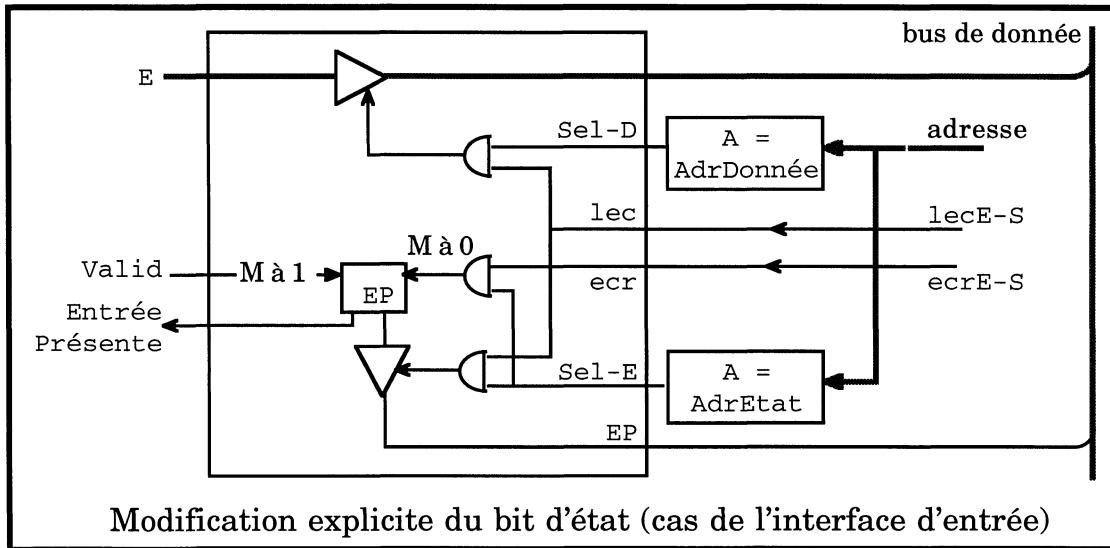
Le programme doit accéder au bit d'état pour tester sa valeur. Celui-ci est donc rendu accessible, à une adresse de port "AdrEtat", et sa valeur est transmise par le bus de données.



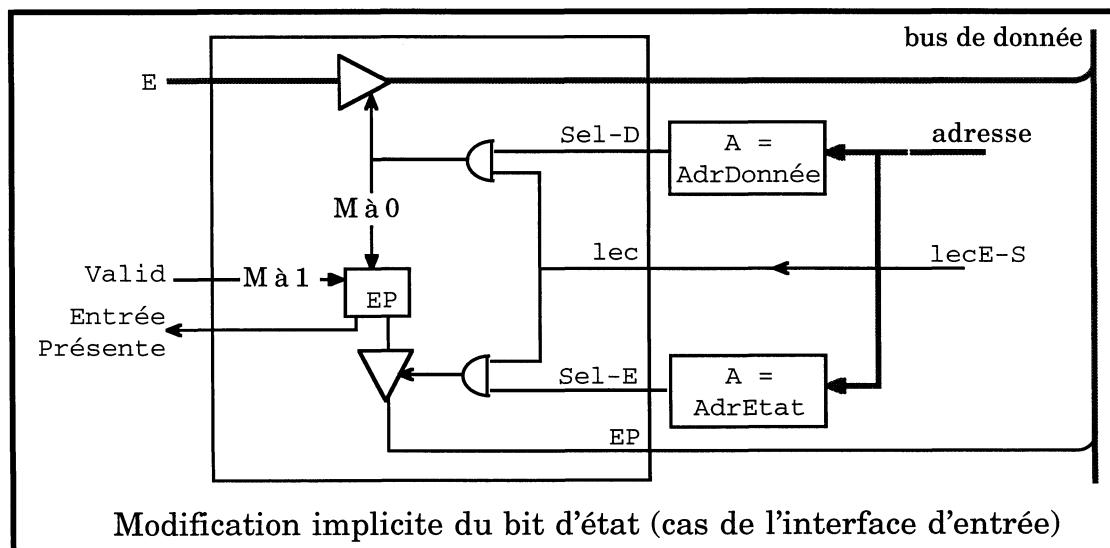
L'interface dispose, comme ici, de deux entrées distinctes pour valider le port d'état et le port de données (Sel-Donnée et Sel-Etat), ou d'une seule entrée de validation et d'une ligne pour spécifier le port, donnée ou état (Sel et Donnée/Etat).

#### **2.2.4 - Modification du bit d'état**

Lorsqu'il fait l'accès à la donnée (lecture ou affichage ...), le programme doit modifier le bit d'état. Il peut le faire par un ordre spécifique d'écriture à une adresse de port (modification explicite).



Le positionnement est habituellement fait systématiquement à chaque accès à la donnée ; il est alors plus simple de le faire automatiquement lors de l'accès lui-même (modification implicite).

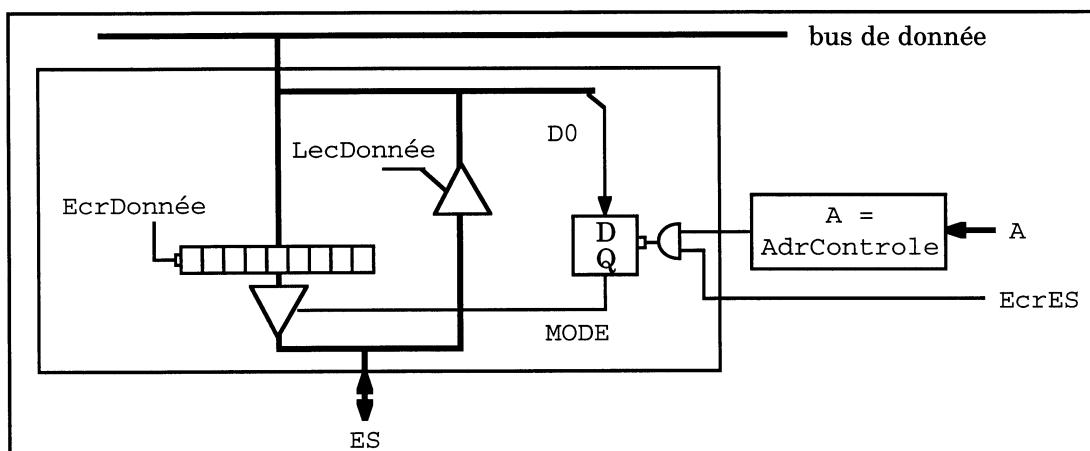


## 3 - Compléments sur les interfaces

### 3.1 - Commandes de configuration

Les interfaces sont souvent des circuits assez complexes pouvant travailler dans différents modes, et donc nécessitant une "configuration". Elle se fait par envoi de certains codes sur des ports réservés à cet effet, les "ports de contrôle".

Ceci est illustré sur l'exemple simple suivant : il s'agit d'une interface permettant, de réaliser des entrées ou bien des sorties de données :



Cette interface peut être configurée en entrée, en écrivant 0 à l'adresse de port `AdrContrôle`, ou en sortie, en écrivant 1 à l'adresse de port `AdrContrôle`.

L'envoi de ces commandes a pour effet de positionner la mémoire un bit `MODE`, interne à l'interface, qui contrôle la présentation des données en sortie.

La plupart des interfaces existantes ont de nombreuses commandes de configuration permettant le choix entre :

- divers modes de transmission,
- divers mécanismes de détection d'erreurs,
- diverses façons de générer les signaux de synchronisation,
- ...

### 3.2 - Interfaces multiples

On trouve très souvent des interfaces à fonctions multiples (échanges bidirectionnels avec un périphérique, ou même échanges avec des périphériques distincts, comme un écran et un clavier ...).

Leur structure est assez proche des interfaces mono-usage, à quelques remarques près :

- Il y a une seule entrée de validation de l'interface ; la liaison concernée est précisée au moyen de bits d'adresse interne : chaque liaison possède son propre port de donnée.

Au niveau du processeur, l'adresse du port est constituée d'une partie "adresse d'interface" qui sert à la validation à travers le circuit de décodage d'adresse, et d'une partie "adresse interne", traitée directement par l'interface.

Dans le cas d'une interface pour une entrée et une sortie, on utilise la même adresse ; la distinction se fait en interne par les signaux lecture ou écriture.

- Il faut un bit d'état pour chaque liaison ; ces différents bits sont généralement observables par le processeur sous la forme d'un "mot d'état" lu en bloc à l'adresse du port d'état. Il suffit ensuite d'isoler le bit concerné par une opération avec masque.

Chaque bit d'état est par ailleurs géré classiquement par les accès à la liaison qu'il représente.

### 3.3 - Interface et port d'entrée-sortie

On l'a bien vu, les notions d'interface et de port d'entrée-sortie sont bien distinctes : une même interface peut comporter (c'est même le cas le plus fréquent) plusieurs ports ; réciproquement une même adresse de port peut correspondre à plusieurs liaisons d'une interface (par exemple dans le cas de l'interface bidirectionnelle).

### 3.4 - Adresses mémoire et ports d'entrée-sortie

Aux adresses mémoires correspondent de simples mots mémoires, ce qui permet d'attribuer une signification simple et immuable aux instructions qui utilisent la mémoire. Par exemple l'instruction "MOV R,Adr" charge le mot d'adresse Adr dans le registre R, à savoir :  $R := \text{MEM}[\text{Adr}]$ .

Par contre aux adresses de port correspondent des organes plus complexes et non fixés à l'avance. Par exemple, la lecture du port AdrDonnée provoque la mise à 0 du bit d'état. On ne peut donc pas attribuer une sémantique précise aux instructions d'entrée-sortie (IN et OUT) car la réaction est spécifique à chaque circuit d'interface. Pour cette interface particulière par exemple, l'instruction "IN R,AdrDonnée" provoque : R:= E ; EP:=0.

## Exercices

### Exercice 1

Une interface d'entrée pour données de huit bits permet de lire les données sur l'adresse de port DONNEE=310h. Le bit d'état EP est accessible à l'adresse de port ETAT=311h, en tant que bit 0 d'un octet. L'unité centrale est un 8086. Rédiger la procédure de lecture synchronisée dans les cas suivants :

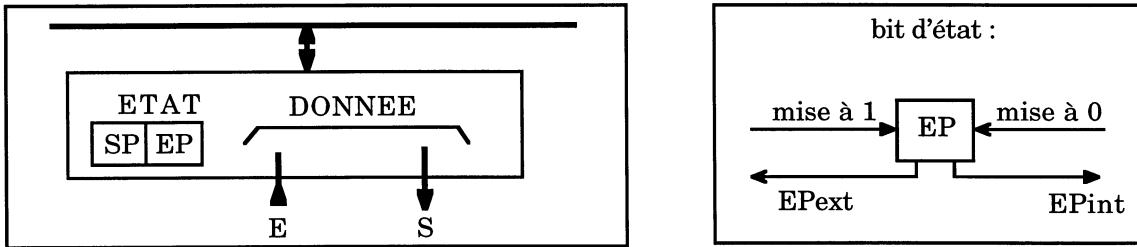
- 1 - La mise à 0 du bit d'état est à faire explicitement.
- 2 - La mise à 0 du bit d'état est automatique à la lecture de la donnée.

### Exercice 2

1 - Donner le schéma interne d'une interface, compatible avec le bus PC-XT, pour réaliser des entrées et des sorties de séquences d'octets. L'interface sera accessible par deux adresses de port seulement :

- adresse DONNEE=310h, qui permet :
  - en lecture, de consulter un octet affiché en entrée E,
  - en écriture, d'afficher un octet sur la sortie S.
- adresse de port ETAT, qui permet :
  - en lecture, d'accéder à deux bits d'état, EP pour la synchronisation des entrées, et SP pour celle des sorties. Ces bits d'états seront présenté respectivement comme les bits 0 et 1 de l'octet lu, et seront affectés automatiquement comme il convient lors de l'accès aux données.

Les bits d'état seront réalisés à l'aide d'un module illustré ci-dessous, que l'on ne détaillera pas ici (c'est l'objet de l'exercice suivant).

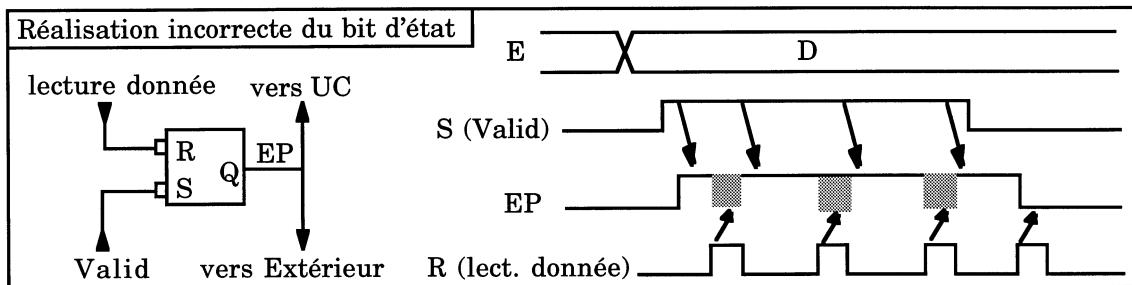


Chaque bit d'état est doté d'entrées de mise à 1 et mise à 0, et de deux sorties EPext (vers l'extérieur) et EPint (vers l'intérieur).

**2 -** Ecrire, en langage machine 8086, la procédure ECRIRE\_ELT de sortie de donnée avec synchronisation par attente active.

### Exercice 3 Problème des affectations concurrentes du bit d'état

On réalise généralement les bits d'état à l'aide d'un verrou RS (R : reset, S : set). Les affectations du bit d'état par l'intérieur et par l'extérieur posent problème. La réalisation simple suivante est incorrecte :



Cette réalisation ne fonctionne pas bien car les mises à 0 et à 1 de EP durent tant que R et S sont maintenus actifs. Si par exemple le calculateur est "rapide" vis à vis de l'extérieur (c'est une question de vitesse relative) il est capable de tester EP=1 et de lire le port de donnée plusieurs fois pendant la durée d'activation du signal Valid par l'extérieur ; le programme tente de mettre EP à 0, mais l'extérieur est encore en train de le mettre à 1 ; ceci provoque plusieurs remises à 1 de EP pour une seule occurrence de valeur d'entrée, et donc des lectures multiples du même élément.

Le problème symétrique se pose également, si l'extérieur est "rapide" relativement à la vitesse du calculateur.

Pour résoudre ce problème, il faut "faire voir" EP à 1 seulement lorsque l'extérieur a fini la mise à 1, et réciproquement ne montrer à l'extérieur "Entrée prête" à 0 que lorsque le calculateur a fini la mise à 0.

- Dans l'exemple précédent, pour un extérieur “lent” et un intérieur “rapide”, la même donnée est lue plusieurs fois. Montrer ce qu'il peut se passer dans le cas d'un extérieur “rapide” et d'un intérieur “lent”.
- Donner un schéma correct du bit d'état, utilisant un verrou RS et des portes. Montrer son fonctionnement par un diagramme temporel.
- On peut également réaliser le bit d'état, à l'aide d'une bascule D avec changeant d'état sur front montant d'horloge. On utilise dans ce cas les signaux de mise à 1 et de mise à 0 comme horloge : donner le schéma d'une telle solution.

# ENTREES-SORTIES SERIES

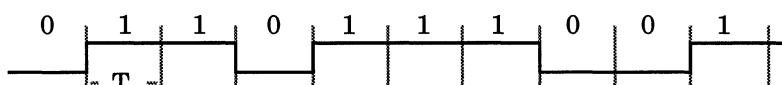
---

## 1 - Transmissions séries

### 1.1 - Généralités

Dans une transmission série, les données sont transmises bit à bit sur une ligne. Ces transmissions sont utilisées pour connecter un ordinateur avec des périphériques (terminaux, imprimantes), ou des ordinateurs entre-eux.

Chaque bit est transmis pendant une durée fixe  $T$ .



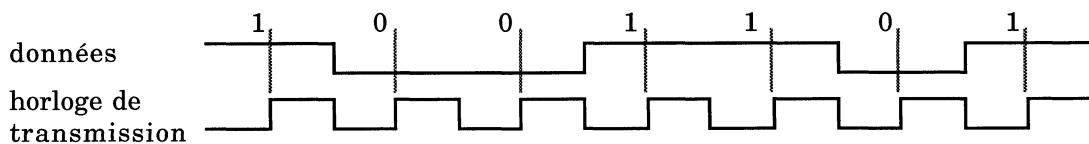
La "vitesse de transmission" est la fréquence  $1/T$ . On l'exprime en bauds : une vitesse de 4800 bauds signifie que la durée de chaque bit est  $1/4800$  s.

Le "débit de transmission" est le nombre de bits de données utiles transmis par seconde. On l'exprime généralement en bits/s et il est légèrement inférieur à la vitesse de transmission car certains intervalles de temps sont utilisés pour transmettre des signaux de contrôle ou de synchronisation.

Il existe deux classes de transmissions série : "synchrone" et "asynchrone".

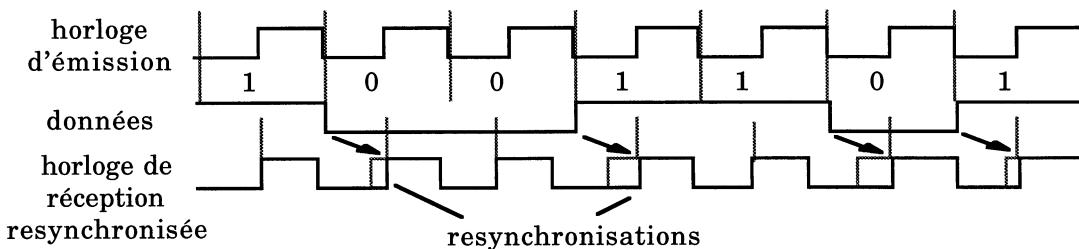
### 1.2 - Transmissions synchrones

En transmission synchrone, l'émetteur et le récepteur disposent d'une horloge de transmission commune au niveau du bit, qui permet de reconstituer la suite des bits à la réception.

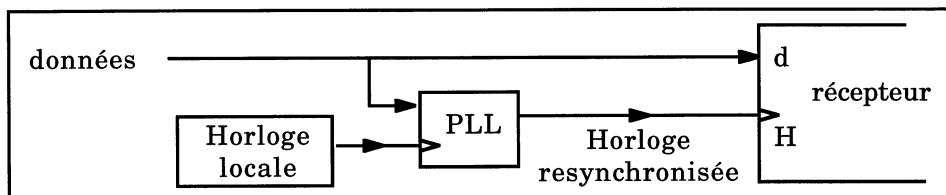


En général, l'horloge n'est pas transmise, elle est reconstituée localement à la réception à partir du signal de données lui-même (il serait coûteux de transmettre l'horloge sur une ligne séparée, et de toute façon impraticable à longue distance et à fort débit : on ne pourrait pas assurer des délais identiques sur la ligne de données et sur la ligne d'horloge).

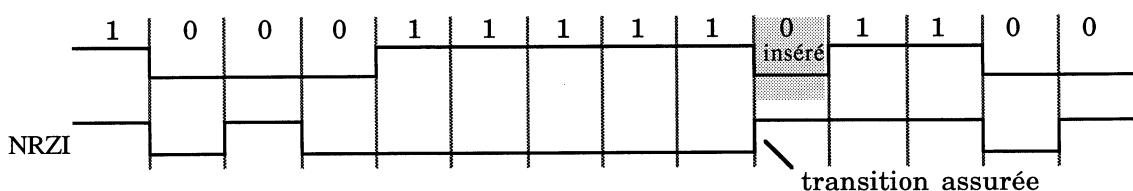
Le récepteur dispose d'un générateur d'horloge de même fréquence que l'horloge d'émission, assez précis pour recevoir environ une dizaine de bits sans problème de dérive (deux horloges de "même" fréquence ne sont jamais parfaitement identiques). Pour corriger la dérive, l'horloge locale est resynchronisée à l'aide des transitions du signal de données.



Le dispositif de resynchronisation, appelé "boucle de verrouillage de phase" (PLL : Phase Locked Loop), peut être réalisé par une méthode électronique ou par un automate qui travaille à partir d'une horloge plus rapide que l'horloge de transmission, par exemple seize fois, et du signal de donnée.



La resynchronisation nécessite des transitions fréquentes dans le signal de données. Ceci est réalisé par un codage spécial de la séquence de bits en signaux physiques. On peut citer le codage NRZI (Non Return to Zero Inverted) avec insertion de zéros : un "0" est représenté par un changement d'état de la ligne (une suite de "0" est donc représentée par une suite alternée de 0 et de 1) et pour des chaînes de "1" de longueur > 5, l'émetteur insère un "0" pour forcer une transition. Les "0" rajoutés sont facilement supprimés par le récepteur.



On distingue deux formes principales de transmissions synchrones :

- les transmissions de caractères (MONOSYNC - BISYNC)
- les transmissions de chaînes de bits quelconques (SDLC - HDLC).

### Transmission de caractères (MONOSYNC)

Ce système est ancien et de moins en moins utilisé. Les données sont formées de caractères, généralement de sept ou huit bits. Un caractère spécial SYNC est émis lorsque aucune donnée n'est transmise. Ce caractère sert à maintenir la synchronisation entre l'émetteur et le récepteur.

SYNC	Car1	Car2	SYNC	SYNC	Car3	Car4	Car5	SYNC	Car6	SYNC
------	------	------	------	------	------	------	------	------	------	------

Une phase d'initialisation est nécessaire, pendant laquelle l'émetteur n'émet que des SYNC pour permettre au récepteur de se cadrer sur les caractères.

Les caractères SYNC sont supprimés par le récepteur. Le caractère SYNC ne doit pas apparaître dans le flot des données, ce qui est une limitation gênante. Ce système est utilisable pour transmettre des caractères (ASCII par exemple), auquel cas le code SYNC est choisi parmi les codes inutilisés, mais il est mal adapté pour la transmission de chaînes de bits quelconques. Le système BISYNC est similaire, mais utilise une paire de caractères spéciaux SYNC1 et SYNC2 lorsque aucune donnée n'est transmise.

### Transmission de chaînes de bits (SDLC)

Les données transmises sont chaînes de bits de taille quelconque. En l'absence de données, il est transmis une configuration spéciale de huit bits, le "drapeau" (flag) : 01111110. Les données sont transmises en un paquet ininterrompu appelé "trame". La fin de la trame est caractérisée par la rencontre d'un drapeau.

