

mini Manuel

Architecture de l'ordinateur

Jacques Jorda
Abdelaziz M'zoughi

→ DUT
→ L1/L2

**Cours
+ exos
corrigés**

DUNOD

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.

Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements

d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour

les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du

droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).



© Dunod, Paris, 2012
ISBN 978-2-10-057763-7

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Table des matières

Avant-propos	1
1 Numération	3
1.1 Système de numération	3
1.2 Principe des opérations de base	6
1.3 Changements de bases	11
<i>Exercices</i>	17
<i>Solutions</i>	17
2 Codage de l'information	25
2.1 Codage des informations numériques	25
2.2 Codage des informations non numériques et codes spéciaux	30
<i>Exercices</i>	36
<i>Solutions</i>	37
3 Algèbre de Boole	43
3.1 Définition et théorèmes	43
3.2 Fonctions booléennes et représentations	46
3.3 Simplifications des fonctions booléennes	56
<i>Exercices</i>	63
<i>Solutions</i>	64
4 Logique combinatoire	71
4.1 Définition	71
4.2 La fonction de codage	73
4.3 La fonction de multiplexage	75
4.4 L'additionneur binaire	77

<i>Exercices</i>	81
<i>Solutions</i>	83
5 Logique séquentielle	91
5.1 Introduction	91
5.2 Éléments de mémorisation élémentaire	92
5.3 Les verrous	93
5.4 Les bascules	103
5.5 Comportement des E/S des verrous et des bascules	108
5.6 Classification des verrous et bascules sous la dénomination « BASCULE »	109
5.7 Association de bascules	109
<i>Exercices</i>	121
<i>Solutions</i>	124
6 Les mémoires	139
6.1 Introduction	139
6.2 Classification des mémoires	140
6.3 Les mémoires vives	142
6.4 Les mémoires mortes	153
7 L'unité centrale	157
7.1 Organisation d'un ordinateur	157
7.2 Architecture générale d'une unité centrale	159
7.3 Introduction à la programmation en assembleur du processeur ARM	167
7.4 Instructions de traitement des données	170
7.5 Instructions de référence mémoire	178
7.6 Instructions de contrôle de programme	183
7.7 Syntaxe d'un fichier source en langage d'assemblage	188
<i>Exercices</i>	191
<i>Solutions</i>	192
Index	205

Avant-propos

L'architecture est la branche de l'informatique qui étudie la structure des ordinateurs. Il s'agit des couches les plus basses, puisqu'allant du transistor aux composants logiciels permettant le dialogue avec les périphériques. La connaissance des mécanismes qui régissent ces couches est fondamentale pour qui veut comprendre le fonctionnement d'un système d'exploitation ou l'implémentation des structures logicielles des langages de haut niveau.

C'est la méconnaissance de ces couches basses qui a rendu possible les attaques logicielles par saturation de *buffer* – à l'origine de bien des failles de sécurité. Leur étude est donc plus que jamais d'actualité.

Nous commencerons par aborder la numération, point de départ de cet ouvrage puisque toutes les informations manipulées par l'ordinateur sont numériques. Nous verrons ensuite comment les informations sont codées, et comment il est possible d'optimiser leur stockage.

Puis nous étudierons l'algèbre de Boole, structure mathématique sous-jacente à la construction des opérateurs binaires complexes. Ces opérateurs seront présentés dans le chapitre suivant, relatif à la logique combinatoire. Nous poursuivrons par l'introduction de la notion de temps et verrons les implications au niveau logique dans le chapitre relatif à la logique séquentielle.

Nous terminerons par l'étude de la structure interne d'un processeur, et présenterons les mémoires, l'architecture et le langage d'assemblage des processeurs ARM dans le dernier chapitre.



Numération

PLAN

- 1.1 Système de numération
- 1.2 Principe des opérations de base
- 1.3 Changements de bases

OBJECTIFS

- Comprendre le système de numération et les opérations de base
- Savoir représenter les entiers naturels et relatifs, et les nombres réels

La **numération** est la méthode qui permet de décrire la forme utilisée pour représenter les nombres. Nous allons présenter le système de numération classique, et montrer comment il permet d'écrire des nombres entiers ou fractionnaires. Nous rappellerons ensuite comment sont implémentées les opérations de base que sont l'addition, la soustraction, la multiplication et la division, puis nous verrons comment changer de base. Nous terminerons par quelques exercices sur le sujet.

1.1 SYSTÈME DE NUMÉRATION

Le système de numération classique est un **système positionnel**. Cela signifie qu'un nombre est formé par une succession de chiffres dont la position détermine l'importance dans le nombre écrit.

Définition : un système de numération positionnel pondéré de base b est un triplé (X, A, f) où X note l'ensemble à énumérer, A est un ensemble de b chiffres et f une application injective qui fait

correspondre au plus une représentation à tout nombre de X :

$$\begin{aligned} f : X &\longrightarrow A^n \\ x &\longrightarrow (a_n)_{n \geq 1} \end{aligned}$$

Représentation des nombres entiers et fractionnaires

La représentation positionnelle des nombres est aussi appelée **représentation b-adique**. Tout entier peut s'écrire sous la forme :

$$N = \varepsilon_{n-1}(N) \cdot b^{n-1} + \dots + \varepsilon_1(N) \cdot b^1 + \varepsilon_0(N) \cdot b^0 = \sum_{i=0}^{n-1} \varepsilon_i(N) \cdot b^i$$

où $0 \leq \varepsilon_i(N) < b$. Chaque chiffre possède donc un **poids**, ou **rang** : il s'agit de l'indice qui lui est associé dans la représentation, c'est-à-dire de l'exposant de la base par lequel il est multiplié. Ce principe peut être étendu aux nombres fractionnaires. Un **nombre fractionnaire** N peut donc s'écrire :

$$N = \varepsilon_{n-1}(N) \cdot b^{n-1} + \dots + \varepsilon_1(N) \cdot b^1 + \varepsilon_0(N) \cdot b^0 + \varepsilon_{-1}(N) \cdot b^{-1} + \varepsilon_{-2}(N) \cdot b^{-2} + \dots + \varepsilon_{-p}(N) \cdot b^{-p}$$

soit, sous forme condensée :

$$N = \sum_{i=-p}^{n-1} \varepsilon_i(N) \cdot b^i$$

soit encore, sous forme condensée :

$$N = (\varepsilon_{n-1}(N) \dots \varepsilon_1(N) \varepsilon_0(N), \varepsilon_{-1}(N) \varepsilon_{-2}(N) \dots \varepsilon_{-p}(N))_b$$



Représentation simplifiée des nombres

Le nombre $N = 1 \cdot 10^4 + 7 \cdot 10^3 + 4 \cdot 10^2 + 9 \cdot 10^1 + 2 \cdot 10^0$ s'écrira donc $(17492)_{10}$.

Usuellement, on a $\varepsilon_i(N) \neq 0$ c'est-à-dire que le chiffre de plus fort poids est non nul – sinon il est non significatif. Par contre, le système étant positionnel, il ne faut pas omettre les autres zéros car ils sont significatifs : ils permettent d'associer le bon rang à chaque chiffre.

Représentation d'un nombre dans différentes bases

Les principales bases dont nous traiterons sont la base 10, la base 2 (numération binaire : $A = \{0, 1\}$), la base 8 (numération octale : $A = \{0, 1, 2, 3, 4, 5, 6, 7\}$) et la base 16 (numération hexadécimale : $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$).

Un nombre N possède une valeur intrinsèque qui est constante, mais a une représentation différente dans chaque base. Il ne faut donc pas confondre un nombre (la valeur de ce nombre) qui est unique et ses représentations.



Représentation de 14792 dans différentes bases

Le nombre 14792 s'écrit, $(14792)_{10}$, $(39C8)_{16}$, $(34710)_8$ ou encore $(111001110010000)_2$.

Valeur d'un nombre

Puisqu'un nombre possède une valeur intrinsèque bien qu'ayant une représentation différente pour chaque base, il peut être utile de savoir calculer la valeur du nombre correspondant à une représentation particulière.

Comme nous manipulons naturellement la base 10 (c'est ce système de numération que nous avons appris à l'école primaire), cela revient à convertir un nombre d'une représentation en base B à sa représentation en base 10. Nous étudierons une généralisation de ce procédé au paragraphe *Changement de base*.

La méthode utilisée est celle du **schéma de Hörner**. Cette méthode permet de calculer rapidement et simplement la valeur d'un polynôme en un point, et d'obtenir une factorisation de ce polynôme si la valeur utilisée pour le calcul est une de ses racines.

Considérons le nombre :

$$N = (\varepsilon_{n-1}(N) \dots \varepsilon_1(N) \varepsilon_0(N))_b$$

Calculer sa valeur revient à calculer :

$$N = \varepsilon_{n-1}(N) \cdot b^{n-1} + \dots + \varepsilon_1(N) \cdot b^1 + \varepsilon_0(N) \cdot b^0$$

c'est-à-dire calculer la valeur au point b du polynôme :

$$P_0(X) = \varepsilon_{n-1}(N) \cdot X^{n-1} + \varepsilon_{n-2}(N) \cdot X^{n-2} + \dots + \varepsilon_2(N) \cdot X^2 \\ + \varepsilon_1(N) \cdot X^1 + \varepsilon_0(N)$$

Le principe du schéma de Hörner consiste à effectuer des factorisations successives par X . On effectue une première factorisation qui permet d'écrire le polynôme sous la forme :

$$P_0(X) = X \cdot (\varepsilon_{n-1}(N) \cdot X^{n-2} + \varepsilon_{n-2}(N) \cdot X^{n-3} + \dots + \varepsilon_2(N) \cdot X + \varepsilon_1(N)) + \varepsilon_0(N)$$

$$\text{soit } P_0(X) = X \cdot P_1(X) + \varepsilon_0(N)$$

avec

$$P_1(X) = \varepsilon_{n-1}(N) \cdot X^{n-2} + \varepsilon_{n-2}(N) \cdot X^{n-3} + \dots + \varepsilon_2(N) \cdot X + \varepsilon_1(N)$$

On recommence alors le procédé avec $P_1(X)$, puis $P_2(X)$, etc. On obtient alors, de manière récursive la forme suivante :

$$P_0(X) = (\dots ((\varepsilon_{n-1}(N) \cdot X + \varepsilon_{n-2}(N)) \cdot X + \varepsilon_{n-2}(N)) + \dots + \varepsilon_1(N)) \cdot X + \varepsilon_0(N)$$

On peut organiser le calcul sous forme d'un tableau ou sous forme d'une suite. Dans ce dernier cas, on calcule, dans l'ordre, les termes suivants :

- $t_0 = P_{n-1}(b) = \varepsilon_{n-1}(N)$
- $t_1 = P_{n-2}(b) = b \cdot P_{n-1}(X) + \varepsilon_{n-2}(N) = b \cdot t_0 + \varepsilon_{n-2}(N)$
- ...
- $t_{n-1} = P_0(b) = b \cdot P_1(X) + \varepsilon_0(N) = b \cdot t_{n-2} + \varepsilon_0(N)$



SCHÉMA DE HÖRNER SOUS FORME D'UNE SUITE

Soit à calculer la valeur de $(39C8)_{16}$. On procède par itérations successives :

$$\begin{aligned} t_0 &= 3 \\ t_1 &= 16 \cdot 3 + 9 = 57 \\ t_2 &= 16 \cdot 57 + 12 = 924 \\ t_3 &= 16 \cdot 924 + 8 = 14792 \end{aligned}$$

1.2 PRINCIPE DES OPÉRATIONS DE BASE

Nous allons rappeler le principe de réalisation des quatre opérations de base : l'addition, la soustraction, la multiplication et la division.

L'addition

Soit à calculer la somme $S = \varepsilon_{s-1}(S)\varepsilon_{s-2}(S) \dots \varepsilon_0(S)$ de deux nombres $X = \varepsilon_{x-1}(X)\varepsilon_{x-2}(X) \dots \varepsilon_0(X)$ et $Y = \varepsilon_{y-1}(Y)\varepsilon_{y-2}(Y) \dots \varepsilon_0(Y)$.

On procède des faibles poids vers les forts poids : on **additionne** les deux chiffres de rang courant (le rang i) avec l'éventuelle retenue générée à l'étape précédente (la retenue entrante du rang 0 étant 0).

On procède de la sorte tant qu'il reste des chiffres à additionner.



EXEMPLE : ADDITION EN BASE 2

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

Ce principe reste applicable quelle que soit la base.

La soustraction

Soit à calculer la soustraction $S = \varepsilon_{s-1}(S)\varepsilon_{s-2}(S) \dots \varepsilon_0(S)$ de $Y = \varepsilon_{y-1}(Y)\varepsilon_{y-2}(Y) \dots \varepsilon_0(Y)$ à $X = \varepsilon_{x-1}(X)\varepsilon_{x-2}(X) \dots \varepsilon_0(X)$. On procède des faibles poids vers les forts poids : on **soustrait** le chiffre de rang i de Y au chiffre de rang i de X . Si une retenue r_i a été générée au rang précédent, elle doit aussi être soustraite du chiffre de rang i de X (la retenue entrante du rang 0 étant 0).

On continue de la sorte tant qu'il reste des chiffres à traiter.



EXEMPLE : SOUSTRACTION EN BASE 16

$$\begin{array}{r}
 A \\
 - \\
 \hline
 8
 \end{array}$$

Ce principe est applicable quelle que soit la base.

La multiplication

La **multiplication** de deux nombres en base b nécessite de savoir multiplier un nombre par un chiffre en base b , de savoir multiplier un nombre par une puissance de la base, et enfin de savoir réaliser une addition.

Nous venons d'étudier les mécanismes de l'addition en base b . Examinons dans un premier temps la multiplication d'un nombre par une puissance de la base.

Multiplication d'un nombre par une puissance de la base

Soit $X = \varepsilon_{x-1}(X)\varepsilon_{x-2}(X) \dots \varepsilon_0(X)$ un nombre écrit en base b , et supposons qu'on désire calculer $X \cdot b^a$. On sait que peut s'écrire :

$$X = \varepsilon_{x-1}(X) \cdot b^{x-1} + \dots + \varepsilon_1(X) \cdot b^1 + \varepsilon_0(X) \cdot b^0$$

On en déduit que l'expression de $X \cdot b^a$ est :

$$X \cdot b^a = \varepsilon_{x-1}(X)\varepsilon_{x-2}(X) \dots \varepsilon_0(X)0 \dots 00$$

Pour multiplier un nombre X par b^a , il suffit donc de décaler le nombre X de a positions vers la gauche en introduisant des zéros à droite (en plus faibles poids).

La dernière étape avant de s'intéresser à la multiplication de deux nombres en base b concerne la multiplication d'un nombre par un chiffre.

Multiplication d'un nombre par un chiffre

Supposons donc que l'on cherche à calculer le résultat $R = \varepsilon_{r-1}(R)\varepsilon_{r-2}(R) \dots \varepsilon_0(R)$ de la multiplication de $X = \varepsilon_{x-1}(X)\varepsilon_{x-2}(X) \dots \varepsilon_0(X)$ par un chiffre y (y étant un chiffre, on a $0 \leq y \leq b$). On va procéder par étapes successives, des plus faibles poids vers les plus forts poids : on multiplie le chiffre de X de rang i par y , et on ajoute l'éventuelle retenue générée lors de l'étape précédente.

On continue de la sorte tant qu'il reste des chiffres du nombre X à multiplier par y .



MULTIPLICATION PAR UN CHIFFRE EN BASE 16

Multiplions $X = (B3D0)_{16}$ par $y = (B)_{16}$.

Nous commençons par multiplier 0 par B . Le résultat est 0, le chiffre de plus faible poids du résultat, et la retenue générée r_i est 0.

Nous multiplions ensuite D par B . N'étant pas familier avec les tables de multiplication en base 16, nous allons passer par la base 10. $D \times B = 13 \times 11 = 143$. En le mettant sous la forme $r_{i+1} \cdot b + \varepsilon_i(R)$, on obtient $D \times B = 143 = 8 \cdot 16 + 15$. On en déduit que le chiffre de rang 1 est F (la valeur 15), et la retenue propagée est 8.

On continue de la sorte jusqu'à la dernière étape, qui consiste à ajouter la retenue issue de l'étape précédente (la valeur 7) à la multiplication par B du chiffre de rang 4 de X (un zéro non significatif). On a donc, pour le calcul du chiffre de rang 4 du résultat : $0 \times B + 7 = 7$. Le chiffre de rang 4 du résultat est donc 7. En reprenant tous les chiffres calculés au cours des étapes successives, on obtient le résultat suivant : $B3D0 \times B = 7B9F0$.

Multiplication d'un nombre par un nombre

Le résultat $S = \varepsilon_{s-1}(S)\varepsilon_{s-2}(S) \dots \varepsilon_0(S)$ de la multiplication d'un nombre $X = \varepsilon_{x-1}(X)\varepsilon_{x-2}(X) \dots \varepsilon_0(X)$ par un nombre $Y = \varepsilon_{y-1}(Y)\varepsilon_{y-2}(Y) \dots \varepsilon_0(Y)$ nécessite deux étapes : le calcul des produits intermédiaires, puis l'addition de ces produits intermédiaires.

- **Étape 1** : calcul des produits intermédiaires. Pour chaque chiffre de Y $\varepsilon_i(Y)$, avec $0 \leq i < y$, on va procéder à :
 - la multiplication de X par $\varepsilon_i(Y)$. On obtient alors le produit $X \cdot \varepsilon_i(Y)$;
 - la multiplication de $X \cdot \varepsilon_i(Y)$ par b^i puisque le rang de $\varepsilon_i(Y)$ est i .
- **Étape 2** : addition des produits intermédiaires. Tous les produits intermédiaires obtenus au cours de l'étape 1 sont additionnés afin d'obtenir le résultat. Notons que cette addition pouvant comporter un nombre de nombres élevés (il y a autant de produit intermédiaires que de chiffres dans Y), la gestion de la retenue peut être délicate : elle est susceptible de se propager sur plusieurs rang consécutivement.



MULTIPLICATION EN BASE 16

				A	3	F	7
				C	5	F	
×							
			9	9	B	7	9
		3	3	3	D	3	0
	7	A	F	9	4	0	0
	7	E	C	6	C	A	9

Si l'addition directe des trois nombres est trop complexe à mettre en œuvre, on peut la scinder en deux additions successives : une première addition visant à produire le résultat de l'addition des deux premiers produits intermédiaires, et une seconde visant à additionner ce résultat avec le troisième produit intermédiaire.

La division

La division euclidienne de $X = \varepsilon_{x-1}(X)\varepsilon_{x-2}(X)\dots\varepsilon_0(X)$ par $Y = \varepsilon_{y-1}(Y)\varepsilon_{y-2}(Y)\dots\varepsilon_0(Y)$ consiste à calculer les nombres $Q = \varepsilon_{q-1}(Q)\varepsilon_{q-2}(Q)\dots\varepsilon_0(Q)$ et $R = \varepsilon_{r-1}(R)\varepsilon_{r-2}(R)\dots\varepsilon_0(R)$ vérifiant $X = Q \cdot Y + R$ avec $0 \leq R < Y$. X est appelé le dividende, Y est appelé le diviseur, Q est le quotient et R le reste.

Si $X < Y$, la solution est triviale : $Q = 0$ et $R = X$. Nous considérerons donc dorénavant que $X \geq Y$.

L'équation $X = Q \cdot Y + R$ peut s'écrire :

$$X = (\varepsilon_{q-1}(Q) \cdot b^{q-1} + \varepsilon_{q-2}(Q) \cdot b^{q-2} + \dots + \varepsilon_0(Q) \cdot b^0) \cdot Y + R$$

$$\text{Soit } R = X - (\varepsilon_{q-1}(Q) \cdot b^{q-1} + \varepsilon_{q-2}(Q) \cdot b^{q-2} + \dots + \varepsilon_0(Q)) \cdot Y$$

$$R = (\dots ((X - \varepsilon_{q-1}(Q) \cdot b^{q-1} \cdot Y) - \varepsilon_{q-2}(Q) \cdot b^{q-2} \cdot Y) - \dots) - \varepsilon_0(Q) \cdot Y$$

Posons $R_q = X$ et $R_i = R_{i+1} - \varepsilon_i(Q) \cdot b^i \cdot Y$. En calculant successivement tous les R_i , nous obtiendrons les chiffres $\varepsilon_i(Q)$ du quotient Q , et le dernier calcul (celui de R_0) nous donnera le reste de la division. Nous allons donc procéder par étapes successives.

- **Étape préliminaire** : détermination de $q - 1$, le rang du chiffre de plus fort poids du quotient. Le coefficient $q - 1$ est la plus petite valeur vérifiant $X < Y \cdot b^q$, c'est-à-dire $Y \cdot b^{q-1} \leq X < Y \cdot b^q$. Compte tenu de l'écriture de X et de Y , on détermine aisément que soit $q - 1 = x - y$, soit $q - 1 = x - y - 1$.
- **Étape 0 (étape initiale)** : calcul de $R_{q-1} = X - \varepsilon_{q-1}(Q) \cdot b^{q-1} \cdot Y$. Le rang $q - 1$ étant connu, on doit déterminer le chiffre $\varepsilon_{q-1}(Q)$. D'après le théorème de la division euclidienne, il s'agit de l'unique chiffre vérifiant $\varepsilon_{q-1}(Q) \cdot b^{q-1} \cdot Y \leq X < (\varepsilon_{q-1}(Q) + 1) \cdot b^{q-1} \cdot Y$.
- **Étape i** : ayant déterminé à l'étape précédente le coefficient le chiffre $\varepsilon_{i+1}(Q)$, on a pu calculer R_{i+1} . On va donc déterminer maintenant $R_i = R_{i+1} - \varepsilon_i(Q) \cdot b^i \cdot Y$. Toujours d'après le théorème de la division euclidienne, il s'agit de l'unique chiffre vérifiant $\varepsilon_i(Q) \cdot b^i \cdot Y \leq R_{i+1} < (\varepsilon_i(Q) + 1) \cdot b^i \cdot Y$.
- La dernière étape (l'étape $q - 1$) permet de déterminer à la fois $\varepsilon_0(Q)$ et R_0 .