01111110	01111110	adresse	contrôle	message		CRC 16 bits	01111110
----------	----------	---------	----------	---------	---	-------------	----------

Lorsque plus de cinq "1" se suivent dans les données à transmettre, l'émetteur insère un "0", ce qui assure les transitions nécessaires à la resynchronisation de l'horloge en codage NRZI, ainsi que l'absence d'occurrence du drapeau dans le codage du message.

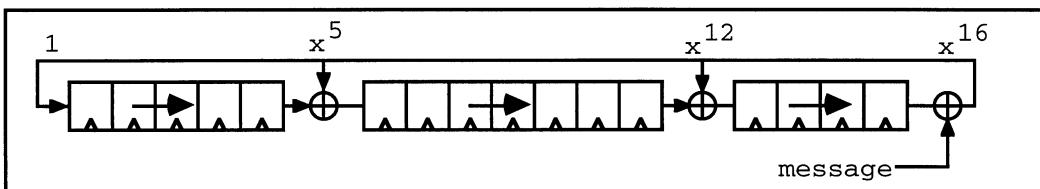


Dans le format SDLC (Synchronous Data Link Control) une trame est composée d'un champ adresse de huit bits pour identifier le récepteur du message (dans le cas de transmissions dans un réseau), d'un champ contrôle de huit bits pour indiquer un type de message (données, acquittement, ...), le message et enfin un champ CRC de seize bits (code de redondance cyclique) qui permet de détecter les erreurs de transmission.

Il y a deux standards de CRC, donnés par les “polynômes générateurs” :

$$\text{standard CRC16 : } x^{16}+x^{15}+x^2+1 \quad \text{standard CCITT : } x^{16}+x^{12}+x^5+1$$

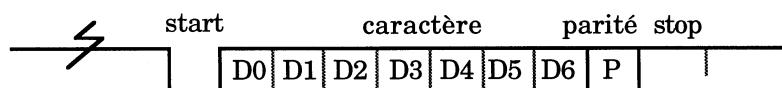
Le CRC est calculé à partir des bits du message à l'aide de registres à décalage et de ou-exclusifs. Le schéma suivant illustre le calcul du CRC CCITT : à chaque terme  $x^i$  du polynôme générateur correspond un ou exclusif en position  $i$  dans le registre. Le registre est initialisé à 0 et il contient le CRC en fin de transmission du message.



Ces CRC permettent de détecter toutes les erreurs concernant un, deux ou trois bits, et toutes les erreurs sur une chaîne de bits consécutifs de longueur inférieure ou égale à seize (cas fréquent appelé “salves d’erreurs”).

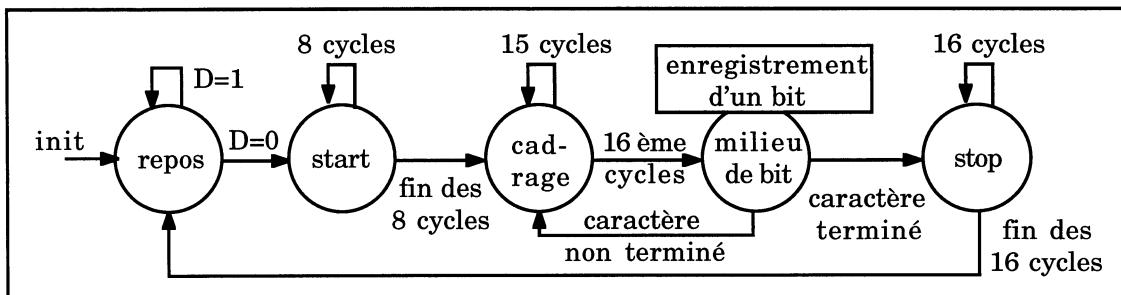
### 1.3 - Transmissions asynchrones

En transmission asynchrone, émetteur et récepteur n'ont pas d'horloge commune pour séquencer les bits. Les données sont transmises par caractères de petites tailles (5 à 9 bits). Le récepteur reconnaît l'arrivée d'un caractère grâce à une marque appelée START, puis échantillonne les bits du caractère avec une horloge locale. Un caractère est suffisamment petit pour que ses bits soient correctement enregistrés malgré la dérive de l'horloge locale.



- En l'absence d'émission, la ligne est dans l'état de repos “1”.
- La transmission d'un caractère commence par la marque START (bit “0”), qui dure une période de base (on l'appelle le “start bit”).
- Puis suivent les bits du caractère, en commençant par les poids faibles.
- De façon optionnelle, il peut y avoir ensuite un bit de parité P, soit paire ( $P = D_0 \oplus D_1 \oplus \dots \oplus D_{n-1}$ ), soit impaire ( $P = D_0 \oplus D_1 \oplus \dots \oplus D_{n-1} \oplus 1$ ).
- La transmission du caractère se termine par un retour à l'état de repos (bit “1”), qui dure au moins (selon option), 1, 1.5 ou 2 périodes de base (on dit qu'il y a 1, 1.5 ou 2 “stop bits”); ceci pour éviter toute confusion entre un zéro en fin de caractère et la marque START du caractère suivant.

La réception est réalisée par un automate qui fonctionne avec une horloge 8 à 64 fois plus rapide que la fréquence des bits. Avec par exemple une horloge 16 fois plus rapide, cet automate peut être schématisé ainsi :



Dans l'état "repos", le récepteur attend la marque START. Quand elle est détectée ( $D=0$ ), il attend 8 cycles pour se situer au milieu de la marque START. Ensuite vient la boucle de lecture des bits : attente de 15 cycles pour se situer au milieu du bit de donnée, et enregistre le bit en l'accumulant dans un registre à décalage. Enfin, après avoir reçu tous les bits du caractère, il attend 16 cycles pour se placer au milieu de la marque STOP puis il se met en attente du caractère suivant.

Les bits sont échantillonnés près du milieu de leur durée, pour mieux tolérer les différences de fréquence entre horloges d'émission et de réception.

#### 1.4 - Usage des transmissions séries

Les vitesses de transmission standard sont :

75, 110, 300, 600, 1200, 2400, 4800, 9600, 19200, 48k, 64k, 128k,... 1Méga ...

Les vitesses lentes ( $\leq 9600$  bauds), sont généralement utilisées en transmission asynchrone (mode de transmission économique et robuste, réservé aux communications avec des terminaux, des imprimantes ...). Les vitesses élevées ( $\geq 19200$  bauds), sont plutôt utilisées en transmission synchrone, essentiellement pour des communications entre ordinateurs, souvent à travers un réseau.

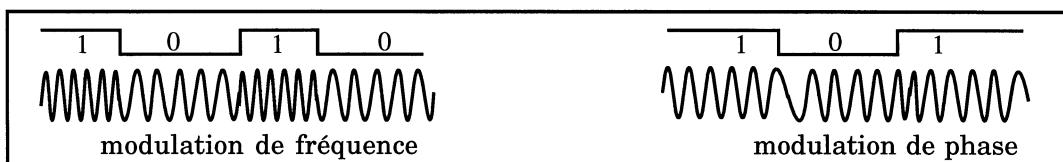
Le support de transmission est différent selon la vitesse de transmission :

A moins de 9600 bauds, des fils normaux suffisent. La norme électrique la plus répandue est la norme RS232 : le "0" et le "1" sont représentés par les tensions -12v et +12v par rapport à la masse.

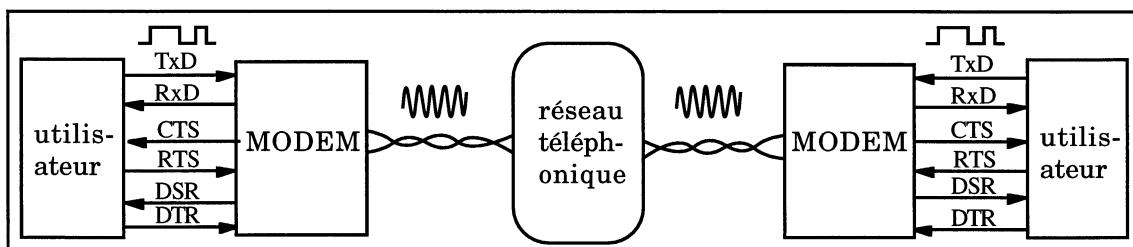
Entre 9600 et 100Kbauds, on utilise des paires de fils torsadées ou des câbles coaxiaux qui offrent des caractéristiques électriques meilleures, ainsi qu'une adaptation électrique soignée entre l'émetteur et la ligne pour éviter les réflexions du signal.

## Modems

Les transmissions séries sont souvent utilisées à travers le réseau téléphonique. Les fréquences du signal transmis par téléphone doivent être supérieures à 300Hz : un signal continu ou lentement variable ne passe pas. De même, les fréquences trop élevées, au-delà de 5000Hz, ne passent pas non plus (cela limite le débit d'informations). Il faut transformer le signal de données, en un signal qui puisse passer à travers le réseau téléphonique. Ceci est réalisé par une technique appelée "modulation". On transmet un signal sinusoïdal, avec une fréquence qui passe à travers le téléphone (environ 1000Hz), modifié (on dit modulé) par le signal des données à transmettre. Le signal sinusoïdal pur s'appelle la porteuse. Il y a plusieurs façons de moduler une porteuse. Les plus simples sont : la modulation d'amplitude, qui fait varier l'amplitude de la porteuse (peu utilisée) - la modulation de fréquence, qui fait varier la fréquence (pour des données binaires, cela revient à utiliser deux fréquences, l'une pour le 0 et l'autre pour le 1) - la modulation de phase, qui fait varier la phase (décalage la sinusoïde vers le passé ou le futur selon la valeur du bit de donnée).



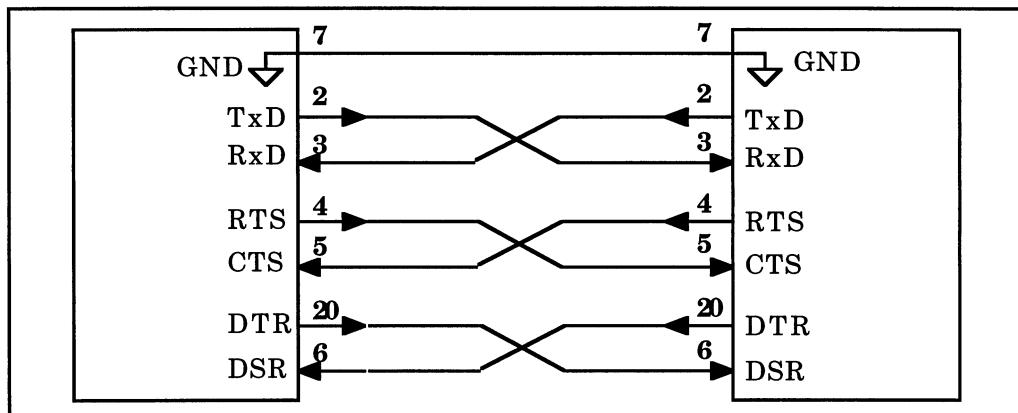
L'appareil qui réalise la modulation à l'émission et la démodulation à la réception s'appelle un MODEM (modulateur-démodulateur).



L'usage de modems a introduit des signaux supplémentaires appelés "signaux de contrôle de modem". Les quatre principaux s'appellent DTR, DSR, RTS et CTS. Leur signification usuelle est la suivante :

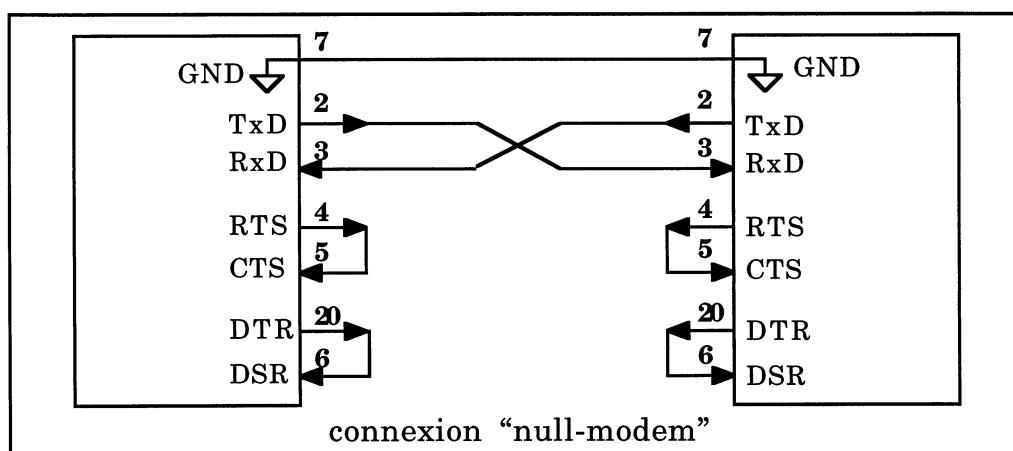
- DTR : "Data Terminal Ready", indique que l'appareil est connecté,
- DSR : "Data Set Ready", indique à l'appareil que la connexion est établie,
- RTS : "Request To Send", dit au modem que le partenaire peut émettre,
- CTS : "Clear To Send", indique à l'appareil qu'il peut émettre les données.

Ces signaux sont parfois utilisés même lorsqu'il n'y a pas de MODEM. Dans ce cas les partenaires sont directement connectés en pratiquant les connexions croisées DTR-DSR et RTS-CTS, comme le montre le schéma ci-dessous (les numéros indiqués sont les numéros de broche pour un connecteur 25 broches standard).



Chaque protagoniste active sa sortie DTR, de sorte que son partenaire peut savoir s'il est connecté en testant son entrée DSR.

La connexion RTS-CTS a un usage plus dynamique : elle sert à pratiquer ce qu'on appelle le “contrôle de flux”, qui consiste à freiner le débit d'émission des données quand c'est nécessaire. Quand un protagoniste est prêt à recevoir, il active sa sortie RTS, ce qui autorise son partenaire à émettre par le biais de son entrée CTS. S'il n'est plus disposé à recevoir, par exemple lorsque ses tampons de réception de données sont pleins, il désactive son RTS ce qui bloque l'émission de son partenaire.

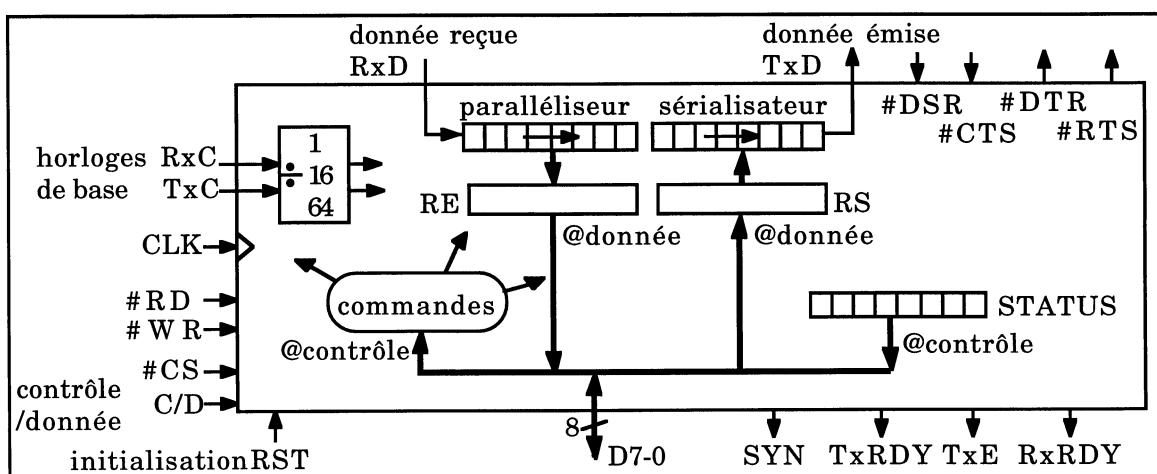


## 2 - Interface d'entrées-sorties séries : USART

Un interface d'entrées-sortie série émet en série les données fournies sous forme d'octets par le calculateur, et présente au calculateur sous forme d'octets les données reçues en série. Il existe de nombreux interfaces séries ; l'USART (Intel 8251, “Universal Synchronous Asynchronous Receiver Transmitter”) en est un, relativement simple, de conception assez ancienne mais très répandu.

### 2.1 - Structure générale de l'USART

La figure suivante montre la structure logique de l'USART. Les données séries reçues sur RxD sont introduites bit à bit dans un registre à décalage, puis rangées dans le registre d'entrée RE. Les données à émettre sont prélevées depuis le registre de sortie RS, rangées dans un registre à décalage et émises bit à bit sur TxD. Les horloges qui définissent la vitesse de transmission sont fournies par l'extérieur (depuis un oscillateur local) sur RxC et TxC ; les vitesses d'émission et de réception peuvent être différentes, mais c'est rarement le cas. La fréquence de ces horloges est divisée par l'un des trois facteur 1, 16 ou 64 pour fournir l'horloge de transmission. Par exemple, si TxC est à 19200Hz et que l'on choisit le facteur 16, la vitesse de transmission sera  $19200/16=1200$  baud.



L'entrée RST (“reset”) est le signal d'initialisation. Les sorties TxRDY, TxE et RxRDY sont les bits du registre d'état de même nom. La sortie SYN, utilisée en mode synchrone, indique que le récepteur est synchronisé. Les broches #DSR, #DTR, #CTS et #RTS sont les signaux de contrôle de modem. L'horloge système CLK fait fonctionner les divers automatismes internes ; elle doit être de fréquence au moins 20 fois supérieure à RxC et TxC.

L'USART admet divers modes de fonctionnement :

- Elle peut réaliser des transmissions synchrones (MONOSYNC et BISYNC), ou asynchrones.
- Elle admet diverses options, telles que le choix de la parité dans le mode asynchrone, ou encore le nombre de bits par caractère.

Ces divers choix sont spécifiés par des commandes de configuration, communiquées à l'USART par le calculateur.

Un registre d'état de huit bits, STATUS, contient diverses informations concernant l'état de la transmission : indicateurs de présence de caractères en entrée et en sortie, indicateurs d'erreurs ...

L'adressage de l'USART se fait par deux adresses de port :

adresse @contrôle: lecture du registre d'état (STATUS),  
écriture des commandes à l'USART.

adresse @donnée : lecture des données reçues (RE),  
écriture d'un caractère à émettre (RS).

## 2.2 - Commandes de l'USART

Avant de pouvoir utiliser l'USART, il faut l'initialiser en choisissant diverses options. Ceci se fait en envoyant trois octets de commande à l'adresse @contrôle. Ces trois octets de commande s'appellent :

la commande *Reset*,  
la commande de *Mode*  
et la commande de *Fonctionnement*.

Les commandes doivent être envoyées dans cet ordre.

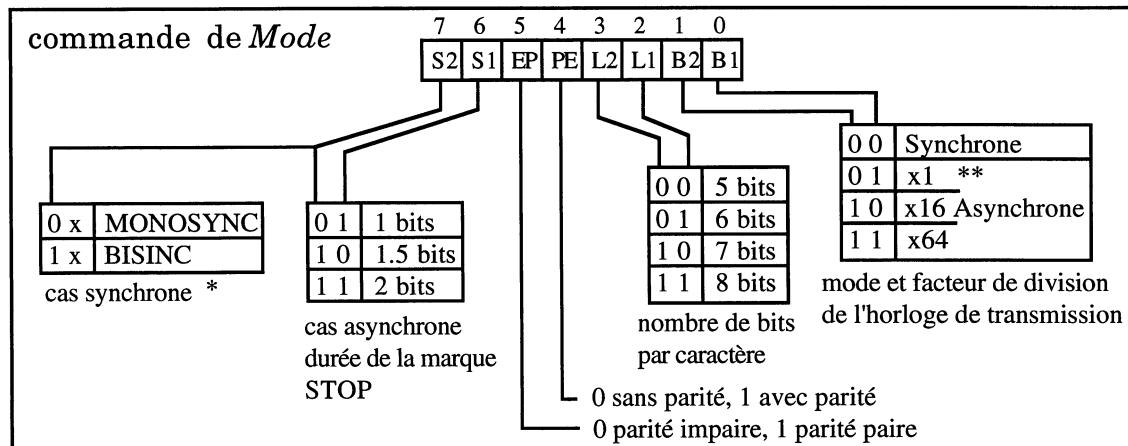
### Commande de Reset

Avant toute initialisation particulière, il faut remettre l'USART dans un état où elle accepte d'être initialisée. Ceci est fait par la commande *Reset* :

commande <i>Reset</i>	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	0	0	0	0	0	0	= 40H
0	1	0	0	0	0	0	0			

## Commande de Mode

Cette commande indique le mode (synchrone/asynchrone) et dans le cas asynchrone indique le facteur de division de l'horloge de transmission, la taille des caractères, la sorte de parité et la durée de la marque STOP.

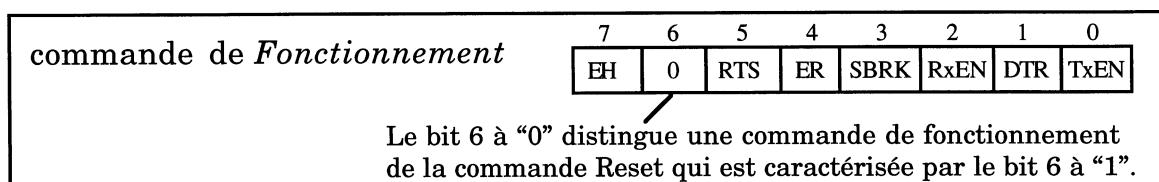


\* : En mode synchrone MONOSYNC il faut de plus envoyer, sous forme d'une commande, le code du caractère de synchronisation choisi (deux caractères en mode BISYNC).

\*\* : L'option x1 est inutilisable en réception, sauf si l'horloge de réception est parfaitement synchronisée avec les données reçues.

## **Commande de Fonctionnement**

Cette commande peut être envoyée plusieurs fois en cours de fonctionnement.



TxEN (*Transmitter Enable*) doit être à “1” pour que l'émetteur fonctionne (aucun caractère n'est transmis sinon).

DTR : permet de positionner la sortie #DTR à la valeur indiquée.

RxEN (*Receiver Enable*) doit être à “1” pour que le récepteur fonctionne (sinon les données reçues sur la ligne sont ignorées).

SBRK (*Send Break*) à “1” provoque l’émission d’un “Break” (un signal à “0” qui dure plus longtemps qu’un caractère ; c’est par exemple ce qu’un terminal génère si on active la touche Break).