DIVISION EN BASE 16

Montrons comment calculer la division de $X = A3F7$ par $Y = 5F$. Le rang du chiffre de plus fort poids du quotient doit être déterminé en premier. On a vu que $q - 1 = x - y = 2$ ou $q - 1 = x - y - 1 = 1$. Dans notre cas $5F00 \leq A3F7 < 5F000$ soit $5F \cdot 16^2 \leq A3F7 < 5F \cdot 16^3$; le rang recherché est donc 2.

L'étape 0 consiste à calculer R_2 . On cherche donc $\varepsilon_2(Q)$ vérifiant $\varepsilon_2(Q) \cdot 16^2 \cdot 5F \leq A3F7 < (\varepsilon_2(Q) + 1) \cdot 16^2 \cdot 5F$, soit $\varepsilon_2(Q) \cdot 5F00 \leq A3F7 < (\varepsilon_2(Q) + 1) \cdot 5F00$. $\varepsilon_2(Q) = 0$ ne convient pas car $A3F7 \geq (0 + 1) \cdot 5F00$. Par contre, $1 \cdot 5F00 \leq A3F7 < (1 + 1) \cdot 5F00 = BE00$; on a donc $\varepsilon_2(Q) = 1$, et $R_2 = A3F7 - 1 \cdot 16^2 \cdot 5F = A3F7 - 5F00 = 44F7$.

L'étape 1 concerne le calcul de R_1 . On cherche donc l'unique $\varepsilon_1(Q)$ vérifiant la relation $\varepsilon_1(Q) \cdot 16^1 \cdot 5F \leq 44F7 < (\varepsilon_1(Q) + 1) \cdot 16^1 \cdot 5F$, soit $\varepsilon_1(Q) \cdot 5F0 \leq 44F7 < (\varepsilon_1(Q) + 1) \cdot 5F0$. On doit procéder par multiplication successives en partant de $\varepsilon_1(Q) = 0$ et en incrémentant sa valeur pour déterminer le bon chiffre. Ici, $B \cdot 5F0 = 4150 \leq 44F7 < 4740 = C \cdot 5F0$; on en déduit que : $\varepsilon_1(Q) = B$, et par voie de conséquence $R_1 = 44F7 - B \cdot 16^1 \cdot 5F = 44F7 - 4150 = 3A7$.

La troisième et dernière étape permet de calculer R_0 . On cherche donc $\varepsilon_0(Q)$ vérifiant $\varepsilon_0(Q) \cdot 16^0 \cdot 5F \leq 3A7 < (\varepsilon_0(Q) + 1) \cdot 16^0 \cdot 5F$, soit $\varepsilon_0(Q) \cdot 5F \leq 3A7 < (\varepsilon_0(Q) + 1) \cdot 5F$. Toujours grâce à des multiplications successives par incrémentation de $\varepsilon_0(Q)$, on obtient $9 \cdot 5F = 357 \leq 3A7 < 3B6 = A \cdot 5F$. Il en résulte que $\varepsilon_0(Q) = 9$ et donc $R_0 = 3A7 - 9 \cdot 16^0 \cdot 5F = 3A7 - 375 = 50$.

Le quotient de la division de $A3F7$ par $5F$ est donc $1B9$ et le reste 50.

1.3 CHANGEMENTS DE BASES

Puisqu'un nombre possède une représentation différente pour chaque base, il peut être utile de passer de la représentation d'un nombre dans une base à la représentation de ce même nombre dans une autre base. On parle alors de changement de base.

Conversion de la partie entière

Soit $X = (\varepsilon_{x_0-1}^{b_0}(X) \varepsilon_{x_0-2}^{b_0}(X) \dots \varepsilon_0^{b_0}(X))_{b_0}$ la représentation en base b_0 du nombre entier naturel X , et $X = (\varepsilon_{x_1-1}^{b_1}(X) \varepsilon_{x_1-2}^{b_1}(X) \dots \varepsilon_0^{b_1}(X))_{b_1}$ sa représentation en base b_1 . Le problème posé ici est le passage de la première forme (base b_0) à la seconde (base b_1).

Méthode générale

On va procéder par divisions entières successives de X par b_1 dans la base b_0 . À chaque itération, on divise le quotient courant Q_{i-1} (initialement, X) par b_1 , et on obtient :

- un quotient Q_i , utilisé lors de l'itération suivante ;
- un reste R_i qui correspond au chiffre de rang i du nombre exprimé en base b_1 .

Le processus est poursuivi tant que $Q_1 \geq b_1$.

Il convient de bien noter que les calculs de divisions successives se font en base b_0 . Cet exercice étant très délicat pour $b_0 \neq 10$, on n'utilisera cette technique que lorsque la base de départ est 10.

Utilisation du schéma de Hörner

Lorsqu'on doit passer de la base n à la base p avec $n, p \neq 10$, on utilisera la base 10 comme base intermédiaire et on s'appuiera sur le schéma de Hörner pour passer de la base n à la base 10.



CONVERSION DE LA BASE 7 À LA BASE 12

Reprenons l'exemple de la conversion de $X = (4263)_7$ en base 12. Nous allons d'abord convertir X en base 10, puis nous l'exprimerons en base 12.

La conversion de X en base 10 à l'aide du schéma de Hörner est relativement simple : il nous suffit d'écrire que

$$X = 4 \cdot 7^3 + 2 \cdot 7^2 + 6 \cdot 7^1 + 3 \cdot 7^0 = 1372 + 98 + 42 + 3 = 1515.$$

On procède maintenant aux divisions successives par 12 en partant de ce résultat, le tout en base 10.

1515 divisé par 12 donne 126 de quotient et 3 de reste.

126 divisé par 12 donne 10 de quotient et 6 de reste.

10 divisé par 12 donne 0 de quotient et 10 de reste, soit A en base 12.

On retrouve donc bien $X = (4263)_7 = (A63)_{12}$.

Conversion de la partie fractionnaire

Le mécanisme est semblable à celui de la partie entière, à ceci près que les exposants des puissances de la base étant négatifs, on va procéder par multiplications au lieu d'utiliser des divisions. La conversion des parties entières ayant été détaillée au paragraphe précédent, nous allons nous focaliser ici sur la conversion des parties fractionnaires.

Notons en préambule que les zéros non significatifs pour une partie fractionnaire sont situés à droite, alors qu'ils sont situés à gauche pour les parties entières : 0,5 est bien égal à 0,50 !

Principe général

Soit $X = (0, \varepsilon_{-1}^{b_0}(X) \varepsilon_{-2}^{b_0}(X) \dots \varepsilon_{-x_0}^{b_0}(X))_{b_0}$ la représentation en base b_0 du nombre fractionnaire X , et $X = (0, \varepsilon_{-1}^{b_1}(X) \varepsilon_{-2}^{b_1}(X) \dots \varepsilon_{-x_1}^{b_1}(X))_{b_1}$ sa représentation en base b_1 . Le problème posé ici est le passage de la première forme (base b_0) à la seconde (base b_1). On va procéder de manière itérative. À chaque itération, on multiplie la partie fractionnaire courante (initialement, X) par b_1 afin d'obtenir deux nombres : une partie entière qui représente le chiffre de rang courant en base b_1 , et une partie fractionnaire utilisée dans la multiplication à l'itération suivante.

La question qui se pose est : jusqu'à quand poursuit-on ce processus ? On peut imaginer répéter les étapes décrites précédemment tant que $F_i \neq 0$. Le problème est que le processus peut être infini.



CONVERSION DE LA BASE 10 À LA BASE 2

On souhaite convertir $X = (0,1)_{10}$ en base 2. Comme décrit ci-dessus, on va opérer par multiplications successives par 2 en base 10.

On note, dans la dernière ligne du tableau suivant, que l'on retrouve la deuxième ligne. Les mêmes causes produisant les mêmes effets, il apparaît que le processus de conversion va amener un nombre d'étapes infini.

Opération	Partie entière	Partie fractionnaire
$0,1 \times 2 = 0,2$	0	0,2
$0,2 \times 2 = 0,4$	0	0,4
$0,4 \times 2 = 0,8$	0	0,8
$0,8 \times 2 = 1,6$	1	0,6
$0,6 \times 2 = 1,2$	1	0,2
$0,2 \times 2 = 0,4$	0	0,4

On peut donc écrire $X = (0,1)_{10} = (0,0001100110011\dots)_2$ avec une alternance de termes 00 et 11 sans fin.

Deux solutions permettent de limiter le nombre de chiffres lors de la conversion d'une partie fractionnaire d'une base vers une autre base.

Précision de la partie fractionnaire

La première solution consiste à fixer de manière arbitraire le nombre de chiffres après la virgule. C'est la technique couramment employée pour coder les nombres en **virgule fixe** dans les ordinateurs (on précisera cette représentation plus loin dans ce chapitre).

La deuxième solution vise à conserver pour le nombre converti une **précision** au moins égale à celle du nombre d'origine. Soit $X = (0, \varepsilon_{-1}^{b_0}(X) \varepsilon_{-2}^{b_0}(X) \dots \varepsilon_{-x_0}^{b_0}(X))_{b_0}$ la représentation en base b_0 du nombre fractionnaire X , et $X = (0, \varepsilon_{-1}^{b_1}(X) \varepsilon_{-2}^{b_1}(X) \dots \varepsilon_{-x_1}^{b_1}(X))_{b_1}$ sa représentation en base b_1 . On souhaite obtenir cette deuxième représentation en garantissant que la précision de cette dernière soit au moins égale à la précision de l'expression d'origine.

► **Première étape :** détermination de la précision d'origine. On a

$X = \varepsilon_{-1}^{b_0}(X) \cdot b_0^{-1} + \varepsilon_{-2}^{b_0}(X) \cdot b_0^{-2} + \dots + \varepsilon_{-x_0}^{b_0}(X) \cdot b_0^{-x_0}$. Il en résulte que la précision de cette représentation est de $b_0^{-x_0}$.

► **Deuxième étape :** calcul de la précision requise dans la base cible.

Pour maintenir une précision au moins égale, il faut que l'expression de X en base b_1 contienne au moins x_1 chiffre, avec x_1 vérifiant $b_1^{-x_1} \leq b_0^{-x_0}$. On en déduit que x_1 doit être tel que :

$$x_1 \geq x_0 \cdot \frac{\log(b_0)}{\log(b_1)}$$



CONVERSION DE LA BASE 10 À LA BASE 2 AVEC MAINTIEN DE LA PRÉCISION

On souhaite convertir $X = (0,10)_{10}$ en base 2 en maintenant la précision d'origine. On cherche donc x_1 tel que :

$$x_1 \geq 1 \cdot \frac{\log(10)}{\log(2)} \approx 3,3$$

On a donc $x_1 = 4$, et on peut écrire $X = (0,1)_{10} = (0,0001)_2$.