ER (*Error Reset*) à “1” force à “0” les indicateurs d’erreur (dans le mot d’état, voir ci-après) ; on le fait à l’initialisation, et après avoir pris en compte une erreur : c’est le seul moyen de mettre à “0” les indicateurs d’erreur.

RTS : permet d'afficher la sortie #RTS à la valeur indiquée.

EH (*Enter Hunt mode*) utilisé uniquement en mode synchrone ; force le récepteur à abandonner toute réception, et à attendre la réception d'un caractère SYNC (permet d'initialiser correctement la réception en mode synchrone et de se resynchroniser en cas d'erreur de transmission qui aurait fait perdre la synchronisation).

### Usage des commandes de l'USART

Pour utiliser convenablement l'USART, il faut :

- l'initialiser selon les modes de fonctionnement choisis, ce qui se fait par une commande *Reset* suivie d'une commande de *Mode*,
- la mettre en marche par une commande de *Fonctionnement* ,

En cours de fonctionnement, on peut envoyer autant de commandes de *Fonctionnement* que l'on veut, par exemple pour arrêter la réception, la redémarrer, remettre à "0" les indicateurs d'erreur ou réinitialiser entièrement l'USART.

### 2.3 - Mot d'état de l'USART

Le registre d'état a la structure suivante :

registre d'état :	7	6	5	4	3	2	1	0
	DSR	BRK	FE	OE	PE	TxE	RxRDY	TxRDY

- TxRDY (*Transmitter Ready*) indique que le registre RS est libre, c'est-à-dire susceptible de recevoir un nouveau caractère à transmettre : ce bit permet la synchronisation des émissions ; il est mis à "1" initialement et chaque fois qu'un caractère est prélevé depuis RS pour être transmis, et mis à "0" lorsque le calculateur écrit un caractère dans RS.
- RxRDY (*Receiver Ready*) indique qu'un caractère reçu et non consommé est présent dans le registre d'entrée RE : ce bit permet la synchronisation des lectures ; il est mis à "1" lorsqu'un caractère est reçu, et mis à "0" lorsque le registre RE est lu par le calculateur.
- TxE (*Transmitter Empty*) indique que le tampon de sortie RS et le registre de transmission sont vides, (aucune transmission n'est en cours). Ce bit, peu utilisé, permet de savoir qu'un caractère est effectivement transmis.
- PE, OE et FE sont les indicateurs d'erreur de réception (actifs à "1") :
  - PE (*Parity Error*) : un caractère a été reçu avec un bit de parité incorrect.

- OE (*Overrun Error*) : un caractère a été reçu alors que le précédent n'a pas été prélevé par le calculateur (dans ce cas, l'USART "écrase" l'ancien caractère par le nouveau, et l'ancien est donc perdu).
- FE (*Frame Error*) : le signal reçu sur RxD n'a pas une allure acceptable pour un signal de données ; par exemple, en mode asynchrone, après avoir reçu les bits d'un caractère, on s'attend à ce que le signal repasse à l'état de repos, pour au moins la durée de la marque STOP ; si ce n'est pas le cas, l'USART positionne FE à "1".
  - BRK (*Break*) signale la réception d'un "Break" sur RxD.
  - DSR est directement l'état de l'entrée #DSR du circuit.

L'entrée #CTS n'est pas consultable par lecture. Elle permet de bloquer l'émission des données tant qu'elle est inactive. Si #CTS est désactivé alors qu'un caractère est en cours d'émission, la transmission complète de ce caractère est tout de même assurée.

En réception asynchrone avec parité et moins de huit bits par caractères, le bit de parité reçu est également rangé dans le registre d'entrée : en règle générale, une procédure de lecture de caractères doit donc mettre ce bit à "0", au moyen d'un masque, car il ne fait pas partie du code du caractère reçu.

## Exercices

### Exercice 1

Dessiner le diagramme temporel correspondant à l'émission de la séquence de caractères "ABC", pour une transmission série asynchrone avec sept bits de données, bit de parité paire et deux stop bit. A 1200 bauds, combien de temps dure cette séquence ?

Rappels : Codes ASCII : "A" = 41h, "B" = 42h, "C" = 43h,

En transmission série, les poids faibles sont transmis en premier.

### Exercice 2

Un émetteur transmet des données sept bits sur une ligne série asynchrone, à 1200 bauds, avec parité paire et un stop bit.

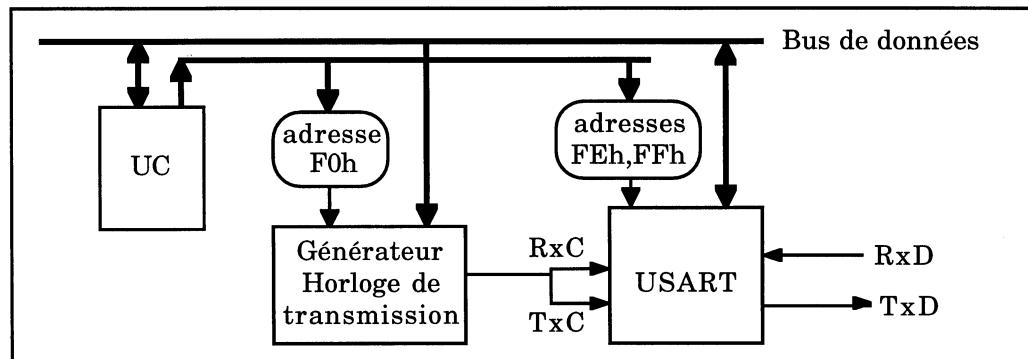
Un récepteur USART analyse ces données à l'autre extrémité de la ligne, mais il est configuré en 4800 bauds, 7 bits de données, parité paire. L'émetteur envoie le code ASCII du caractère "A".

Que reçoit le récepteur dans son registre d'entrée ?

Quels indicateurs d'erreur sont positionnés ?

### Exercice 3

On considère l'installation suivante :



L'USART est accessible aux adresses de port FEh (données) et FFh (contrôle). Les horloges RxC et TxC proviennent d'un générateur d'horloge de fréquence programmable par envoi, à l'adresse de port F0h, d'un code de trois bits (sur D2-0) selon la correspondance suivante :

code 3 bits	0	1	2	3	4	5	6	7
fréquence du générateur	1200	2400	4800	9600	19200	38400	76800	153600

- 1 - Ecrire (en langage machine 8086) la procédure USARTINIT qui configure le générateur d'horloge et l'USART pour une transmission asynchrone à 1200 bauds, 7 bits de donnée, parité paire et un stop bit.
- 2 - Ecrire les procédures de lecture et d'écriture d'un caractère sur cette liaison série :
  - LIRECAR : lecture d'un caractère, résultats :  
AL = code ASCII du caractère reçu, sur sept bits, bit 7 forcé à "0",  
BL = Les trois indicateurs d'erreur du mot d'état (0 si pas d'erreur).
  - ECRICAR : émission d'un caractère, paramètre:  
AL = code ASCII du caractère à émettre.



# MECANISME D'INTERRUPTIONS

---

## 1 - Principe et utilisation des interruptions

### 1.1 - Insuffisance du test d'état programmé

Pendant le fonctionnement normal d'une machine, les actions qu'elle effectue sont prévues par avance et codifiées dans un programme. Les entrées-sorties notamment ne peuvent être provoquées que par une action du processeur (l'exécution d'un "IN" ou d'un "OUT"). Cela ne convient pas toujours, car pour pouvoir réagir correctement à certains événements externes survenant à des moments aléatoires, il faut que le monde extérieur soit capable d'intervenir sur le déroulement de l'activité du processeur. On rencontre ce besoin principalement dans trois cas.

- Les **traitements d'urgence** : le processeur doit répondre rapidement à une situation inhabituelle (déttection d'alarme, chute de tension d'alimentation, etc). Il n'est pas réaliste de tester ces situations par programme.
- Les **entrées-sorties non bloquantes** : on désire faire des lectures ou des écritures synchronisées, mais sans bloquer le processeur quand l'extérieur n'est pas prêt (on veut utiliser le processeur pour exécuter d'autres tâches plutôt que d'attendre). C'est ce que fait un système multitâches : quand une tâche tombe en attente, c'est une autre qui est activée.
- Les **entrées-sorties et traitements temps réel** avec calculs simultanés : on doit enregistrer des données quand elles surviennent, alors que les traitements à effectuer sur les données précédentes ne sont pas terminés. Il faut donc suspendre l'activité de calcul le temps de l'acquisition.

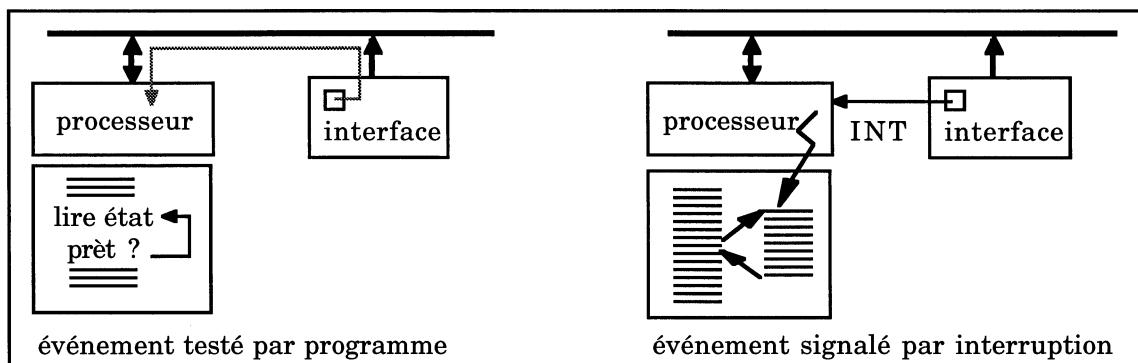
Ces trois cas présentent une grande similitude : alors que le processeur effectue un certain traitement, il doit pouvoir être dérouté vers un traitement spécifique (traitement d'urgence, lancement ou réactivation d'une tâche, opération d'entrée-sortie ...) dès que survient un événement extérieur. Après ce déroutement, le processeur doit pouvoir reprendre l'activité interrompue au point où il l'avait laissée.

## 1.2 - Le mécanisme d'interruption

Pour permettre à l'environnement d'intervenir sur le déroulement de ses activités, le processeur dispose d'une broche sur laquelle les interfaces peuvent signaler leurs demandes d'interruption. Ce signal provoque, dès que possible, le déroutement du programme en cours (force le compteur ordinal à une valeur déterminée). Cette broche est testée à chaque instruction par l'automate de contrôle du processeur, lors de la phase recherche de la prochaine instruction.

L'entrée d'interruption peut être invalidée par un masque d'interruption (un indicateur dans le processeur), qui peut être activé ou désactivé par programme, rendant ainsi le traitement des interruptions effectif ou non.

Les processeurs possèdent généralement deux entrées d'interruptions : l'une est masquable (INT : "interrupt"), l'autre non (NMI : "non maskable interrupt") ; celle-ci provoque systématiquement le déroutement. Elle est habituellement réservée pour les situations d'urgence.



L'activité interrompue doit pouvoir reprendre dans l'état où elle était lors de la suspension. Cela nécessite une sauvegarde de tout ce qui caractérise cet état : au minimum le compteur ordinal, certains registres du processeur (généralement tous), et parfois même certaines mémoires de travail, qui seraient utilisées par convention comme des registres.

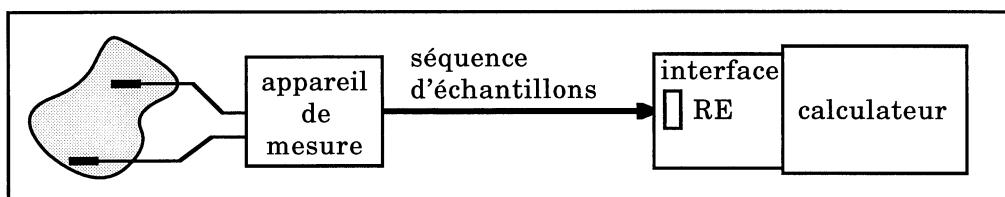
Le processeur fait automatiquement une telle sauvegarde, dans la pile courante ou dans une pile réservée au système. Certains processeurs ne sauvegardent que le compteur ordinal : les autres sauvegardes doivent alors être faites au début des procédures de traitement d'interruptions. D'autres assurent une sauvegarde plus complète du contexte : registres, indicateurs ...

Pour revenir à la tâche interrompue, il faut restaurer le contexte à la fin de chaque procédure de traitement d'interruption.

### 1.3 - Exemple d'utilisation : traitement de données en temps réel

Pour montrer l'intérêt du mécanisme d'interruptions, nous étudierons un exemple simple qui illustre les aspects importants.

Un appareil de mesures, connecté sur un phénomène réel, envoie des données au calculateur. Ces données, que nous appellerons "échantillons", sont émises de façon irrégulière et imprévisible. Chaque échantillon doit donner lieu à un traitement par le calculateur.

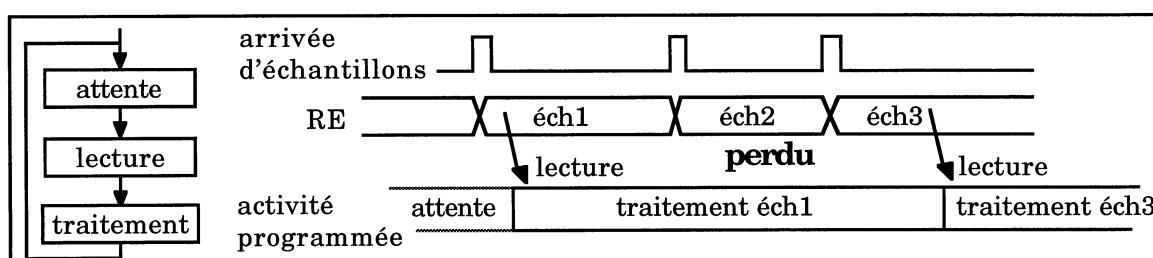


Nous faisons les hypothèses suivantes :

- la durée qui sépare l'arrivée de deux échantillons est supérieure à 0.01 s ;
- il n'arrive pas plus de 100 échantillons en 100 secondes ;
- le traitement d'un échantillon nécessite moins de 1 s.

Le débit des échantillons étant inférieur à 1 par seconde en moyenne, cela est réalisable sans perte d'information.

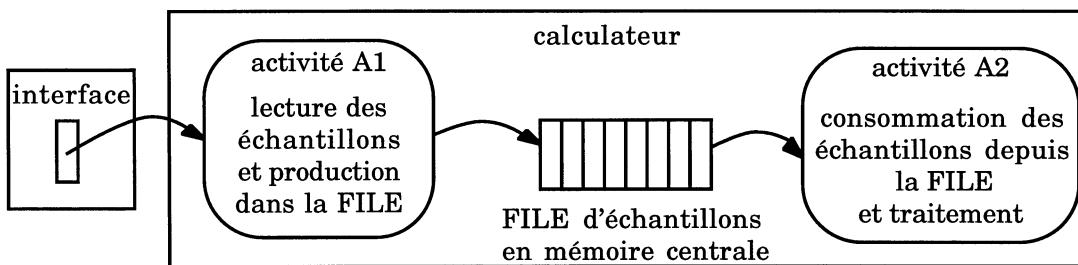
Cependant, si on attend un échantillon entre chaque traitement, certains échantillons risquent d'être perdus. C'est le cas s'il arrive plusieurs échantillons pendant un traitement (l'interface ne dispose que d'un seul registre d'entrée, RE).



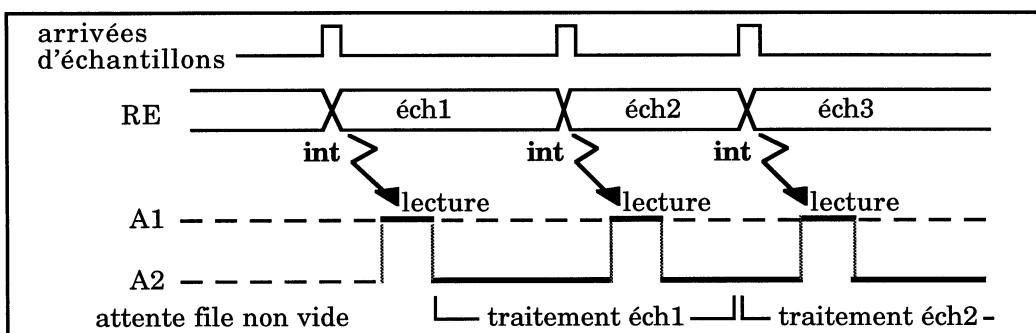
Pour résoudre ce problème de débit irrégulier, on utilise une file (FIFO) pour mémoriser les échantillons produits mais pas encore traités. On pourrait utiliser une file matérielle, localisée dans l'interface, mais on préfère la réaliser par logiciel dans la mémoire centrale, car c'est moins coûteux et plus souple (quand c'est possible, il vaut toujours mieux utiliser du logiciel plutôt que du matériel : c'est plus simple et plus fiable).

Le calculateur supporte alors deux activités programmées :

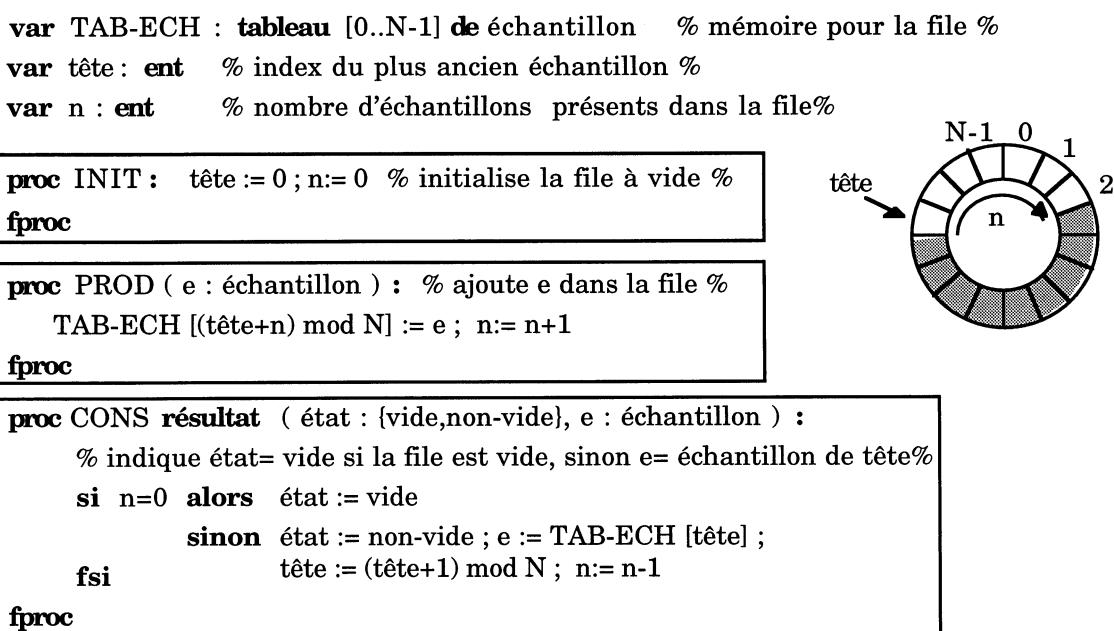
- Une activité A1 (rangement de l'échantillon dans la file), déclenchée par une interruption chaque fois que l'interface reçoit un échantillon.
- Une activité A2 (consommation des échantillons de la file et traitement).



Le diagramme temporel suivant illustre ce fonctionnement :



La file peut être réalisée selon n'importe quelle technique convenable, par exemple selon la méthode classique de gestion circulaire d'un tableau :



La programmation des activités A1 et A2 est indiquée sur la figure suivante. L'aspect important à remarquer ici est que les deux activités sont relativement indépendantes. L'exécution des instructions de A1 peut survenir à tout moment par rapport à l'exécution des instructions de A2, puisque l'activation de A1 dépend des conditions extérieures.

```
A1 : traitement d'interruption % arrivée d'échantillon %
    var éch : échantillon ;
    éch := lire-interface ;
    PROD (éch)
fin interruption
```

```
A2 : var état : {vide, nonvide} ;
    var éch : échantillon ;
    jusque toujours faire % boucle éternelle %
        jusque échantillon-obtenu faire % prélève un échantillon%
            état, éch := CONS ;
            quand état = nonvide sortir
        fait ;
        TRAITEMENT (éch)
    fait
```

En l'absence de demande d'interruption, le processeur exécute les instructions de l'activité A2 : on dit que A2 est exécutée “au niveau normal” ou en “tâche de fond”.

Lors d'une demande d'interruption, le processeur termine l'instruction en cours, sauvegarde automatiquement certains registres en mémoire (au moins le compteur ordinal, mais parfois plus, cela dépend du processeur), et exécute alors les instructions de A1 (implantées à partir d'une adresse prédéfinie).

Dans les machines simples, chaque activation de A1 se fait à la même adresse, le début du programme de A1. Sur des machines plus sophistiquées, par exemple les 80286 et 80386 (en mode non-dégénéré), la réactivation d'une activité d'interruption restaure l'état des registres de la fin de sa précédente activation, y compris le compteur ordinal : on parle alors de “gestion matérielle de processus”, plutôt que de simple “mécanisme d'interruptions”.

Pour le processeur, la prise en compte des interruptions se fait par test de son entrée INT, réalisé entre chaque exécution d'instruction.