Méthode de conversion simplifiée

La conversion d'une partie fractionnaire d'une base b_0 vers une base b_1 avec $b_0, b_1 \neq 10$ pose les mêmes problèmes que la conversion d'une partie entière entre ces bases : les opérations arithmétiques devant être effectuées dans la base b_0 , les calculs peuvent vite devenir fastidieux. On peut alors utiliser le même biais que dans le cas des parties entières pour simplifier les traitements à effectuer : on passe par la base 10 comme base intermédiaire. On fait donc deux conversions : une conversion de la base b_0 vers la base 10 (en calculant la valeur du nombre), et une conversion de la base 10 vers la base b_1 (par multiplications successives).



CONVERSION DE LA BASE 7 VERS LA BASE 12

On souhaite convertir $X = (0,423)_7$ en base 12 avec maintien de la précision.

La première étape consiste à calculer la valeur en base 10, avec maintien de la précision. On cherche donc x_{10} tel que :

$$x_{10} \geq 3 \cdot \frac{\log(7)}{\log(10)} \approx 2,4$$

On prend donc $x_{10} = 3$, et on calcule l'expression de X grâce à $X = 4 \cdot 7^{-1} + 2 \cdot 7^{-2} + 3 \cdot 7^{-3}$. On obtient $X = (0,620)_{10}$.

La deuxième étape consiste à passer de la base 10 à la base 12, avec maintien de la précision initiale. On cherche donc x_{12} tel que :

$$x_{12} \geq 3 \cdot \frac{\log(7)}{\log(12)} \approx 2,3$$

On prend donc $x_{12} = 3$, et on applique le principe des multiplications successives pour obtenir les chiffres de la partie fractionnaire en base 12. On obtient alors $X = (0,423)_7 = (0,753)_{12}$.

Cas particulier des conversions entre bases puissances

On s'intéresse ici à la conversion d'un nombre d'une base b_0 vers une base $b_1 = (b_0)^n$, ou d'une base $b_1 = (b_0)^n$ vers la base b_0 . Ce cas particulier est très utile puisque les ordinateurs travaillent en base 2, mais cette base est difficile à manipuler du fait du nombre de chiffres élevés nécessaires à la représentation des nombres. On lui préfère donc la base 16, et les conversions entre ces deux bases sont donc très fréquentes.

Conversion de la base b_0 vers la base $b_1 = (b_0)^n$

Soit $X = (\varepsilon_{x_0-1}^{b_0}(X) \dots \varepsilon_0^{b_0}(X), \varepsilon_{-1}^{b_0}(X) \dots \varepsilon_{-x_0'}^{b_0}(X))_{b_0}$ la représentation d'un nombre fractionnaire en base b_0 . On va procéder par regroupement des chiffres de part et d'autre de la virgule en n -uplets et convertir les n -uplets indépendamment les uns des autres.

Notons que s'il n'y a pas suffisamment de chiffres en fort poids pour constituer un n -uplet, on rajoute des zéros non significatifs.

La conversion de la partie fractionnaire repose sur le même principe : on constitue des n -uplets (en rajoutant le cas échéant des zéros non significatifs en partie droite de la partie fractionnaire), puis on convertit chaque n -uplet en chiffre de la base $b_1 = (b_0)^n$.



CONVERSION DE LA BASE 2 VERS LA BASE 16

Convertissons le nombre $X = (111100,001)_2$ pour l'exprimer en base 16. On peut écrire :

$$\begin{aligned} X &= (0011 \quad 1100 \quad ,0010)_2 \\ X &= (3 \quad C \quad ,2)_{16} \end{aligned}$$

Conversion de la base $b_1 = (b_0)^n$ vers la base b_0

Cette conversion repose sur le principe inverse de celui que l'on vient de détailler. Ici, chaque chiffre du nombre X exprimé en base $b_1 = (b_0)^n$ va être converti en n chiffres de la base b_0 .



CONVERSION DE LA BASE 16 VERS LA BASE 2

Convertissons le nombre $X = (F4,C)_{16}$ pour l'exprimer en base 2. On peut écrire :

$$\begin{aligned} X &= (F \quad 4 \quad ,C)_{16} \\ X &= (1111 \quad 0100 \quad ,1100)_2 \end{aligned}$$

EXERCICES

1-1 Numération

1) Réalisez les opérations suivantes en binaire :

a) $11010011 + 01101110$

b) $11010011 - 01101110$

c) $11010011 * 1011$

d) $11010011 / 1011$

e) $(11010011 - 1011) * 101$

2) Réalisez les opérations suivantes en base 16 :

a) $3A2B75 + FAB7$

b) $3A2B75 - C53AF$

c) $3A2B75 * 2B$

d) $3A2B75/A3$

e) $(3A2B75 - C3B5F) * 3B$

1-2 Changements de bases

1) Effectuez les conversions suivantes :

a) $(97435)_{10} = (?)_{16} = (?)_2$

b) $(2BA35F0)_{16} = (?)_{10} = (?)_2$

c) $(1101011010010011)_2 = (?)_{10} = (?)_{16}$

2) Effectuez les conversions suivantes en conservant au moins la précision initiale du nombre donné :

a) $(97435,4832)_{10} = (?)_{16} = (?)_2$

b) $(2BA35F0, A2BC5)_{16} = (?)_{10} = (?)_2$

c) $(1101011010010011, 10000001)_2 = (?)_{10} = (?)_{16}$

SOLUTIONS

1-1 Numération

1) Réalisez les opérations suivantes en binaire :

a) $11010011 + 01101110$

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

On va procéder en deux étapes. D'abord la soustraction :

$$\begin{array}{cccccccc}
 & & & & 1 & & & \\
 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 - & & & & & 1 & 0 & 1 & 1 \\
 & & & & 1 & & & & \\
 \hline
 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0
 \end{array}$$

			1	1	0	0	1	0	0	0
*								1	0	1
			1	1	0	0	1	0	0	0
		0	0	0	0	0	0	0	0	
	1	1	0	0	1	0	0	0		
	1	1	1	1	1	0	1	0	0	0

a) 3A2B75 + FAB7

$$\begin{array}{rcccccc} & & 1 & 1 & 1 & & \\ & 3 & A & 2 & B & 7 & 5 \\ + & & & F & A & B & 7 \\ \hline & 3 & B & 2 & 6 & 2 & C \end{array}$$

$$\begin{array}{rcccccc}
 & & 1 & 1 & & 1 & 1 \\
 & 3 & A & 2 & B & 7 & 5 \\
 - & & C & 5 & 3 & A & F \\
 & 1 & 1 & & 1 & 1 & \\
 \hline
 & 2 & D & D & 7 & C & 6
 \end{array}$$

	3	A	2	B	7	5
*					2	B
	1	1	1			
2	7	F	D	E	0	7
7	4	5	6	E	A	
	9	C	5	4	C	A
						7

Copyright © 2012 Dunod.

$$\begin{array}{r}
 3 \ A \ 2 \ B \ 7 \ 5 \mid A \ 3 \\
 \hline
 3 \ 2 \ F \mid 5 \ B \ 5 \ B \\
 7 \ 3 \ B \mid \\
 7 \ 0 \ 1 \mid \\
 3 \ A \ 7 \mid \\
 2 \ F \mid \\
 7 \ 8 \ 5 \mid \\
 0 \ 1 \mid \\
 8 \ 4 \mid
 \end{array}$$

On procède à l'identique de ce qu'on a vu pour la division en base 2. On a utilisé la même convention typographique. Le quotient de 3A2B75 par A3 est donc 5B5B, le reste 84.

e) $(3A2B75 - C3B5F) * 3B$

On procède ici aussi en deux étapes. D'abord la soustraction :

$$\begin{array}{r}
 \\
 3 \ A \ 2 \ B \ 7 \ 5 \\
 - C \ 3 \ B \ 5 \ F \\
 \hline
 1 \ 1 1 \\
 \hline
 2 \ D \ F \ 0 \ 1 \ 6
 \end{array}$$

Puis la multiplication :

$$\begin{array}{r}
 \\
 * 3 \ B \\
 \hline
 \\
 1 \ F \ 9 \ 5 \ 0 \ F \ 2 \\
 8 \ 9 \ D \ 0 \ 4 \ 2 \\
 \hline
 A \ 9 \ 6 \ 5 \ 5 \ 1 \ 2
 \end{array}$$

1-2 Changements de bases

1) Effectuez les conversions suivantes :

a) $(97435)_{10} = (?)_{16} = (?)_2$

Pour convertir en base 16 ce nombre donné en base 10, on procède par divisions successives par 16.

$$\begin{array}{ll}
 97435 = 16 * 6089 + 11 & B \\
 6089 = 16 * 380 + 9 & 9 \\
 380 = 16 * 23 + 12 & C \\
 23 = 16 * 1 + 7 & 7 \\
 1 = 16 * 0 + 1 & 1
 \end{array}$$

On obtient alors les chiffres du nombre en base 16, des faibles poids vers les forts poids. On peut donc écrire :

$$(97435)_{10} = (17C9B)_{16}$$

Le principe est le même pour passer à la base 2, à ceci près que les divisions successives sont des divisions par 2.

$$\begin{array}{rcl} 97435 & = & 2 * 48717 + 1 & 1 \\ 48717 & = & 2 * 24358 + 1 & 1 \\ 24358 & = & 2 * 12179 + 0 & 0 \\ 12179 & = & 2 * 6089 + 1 & 1 \\ & & \dots & \end{array}$$

On continue ainsi jusqu'à obtenir un quotient nul. Les chiffres obtenus sont ceux du résultat de la conversion, des faibles poids vers les forts poids. On a donc :

$$(97435)_{10} = (10111110010011011)_2$$

b) $(2BA35F0)_{16} = (?)_{10} = (?)_2$

La conversion de la base 16 à la base 10 se fait par le biais du schéma de Hörner. On a :

$$\begin{aligned} (2BA35F0)_{16} &= 2 \cdot 16^6 + B \cdot 16^5 + A \cdot 16^4 + 3 \cdot 16^3 + 5 \cdot 16^2 \\ &\quad + F \cdot 16^1 + 0 \cdot 16^0 \\ &= 33554432 + 11534336 + 656360 + 12288 + 1280 \\ &\quad + 240 + 0 \\ &= (45757936)_{10} \end{aligned}$$

La conversion vers la base 2 se fait en groupant et en convertissant chaque chiffre hexadécimal en quartet binaire.

$$(2BA35F0)_{16} = (0010 \ 1011 \ 1010 \ 0011 \ 0101 \ 1111 \ 0000)_2$$

c) $(1101011010010011)_2 = (?)_{10} = (?)_{16}$

La conversion de la base 2 vers la base 10 se fait, dans ce cas aussi, en utilisant le schéma de Horner. On a :

$$\begin{aligned} (1101011010010011)_2 &= 1 \cdot 2^{15} + 1 \cdot 2^{14} + 0 \cdot 2^{13} + 1 \cdot 2^{12} + 0 \cdot 2^{11} \\ &\quad + 1 \cdot 2^{10} + 1 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 \\ &\quad + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 \\ &\quad + 1 \cdot 2^0 \\ &= (54931)_{10} \end{aligned}$$

2) Effectuez les conversions suivantes en conservant au moins la précision initiale du nombre donné :

a) $(97435,4832)_{10} = (?)_{16} = (?)_2$

La précision initiale du nombre est de 10^{-4} . On cherche la précision en base 16 :

$$p \geq 4 \cdot \frac{\log(10)}{\log(16)} \approx 3,32$$

On choisit donc $p = 4$. La conversion de la partie entière a été effectuée à la première question ; on utilisera donc directement le résultat.

Pour la partie fractionnaire, on procède par multiplications successives par 16, en retenant à chaque fois la partie entière, et ce jusqu'à concurrence du nombre de bits requis.

$$\begin{array}{rcl} 0,4832 * 16 = 7,7312 & & 7 \\ 0,7312 * 16 = 11,6992 & 11 = & B \\ 0,6992 * 16 = 11,1872 & 11 = & B \\ 0,1872 * 16 = 2,9952 & & 2 \end{array}$$

On a donc : $(97435,4832)_{10} = (17C9B,7BB2)_{16}$.

La précision initiale du nombre est de 10^{-4} . On cherche la précision en base 2 :

$$p \geq 4 \cdot \frac{\log(10)}{\log(2)} \approx 13,29$$

On choisit donc $p = 14$. La conversion de la partie entière a été effectuée à la question précédente ; on utilisera donc directement le résultat.

Pour la partie fractionnaire, on procède par multiplications successives par 2, en retenant à chaque fois la partie entière, et ce jusqu'à concurrence du nombre de bits requis – soit 14.

$$\begin{array}{rcl} 0,4832 * 2 = 0,9664 & & 0,8496 * 2 = 1,6992 \\ 0,9664 * 2 = 1,9328 & & 0,6992 * 2 = 1,3984 \\ 0,9328 * 2 = 1,8656 & & 0,3984 * 2 = 0,7968 \\ 0,8656 * 2 = 1,7312 & & 0,7968 * 2 = 1,5936 \\ 0,7312 * 2 = 1,4624 & & 0,5936 * 2 = 1,1872 \\ 0,4624 * 2 = 0,9248 & & 0,1872 * 2 = 0,3744 \\ 0,9248 * 2 = 1,8496 & & 0,3744 * 2 = 0,7488 \end{array}$$

On a donc :

$$(97435,4832)_{10} = (10111110010011011,01111011101100)_2$$

b) $(2BA35F0, A2BC5)_{16} = (?)_{10} = (?)_2$

La précision requise en base 10 est :

$$p \geq 5 \cdot \frac{\log(16)}{\log(10)} \approx 6,02$$

On prendra donc $p = 7$. On utilise le schéma de Hörner pour passer de la base 16 à la base 10. On peut écrire :

$$\begin{aligned} (2BA35F0, A2BC5)_{16} &= 2 \cdot 10^6 + B \cdot 16^5 + A \cdot 16^4 + 3 \cdot 16^3 \\ &\quad + 5 \cdot 16^2 + F \cdot 16^1 + 0 \cdot 16^0 + A \cdot 16^{-1} \\ &\quad + 2 \cdot 16^{-2} + B \cdot 16^{-3} + C \cdot 16^{-4} + 5 \cdot 16^{-5} \\ &= 45757936 + 0,625 + 0,0078125 \\ &\quad + 0,002685546875 \\ &= 45757936,6356859 \end{aligned}$$

La conversion de la base 16 à la base 2 se fait en convertissant chaque chiffre de la base 16 en quartet binaire. La précision sera nécessairement respectée. On obtient donc :

$$\begin{aligned} (2BA35F0, A2BC5)_{16} \\ = (10101110100011010111110000, 10100010101111000101)_2 \end{aligned}$$

c) $(1101011010010011, 10000001)_2 = (?)_{10} = (?)_{16}$

Pour passer de la base 2 à la base 10, on utilise encore une fois le schéma de Hörner, avec comme précision requise :

$$p \geq 8 \cdot \frac{\log(2)}{\log(10)} \approx 2,4$$

On prendra donc trois chiffres après la virgule. La partie entière a été convertie au cours de la question précédente. On se concentrera donc sur la partie fractionnaire.

$$\begin{aligned} (1101011010010011, 10000001)_2 &= 54931 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} \\ &\quad + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 0 \cdot 2^{-5} \\ &\quad + 0 \cdot 2^{-6} + 0 \cdot 2^{-7} + 1 \cdot 2^{-8} \\ &= 54931 + 0,5 + 0,00390625 \\ &= 54931,500 \end{aligned}$$

CHAPITRE 2

Codage de l'information

PLAN

- 2.1 Codage des informations numériques
- 2.2 Codage des informations non numériques

OBJECTIFS

- Étudier les formats de représentation numériques alphanumériques courants
- Appréhender les codages spéciaux permettant d'assurer l'intégrité des données ou leur compactage.

Nous allons nous pencher sur la manière de représenter les informations dans un ordinateur. Nous étudierons donc le codage des informations numériques (entières ou non) et le codage des informations non numériques (notamment le texte). Nous verrons ensuite comment certains codages permettent de garantir la détection des erreurs lors de la transmission des informations et leur correction éventuelle, ou comment il est possible de compacter les données pour les transmettre et/ou les stocker. Nous terminerons par quelques exercices sur le sujet.

2.1 CODAGE DES INFORMATIONS NUMÉRIQUES

Nous avons vu comment coder des nombres dans différentes bases, comment effectuer des opérations sur ces nombres et comment convertir des nombres d'une base à une autre. Nous allons maintenant nous pencher sur la manière de représenter les informations dans un ordinateur, en commençant par les nombres.

Nous allons présenter dans un premier temps le codage des entiers naturels et relatifs. Nous verrons ensuite le codage des nombres fractionnaires et nous terminerons par les nombres réels.

Codage des nombres entiers naturels

Les ordinateurs étant construits à partir d'éléments bistables, c'est naturellement la base 2 qui est utilisée pour stocker et manipuler les données. Chaque chiffre binaire est appelé **bit** (contraction de *Binary Digit*) ; l'association de huit bits est appelée **octet** (*byte* en anglais). Les microprocesseurs travaillent sur des mots dont la taille est une puissance de 2 ; actuellement, la taille des mots manipulés par la majorité des processeurs est 64 bits (soit 8 octets). Plus l'ordinateur est puissant, plus la taille des mots manipulés est importante.

Le codage d'un entier naturel se fait donc en base 2 dans un ordinateur, sur un mot dont la taille est liée à la puissance du système. Il convient de noter que les opérations arithmétiques peuvent entraîner un **dépassement de capacité**. Cela se produit lorsque la taille des mots manipulés par le processeur ne permet pas de représenter le résultat d'une opération.



DÉPASSEMENT DE CAPACITÉ LORS D'UNE ADDITION

Considérons un processeur travaillant sur des octets (i. e. des mots de huit bits) et réalisant l'opération suivante : $209 + 123$. En base 2, cette opération, s'écrit :

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

Le résultat est 332, soit 101 001 100 en base 2. On constate que ce nombre n'est pas représentable sur huit bits, mais qu'il en nécessite neuf pour être codé. Le microprocesseur ne travaillant que sur huit bits, il ne peut donc pas donner un résultat exact de l'opération : il y a débordement, ou dépassement de capacité. Il donnera donc comme résultat les huit bits de plus faible poids, soit 01 001 100 (76 en base 10), et il mettra à 1 un bit spécial pour indiquer que le résultat donné n'est pas correct.

Le plus petit nombre entier naturel représentable sur n bits est 0, quel que soit n . Le plus grand nombre est $2^n - 1$, lorsque tous les bits sont à 1. On peut donc représenter 2^n entiers naturels sur n bits : il s'agit des nombres compris dans l'intervalle $[0; 2^n - 1]$.

Codage des nombres entiers relatifs

Pour coder un entier relatif, il faut à la fois représenter son signe (+ ou –) et sa valeur absolue. Plusieurs solutions existent.

Représentation en valeur absolue plus signe

C'est la représentation la plus intuitive : on réserve un bit du mot pour représenter le signe, les autres bits servant à coder la valeur absolue.

Généralement, le bit de signe est celui de plus fort poids, et la convention retenue est que 0 représente le signe +, et 1 le signe –.

Si le processeur travaille sur des mots de n bits, il n'en reste donc que $n - 1$ pour coder la valeur absolue. On peut donc représenter les intervalles suivants :

- $[0; 2^{n-1} - 1]$ pour les nombres positifs ;
- $[-(2^{n-1} - 1); 0]$ pour les nombres négatifs.

L'intervalle est donc $[-2^{n-1} + 1; 2^{n-1} - 1]$, avec une double représentation du 0 (–0 et +0).

L'inconvénient majeur de cette représentation est que les opérations arithmétiques ne se font pas de la même manière sur les nombres naturels et relatifs. Par exemple, l'addition de –1 avec +1 sur des mots de huit bits donne :

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

Soit –2, ce qui est faux. Pour que le résultat donne 0, il faudrait jouer sur la notion de dépassement de capacité, et s'arranger pour que l'addition de –1 avec +1 donne 2^n .

Représentation en complément à deux

L'addition de $+x$ avec $-x$ ne pouvant pas donner 0, on utilise le débordement de capacité à notre avantage. Considérons des mots de n bits. Le plus grand nombre entier naturel représentable est $2^n - 1$, ce qui signifie qu'on a un débordement à partir du nombre 2^n .



Par exemple, sur huit bits, le plus petit débordement correspond au bit de rang 8 à 1, les huit autres bits (rang 7 à 0) étant à 0.

On va donc s'arranger pour que la somme de $+x$ avec $-x$ donne 2^n . Il suffit pour cela de coder $-x$ par $2^n - x$.



CODAGE DE -6 SUR HUIT BITS

La représentation de +6 sur huit bits est 00000110. Pour coder -6, on va réaliser l'opération $2^8 - 6$.

$$\begin{array}{r}
 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
 -\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 \\
 \hline
 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0
 \end{array}$$

Restreint aux huit bits de plus faible poids, le résultat est 1111010 ; il s'agit de la représentation de -6.

La représentation de $2^n - x$ s'appelle le complément à deux de x . Plusieurs techniques permettent de l'obtenir :

- la soustraction de x à 2^n comme nous venons de le voir ;
- l'addition de 1 au complément à 1 de x . Le complément à 1 de x est défini comme le résultat de la soustraction $2^n - 1 - x$; il peut être obtenu rapidement en inversant chaque bit de x ;
- la recopie des bits de x du plus faible poids vers les plus forts poids jusqu'au premier 1 inclus, puis l'inversion de tous les autres bits.