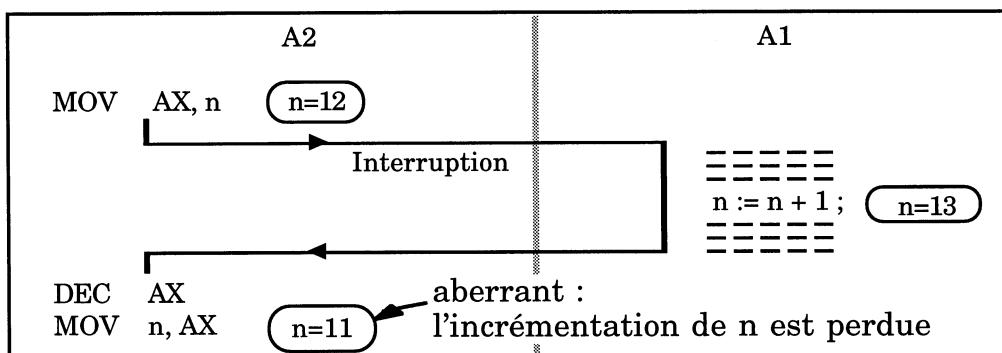
## 1.4- Masquage des interruptions - réalisation des exclusions

On ne maîtrise pas l'ordonnancement du mélange des instructions de A1 et de A2 : ceci pose un problème quand à la cohérence des variables partagées par les deux activités. Ici les variables partagées sont tout ce qui constitue l'état de la file. Si les deux activités manipulent la file “simultanément”, on court le risque d'avoir un fonctionnement incohérent. Il peut par exemple se passer le phénomène suivant :

Considérons dans CONS l'affectation “ $n := n - 1$ ” et dans PROD l'affectation “ $n := n + 1$ ”. On peut coder “ $n := n - 1$ ” en langage machine comme ceci :

dans CONS : $n := n - 1 ;$	MOV      AX, n      *** DEC      AX MOV      n, AX
----------------------------	--

Si une interruption a lieu au moment où l'activité A2 en est arrivée au point marqué par \*\*\*, il se produit la catastrophe suivante :



Avant l'interruption, A2 capte une partie de l'état de la file, la variable n. Là dessus, A1 met à jour correctement la variable n, et quand A2 continue, elle affecte à nouveau n en utilisant son ancienne valeur (qui ne reflète plus l'état présent de la file). On peut se demander “où est la faute ?”. Les programmes qui réalisent la file sont pourtant “corrects”. Ils le sont effectivement, mais à condition de considérer la file comme un tout, sans “morceaux”. Les composantes TAB-ECH, n, tête, forment un tout qui doit évoluer de façon coordonnée. Il faut donc considérer les actions PROD et CONS comme des actions “atomiques”, indivisibles.

Les actions sur la file doivent être une suite de PROD et CONS, sans mélange de ces actions (au plus une activité est en train d'agir sur la file) : on appelle cela assurer **l'exclusion mutuelle** sur les manipulations de la file.

La technique usuelle pour assurer cette exclusion est le masquage des interruptions. Il suffit que A2 inhibe les interruptions de la machine pendant qu'il exécute une action sur la file (CONS dans notre exemple) : ainsi on est certain que A1 ne peut pas s'exécuter pendant cette période. Le problème inverse, empêcher que A2 manipule la file pendant que A1 la manipule, est d'emblé résolu car l'activité A2 est suspendue tant que A1 n'a pas terminé (un mécanisme d'interruptions est une attribution disymétrique du processeur : les traitements d'interruption sont prioritaires).

Le processeur dispose de deux instructions pour autoriser et interdire la prise en compte des interruptions :

**AUTORISE-INT** : autorise la prise en compte des interruptions,

**INTERDIT-INT** : interdit la prise en compte des interruptions.

Ces instructions agissent sur le bit d'état du processeur appelé "masque des interruptions". Le processeur ne teste l'entrée INT entre chaque instruction que si le masque est dans l'état "validé".

Selon les machines, ces instructions s'appellent **EI, STI, ...** ("enable interrupt", "set interrupt") pour l'autorisation et **DI, CLI, ...** ("disable interrupt" ou "clear interrupt") pour l'interdiction.

La programmation correcte de A2 sera alors :

```

A2 : jusqu toujours faire
      jusqu échantillon-obtenu faire
          INTERDIT-INT ;
          état, éch := CONS ;    % accès exclusif à la file%
          AUTORISE-INT ;
          quand état = nonvide   sortir
          fait ;
          TRAITEMENT (éch)
      fait
  
```

Une prise en compte d'interruption peut donc être retardée à cause du masquage des interruptions. La demande d'interruption doit donc être mémorisée dans l'interface, et maintenue tant qu'elle n'est pas prise en compte.

Les séquences ininterruptibles doivent cependant rester relativement brèves : dans notre exemple, plus brèves que la durée minimum entre deux arrivées d'échantillons (sinon on risque à nouveau d'en perdre).

C'est habituellement l'action sur l'interface (effectuée par le programme d'interruption) qui fait disparaître la demande. Quand le processeur prend en compte une demande, il masque automatiquement les interruptions pour ne pas être aussitôt ré-interrompu (par la même demande).

Par exemple pour une interface d'entrée, c'est la lecture de la donnée qui provoque la suppression de la demande d'interruption (c'est donc le bit d'état indiquant la présence d'une donnée en entrée).

Dans les systèmes simples, les interruptions restent inhibées jusqu'à la fin du traitement d'une interruption. Avant de rendre le contrôle à l'activité suspendue, les interruptions doivent être validées à nouveau afin de recevoir la prochaine interruption, soit explicitement par une instruction **AUTORISE-INT**, soit automatiquement par l'instruction qui rend le contrôle en fin d'interruption (**IRET** sur 8086) :

```

A1 : traitement d'interruption
    var éch : échantillon ;
    % interruptions automatiquement inhibées %
    éch := lire-interface ;
    PROD (éch) ;
    AUTORISE-INT
fin interruption

```

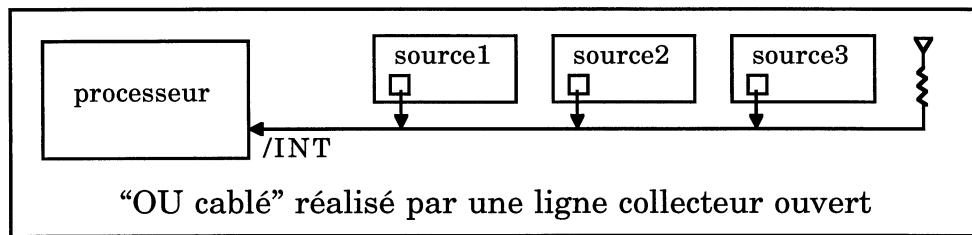
## 1.5 - Sources d'interruptions multiples

Un système comprend normalement plusieurs sources d'interruptions, pour divers événements, et chacune nécessite l'exécution d'un programme spécifique ; il faut donc pouvoir identifier les sources.

### 1.5.1 - Scrutation programmée des sources

Dans un mécanisme d'interruptions rudimentaire, il n'y a qu'un seul programme général de traitement d'interruptions, dont l'adresse est connue du processeur. Ce programme lit séquentiellement les bits d'états des interfaces pour déterminer une source active (il peut y en avoir plusieurs), et appelle la procédure associée à cette source : c'est la scrutation programmée des sources.

Sur le plan matériel, la ligne #INT de demande d'interruption est active à "0" ; les sources sont connectées via une ligne à collecteur ouvert ("OU cablé") : la demande d'interruption est active quand une source au moins est active.



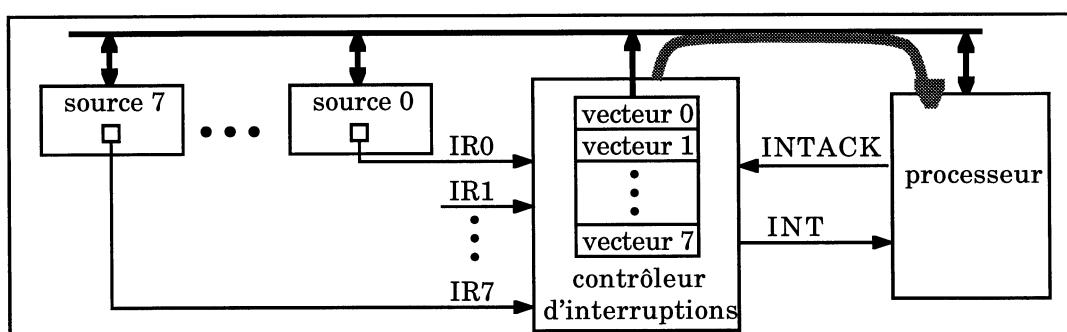
### 1.5.2 - Interruptions vectorisées

Le mécanisme des interruptions vectorisées est beaucoup plus élaboré ; le processeur identifie automatiquement la source active et appelle directement le programme associé à cette source.

- A chaque source est associée un numéro, son “vecteur d’interruption” .
- A un emplacement prédéfini (connu du processeur) de la mémoire centrale, on trouve une table des procédures d’interruption, qui contient les adresses des programmes associés à chaque source.
- Quand une interruption est prise en compte, le processeur exécute un cycle “d’acquittement d’interruption” (cycle bus spécial), qui lui fournit (via le bus de données) le vecteur d’interruption d’une source active. Le processeur indexe la table avec cette valeur, et lance la procédure ainsi désignée (saut indirect).

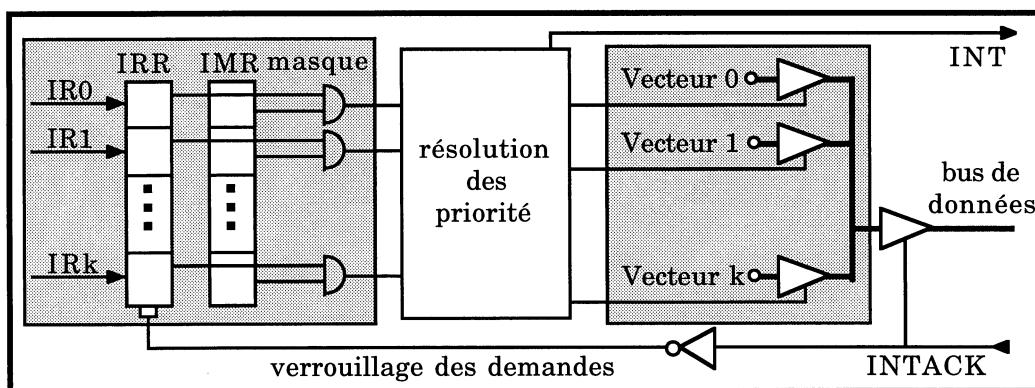
Pour choisir une seule source lorsque plusieurs sources sont actives simultanément, une logique matérielle de priorité assure que seul le vecteur de la source active la plus prioritaire est présentée sur le bus pendant le cycle de prise en compte d’interruption.

Sur la plupart des machines, les vecteurs d’interruption sont fournis par un composant spécifique, le “contrôleur d’interruptions”. Un tel contrôleur reçoit les diverses sources d’interruptions (IR0... IR7), et génère la demande d’interruption INT. Le processeur répond par INTACK pour recevoir le vecteur de la source active de plus forte priorité.



### 1.5.3 - Fonctionnement d'un contrôleur d'interruption

Un contrôleur d'interruptions doit gérer les demandes d'un certain nombre de sources : assurer la transmission des demandes, sélectionner la plus prioritaire lorsque le processeur active son cycle de reconnaissance d'interruptions, éventuellement gérer des niveaux d'interruptibilité (une interruption en cours de service peut-elle être interrompue par une autre ?). Il comprend pour cela trois parties, qui assurent respectivement la gestion des demandes en cours ou en attente, la résolution des priorités, et la génération des vecteurs d'interruption.



Les demandes d'interruptions  $IR_0..IR_k$  sont présentées en entrée d'un verrou IRR (interrupt request register). Un registre IMR (interrupt mask register), accessible par programme, permet de masquer individuellement les sources (en plus du masquage général effectué par le bit d'interruptibilité au niveau du processeur). Une demande d'interruption (INT) est émise si une source non masquée est active. Lorsque le processeur prend en compte l'interruption, il génère INTACK ce qui verrouille le registre IRR (pour que le circuit de priorité puisse effectuer le calcul du vecteur à partir d'un jeu de demandes stables). Le vecteur est présenté sur le bus de données.

Deux mécanismes de priorité sont généralement offerts :

- priorité fixe, définie par le numéro de la source ;
- priorité tournante : les sources sont rangées “circulairement” ; quand une source est prise en compte, sa suivante devient la plus prioritaire.

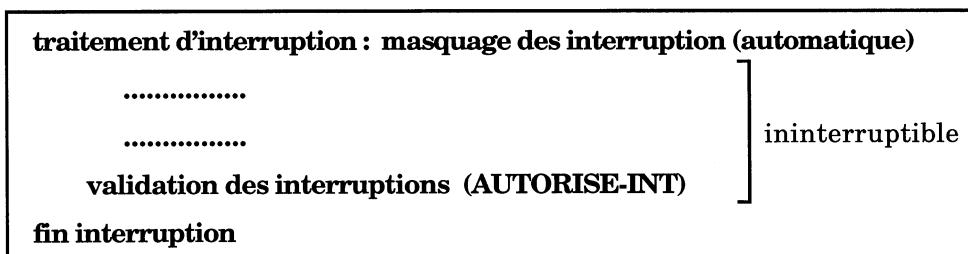
Ceci garantit que toute demande est prise en compte en un temps borné par la somme des durées maximales de traitement des autres sources (dans le cas de la priorité fixe, quelques sources peuvent monopoliser le temps disponible et ne jamais laisser passer les sources moins prioritaires).

### Traitement séquentiel ou prioritaire des interruptions

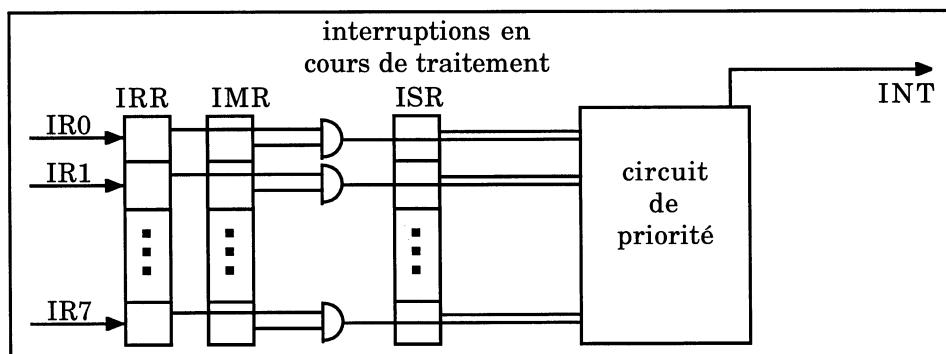
On peut gérer les traitements d'interruption de deux façons :

- séquentiellement : une interruption est toujours traitée complètement avant d'en commencer une autre ;
- prioritairement : les demandes sont assorties d'un niveau de priorité ; un traitement d'interruption plus prioritaire interrompt un traitement en cours.

L'organisation séquentielle, la plus simple, est souvent suffisante. Le circuit de priorité ne sert alors qu'à choisir une source parmi plusieurs au moment de la prise en compte. Le traitement séquentiel des interruptions est assuré par le masque général interne au processeur. Les interruptions, masquées lors de la prise en compte d'une interruption, ne sont validées à nouveau qu'à la fin du traitement d'interruption.



Dans le cas d'un traitement prioritaire, le contrôleur d'interruptions doit connaître le niveau de priorité de l'interruption en cours de traitement, afin de ne laisser passer que les interruptions plus prioritaires. Il faut un registre de plus dans le contrôleur, le registre des interruptions en cours de traitement ("ISR" : interrupt service register).

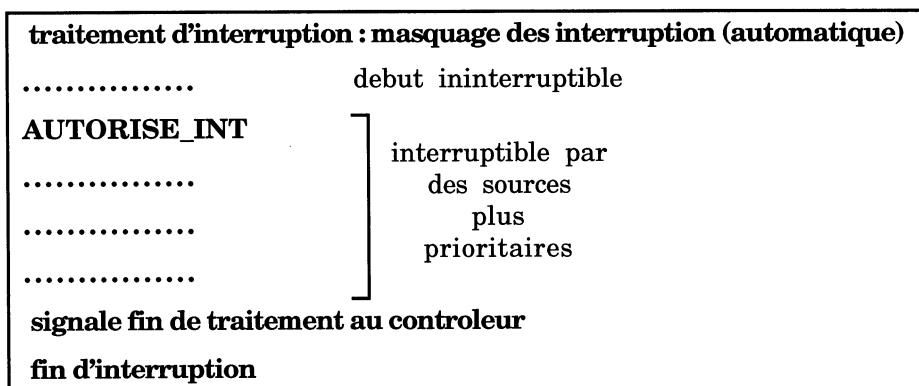


Le contrôleur n'émet une interruption (INT) que pour une demande non masquée plus prioritaire que celles enregistrées dans ISR. Lors de la prise en compte (INTACK), le bit correspondant dans ISR est mis à "1".

Le contrôleur doit être prévenu de la fin des traitements d'interruption : à ce signal, le bit de plus forte priorité de ISR est remis à “0”.

Le signal de fin de traitement d'interruption peut être généré automatiquement par le processeur (lors de l'instruction de fin de traitement d'interruption), ou par une commande émise explicitement par programme, à une adresse de port permettant l'accès au contrôleur (cas du PIC 8259 utilisé avec les processeurs Intel).

Pour que le système de traitement prioritaire soit effectif, toute procédure d'interruption doit revalider les interruptions le plus tôt possible ; elle fait ensuite son travail normal (transfert d'une donnée par exemple), pendant lequel la machine est interruptible par les sources plus prioritaires. Enfin elle signale la fin de traitement avant de terminer pour rendre à nouveau possible la prise en compte d'interruptions de plus faible priorité.



## 2 - Interruptions du 8086

Nous décrivons ici le fonctionnement des interruptions sur un 8086 doté d'un contrôleur d'interruptions PIC 8259.

### 2.1 - Table des procédures d'interruptions

La table des procédures (ou “routines”) d’interruption du 8086 est implantée à partir de l’adresse 0 (“00000 H”). Elle possède 256 entrées de quatre octets, contenant les adresses des procédures, sous la forme de deux mots : déplacement (IP) et segment de code (CS).

000000	IP0	CS0	INT0 division par zéro
000004	IP1	CS1	INT1 pas à pas
000008	IP2	CS2	INT 2 NMI (interruption non masquable)
00000C	IP3	CS3	INT 3 (appel système codé sur un seul octet)
000010	IP4	CS4	INT 4 overflow
<hr/>			
000020	IP8	CS8	INT 8 IR0
000024	IP9	CS9	INT 9 IR1
000028	IP10	CS10	INT10 IR2
<hr/>			
00003C	IP15	CS15	INT15 IR7
000040	IP16	CS16	INT16
000044	IP17	CS17	INT 17
<hr/>			
0003FC	IP255	CS255	INT 255

interruptions externes  
(cas du PC - XT)

interruptions logicielles  
(cas du PC - XT sous MS-DOS)

Les huit premiers emplacements sont réservés par le constructeur pour des usages bien déterminés (en fait, les cinq premiers sont effectivement spécifiés). Les autres sont disponibles pour le système ou l’utilisateur.

NMI correspond une à interruption non masquable (détectée sur une broche spéciale du processeur), pour des traitements d’urgence absolue.

Les “division par zéro” et “overflow” sont provoqués par certaines conditions lors de l’exécution d’instructions (statut un peu particulier : ces événements sont provoqués par le processeur lui-même, donc sans indéterminisme temporel ; on parle de “traitement d’exception”).

La procédure “pas à pas” est activée, en mode trace, à chaque instruction.

La procédure INT 3 est une procédure dont l’appel est codé sur un seul octet (“0CC h”), ce qui permet de réaliser les “points d’arrêt” pour la mise au point de programmes.

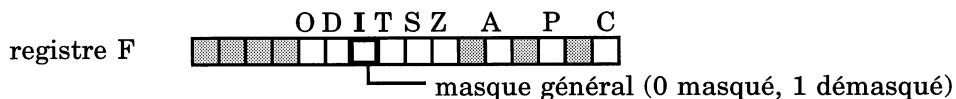
Les emplacements restant sont disponibles pour des procédures de traitement d'interruption.

En fait, le 8086 ne peut recevoir que 64 sources d'interruptions au plus (dont huit seulement sont connectées sur un PC\_XT, quinze sur AT et au-delà). L'exemple de la figure ci-dessus utilise les emplacements 8 à 15 pour le traitement des sources d'interruptions externes IR0 à IR7 : c'est ainsi que fonctionne le système MS-DOS pour les PC-XT.

Une instruction particulière du répertoire (mnémonique : "INT *nn*") permet d'activer toute procédure associée à un vecteur d'interruption (*nn* étant le numéro du vecteur, de 0 à 255). Cette instruction "d'interruption logicielle" est largement utilisée dans les systèmes d'exploitation pour l'accès aux procédures système (notamment sur PC, sous MS-DOS). Il faut bien comprendre qu'il ne s'agit pas réellement d'une interruption, mais simplement d'un saut "indirect" utilisant le mécanisme d'adressage spécial des instructions qu'est la table d'interruptions.

## 2.2 - Prise en compte des interruptions - Masque général

Sur le 8086, le masque d'interruption est un des bits du registre des indicateurs (le bit I du registre F).



Lors de la prise en compte d'une interruption, le 8086 sauvegarde le registre des indicateurs F, le registre segment de code CS et le compteur ordinal IP, au sommet courant de la pile (pointé par SS:SP), puis les interruptions sont automatiquement masquées (masque I mis à 0) avant de donner le contrôle à la procédure d'interruption. La procédure d'interruption doit sauvegarder les registres qu'elle utilise, par exemple dans la pile. En fin de programme d'interruption, le contrôle doit être rendu par l'instruction :

**IRET**   retour de traitement d'interruption

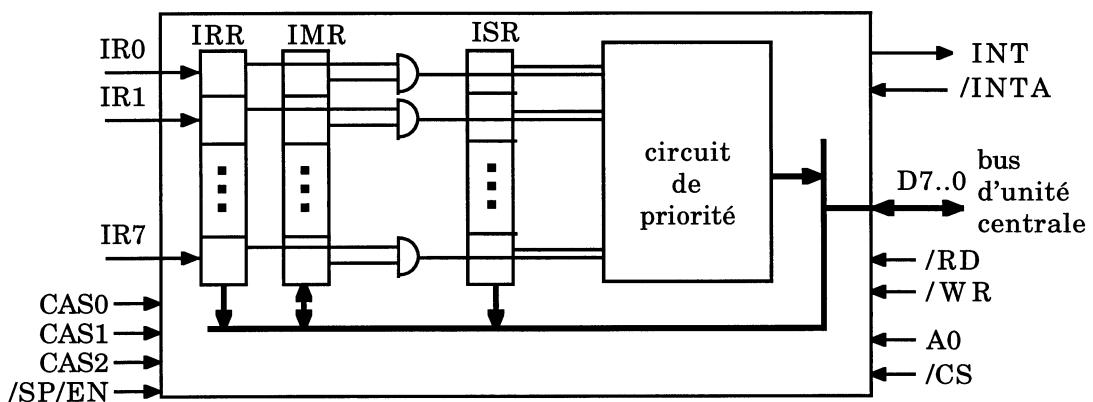
Cette instruction restaure les registres IP, CS et F (les interruptions sont donc démasquées automatiquement).