CODAGE DE -40 SUR HUIT BITS

Le nombre entier 40 s'écrit en base 2 : 00101000. Pour coder -40, on peut donc utiliser au choix l'une des trois méthodes ci-dessus. Dans tous les cas, on obtient la même représentation : 11011000.

Avec une telle représentation, le bit de plus fort poids indique toujours le signe : 0 si le nombre est positif, 1 si le nombre est négatif. Sur n bits, on peut donc représenter des nombres compris entre :

- 00...0 et 011...1 pour les nombres positifs, soit l'intervalle $[0; 2^{n-1} - 1]$;
- 10...0 et 111...1 pour les nombres négatifs, soit les nombres négatifs dont la valeur absolue est comprise entre 10...0 et 00...1. Cela nous donne l'intervalle $[-2^{n-1}; 0]$.

L'intervalle représentable est donc $[-2^{n-1}; 2^{n-1} - 1]$, et on a une seule représentation pour le zéro. L'avantage considérable de cette représentation est que les opérations arithmétiques se font de la même manière, que les nombres soient des entiers naturels ou des entiers relatifs. Le proces-

seur peut donc ignorer la nature des données qu'il manipule ce qui simplifie sa conception et le jeu d'instructions ; seul le programmeur connaît la sémantique exacte des informations.

Codage des nombres réels en virgule fixe

Le codage d'un nombre réel selon une représentation en **virgule fixe** consiste à déterminer, dans la représentation cible, le nombre de bits affectés à la représentation de la partie entière, et le nombre de bits affectés à la représentation de la partie fractionnaire.

Considérons des mots de n bits. On va fixer arbitrairement p bits sur les n du mot pour le codage de la partie fractionnaire, les $n - p$ bits restant étant utilisés pour coder la partie entière. En utilisant ce principe, on peut effectuer des calculs sur des valeurs réelles (ou plus exactement sur des valeurs décimales) en utilisant des circuits dédiés à l'arithmétique entière.

L'inconvénient majeur de cette notation est son incapacité à s'adapter à l'échelle du nombre à représenter : la précision reste fixe (dans notre cas 2^{-p}) que le nombre soit grand ou petit. Pour pallier à ce problème, on a usuellement recours à la représentation en virgule flottante.

Codage des nombres réels en virgule flottante

La représentation d'un nombre réel en **virgule flottante** consiste à coder la notation scientifique de ce nombre en base deux.

Notation scientifique et codage des nombres

On rappelle que la **notation scientifique** d'un nombre revient à le représenter sous la forme $\pm man \cdot b^{exp}$, où b représente la base, man s'appelle la **mantisse** ou significande et est tel que $0 \leq man < b$, et exp s'appelle l'**exposant**. Si man ne vérifie pas la relation $0 \leq man < b$, on dit que la représentation est **dénormalisée**.



En base 10, la notation scientifique normalisée des nombres réels est de la forme $\pm man \cdot 10^{exp}$ avec $man \in [0; 10[$.

Cette représentation présente l'avantage de connaître immédiatement l'ordre de grandeur du nombre, donné par la valeur positive ou négative de l'exposant exp . Par ailleurs, si on fixe le nombre de chiffres de la mantisse, on obtient une représentation dont la précision s'adapte à l'ordre de grandeur du nombre codé.

La norme IEEE754

Cette norme définit le codage binaire des nombres en virgule flottante selon différentes précisions : la simple précision (où les nombres sont codés sur 32 bits), la double précision (qui permet de coder des nombres sur 64 bits), et la quadruple précision (où les nombres sont codés sur 128 bits).

Quelle que soit la précision, la mantisse est toujours codée sur les plus faibles poids, et le bit de signe en plus fort poids – l'exposant étant donc codé au milieu.

Il convient de noter que l'exposant est **codé à l'excès**, c'est-à-dire qu'il est décalé. Si e le nombre de bits utilisés pour coder l'exposant, alors on soustraira $2^{e-1} - 1$ à l'exposant qui sera considéré comme un entier naturel. Ceci permet d'effectuer rapidement et simplement des comparaisons entre nombres flottants.



Si on utilise le format simple précision de la norme IEEE754, alors les nombres sont codés sous la forme $(-1)^{sign} \cdot man \cdot 2^{exp-127}$, où *sign* représente le bit de signe (bit de rang 31), *exp* représente l'exposant (bits de rang 30 à 23) et *man* représente la mantisse (bits de rang 22 à 0).

Certaines valeurs de l'exposant permettent de modifier l'interprétation du nombre. Supposons que le nombre de bits utilisés pour coder l'exposant soit noté e . Cet exposant peut donc recevoir comme valeur soit 0 (nombre dénormalisé), soit $2^e - 1$ (significations spéciales), soit un nombre compris entre 1 et $2^e - 2$. Dans ce cas, le nombre est normalisé et la forme :

s	exp	mantisse
---	-----	----------

code le nombre $(-1)^s \cdot 1, mantisse \cdot 2^{exp-(2^{e-1}-1)}$

Notons que lorsque le nombre est normalisé, le 1 précédant la virgule est omis dans le codage de la mantisse.

2.2 CODAGE DES INFORMATIONS NON NUMÉRIQUES ET CODES SPÉCIAUX

Outre les nombres, les ordinateurs doivent manipuler des informations textuelles ou des données binaires brutes. Lorsque ces données doivent être échangées ou stockées, il peut être intéressant d'une part de vérifier que des erreurs ne se sont pas introduites dans les données au cours de

la transmission ou de l'archivage, et d'autre part de chercher à réduire le volume des données.

Codage du texte

Historiquement le texte dans les ordinateurs est codé en **ASCII** (acronyme de *American Standard Code for Information Interchange*). Il s'agit d'un code sur 7 bits permettant de représenter les lettres utilisées dans les langues anglo-saxonnes – donc pas les caractères accentués, ainsi que des caractères de contrôles utilisés par les télétypes à l'époque.

Des extensions à ce code ont été ajoutées en utilisant huit bits par la suite pour permettre la représentation des caractères accentués. Néanmoins, compte tenu du nombre de caractères nécessaires à l'écriture de l'ensemble des langues, aucune extension ne permettait un codage exhaustif.

L'évolution des technologies a permis l'émergence d'un nouveau standard, appelé **Unicode**. Cette norme définit les caractères et symboles graphiques utiles, leurs représentations et leurs codages suivant la taille des mots utilisés (8 bits, 16 bits, 32 bits).

Codes détecteurs d'erreur

Lors de la transmission ou du stockage d'une information, des erreurs sont susceptibles de se produire. Un **code détecteur d'erreur** a vocation à permettre la détection de ces erreurs. Il est nécessaire cependant que le nombre d'erreurs ne soit pas trop important, sans quoi le code risque de ne pas détecter les erreurs se produisant.

Pour pouvoir détecter les erreurs, l'information transmise doit être redondante ; cela signifie que des bits supplémentaires doivent être insérés dans l'information.

La manière la plus répandue de détecter des erreurs est d'ajouter, pour chaque octet, un bit de parité. Ce bit surnuméraire peut, au choix, permettre d'avoir un nombre total de 1 pair ou impair selon la stratégie retenue. On parle alors de **parité paire** (le nombre de bits, y compris le bit de parité, est pair) ou de **parité impaire** (le nombre total de bits, y compris le bit de parité, est impair).

Codes correcteurs

On a vu que des erreurs étaient susceptibles de se produire lors de la transmission ou du stockage d'une information. Un **code correcteur**

d'erreur a vocation à permettre la détection et la correction de ces erreurs. Comme dans le cas des codes détecteurs d'erreur, il est nécessaire que le nombre d'erreurs ne soit pas trop important.

Les techniques utilisées pour la correction des erreurs reposent aussi sur le principe d'une information redondante transmise – ou stockée – en même temps que l'information de base.

Un code correcteur typique est celui de **Hamming**. Ce dernier a formalisé une famille de codes correcteurs. Étudions le code binaire (7,4) à titre d'exemple. Il permet de transférer quatre bits de données et trois bits de contrôle, et de détecter et corriger une erreur sur les sept bits transmis. Les bits de contrôles sont placés dans le message final aux positions correspondant aux puissances de deux successives, donc dans notre cas aux positions 1, 2 et 4 soit aux rangs 0, 1 et 3.

Calcul des bits de parité

Pour établir les valeurs des bits de contrôle, ou **bits de parité**, on utilise une **matrice de parité**. Cette matrice M_p est constituée des nombres de 1 à 7 codés en colonne :

$$M_p = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Si le message à transmettre est constitué des bits m_0, m_1, m_2 et m_3 et qu'on nomme les bits de contrôle c_0, c_1 et c_2 alors en définissant le vecteur V_m par :

$$V_m = \begin{pmatrix} c_0 \\ c_1 \\ m_0 \\ c_2 \\ m_1 \\ m_2 \\ m_3 \end{pmatrix}, \text{ on doit avoir } M_p \cdot V_m = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

On en déduit les équations suivantes (les additions se faisant modulo 2) :

$$\begin{cases} c_2 + m_1 + m_2 + m_3 = 0 \\ c_1 + m_0 + m_2 + m_3 = 0 \\ c_0 + m_0 + m_1 + m_3 = 0 \end{cases}$$

Ces équations permettent de calculer les bits de contrôle. Le message envoyé est alors $m_3 m_2 m_1 c_2 m_0 c_1 c_0$.



CODAGE DE HAMMING D'UN MESSAGE

On souhaite envoyer le message 1101 en utilisant le code de Hamming (7,4). Il nous faut donc résoudre le système d'équations :

$$\begin{cases} c_2 + 0 + 1 + 1 = 0 \\ c_1 + 1 + 1 + 1 = 0 \\ c_0 + 1 + 0 + 1 = 0 \end{cases}$$

On a donc :

$$\begin{cases} c_2 = 0 \\ c_1 = 1 \\ c_0 = 0 \end{cases}$$

Il en résulte que le message à envoyer est : 1100110.

Vérification du message reçu

Soit V_r le vecteur représentant le message reçu, défini par :

$$V_r = \begin{pmatrix} p_0 \\ p_1 \\ r_0 \\ p_2 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix}$$

Si aucune erreur de transmission ne s'est produite, alors on a :

$$M_p \cdot V_r = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

On en déduit, dans ce cas, que $V_n = V_m$. L'information est correcte, et pour retrouver le message d'origine, il suffit d'ôter les bits de parité dans le message reçu.



MESSAGE REÇU VALIDE

Supposons qu'on reçoive 1001100. Deux questions se posent : quelle est l'information contenue, et cette information est-elle correcte ?

Pour le savoir, nous devons calculer le produit matriciel du vecteur reçu avec la matrice de parité.

On a :

$$M_p \cdot V_r = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Le vecteur étant nul, la réception est correcte. L'information initialement contenue est : 1001.