Des instructions permettent aussi de masquer et démasquer explicitement les interruptions : **CLI**   masque les interruptions (I:=0)  
**STI**   démasque les interruptions (I:=1)

## 2.3 - Contrôleur d'interruptions PIC 8259

Le contrôleur d'interruption PIC 8259 est spécifiquement dédié aux machines à base de processeur de la gamme Intel 8086.

Il reçoit 8 demandes d'interruptions IR0..IR7. Un registre de masque IMR permet de masquer individuellement les sources et un registre ISR indique les interruptions en cours de traitement.



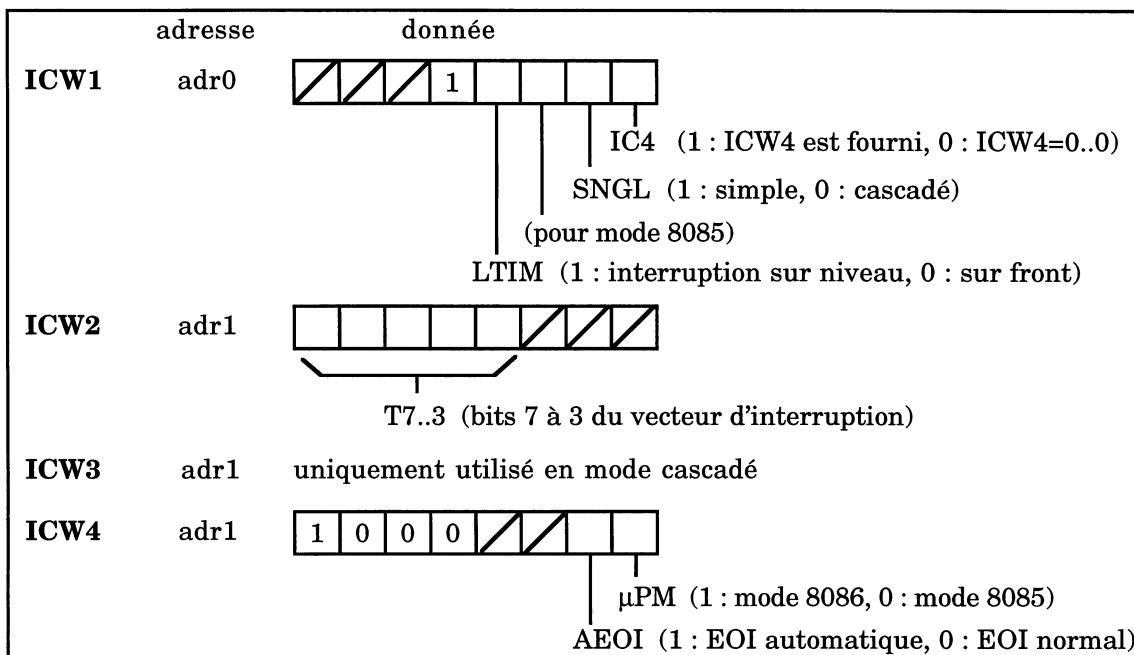
On peut utiliser plusieurs circuits PIC organisés hiérarchiquement en un “maître” et des “esclaves” (un seul niveau de hiérarchie), ce qui permet de gérer jusqu’à 64 sources d’interruptions ; nous ne décrirons pas ici ce mode, appelé “cascadé” (l’entrée /SP/EN configure le PIC en maître ou en esclave ; le PIC maître sélectionne l’esclave demandeur le plus prioritaire grâce aux broches CAS2..0).

Le PIC est prévu pour fonctionner avec un processeur 8080 et 8085 (mode 8085), ou bien avec un 8086, 8088, 80286, ... (mode 8086). Nous ne décrirons ici que les options du mode 8086.

Le PIC est accessible par deux adresses adr0 et adr1 distinguées par le bit A0. Dans un système, ces adresses sont généralement des adresses de port d’entrée-sortie. Elles permettent :

- d’envoyer des commandes au PIC ; il y a quatre commandes d’initialisation (ICW1, ICW2, ICW3, ICW4) et trois commandes de fonctionnement (OCW1, OCW2, ICW3) ;
- d’écrire ou lire le masque IMR ;
- de lire les registres IRR et ISR.

## Adresse et format des commandes



Remarque : ces commandes se distinguent les unes des autres par une combinaison “subtile” des adresses et des bits de données. Par exemple, ICW1 est à écrire à l’adresse adr0, le bit de donnée D4 valant “1”. Après ICW1, le premier mot écrit à l’adresse adr1 est interprété comme étant ICW2 ; les suivants éventuels sont ICW3 (si SNGL=0) puis ICW4 (si IC4=1).

IC4 = 1 signifie que l’on fournit ICW4, sinon ICW4 est nul par défaut.

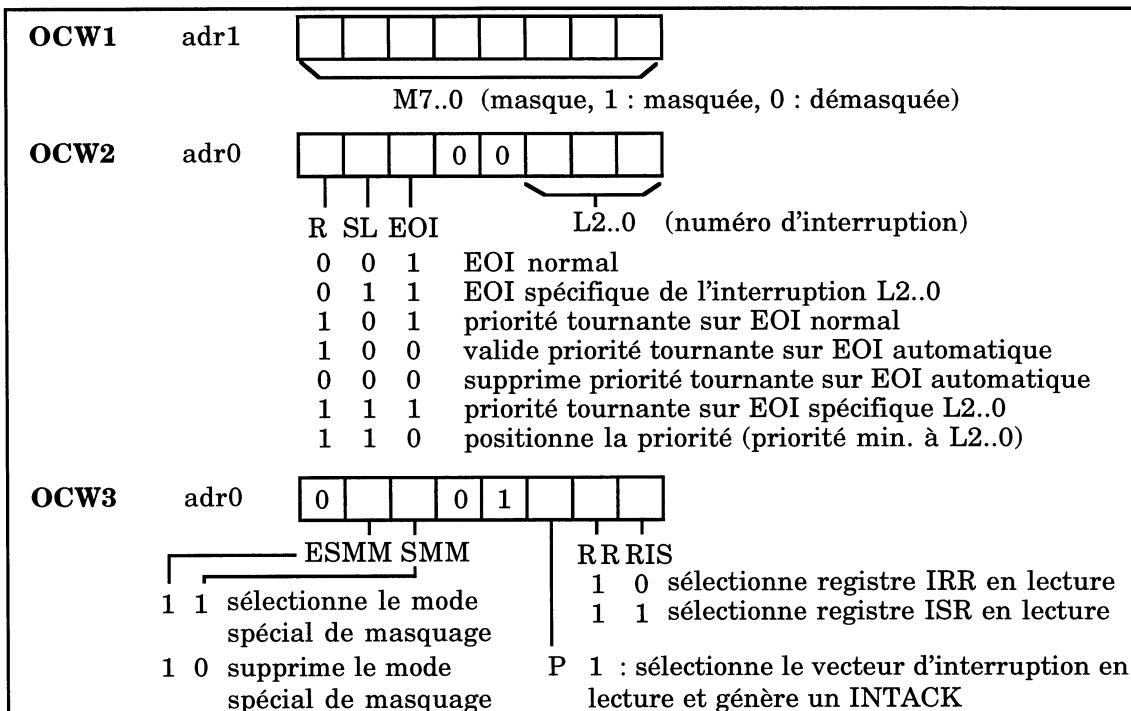
SNGL = 1 sélectionne le mode simple, sinon le mode cascadé est choisi.

LTIM = 1 signifie que les demandes d’interruption IR0..7 sont des niveaux (level mode), sinon ce sont des fronts montants (trigger mode).

T7..3 sont les 5 bits de poids fort des vecteurs d’interruption (les 3 bits de poids faibles T2..0 sont déterminés par le numéro de la demande d’interruption). Ceci fixe les numéros des interruptions matérielles dans la table des interruptions ; par exemple T7..3 = 00001 attribue les numéros 8 (00001000) à 15 (00001111) aux interruptions matérielles.

µPM fixe le mode de fonctionnement (8085 ou 8086).

AEOI = 1 spécifie la génération automatique de la “fin de traitement” d’une interruption lors de la prise en compte de cette interruption. Ceci est utilisé en traitement séquentiel des interruptions (dans ce mode, la priorité de traitement est sans signification) et cela évite de devoir signaler explicitement la fin de traitement (par EOI).



La commande OCW1 provoque l'écriture du registre de masque IMR.

La commande OCW2 permet de signaler explicitement la fin de traitement d'interruption. Il y a deux façons de signaler une fin : le signalement normal, sans mention de numéro d'interruption, indique la fin de traitement de l'interruption en cours de plus grande priorité ; le signalement spécifique précise explicitement le numéro de l'interruption qui se termine.

La commande OCW3 permet également de contrôler le mécanisme de priorités (peu utilisé) : on peut notamment demander au PIC de fonctionner en "priorités tournantes", avec diverses variantes.

La commande OCW3 permet de préparer la lecture soit de IRR (RR,RIS=10), soit de ISR (RR,RIS=11) : après une telle commande, la prochaine lecture du PIC provoque la lecture du registre indiqué. Le bit P=1 permet de même de préparer la lecture explicite du vecteur d'interruption, avec en plus simulation d'un acquittement d'interruption (peu utilisé). Les bits ESMM et SMM concernent le positionnement d'un mode de masquage spécial dans lequel le démasquage d'une interruption autorise sa prise en compte, même si elle est moins prioritaire qu'une interruption en cours de traitement.

#### Lecture du masque IMR :

Le registre IMR est accessible en lecture à l'adresse adr1. Pour changer un bit du masque en laissant les autres inchangés, il faut lire le masque, changer le bit (instruction logique avec masquage), puis ré-écrire le masque.

## 2.4 - Exemple d'application

### 2.4.1 - Initialisation du PIC

Sur un PC-XT, le PIC est implanté aux adresses de port PIC0=20h et PIC1=21h.

On veut initialiser le PIC pour satisfaire aux options suivantes : les demandes d'interruptions sont des fronts, on désire des priorités de traitement fixes, les numéros associés aux interruptions matérielles sont 8..15, et initialement toutes les interruptions doivent être masquées.

PIC0	EQU	20H	
PIC1	EQU	21H	
ICW1	EQU	0001 0011B	; LTIM=1 (front), SNGL=1 (non cascadé), IC4=1
ICW2	EQU	08H	; vecteurs d'interruption 8..15
ICW4	EQU	1000 0001B	; AEOI=0 (EOI normal), $\mu$ PM=1 (mode 8086)
OCW1	EQU	1111 1111B	; Masque d'interruption : toutes masquées
EOI	EQU	0010 0000B	; Commande de fin d'interruption (EOI normal)

```
CSEG      ; Initialisation du PIC
          ; Remarque : l'UC doit être ininterruptible
          ; pendant l'initialisation du PIC
```

```
PICINIT : PUSH AX
          MOV AL,ICW1
          OUT PIC0,AL
          MOV AL,ICW2
          OUT PIC1,AL
          MOV AL,ICW4
          OUT PIC1,AL
          MOV AL,OCW1
          OUT PIC1,AL
          POP AX
          RET
```

### 2.4.2 - Utilisation

Une horloge génère une interruption toute les 10ms sur l'entrée IR3. On veut l'utiliser pour gérer un clignotant de période 2 secondes (une seconde allumé, une seconde éteint). La lampe du clignotant est commandée par une bascule accessible en écriture en tant que bit 0 à l'adresse de port 312h.

On veut rédiger les procédures suivantes (pour 8086) :

- CLIGNOTANT\_DEBUT mise en fonctionnement du clignotant

La mise en marche est réalisée en démasquant les interruptions de l'horloge (sans modifier les autres bits du masque).

- HORLOGE\_TOP procédure de l'interruption d'horloge

- CLIGNOTANT\_FIN arrêt du clignotant (la lampe reste éteinte).

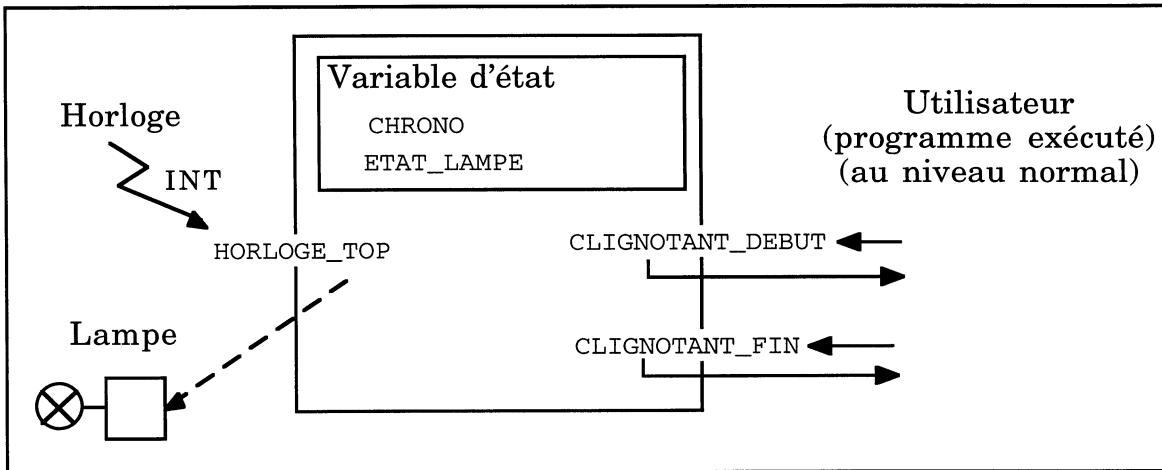
Il faudra notamment indiquer l'emplacement de la table des interruptions à modifier, en donnant la valeur qui doit y figurer, sachant que le segment de code des programmes précédents vaut 2000h.

Une application qui utilise les interruptions à toujours une structure similaire à celle illustrée sur le schéma suivant. C'est essentiellement un "automate" qui réagit à diverses sollicitations :

- De la part de l'utilisateur interne, à travers les procédures de services qui lui sont offertes , CLIGNOTANT\_DEBUT et CLIGNOTANT\_FIN ici.
- De la part de l'environnement, sous forme de demandes d'interruptions : ici, les tops de l'horloge, qui provoquent l'exécution de la procédure d'interruption HORLOGE\_TOP.

Le plus important dans ce type de problème, est de définir l'état de cet automate, c'est-à-dire définir les variables d'état (et leur rôle). Ces variables sont appelées "variables d'état" car elles conservent un état "intéressant" entre les activations successives des procédures du module.

Une brève analyse nous permet d'inventer ces variables d'état. Il faut constituer des périodes d'une seconde à partir de tops espacés de 10ms, et pour cela il nous faut donc compter (ou décompter) ces tops. La première variable sera donc un décompteur (de 100 à 0) que nous baptisons CHRONO. Ensuite, il faut connaître l'état de la lampe, pour décider de l'éteindre si elle est allumée et l'allumer si elle est éteinte. On pourrait rendre la bascule LAMPE accessible en lecture, mais il n'est jamais judicieux de faire du matériel si on peut résoudre le problème par logiciel (rappel : le matériel tombe en panne, le logiciel jamais). Il vaut donc mieux gérer en mémoire un double de l'état de la lampe, dans une variable ETAT\_LAMPE.



; variables d'état

#### DSEG

```

ETAT_LAMPE RB 1           ; état allumé-éteint de la lampe
ALLUME    EQU   01H
ETEINT    EQU   00H
CHRONO   RB    1           ; décompteur pour mesurer une seconde (100x10ms)

LAMPE     EQU   312H       ; adresse de port de la lampe

```

#### CSEG

```

DATASEG R W 1           ; pour mémoriser le segment de données de l'application

DEMASQUE_IR3   EQU 11110111B ; pour démasquer interruption de l'horloge
MASQUE_IR3     EQU 00001000B ; pour masquer interruption de l'horloge
CLIGNOTANT_DEBUT : PUSH AX ; PUSH DX
    MOV CS:DATASEG,DS ; capte le DS de l'application
    MOV AL,ALLUME
    MOV DX,LAMPE
    OUT DX,AL          ; allume la lampe
    MOV ETAT_LAMPE,ALLUME ; note état allumé
    MOV CHRONO,100      ; initialise le décompte d'une seconde
    PUSHF
    CLI                 ; le PIC doit être manipulé sous ininterruptibilité
    IN AL,PIC1; lecture du masque IMR
    AND AL,DEMASQUE_IR3
    OUT PIC1,AL         ; démasque interruption d'horloge
    POPF                ; restaure interruptibilité
    POP DX ; POP AX ; RET

```

```

HORLOGE_TOP : PUSH AX; PUSH DX; PUSH DS
    MOV  AX,CS:DATASEG ; installe le DS de l'application
    MOV  DS,AX
    STI  ; autorise interruptions plus prioritaires (sans grand intérêt)
    DEC  CHRONO
    JNZ  FIN
    XOR  ETAT_LAMPE,ALLUME ; complémente l'état de la lampe
    MOV  AL,ETAT_LAMPE
    MOV  DX,LAMPE
    OUT  DX,AL      ; change l'état de la lampe
    MOV  CHRONO,100 ; réinitialise le décompte d'une seconde
FIN :   MOV  AL,EOI      ; signale fin de traitement d'interruption
        OUT PIC0,AL
        POP  DS; POP DX; POP AX; IRET

```

```

CLIGNOTANT_FIN : PUSH AX ; PUSH DX ; PUSHF
    CLI           ; le PIC doit être manipulé sous ininterruptibilité
    IN  AL,PIC1; lecture du masque IMR
    OR  AL,MASQUE_IR3
    OUT PIC1,AL   ; masque interruption d'horloge
    POPF          ; restaure interruptibilité
    MOV  AL,ETEINT
    MOV  DX,LAMPE
    OUT  DX,AL      ; éteint la lampe
    POP  DX; POP AX; RET

```

### Initialisation de la table des interruptions

L'adresse de la procédure d'interruption HORLOGE\_TOP doit être inscrite à l'emplacement 11 (8+3 car l'horloge est branchée sur IR3) de la table des interruptions (adresse 00002Ch). Si CS vaut 2000h, on aura :

00002C	HORLOGE_TOP	2000H
--------	-------------	-------

Cette initialisation aurait pu être faite, par exemple, dans CLIGNOTANT\_DEBUT, en y ajoutant quelques instructions.

```
MOV AX,0
MOV ES,AX      ; ES:=0, pour atteindre le début de la mémoire
MOV SI, 002Ch
MOV ES: 0[SI], OFFSET HORLOGE_TOP
MOV ES: 2[SI], CS
```

Pour remplir un élément de la table d'interruptions on peut également, sous le système MS\_DOS, utiliser la procédure système INT 21h avec AH=25h.  
Ses paramètres sont :

AL : numéro de l'interruption,  
DS : segment de code de la procédure, DX : offset de la procédure

```
PUSH DS
MOV AX,CS
MOV DS,AX
MOV DX, OFFSET HORLOGE_TOP
MOV AL,11
MOV AH,25h
INT 21H
POP DS
```

# HIERARCHIE DE MEMOIRES

## 1 - Principes généraux

### 1.1 - Importance des performances de la mémoire

Les performances de la mémoire sont un facteur déterminant pour les performances globales d'un calculateur. Ces performances reposent sur deux aspects, à priori contradictoires : la rapidité et la grande taille.

#### Nécessité d'une mémoire à accès rapide

Le temps de cycle de l'horloge des processeurs actuels est souvent compris entre 20 et 50 ns (et, grâce à une exécution en "pipeline", la plupart des instructions sont exécutées en 1 ou 2 cycles). Chaque instruction nécessite au moins un accès mémoire (la lecture de l'instruction), plus éventuellement un ou plusieurs autres, s'il s'agit d'une instruction avec référence mémoire. Le fonctionnement d'un processeur est donc une séquence quasi ininterrompue d'accès à la mémoire.

On comprend donc aisément que l'usage d'un processeur rapide est sans intérêt si on doit le faire fonctionner avec une mémoire "lente" (temps d'accès de l'ordre de 200 ns par exemple).

#### Nécessité d'une mémoire de grande taille

Les systèmes, comme les applications, sont de plus en plus gourmands en mémoire : systèmes conviviaux avec fenêtrage, usage intensif de moyens graphiques, langages de programmation de haut ou très haut niveau (langages déclaratifs par exemple), applications sophistiquées de simulation, de conception assistée ...

On a de plus en plus fréquemment besoin de mémoires de 4 à 16 Méga octets ou plus. Avec la généralisation des systèmes multi-tâches et multi-usagers, la quantité de mémoire principale nécessaire à tout instant pour une station de travail s'approche même plutôt des 100 Méga octets ou plus.

## 1.2 - La hiérarchie : concilier rapidité et grande taille

Au niveau des composants mémoire, ces deux critères sont incompatibles :

- Les mémoires rapides (temps d'accès de l'ordre de 20ns) ont une taille réduite, quelques dizaines de kilo-octets.
- Les mémoires de grande taille (plusieurs Méga-octets), telles les mémoires dynamiques, ont des temps d'accès compris entre 100 et 200 ns. Quand aux mémoires de très grande taille (plusieurs centaines de Méga-octets), leur temps d'accès dépasse les 10 ms ( $10^7$  ns !).

Les accès à la mémoire principale ne sont pas totalement aléatoires (sinon, il n'y aurait pas de solution à ce problème) ; les accès mémoires engendrés par les programmes ont des propriétés de localité : la probabilité d'accès à une adresse dépend fortement des accès précédents. On distingue deux sortes de localité :

- La localité temporelle : les adresses récemment sollicitées ont une forte probabilité d'être sollicitées à nouveau.
- La localité spatiale : les adresses numériquement voisines d'adresses récemment sollicitées ont une forte probabilité d'être sollicitées.

La localité temporelle est "naturelle" : elle signifie qu'un programme s'occupe pendant un certain temps d'un même thème ; il a donc tendance à accéder plusieurs fois aux données et aux procédures propres à ce thème.

La localité spatiale est liée au fait que les structures de données concernant un même thème sont souvent contiguës en mémoire (les champs d'une structure, les éléments d'un tableau, sont rangés à des adresses successives).