Si le vecteur n'est pas nul, cela signifie qu'une erreur a eu lieu durant la transmission. Dans ce cas, on a un vecteur d'erreur de transmission V_e comportant un 1 dans une position et vérifiant $V_r = V_m + V_e$ (les opérations d'addition se faisant modulo deux). En utilisant la matrice de parité, on a :

$$\begin{aligned} V_r = V_m + V_e &\Rightarrow M_p \cdot V_r = M_p \cdot V_m + M_p \cdot V_e \\ &\Rightarrow M_p \cdot V_r = M_p \cdot V_e \end{aligned}$$

Si l'erreur se trouve au niveau du bit de position i (c'est-à-dire au niveau du bit de rang $i - 1$), alors le résultat du produit matriciel $M_p \cdot V_r$ est la colonne i qui code i en binaire. On peut donc facilement retrouver l'origine d'une erreur de transmission et la corriger. Par contre, dans le cas d'erreurs de transmission multiples, on pourra détecter le problème mais on ne pourra pas corriger le message.



MESSAGE REÇU ERRONÉ

Si on reçoit la séquence 1101100 alors on a :

$$M_p \cdot V_r = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

On vérifie que le bit qui a été modifié est bien celui de rang 5, c'est-à-dire le bit de position 6.

Notons que l'on ajoute parfois un bit de parité à un code de Hamming (7,4). On peut ainsi détecter les erreurs supplémentaires susceptibles de se produire lors de la transmission.

Codes optimaux

Les **codages optimaux** visent à réduire la quantité d'informations nécessaires à la représentation d'informations en se basant sur les statistiques d'utilisation de ces différentes informations, et à produire le code le plus court possible compte tenu de ces probabilités. Différents codes de compression des données existent, mais tous ne sont pas optimaux (comme le code de Shannon-Fano par exemple).

Le principal code utilisé est le **code de Huffman**. Il s'agit d'un algorithme de type glouton qui permet la construction d'un arbre pour déduire le code associé à chaque information. Le principe est le suivant :

- **Étape initiale** : on dispose d'un ensemble de nœuds (les informations à coder) associés à leur probabilité (ou à leur fréquence d'apparition).
- **Étape i** : on regroupe les deux nœuds de plus faibles probabilités (ou ayant la plus faible fréquence d'apparition) et on les remplace par un arbre ayant ces deux nœuds comme feuilles et la somme de ces probabilités (ou des fréquences d'apparition) comme racine.
- **Étape finale** : le code est construit pour chaque information en parcourant l'arbre de la racine jusqu'à la feuille concernée, et en concaténant un 0 si la branche suivie est la branche de gauche, un 1 si la branche suivie est celle de droite.



APPLICATION DU CODE DE HUFFMAN

Supposons qu'on cherche à coder « barnabé a mangé un baba à l'armagnac ». On a la liste de lettres suivante, avec les fréquences d'apparition correspondante : a – 9 ; à – 1 ; b – 4 ; c – 1 ; é – 2 ; g – 2 ; l – 1 ; m – 2 ; n – 4 ; r – 2 ; u – 1 ; espace – 6 ; apostrophe – 1.

Les regroupements successifs donnent la figure 2.1.

On en déduit les codes suivants : a – 10 ; à – 00110 ; b – 010 ; c – 00111 ; é – 11110 ; g – 0010 ; l – 01110 ; m – 0110 ; n – 000 ; r – 1110 ; u – 01111 ; espace – 110 ; apostrophe – 1111

Le message s'écrit donc :

0101011100001001011110110101100110100000010111101100111

100011001010010101100011011001110111111011100110100010001000111, soit 119 bits. Un codage classique aurait nécessité 4 bits par lettre (puisque'il y a 13 lettres différentes). Pour coder ce message de 36 lettres, il aurait donc fallu 144 bits.

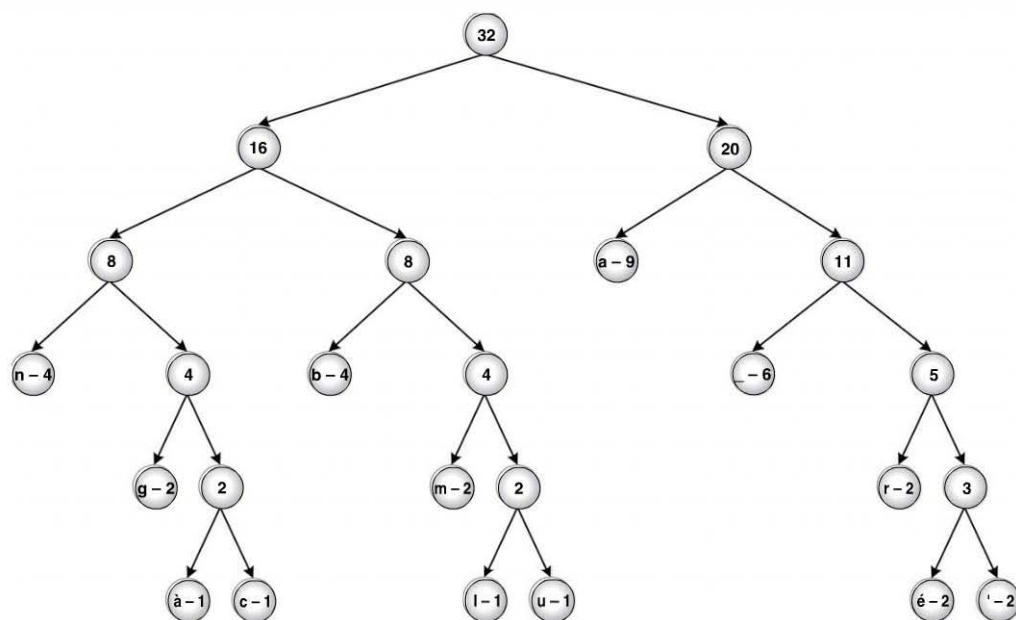


Figure 2.1 Arbre de construction du code de Huffman.

EXERCICES

2-1 Codage des nombres entiers relatifs

En supposant que les nombres sont codés sur 16 bits (2 octets), remplissez le tableau suivant.

En base 10	Valeur absolue en base 2	Valeur absolue en base 16	En valeur absolue + signe	En complément à 2
54621				
-14784				
	1000 0111 0110 1011			
		FDD1		
			0100 1100 0110 0100	
				1011 1110 0010 0000

2-2 Codage des nombres réels

- 1) Donnez la représentation de $(97435,4832)_{10}$.
- 2) Donnez la valeur en base 10 du nombre codé par $(B3C5241F)_{16}$.

2-3 Codes détecteurs et correcteurs d'erreurs, codes optimaux

- 1) On veut transmettre le message suivant par paquets de 9 bits (huit bits de données plus un bit de parité) en utilisant une parité paire : 11010011 10000011 11001110 11110001. Donnez le codage correspondant.
- 2) Reprendre le message précédant pour le transmettre en utilisant le code de Hamming (7,4).
- 3) Utilisez le code de Huffman pour coder le message :
« barbabapaestfanadubaba ».

SOLUTIONS

2-1 Codage des nombres entiers relatifs

Les nombres sont codés sur 16 bits (2 octets).

En base 10	Valeur absolue en base 2	Valeur absolue en base 16	En valeur absolue + signe	En complément à 2
54621	1101 0101 0101 1101	D55D	NR	NR
-14784	0011 1001 1100 0000	39C0	1011 1001 1100 0000	1100 0110 0100 0000
±34667	1000 0111 0110 1011	876B	NR	NR
±64977	1111 1101 1101 0001	FDD1	NR	NR
19556	0100 1100 0110 0100	4C64	0100 1100 0110 0100	0100 1100 0110 0100
-16864	0100 0001 1110 0000	41E0	1100 0001 1110 0000	1011 1110 0010 0000

On rappelle que sur 16 bits, l'intervalle des nombres représentables en valeur absolue plus signe est $[-32767; +32767]$, et l'intervalle des nombres représentables en complément à deux est $[-32768; +32767]$.

- **Première ligne** : le nombre est positif. Le codage de la valeur absolue en base 2 ou 16 se fait donc comme vu précédemment. Le nombre étant positif et la valeur absolue supérieure à 32 768 (soit 2^{15}), le nombre ne peut pas être représenté en valeur absolue plus signe, ni en complément à deux.
- **Deuxième ligne** : le nombre est négatif. La conversion de la valeur absolue en base 2 et 16 se fait comme vu précédemment (en ne tenant pas compte du signe, puisqu'on code la valeur absolue). La valeur absolue étant inférieure à 32 768 (2^{15}), le nombre est représentable (NR) en valeur absolue plus signe (on reprend la valeur absolue, et le nombre étant négatif, on met un 1 sur le bit de plus fort poids) ainsi qu'en complément à deux (le nombre étant négatif, on le code par le complément à deux de sa valeur absolue).
- **Troisième et quatrième ligne** : les nombres étant donnés par leur valeur absolue, on ne peut rien déduire de leur signe. Les valeurs absolues étant supérieures à 32 768, les nombres ne sont représentables ni en valeur absolue plus signe, ni en complément à deux.
- **Cinquième ligne** : le nombre est positif, le bit de plus fort poids de la représentation en valeur absolue plus signe étant égal à 0. Il s'agit donc de la valeur absolue du nombre, que l'on peut recopier directement dans la case 2. Les changements de représentations ne posent aucun problème particulier.
- **Sixième ligne** : le nombre est négatif, le bit de plus fort poids de la représentation en complément à deux étant égal à 1. Pour obtenir sa valeur absolue, on calcule donc le complément à deux que l'on recopie dans la deuxième case.

2-2 Codage des nombres réels

1) $(97435,4832)_{10}$. On a vu, dans l'exercice précédant, la conversion du nombre $(97435,4832)_{10}$ en base 2. Si on réduit l'expression sur 24 bits, on obtient alors : $(97435,4832)_{10} = (10111110010011011,0111101)_2$. Pour que le nombre soit normalisé, il faut le mettre sous la forme « 1,xxx », c'est-à-dire déplacer la virgule vers la gauche de 16 positions. En omettant le premier 1 (implicite), on obtient la mantisse suivante :

0111110010011011011101

Il reste à coder l'exposant. Il faut que $exp - 127$ soit égal 16, donc l'exposant à coder est 143, soit 10001111. La représentation du nombre est donc :

01000111101111100100110110111101

soit, en hexadécimal : $(47BE4DBD)_{16}$.

2) $(B3C5241F)_{16}$ en base 10.

Ce nombre s'écrit, en base 2 :

10110011110001010010010000011111

Le bit de plus fort poids est 1, donc le nombre est négatif. Les huit bits suivants indiquent l'exposant codé à l'excès. Il s'agit de 01100111, soit 103 ; le nombre est normalisé puisque l'exposant est différent de zéro. Le nombre cherché est donc :

$$-1,10001010010010000011111 \cdot 2^{-24}$$

soit :

$$-11000101001001000011111 \cdot 2^{47}$$

soit :

$$-12919839 \cdot 2^{-47} \approx 9,18 \cdot 10^{-8}$$

2-3 Codes détecteurs et correcteurs d'erreurs, codes optimaux

1) 11010011 10000011 11001110 11110001.