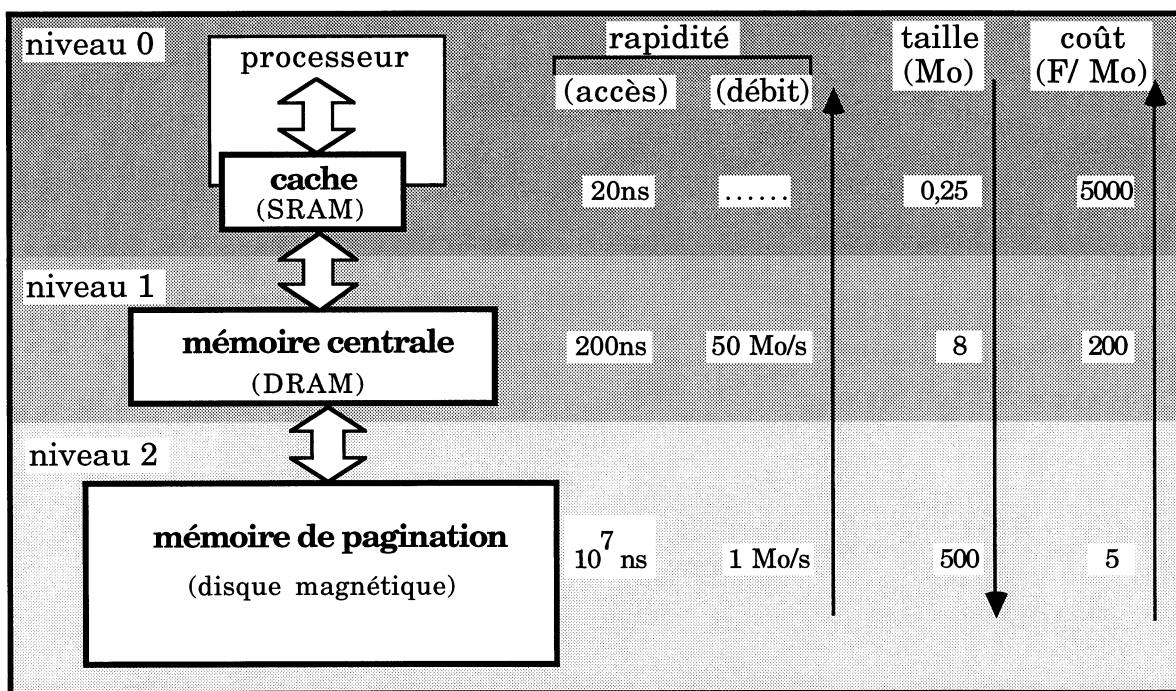
Une hiérarchie de mémoires comprend plusieurs "niveaux" de mémoires. Le sommet de la hiérarchie est une mémoire rapide (coût par bit élevé), mais de petite taille ; vers le bas, la hiérarchie est constituée de mémoires de plus en plus grandes, de moins en moins coûteuses, mais de plus en plus lentes.

L'idée générale est de retenir vers les niveaux hauts de la hiérarchie, proche du processeur, les données ayant la plus forte chance d'être utilisées. Grâce aux propriétés de localité, on espère donner ainsi l'impression d'une mémoire grande et rapide : une mémoire avec un temps d'accès moyen proche de celui de la mémoire de plus haut niveau, mais une taille et un coût proches de ceux des mémoires de plus bas niveau.

Conceptuellement, le nombre de niveaux est quelconque. Dans les systèmes courants, on a généralement trois niveaux :

- Le niveau 0, proche du processeur, est le cache. Il est réalisé par de la mémoire statique (SRAM), et il est parfois intégré au processeur.
- Le niveau 1, la mémoire centrale, est réalisé par des composants de mémoire dynamique (DRAM).
- Le niveau 2, la mémoire de pagination, est réalisé avec un disque magnétique.

En 1995, les ordres de grandeur sont à peu près les suivants :



On appellera dans la suite “mémoire centrale” la mémoire physique de niveau 1, réservant l’appellation “mémoire principale” pour désigner la mémoire globale vue du processeur (l’abstraction réalisée par le système de hiérarchie).

### Notions quantitatives

Etant donné un accès au niveau  $i$ , si le mot cherché s'y trouve, on l'appelle un succès, sinon on l'appelle un défaut. Le taux de succès et le taux de défaut sont la proportion des accès qui sont respectivement des succès et des défauts. Bien évidemment :

$$\text{taux de succès} + \text{taux de défaut} = 1$$

La gestion de la hiérarchie introduit un surcoût. Pour le minimiser, et pour profiter de la localité spatiale, l'unité de transfert d'information (on l'appelle un bloc) doit être plus grosse que l'unité adressable (l'octet). La taille du bloc peut avoir une influence sur les performances de la mémoire. Pour les caches actuels (1995), elle est comprise entre 4 et 64 octets.

En cas de succès, l'accès au niveau i suffit, le temps d'accès  $T_{si}$  est donc à peu près celui du composant de niveau i.

En cas de défaut, le mécanisme de hiérarchie déclenche un accès au niveau  $i+1$  pour transférer dans le niveau i le bloc complet où se trouve la donnée. Le temps d'accès (coût de défaut,  $T_{di}$ ) est plus long. Il est égal au temps d'accès du niveau  $i+1$  augmenté du temps de transfert du bloc. Etant donné qu'un bloc est constitué de k mots d'adresses successives, le temps de transfert du bloc est lié au débit offert par le niveau  $i+1$  (bien plus rapide que le temps de k accès aléatoires).

Par exemple, pour un disque, le temps d'accès est très long (10 ms :  $10^7$  ns), car il nécessite un positionnement mécanique de la tête de lecture, alors que le débit est relativement rapide (1 Moctet/s, soit 1 octet par 1000 ns).

En définitive, le coût de défaut  $T_{di}$ , est de l'ordre de grandeur du temps d'accès du niveau  $i-1$  (1 à 2 fois), et on a :

$$\text{temps d'accès moyen niveau } i = \text{taux de succès} \times T_{si} + \text{taux de défaut} \times T_{di}$$

### **Exemple**

Avec une hiérarchie bien conçue et des programmes "normaux", le taux de succès sera de 95%. Pour une hiérarchie à 2 niveaux, avec un cache de temps d'accès  $T_{s0} = 20$  ns et un coût de défaut  $T_{d0} = 300$  ns on aura :

$$\text{temps d'accès moyen à la mémoire} = 0,95 \times 20 + 0,05 \times 300 = 34 \text{ ns}$$

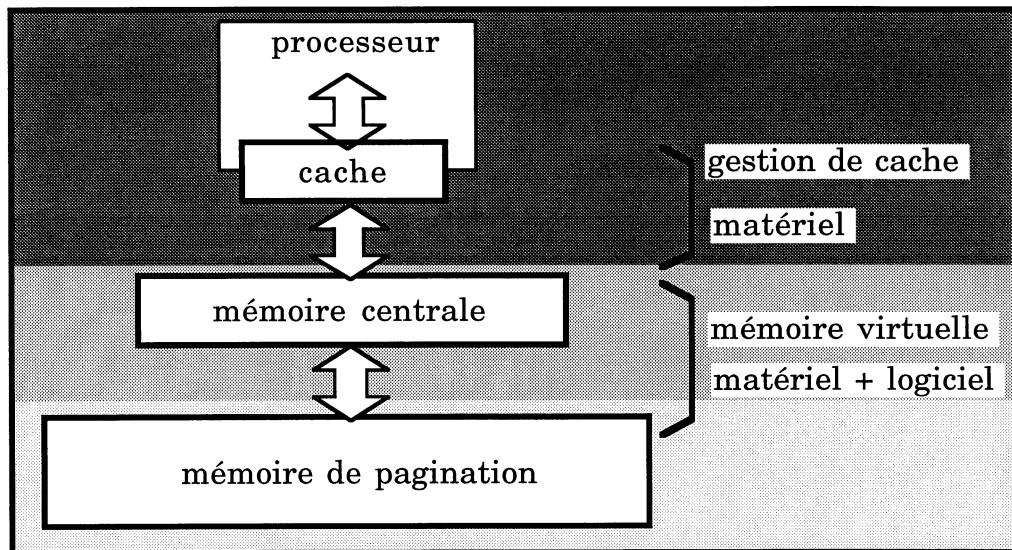
## **1.3 - Mécanismes : gestion de cache et mémoire virtuelle**

Aux trois niveaux usuels de hiérarchie correspondent deux mécanismes :

- La gestion de cache, qui s'occupe des accès au cache et des transferts entre la mémoire centrale et le cache.

- La gestion de mémoire virtuelle, ou pagination, qui s'occupe des accès à la mémoire virtuelle et des transferts entre le disque et la mémoire centrale.

D'un point de vue fonctionnel, ces deux mécanismes sont semblables. Cependant, à cause des aspects quantitatifs très différents (temps d'accès, tailles des mémoires, taux de défauts ...) la mise en œuvre est très différente.



- **Gestion de cache**

La mise en œuvre est nécessairement matérielle, car le temps d'accès de la mémoire centrale est tout de même assez court (quelques centaines de ns). De plus, le taux de défauts, bien que faible, n'est pas négligeable (de l'ordre de 0,05%). Une intervention logicielle pour le transfert des données entre mémoire principale et cache dégraderait complètement les performances.

- **Gestion de mémoire virtuelle** : la mise en œuvre est mixte (matérielle et logicielle). Le traitement des succès est assuré par des moyens matériels : circuits de traduction d'adresses qui, connaissant l'adresse (dite "virtuelle") d'une donnée, calculent son emplacement physique en mémoire centrale.

En revanche, le traitement des défauts est géré par du logiciel, qui transfère des blocs (appelés pages) entre le disque et la mémoire centrale, et met à jour les tables de traduction d'adresses. La gestion logicielle n'est pas pénalisante à ce niveau, car le temps d'accès au disque est long (10 ms, soit le temps d'exécution de 100 000 instructions sur un processeur moyen). De plus le taux de défaut est normalement plus faible que pour un cache (de l'ordre de 0,0001%). Rappelons à ce propos qu'une réalisation en logiciel est toujours préférable si elle est suffisante : le logiciel est moins cher, plus souple et plus fiable (le matériel tombe en panne, le logiciel jamais).

## 2 - Fonctionnement des caches

### 2.1 - Correspondance adresse-emplacement dans le cache

Le cache doit contenir les données, mais il doit également être capable de savoir si une donnée s'y trouve (et dans ce cas pouvoir y accéder).

Il existe deux techniques de correspondance entre les adresses de mémoire principale et les emplacements dans le cache.

- Les caches à correspondance directe : une adresse de la mémoire principale correspondra toujours au même emplacement dans le cache. Ces caches, les plus simples, suffisent généralement.
- Les caches associatifs par ensembles : une adresse de la mémoire principale peut correspondre à divers emplacements dans le cache. Ces caches sont plus complexes.

Pour éviter toute ambiguïté, nous adopterons la terminologie suivante :

adresse : adresse en mémoire principale (telle que la fournit le processeur)

bloc : entité élémentaire de transfert entre mémoire principale et cache (une puissance de 2 mots d'adresses contiguës).

emplacement : zone du cache qui peut recevoir un bloc.

index : adresse d'un emplacement dans le cache.

### 2.2 - Caches à correspondance directe

La taille d'un emplacement est une puissance de 2 ; la taille (nombre d'emplacements) du cache également.

Les bits de l'adresse peuvent être décomposés en trois champs :

- les poids faibles (“numéro”) spécifient le mot dans l'emplacement,
- les bits suivants ( “index”) désignent l'emplacement dans le cache,
- les poids forts (“étiquette”) permettent de déterminer si l'accès est satisfait.

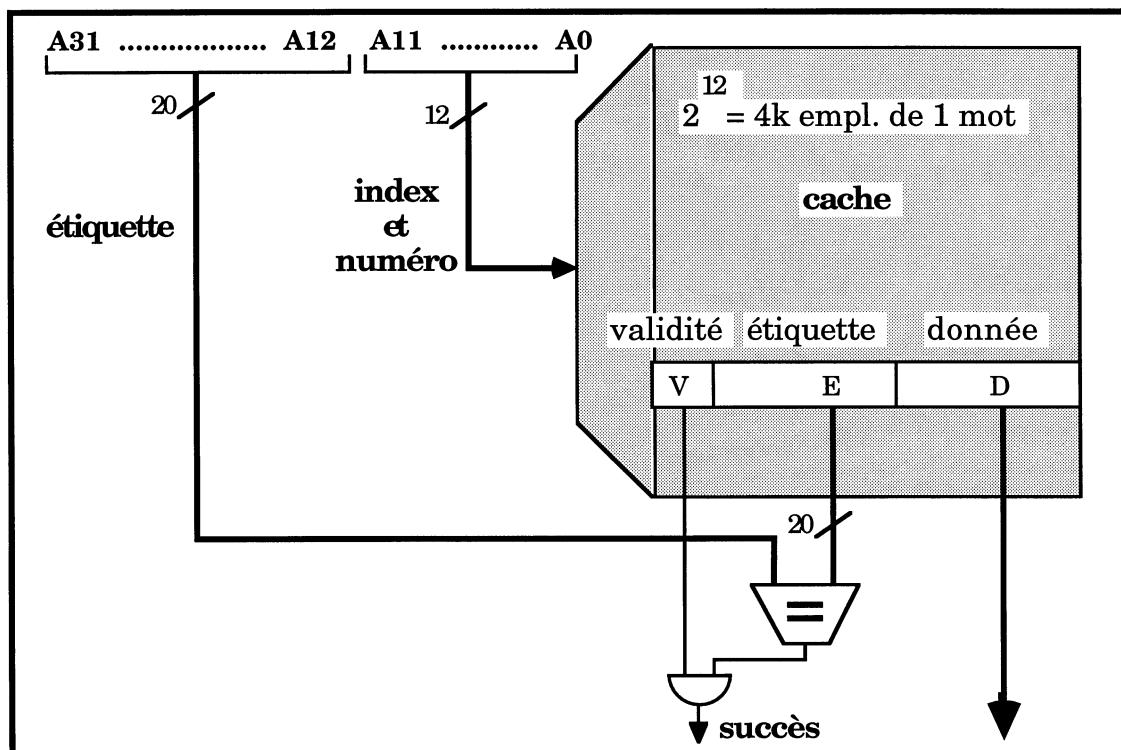
“étiquette”	“index”	“numéro”
-------------	---------	----------

Pour un cache de  $2^k$  blocs de  $2^p$  mots, le numéro est constitué des  $p$  bits de poids faibles, et l'index est formé des  $k$  bits suivants.

Pour savoir à quelle zone (de la mémoire centrale) appartient le bloc situé dans un emplacement, on associe, à chaque emplacement du cache, une étiquette (E) égale aux  $n - (k+p)$  bits de poids forts des adresses des données qui s'y trouvent ; un bit de “validité” (V) indique si des données de la mémoire principale sont effectivement présentes à cet emplacement (à l'initialisation, tous les emplacements sont invalides).

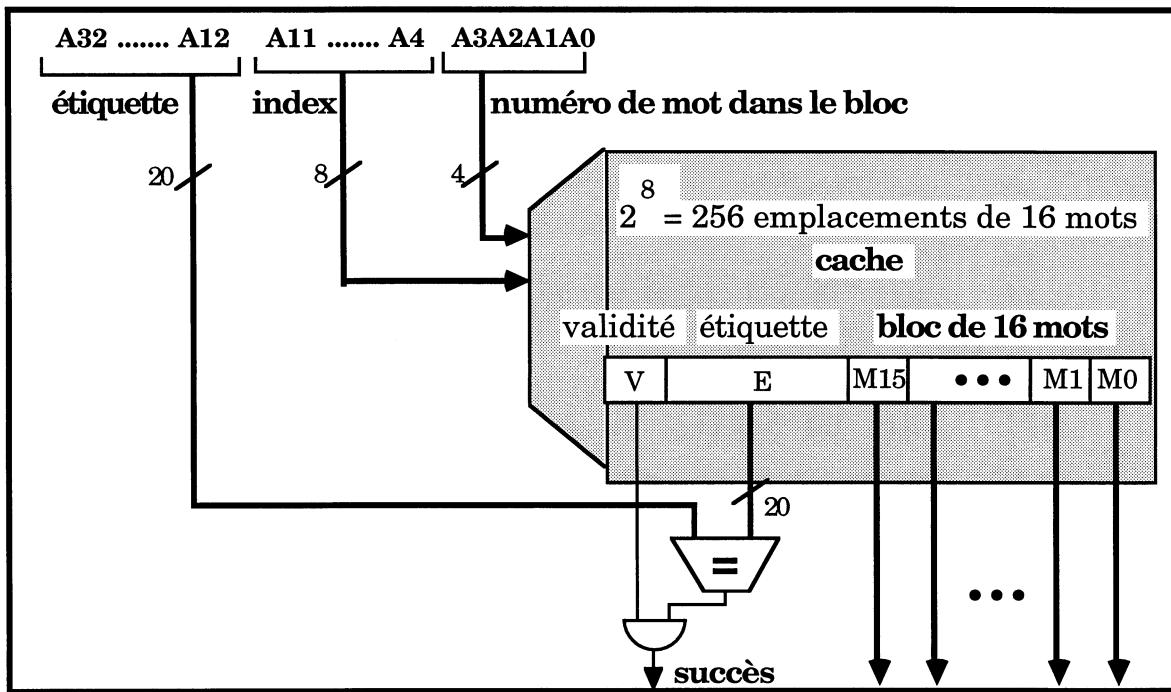
Dans le cas d'une lecture avec succès, le calcul est simple : les champs “index” et “numéro” de l'adresse désignent l'emplacement et le mot ; la validité de cet emplacement et l'égalité de son étiquette avec le champ “étiquette” de l'adresse sont testées. Si le test est positif (succès), la donnée est délivrée au processeur ; sinon (défaut), le processeur est suspendu le temps d'accomplir l'accès à la mémoire centrale.

Par exemple, pour un cache de 4k emplacements de 1 mot, l'index fait 12 bits (pas de bits de “numéro”), et l'étiquette 20 bits :



Le surcoût de gestion des emplacements du cache est inacceptable : il y a 21 bits de “contrôle” (V et E) pour un seul mot de données (D).

Pour diminuer ce surcoût en profitant de la localité spatiale, la taille du bloc serait par exemple de 16 mots (256 emplacements de blocs pour une même taille de 4 Ko). Le surcoût n'est plus alors que de 21 bits de contrôle pour 16 données. Dans ce cas les 4 bits de poids faibles de l'adresse sont le numéro du mot dans le bloc, et l'index ne fait plus que 8 bits (voir figure page suivante).



Dans un cache à correspondance directe, deux données avec le même index ne peuvent coexister dans le cache : elles sont en conflit pour occuper le même emplacement.

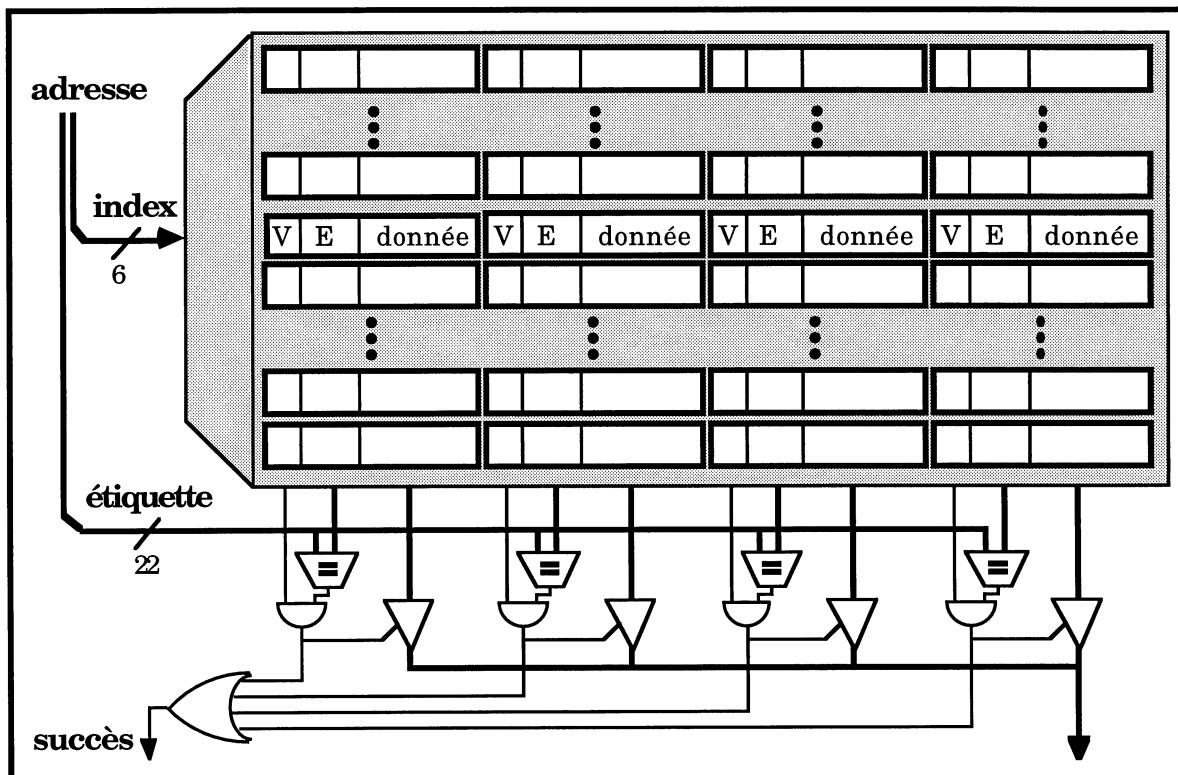
Une telle correspondance est parfois trop rigide : un peu de malchance sur les adresses manipulées par un programme risque de provoquer un grand nombre de défauts. En revanche, le remplacement d'un bloc par un autre est très simple, puisqu'il n'y a pas de choix possible.

Ces caches sont cependant viables, surtout si, comme c'est souvent le cas, le système dispose de deux caches indépendants : un cache pour les données et un cache pour les instructions. La présence de deux caches réduit beaucoup les risques de conflits gênants. De plus le cache d'instructions est plus simple, car il est inutile d'y prévoir l'écriture (le programme est "constant", du moins considéré en tant que programme), et la logique d'écriture est la partie la compliquée d'un cache.

### 2.3 - Caches associatifs par ensembles de k emplacements

Dans un cache associatif par ensembles, l'index associé à une adresse conduit à un ensemble de  $k$  emplacements ( $k \geq 2$ , car  $k=1$  donne un cache à correspondance directe). Un bloc de données peut être placé dans n'importe lequel de ces  $k$  emplacements.

La figure suivante illustre un cache associatif par ensembles de 4 emplacements. Pour une même capacité du cache, l'index aura 2 bits de moins (6 ici), car il y a quatre fois moins de "lignes" que d'emplacements (chaque ligne est un ensemble de 4 emplacements possibles pour un index).



La logique de recherche d'une donnée est simple : c'est 4 fois la logique d'un cache à correspondance directe. Si la donnée se trouve dans un des emplacements de l'ensemble désigné par l'index, le cache délivre la donnée valide dont l'étiquette est égale au champ étiquette de l'adresse cherchée.

Ce qui complique un tel cache est la logique de remplacement d'une donnée dans le cache. En effet, lorsque la donnée d'une nouvelle adresse doit être ramenée dans le cache, il faut maintenant choisir laquelle on va remplacer parmi les 4 possibles. Deux stratégies de remplacement sont utilisées :

- Remplacement du bloc le moins récemment utilisé (LRU).
- Remplacement aléatoire : un bloc quelconque, pris "au hasard".

Le remplacement LRU (least recently used) est un peu meilleur pour le taux de défauts, mais il est assez complexe (il faut mémoriser et mettre à jour l'ordre de dernière utilisation de chaque bloc). Le remplacement aléatoire est plus simple, pour une augmentation minime du taux de défauts. Selon certaines mesures, pour des ensembles de 2 ou 4 emplacements, le remplacement aléatoire ne donnerait que 1,1 fois plus de défauts que le LRU.

## 2.4 - L'écriture dans un cache

Les schémas vus jusqu'ici ne concernaient que des lectures. Les écritures sont un peu plus complexes, et il existe deux façons de procéder.

- **La recopie au remplacement.**

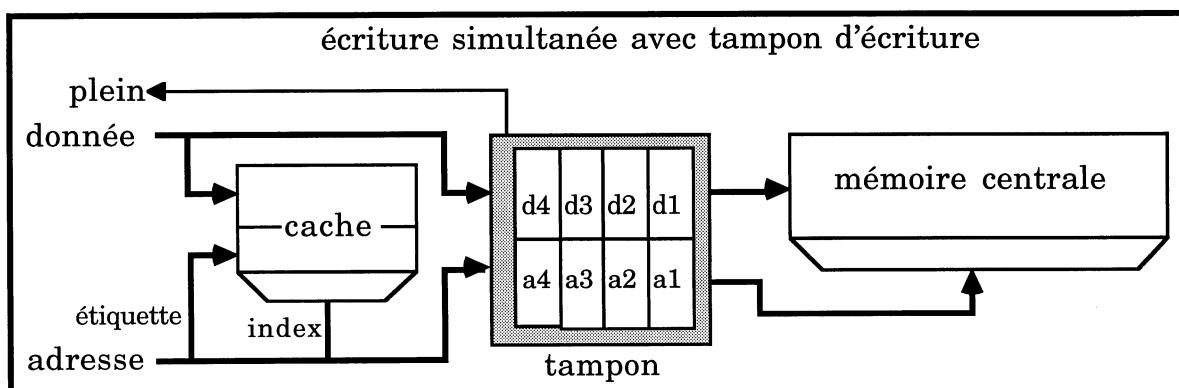
Le bloc, s'il a été modifié, est recopié en entier lors de son remplacement.

Ce mode est efficace : un mot écrit plusieurs fois ne sera rangé qu'une fois en mémoire, et la copie d'un bloc entier profite du débit élevé des mécanismes d'accès à des adresses contiguës. Un bit supplémentaire dans chaque emplacement du cache indique que le bloc a été modifié et qu'il faut le recopier lors du prochain remplacement.

- **L'écriture simultanée.**

Chaque écriture dans un mot du cache est répercutée en mémoire centrale. Cette technique est plus simple à réaliser, et l'usage de tampons d'écriture la rend presque aussi efficace que la recopie au remplacement.

Le débit moyen des écritures est assez faible (avec l'hypothèse, plutôt défavorable, qu'une instruction sur deux fait une référence mémoire, et une fois sur trois seulement en écriture, on obtient une moyenne d'une écriture pour 8 lectures). Ce débit est de l'ordre de ce que peut accepter la mémoire centrale. On peut ainsi utiliser un tampon dans lequel sont rangées les écritures en attente. A partir de ce tampon, les données sont écrites au rythme de la mémoire centrale, sans mettre le processeur en attente.



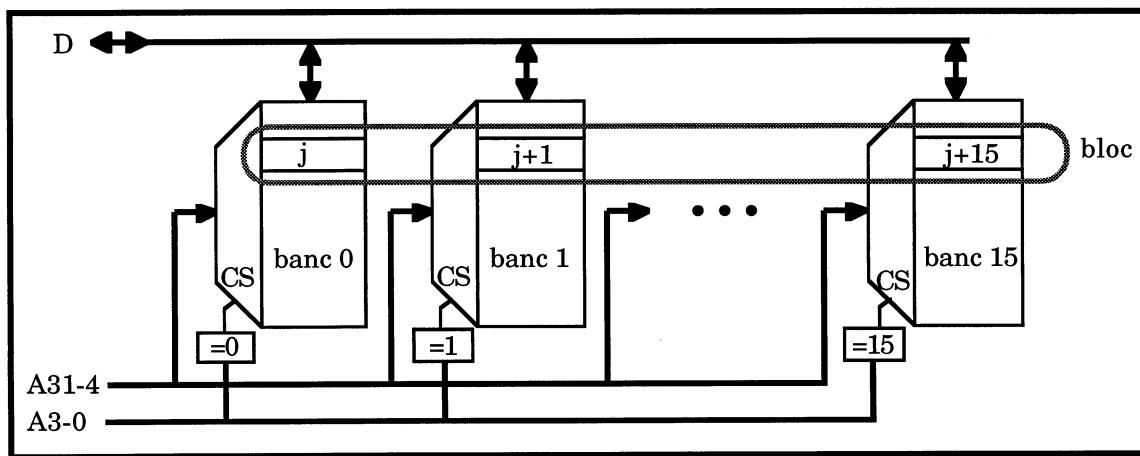
Le tampon d'écriture est une file (FIFO) de paires <donnée,adresse>. Un indicateur *plein* permet de mettre en attente le processeur en cas d'écriture lorsque la file est pleine (ce qui arrive rarement).

## 2.5 - Transferts entre cache et mémoire - Mémoire entrelacée

Un défaut donne lieu à des transferts de données entre la mémoire centrale et le cache, au moins la lecture du bloc recherché, et éventuellement, dans le cas d'un cache avec recopie au remplacement, l'écrire du bloc remplacé s'il a été modifié pendant son séjour dans le cache.

Le bloc comporte plusieurs mots afin de profiter de la localité spatiale. Pour que ce soit un gain effectif, il faut que le transfert d'un bloc de  $k$  mots avec la mémoire principale soit plus rapide que  $k$  transferts de mots isolés. En d'autres termes, il faut que le débit soit élevé (plus élevé que l'inverse du temps d'accès).

Pour augmenter le débit de la mémoire centrale, on réalise une mémoire entrelacée. Comme un bloc est constitué d'adresses contiguës, on peut placer les adresses successives dans des composants mémoires différents : les composants qui détiennent le bloc peuvent être adressés en parallèle, ce qui ne nécessite qu'une seule fois le temps d'accès ; puis ils sont rapidement sollicités tour à tour pour fournir ou enregistrer une donnée. Le schéma suivant montre l'organisation d'une telle mémoire : elle est composée de 16 composants (on les appelle des "bancs"). Un bloc entier est adressé en une fois, par les lignes d'adresse A31-4, puis, le temps d'accès écoulé, une rafale de 16 accès est déclenchée, chaque accès lisant ou écrivant sa propre donnée sur D, accompagné du numéro de banc sur A3-0.



Exemple numérique réaliste : si le temps d'accès d'un banc est de 200 ns, et que le temps de transfert d'une donnée avec chacun des banchs est 10 ns, le temps de transfert d'un bloc sera

$$\text{tempsDeTransfertBloc} = 200 + 16 \times 20 = 520 \text{ ns}$$

alors que 16 accès individuels nécessiteraient  $16 \times 200 = 3200 \text{ ns}$ .

### Utilisation de l'adressage par colonnes des mémoires dynamiques

L'exemple précédent illustre le principe général de l'entrelacement. Ce principe peut donner lieu à plusieurs variantes dans sa réalisation. Un problème majeur à l'heure actuelle est que les mémoires sont trop intégrées (!) : avec des composants DRAM de 4 Méga-bits, organisés en 1 bit de large, il faut réaliser une capacité d'au moins 64 Méga-mots pour avoir 16 bancs en composants individualisés. Si on a des mots de 32 bits, ce qui est la norme actuelle, ceci représente  $16 \times 32 = 512$  composants DRAM, ce qui n'est pas raisonnable.

La solution à ce problème est d'intégrer le découpage en bancs au sein même des composants DRAM. Or ces derniers s'y prêtent bien, à cause de leur organisation interne matricielle ( $N$  lignes  $\times N$  colonnes de bits). On profite alors du fait que l'essentiel du temps d'accès est le temps d'accès à une ligne, le temps d'accès aux bits d'une même ligne étant ensuite beaucoup plus rapide. Les composants DRAM prévoient pour cela un adressage de plusieurs bits consécutifs d'une même ligne, avec un temps voisin de 20 ns entre accès successifs.

## 3 - Fonctionnement d'une mémoire virtuelle

La mémoire virtuelle est le mécanisme qui gère les relations entre les niveaux 1 et 2 de la hiérarchie de mémoire. Nous détaillerons moins son fonctionnement que celui des caches : il est pour moitié réalisé par du matériel et pour moitié par du logiciel (la partie logicielle nous entraînerait trop loin dans le fonctionnement des systèmes d'exploitation).

Nous nous intéressons ici au cas le plus simple de mémoire virtuelle, qui se borne à découper un grand espace d'adresses en blocs de taille fixe (appelés pages) et à transférer ces blocs entre le niveau 1 (mémoire centrale) et le niveau 2 (disque de pagination). Un tel système s'appelle une pagination. Il existe des systèmes plus sophistiqués, appelés segmentation, qui offrent des espaces de tailles variables, accompagnés de moyens de protection.

### 3.1 - Principales caractéristiques d'une mémoire virtuelle

Les aspects quantitatifs de la mémoire virtuelle sont très différents de ceux de la gestion des caches : le rapport des temps d'accès disque/mémoire centrale est beaucoup plus grand que le rapport mémoire centrale/cache ( $\approx 100\,000$  contre  $\approx 10$ ). Le débit du disque étant cependant assez élevé ( $\approx 1$  Moctet/s), une taille de pages assez grande (de l'ordre de 4 ou 16ko) permet de compenser le temps d'accès.

Le coût d'un défaut de page étant très important, tout ce qui peut réduire le taux de défaut est mis à contribution : une page peut être placée dans n'importe quel emplacement de page en mémoire centrale, et la stratégie de remplacement est sophistiquée (**LRU** par exemple).

Les défauts de page sont traités par logiciel, car le surcoût que cela apporte est négligeable.

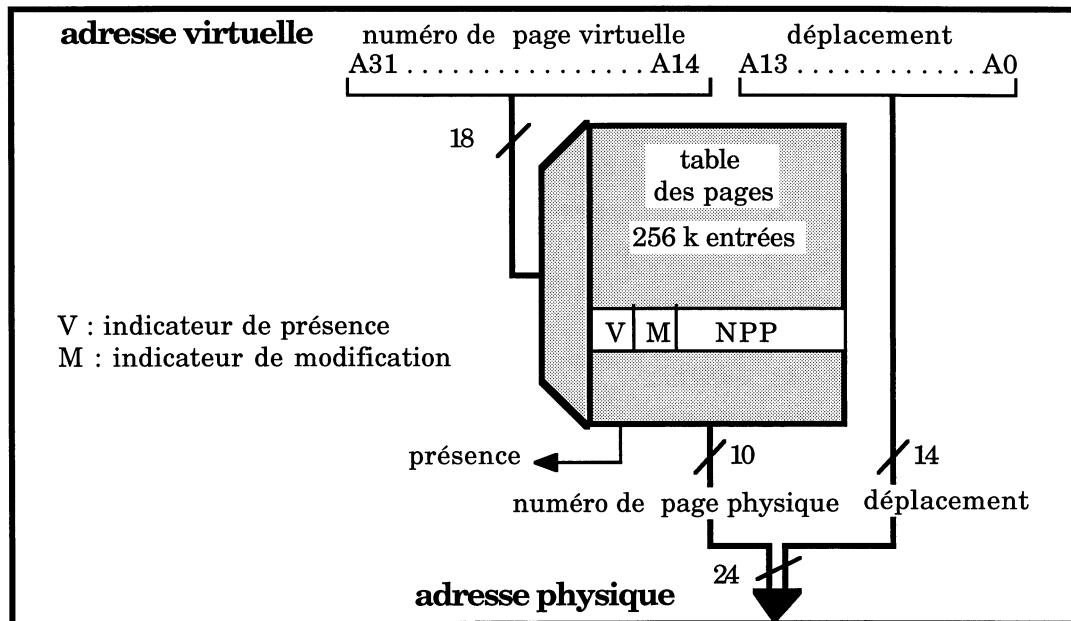
### 3.2 - Correspondance adresse virtuelle - adresse physique

Une adresse dans l'espace total (la mémoire principale) est d'ordinaire appelée "adresse virtuelle", et une adresse en mémoire centrale est appelée "adresse physique".

Les pages sont de taille fixe, une puissance de 2, par exemple 16 ko =  $2^{14}$  o. Une adresse virtuelle, par exemple de 32 bits, se décompose donc en un numéro de page virtuelle (les 18 bits de poids forts) et un déplacement dans la page (les 14 bits de poids faibles).

Les adresses figurant dans les programmes et manipulées par eux sont bien évidemment des adresses virtuelles, et c'est donc au moyen d'une adresse virtuelle que le processeur doit atteindre une donnée. Pour cela il doit la traduire en l'adresse physique où se trouve couramment la donnée.

Un moyen pour établir la correspondance (ce n'est pas le seul) est d'utiliser une table indexée par le numéro de page virtuelle :



Pour chaque page virtuelle cette table indique si elle est présente ou non dans la mémoire centrale (bit V) et si oui en quel emplacement (NPP : numéro de page physique). L'adresse physique s'obtient en concaténant numéro de page physique et déplacement.

Si la page est présente, l'adresse physique sert à adresser la mémoire.

Si elle est absente, l'activité en cours est suspendue et le contrôle est donné au système (donc à du logiciel) pour charger la page en mémoire centrale et, plus tard, redonner le contrôle à l'activité interrompue.

Etant donné le coût des accès au disque, une page est réécrite sur le disque uniquement lors de son remplacement. Afin d'économiser les écritures, seules sont écrites les pages modifiées. Pour cela il existe un indicateur de modification de la page pendant son séjour en mémoire centrale (M).

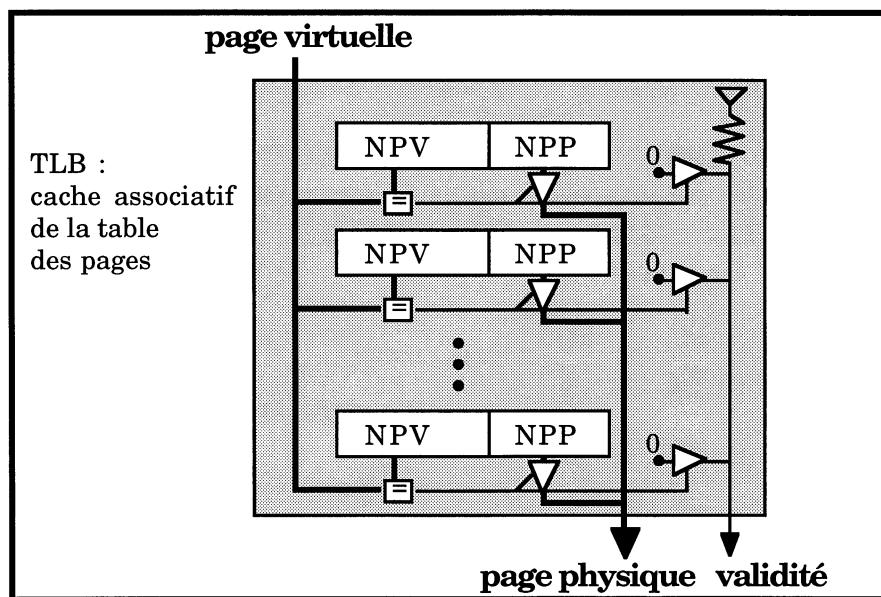
Le mécanisme qui donne le contrôle au système en cas de défaut de page ressemble au mécanisme d'interruptions. Une telle interruption, provoquée par le processeur (ou un organe synchrone avec le processeur) s'appelle une "exception".

La principale différence (outre qu'il n'y a pas besoin de contrôleur d'interruptions), est que l'instruction qui provoque le défaut de page doit pouvoir être reprise. Ce dernier point soulève des problèmes non triviaux de réalisation du répertoire d'instructions : par exemple, pour un défaut de page dû à un accès de donnée, l'instruction ne doit pas avoir commis de changement d'état irréversible au moment de la détection du défaut, sinon il serait impossible de la redémarrer dans le même état ...

### 3.3 - Traduction rapide des adresses - TLB

La table des pages est très grande. Elle doit donc être rangée en mémoire (en mémoire centrale physique, voire même, avec certaines précautions, en mémoire virtuelle). Sans dispositif auxiliaire, l'accès à une donnée nécessite deux accès, un accès à la table suivi d'un accès à la donnée.

Pour accélérer la traduction, on utilise un cache spécial qui contient une petite partie de la table de traduction. Ce cache de traduction s'appelle le TLB (translation lookaside buffer).



Le TLB est généralement un cache matériel totalement associatif : le champ étiquette est un numéro de page virtuelle complet (NPV), et chaque étiquette du TLB est comparée au numéro de la page virtuelle cherchée. Il fait partie intégrante du processeur.

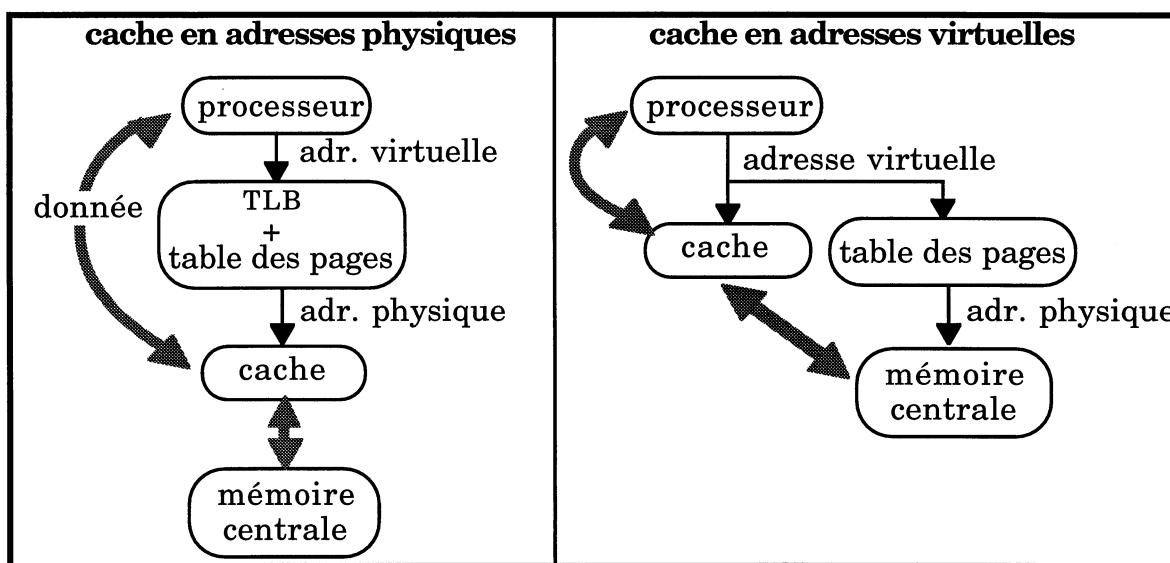
Il y a donc deux sortes de défauts :

- les défauts de TLB, assez fréquents, lorsque le TLB ne détient pas la traduction de la page demandée,
- les défauts de page, beaucoup plus rares, lorsque la page demandée n'est pas en mémoire centrale.

Les défauts de TLB sont, selon les machines, traités soit par matériel (le processeur lui-même accède à la table des page et charge le TLB), soit par logiciel (la mise à jour du TLB est faite par programme, programme déclenché par exception à chaque défaut de TLB). Ce n'est qu'à l'occasion d'un défaut de TLB que le processeur s'aperçoit d'un éventuel défaut de page, en consultant la table des pages en mémoire.

### 3.4 - Interactions entre cache et mémoire virtuelle

Les explications précédentes concernent un système dit de “cache en adresses physiques”, le plus naturel : le processeur traduit les adresses virtuelles en adresses physiques et le cache travaille sur ces adresses physiques pour délivrer la donnée. Il existe également des systèmes dit de “cache en adresses virtuelles” : le processeur s’adresse au cache directement en adresses virtuelles. Ce n’est qu’en cas de défaut de cache qu’il y a traduction en adresse physique. Un tel système, moins naturel, peut épargner l’usage d’un TLB.



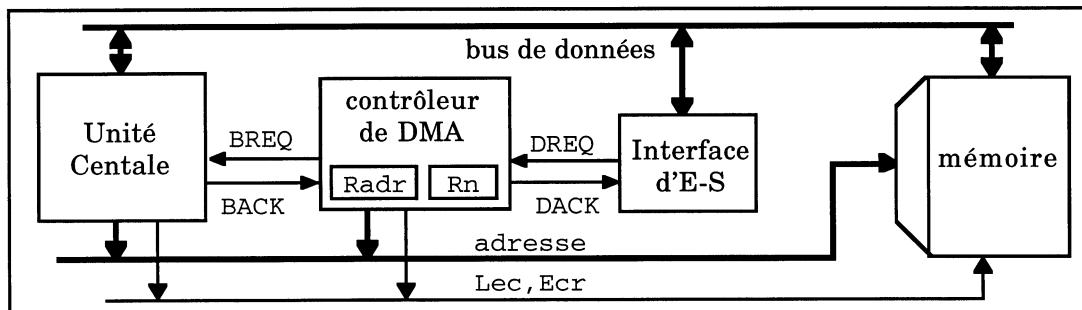
# ACCES DIRECT MEMOIRE - DMA

---

## 1 - Mécanisme d'accès direct mémoire

L'accès direct mémoire permet de donner le contrôle du bus à un composant autre que l'unité centrale. Ce composant, appelé "contrôleur de DMA" ou simplement "DMA", peut alors générer les signaux de commande et les adresses pour réaliser des transferts entre la mémoire et un périphérique (ou parfois de mémoire à mémoire). Les transferts par DMA ont un débit plus élevé que des transferts programmés et l'unité centrale peut exécuter des programmes pendant que des entrée-sorties ont lieu.

On utilise le DMA pour gérer les périphériques à haut débit, tels les disques ou les liaisons réseaux.

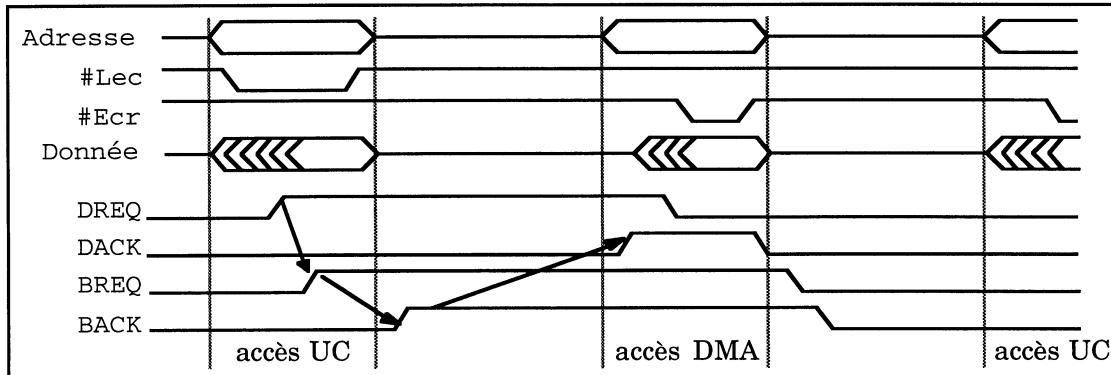


Le contrôleur de DMA possède des registres contenant l'adresse (Radr) et la longueur (Rn) de la zone mémoire concernée.

- L'interface signale au DMA, par *DREQ* (DMA request), qu'un transfert de donnée est possible.
- Le DMA demande à la contrôle du bus par *BREQ* (bus request).
- L'unité centrale termine l'accès en cours, puis lâche le contrôle du bus en laissant libre ses sorties d'adresse et de commande. Elle signale alors au DMA par *BACK* (bus acknowledge) qu'il peut prendre le bus.
- Le DMA réalise le transfert et l'indique à l'interface par *DACK*.

Le DMA utilise les lignes d'adresse et de commandes du bus pour accéder à la mémoire, alors que l'interface est directement sélectionnée par *DACK*.

Le transfert d'un mot est réalisé en un seul accès bus, directement entre la mémoire et l'interface. Le contrôleur conserve le bus tant qu'il maintient sa demande *BREQ*. Le DMA rend généralement le contrôle du bus entre chaque transfert élémentaire (octet ou mot).



Le DMA est accessible par programme à certaines adresses d'entrées-sorties. Un transfert est lancé en envoyant au DMA :

- l'adresse de la zone mémoire concernée (Radr),
- le nombre d'octets à transférer (Rn),
- le sens (mémoire vers interface ou interface vers mémoire),
- l'ordre de démarrage effectif.

La fin du transfert est indiquée par un bit d'état du DMA. La fin du transfert peut également être signalée par une interruption.

La plupart des DMA gèrent plusieurs canaux simultanément : chaque canal est connecté à une interface, et comporte ses propres registres d'adresse et de taille. Les demandes simultanées sont gérées selon une certaine priorité.

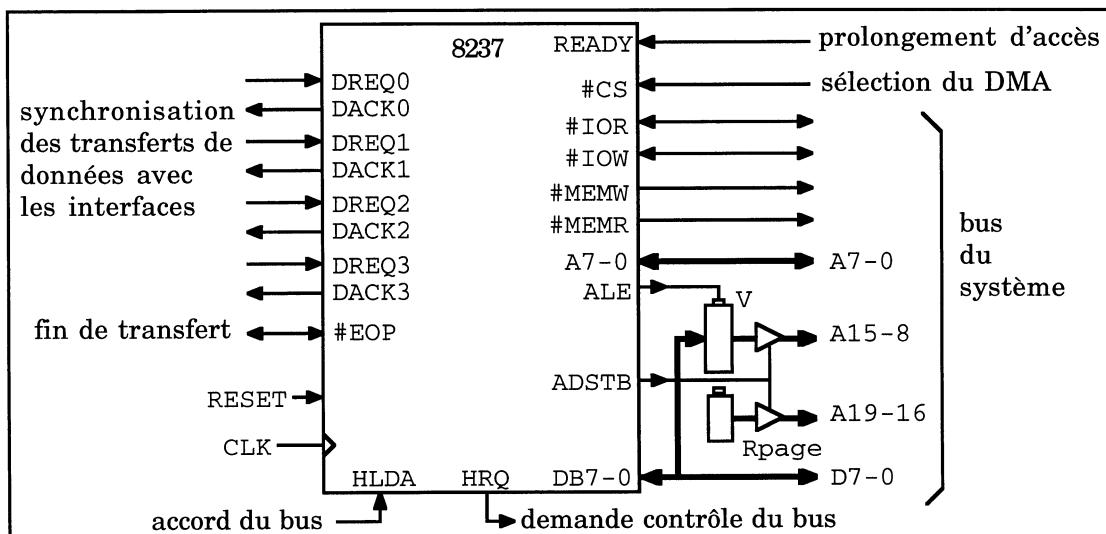
Les DMA ont généralement plusieurs modes de restitution du bus :

- Le mode mot par mot (single transfert) : le DMA rend le bus à l'unité centrale après chaque transfert élémentaire.
- Le mode transfert tant que la demande est active (demand transfert) : le DMA conserve le bus tant que l'interface est disponible à transférer de nouvelles données, c'est-à-dire tant que *DREQ* demeure actif.
- Le mode transfert par bloc (block transfer) : le DMA conserve le bus jusqu'à la fin du transfert de toutes les données.

## 2 - Exemple de contrôleur de DMA : Intel 8237

### 2.1 - Structure du contrôleur de DMA

Ce contrôleur de DMA est capable de gérer quatre canaux simultanément, c'est-à-dire de gérer des transferts entre quatre interfaces et la mémoire.



#### Synchronisation du transfert de données avec les interfaces

Chaque canal  $i$  est connecté à une interface : il reçoit les demandes sur  $DREQ_i$  et répond par  $DACK_i$  pour réaliser le transfert.

#### Connexions au bus du système

Le DMA est accessible par 16 adresses de ports, permettant d'envoyer des commandes ou de consulter l'état, grâce aux signaux de sélection (#CS), de lecture (#IOR) et d'écriture (#IOW), et à quatre bits d'adresse interne.

Lorsque le DMA obtient le contrôle du bus, il réalise le transfert en adressant la mémoire par les signaux #MEMR ou #MEMW, et A19-0. Les adresses gérées par le DMA ne font que les 16 bits A15-0, et les poids forts A15-8 sont multiplexés avec les données sur les broches DB7-0 : lorsque le DMA génère une adresse, il affiche A15-0 sur DB7-0 accompagné du signal ALE (address latch enable) qui permet de charger ces bits d'adresse dans un verrou externe V.

Si les adresses font plus de 16 bits (par exemple avec un 8086 : 20 bits), il faut ajouter un registre pour les poids forts supplémentaires (Rpage). V et Rpage doivent être connectés sur les lignes d'adresse du système quand le DMA a le contrôle du bus : cette connexion est validée par la sortie ADSTB (adres strobe).

### Connexion à l'unité centrale

Le DMA demande le contrôle du bus à l'unité centrale par *HRQ* (hold request). Quand elle accorde le bus, celle-ci le signale par *HLDA* (hold acknowledge).

### Autres signaux

La broche *#EOP* sert à signaler la fin de transfert sur l'un quelconque des canaux. Ce signal est activé pendant un cycle de *CLK* et peut servir à générer une interruption de l'unité centrale en fin de transfert. La même broche peut également servir d'entrée (usage peu fréquent) pour forcer la terminaison du transfert sur le canal pour lequel le DMA répond couramment *DACK*.

## 2.2 - Programmation du DMA

Chaque canal comporte les registres suivants :

*RAi*, *RCi* : adresse de début et nombre d'octets de transfert du canal i (16 bits),

*RACi*, *RCCi*: adresse courante et nombre d'octets restants pour le canal i,

*RMODi* : registre de mode du canal i (6 bits),

*RREQi*, *RMSKi* : registres de requête (1 bit) et de masque (1 bit).

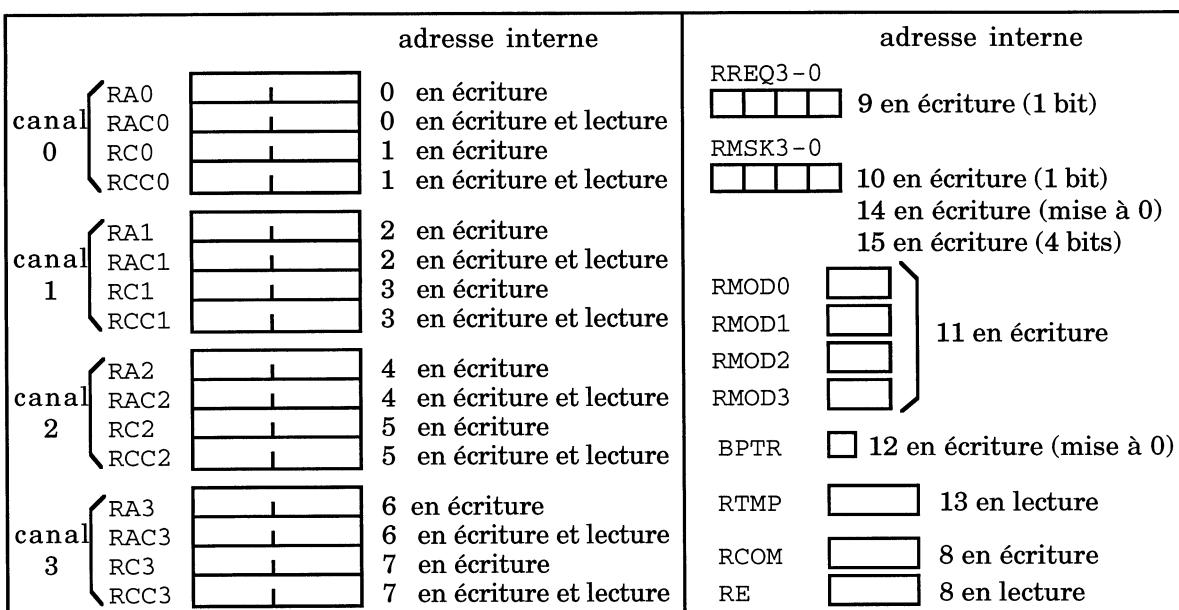
Le DMA contient également des registres communs à tous les canaux :

*RCOM* : registre de commande (8 bits),

*RE* : registre d'état (8 bits),

*BPTR* : bascule de sélection octet faible/fort (1 bits),

*RTMP* : tampon de donnée pour les transferts mémoire-mémoire (8 bits).



Les registres d'adresse et de compte d'octets font 16 bits. Chacun est

accessible à une seule adresse (envoi en deux octets successifs, poids faibles puis poids forts). La bascule *BPTR* (byte pointer) indique l'octet couramment accessible (0: faible, 1: fort). L'écriture d'une valeur quelconque à l'adresse 12, initialise *BPTR* à 0, et elle est mise à 1 lors du premier accès ultérieur à un registre 16 bits (attention : pour transférer N octets, le registre *RCi* doit être initialisés à N-1).

Par exemple, pour écrire 4567h dans le registre *RA2*, on fera :

```
OUT12,AL ; mise à 0 BPTR
MOV AL,67h
OUT4,AL ; écriture poids faibles de RA2
MOV AL,45h
OUT4,AL ; écriture poids forts de RA2
```

### Registre RCOM

Le registre *RCOM* fixe les options concernant l'ensemble des canaux.

RCOM	7	6	5	4	3	2	1	0	écriture adresse 8
									1 : transferts mém-mém, 0 : mém - interface
									1 : adresse du canal 0 fixe (transfert mém-mém spécial)
									0 : valide le fonctionnement du DMA
									0 : accès bus normal (4 cycles), 1 : accès bus rapide (3 cycles)
									0 : priorité fixe entre canaux, 1 : priorité tournante
									0 : signal d'écriture tardif, 1 : précoce
									0 : DREQi actifs à 1, 1 : actifs à 0
									0 : DACKi actifs à 0, 1 : actifs à 1

Les canaux 0 et 1 peuvent être utilisés conjointement pour des transferts de mémoire à mémoire : le canal 0 contient alors l'adresse de la zone origine, le canal 1 celle de la zone destinataire et le nombre d'octets ; chaque octet copié transite par le DMA, dans le registre *RTMP* (registre temporaire).

### Registres RMODi

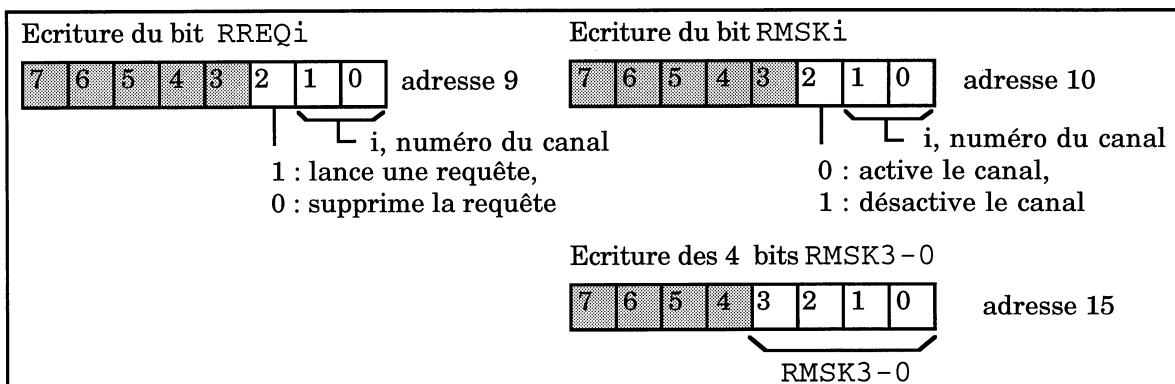
Ils fixent les options concernant un canal. L'écriture sur ces registres se fait à l'adresse 11, en plaçant le numéro du canal concerné dans les bits 1-0.

Ecriture de RMODi	7	6	5	4	3	2	1	0	adresse 11
									i, numéro du canal concerné
									00 : vérification, 01 : écriture mém, 10 : lecture mém
									1 : réinitialisation automatique de <i>RACi</i> et <i>RCCI</i> .
									0 : incrémentation d'adresse, 1 : décrémentation d'adresse
									00 : transfert tant que demande, 01 : octet par octet, 10 : par bloc

Les bits 2 et 3 définissent le sens du transfert (E-S vers mémoire ou mémoire

Les bits 2 et 3 définissent le sens du transfert (E-S vers mémoire ou mémoire vers E-S) ou bien provoquent une simple vérification (aucun signal *#IOW*, *#IOR*, *#MEMW* ou *#MEMR* n'est généré). Le bit 4 spécifie la réinitialisation automatique de *RACi* et *RCCi* à la valeur de *RAi* et *RCi* en fin de transfert (prêt pour un nouveau transfert). Le bit 5 choisit entre un transfert par adresses croissantes ou décroissantes. Les bits 6 et 7 définissent le mode de transfert.

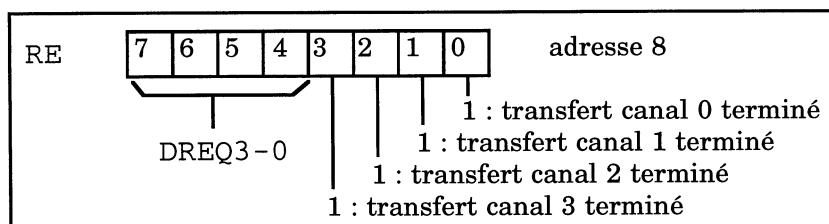
### Registres de requêtes RREQ et de masques RMSK



La mise à 1 du bit *RREQ<sub>i</sub>* permet de démarrer un transfert par logiciel. Ceci a le même effet que l'activation matérielle de l'entrée *DREQ<sub>i</sub>* (utilisable qu'en mode transfert par bloc). Les bits *RREQ<sub>i</sub>* sont accédés à l'adresse 9, en indiquant le numéro de canal dans les bits 1-0 de l'octet.

Le bit *RMSK<sub>i</sub>* permet d'activer ou désactiver le fonctionnement du canal en masquant ou démasquant la prise en compte de l'entrée *DREQ<sub>i</sub>*. Les bits *RMSK<sub>i</sub>* sont accédés à l'adresse 10, en indiquant le numéro de canal dans les bits 1-0 de l'octet. On peut également modifier les quatre bits à la fois en écrivant à l'adresse 15.

### Registre d'état RE



Le registre d'état permet de tester la fin du transfert pour chaque des canal (bits 3-0). Les bits 7-4 indiquent l'état des entrées *DREQ3-0*.

Les bits *RE3-0* sont remis à 0 à chaque lecture du registre d'état.

### Exercice 1

On veut comparer les débits maximums d'acquisition de données entre une lecture programmée sur l'unité centrale et une lecture par DMA. La version programmée est réalisée ainsi :

```
BOUCLE_LEC :
ATTENTE_E :   IN AL, ETAT      ; attente d'une donnée
              AND AL, 01h
              JZ ATTENTE_E
              IN AL, DONNEE    ; lecture de la donnée
              MOV [SI], AL      ; rangement
              INC SI           ; incrémentation de l'adresse de rangement
              DEC CX           ; décrémentation du compte d'octets restant
              JNZ BOUCLE_LEC
```

Nous faisons les hypothèses suivantes :

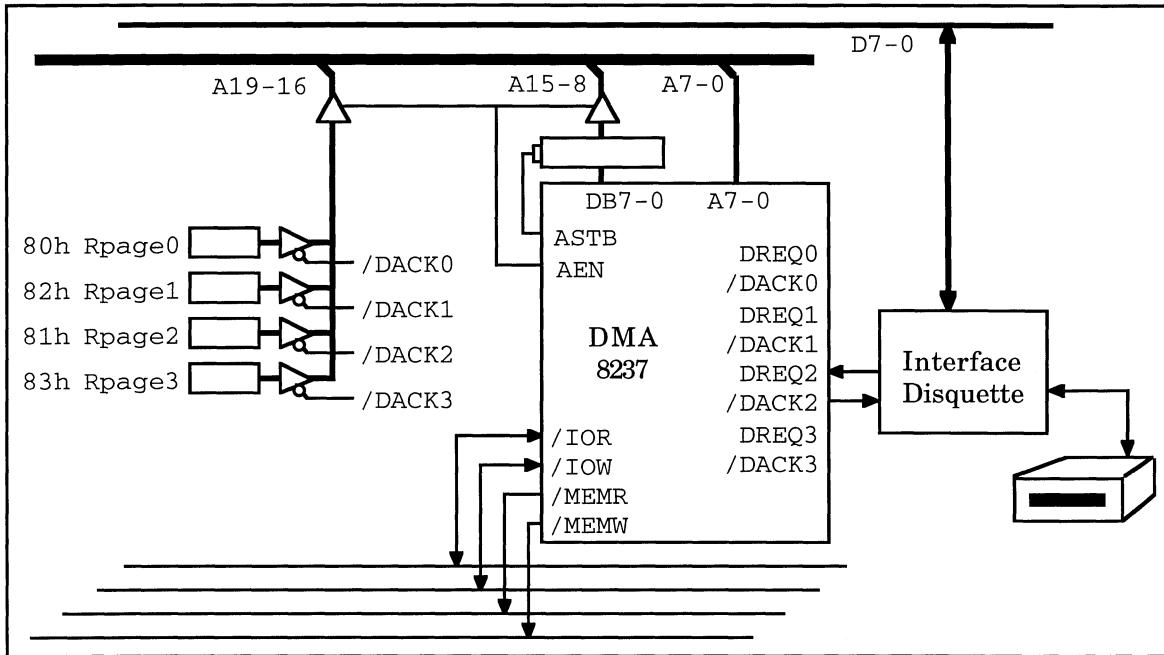
- L'unité centrale a un bus de données de 16 bits, elle lit les instructions par mots de 16 bits et toutes les instructions du programme ci-dessus font 16 bits.
- La durée d'un accès quelconque par le bus est de 400ns.
- Le temps d'exécution d'un programme est uniquement du aux accès bus.
- Le temps d'obtention et de restitution du bus par le DMA est 200ns.

Indiquer les débits maximums d'acquisition de données pour les versions programmée, avec DMA en mode “transfert octet par octet” et avec DMA en mode “transfert tant que la demande est active”.

### Exercice 2

Dans un ordinateur PC, le contrôleur de DMA 8237 est connecté comme indiqué sur la figure. Chaque canal possède son propre registre Rpage qui fournit les poids fort A19-16 de l'adresse de la zone mémoire de transfert. Ces registres sont accessibles en écriture aux adresses indiquées (Rpage1 est à l'adresse 82h et Rpage2 à l'adresse 81h). Chaque registre Rpagei est validé sur le bus d'adresse par le signal #DACKi car ce signal indique justement que le DMA réalise un accès pour le canal i.

Le DMA est situé à partir de l'adresse d'entrée-sortie 00h.



Le canal 2 est connecté à l'interface de disquettes. Les informations sur la disquette sont découpées en secteurs de 512 octets.

- Rédiger le programme qui initialise le DMA et le registre de page pour effectuer une lecture d'un secteur en mémoire, à partir de l'adresse 42000h. On n'initialisera pas le registre RCOM, commun à tous les canaux, qui est supposé déjà initialisé correctement (valeur 00h : pas de transferts mémoire-mémoire, DMA actif, durée d'accès normale, priorité fixe, signal d'écriture tardif, DREQ<sub>i</sub> actifs à 1, DACK<sub>i</sub> actifs à 0).

Remarque : l'adresse du tampon doit être telle que le tampon de 512 octets ne franchit pas une frontière de page de 64k, c'est-à-dire que les quatre bits de poids fort des adresses du tampon sont fixes, car le DMA ne sait pas incrémenter les quatre bits de poids fort (ils sont dans le registre externe Rpage2).