Le premier mot contient 5 bits à 1. Pour avoir un nombre de bits à 1 pair, il faut donc rajouter un bit à 1. Le bit de parité est donc 1.

Le deuxième mot contient 3 bits à 1. Il faut donc, là aussi un bit de parité à 1. Le troisième mot contient 6 bits à 1. Le bit de parité doit donc être à 0 pour conserver un nombre de bit à 1 pair. Enfin, le dernier mot contient 5 bits à 1. Il faut donc un bit de parité à 1 pour que le nombre de bits à 1 soit pair. Le message transmis est donc :

110100111 100000111 110011110 111100011

2) On va scinder le message en quartets (groupes de 4 bits) et travailler sur ces groupes de 4 bits.

► **Premier quartet :** 1101. On utilise la matrice de parité donnée dans le cours. On obtient donc les équations suivantes :

$$\begin{cases} c_2 + 0 + 1 + 1 = 0 \\ c_1 + 1 + 1 + 1 = 0 \\ c_0 + 1 + 0 + 1 = 0 \end{cases}$$

On en déduit $c_2 = 0$, $c_1 = 1$ et $c_0 = 0$. Le groupe de 7 bits transmis est donc $m_3m_2m_1c_2m_0c_1c_0$, soit : 1100110. En ajoutant un bit de parité impaire, on a : 11001101.

- **Deuxième et quatrième quartets : 0011.** On utilise la matrice de parité donnée dans le cours. On en déduit $c_2 = 1$, $c_1 = 1$ et $c_0 = 0$. Le groupe de 7 bits transmis est donc $m_3m_2m_1c_2m_0c_1c_0$, soit : 0011110. En ajoutant un bit de parité impaire, on a : 00111101.
- **Troisième quartet : 1000.** On a $c_2 = 1$, $c_1 = 1$ et $c_0 = 1$. Le groupe de 7 bits transmis est donc $m_3m_2m_1c_2m_0c_1c_0$, soit : 1001101. En ajoutant un bit de parité impaire, on a : 10011011.
- **Cinquième quartet : 1100.** On a $c_2 = 0$, $c_1 = 0$ et $c_0 = 0$. Le groupe de 7 bits transmis est donc $m_3m_2m_1c_2m_0c_1c_0$, soit : 1100000. En ajoutant un bit de parité impaire, on a : 11000001.
- **Sixième et septième quartets : 1111.** On a $c_2 = 0$, $c_1 = 1$ et $c_0 = 0$. Le groupe de 7 bits transmis est donc $m_3m_2m_1c_2m_0c_1c_0$, soit : 1111111. En ajoutant un bit de parité impaire, on a : 11111110.
- **Huitième quartet : 0001.** On a $c_2 = 0$, $c_1 = 1$ et $c_0 = 0$. Le groupe de 7 bits transmis est donc $m_3m_2m_1c_2m_0c_1c_0$, soit : 0000110. En ajoutant un bit de parité impaire, on a : 00001101.

Le message transmis est donc :

11001101 00111101 10011011 00111101 11000001 11111110
11111110 00001101

3) Les occurrences des caractères sont les suivantes :

$a : 8 - b : 4 - d : 1 - e : 1 - f : 1 - n : 1 - r : 1 - p : 2 - s : 1 - t : 1 - u : 1$

On effectue les regroupements suivants : d et e , f et n , r et s , t et u , p et (d, e) , (f, n) et (r, s) , B et (t, u) , $(p, (d, e))$ et $((f, n), (r, s))$, a et $(b, (t, u))$, $(a, (b, (t, u)))$ et $((p, (d, e)), ((f, n), (r, s)))$.

On obtient alors le schéma suivant (voir figure 2.2).

Les lettres se codent donc de la manière suivante : $a : 00 - b : 010 - d : 1010 - e : 1011 - f : 1100 - n : 1101 - r : 1110 - p : 100 - s : 1111 - t : 0110 - u : 0111$

Le code correspondant au message est donc :

010 00 1110 010 00 100 00 100 00 1011 1111
0110 1100 00 1101 00 1010 0111 010 00 010 00

Notons que les espaces ne sont ajoutés que pour des raisons de lisibilité. Ce code est non ambigu, comme tous les codes de Huffman.

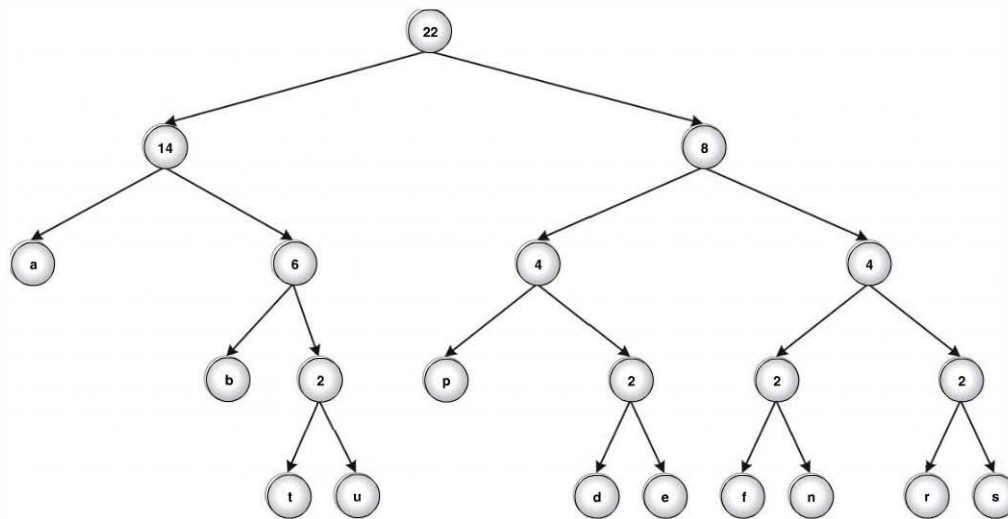


Figure 2.2 Arbre de construction du code de Huffman.

CHAPITRE 3

Algèbre de Boole

PLAN

- 3.1 Définitions et théorèmes
- 3.2 Fonctions booléennes et représentations
- 3.3 Simplification des fonctions booléennes

OBJECTIFS

- Appréhender les algèbres de Boole
- Maîtriser le calcul algébrique dans une algèbre de Boole à deux éléments
- Simplifier les fonctions Booléennes de manière algébriques ou graphique

3.1 DÉFINITIONS ET THÉORÈMES

La définition précise d'une algèbre dépasse le cadre de ce cours. On se reportera pour cela à un cours d'algèbre. Nous allons simplement définir la notion d'algèbre de Boole, en supposant connus les principes de base de l'algèbre, tels que la notion de loi de composition interne.

Définition axiomatique

On appelle **Algèbre de Boole** tout quadruplet $(B, \bar{}, +, \cdot)$ constitué d'un ensemble non vide muni d'au moins deux éléments différents, d'une loi unaire notée $\bar{}$ et de deux lois binaires notées $+$ et $\cdot$, satisfaisant les **postulats d'Huntington** définis par :

Postulat 1 : la loi $+$ (appelée OU logique) est une loi de composition interne : $\forall x, y \in B \quad x + y \in B$.

Postulat 2 : la loi $\cdot$ (appelée ET logique) est une loi de composition interne : $\forall x, y \in B \quad x \cdot y \in B$.

Postulat 3 : il existe un élément, noté 0, neutre pour la loi + :
 $\forall x \in B \quad x + 0 = x.$

Postulat 4 : il existe un élément, noté 1, neutre pour la loi · :
 $\forall x \in B \quad x \cdot 1 = x.$

Postulat 5 : la loi + est commutative : $\forall x, y \in B \quad x + y = y + x.$

Postulat 6 : la loi · est commutative : $\forall x, y \in B \quad x \cdot y = y \cdot x.$

Postulat 7 : la loi + est distributive par rapport à la loi · : $\forall x, y, z \in B$
 $x + (y \cdot z) = (x + y) \cdot (x + z)$

Postulat 8 : la loi · est distributive par rapport à la loi + : $\forall x, y, z \in B$
 $x \cdot (y + z) = (x \cdot y)(x \cdot z)$

Postulat 9 : la loi $\bar{}$ (appelée Complément logique ou opérateur NON) vérifie :

$$\forall x \in B \quad \exists \bar{x} \in B \quad \begin{cases} x + \bar{x} = 1 \\ x \cdot \bar{x} = 0 \end{cases}$$

On peut définir des algèbres de Boole sur des ensembles B infinis, mais on ne s'intéressera ici qu'à l'algèbre de Boole définie sur un ensemble à deux éléments.

Une algèbre de Boole binaire est donc une structure définie par $(B, \bar{}, +, \cdot)$ où $B = \{0, 1\}$. La loi unaire notée $\bar{}$ est appelée **complément logique**, la loi binaire notée + est appelée **ou logique** et la loi binaire notée · est appelée **et logique**.



L'opérateur · est parfois omis dans les expressions algébriques. On note alors xy au lieu de $x \cdot y$.

Nous pouvons remarquer que les différents postulats font apparaître certaines similitudes. C'est le principe de dualité.

Dualité de l'algèbre de Boole

L'observation des **postulats d'Huntington** montre que n'importe lequel reste vrai si on échange la loi + et la loi ·, et si on échange la valeur 0 et la valeur 1. C'est donc vrai en particulier pour l'algèbre de Boole binaire. Il en résulte que toute propriété déduite de ces postulats restera vraie en procédant à cette permutation de lois et de valeurs. C'est le principe de **dualité de l'algèbre de Boole**.

Théorèmes fondamentaux

Nous allons présenter une liste de théorèmes qui sont utiles lors de la manipulation algébrique des fonctions booléennes. Nous donnerons à chaque fois les deux formes correspondant à la dualité de l'algèbre de Boole.

Théorème 1 : éléments absorbants

- $\forall x \in B \quad x + 1 = 1$
- $\forall x \in B \quad x \cdot 0 = 0$

Théorème 2 : idempotence

- $\forall x \in B \quad x + x = x$
- $\forall x \in B \quad x \cdot x = x$

Théorème 3 : involution

$$\forall x \in B \quad \overline{\overline{x}} = x$$

Théorème 4 : absorption

- $\forall x \in B \quad x + x \cdot y = x$
- $\forall x \in B \quad x \cdot (x + y) = x$

Théorème 5 : simplification

- $\forall x, y \in B \quad x + \overline{x} \cdot y = x + y$
- $\forall x, y \in B \quad x \cdot (\overline{x} + y) = x \cdot y$

Théorème 6 : associativité

- $\forall x, y, z \in B \quad x + (y + z) = (x + y) + z = x + y + z$
- $\forall x, y, z \in B \quad x \cdot (y \cdot z) = (x \cdot y) \cdot z = x \cdot y \cdot z$

Théorème 7 : consensus

- $\forall x, y, z \in B \quad x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$
- $\forall x, y, z \in B \quad (x + y) \cdot (\overline{x} + z) \cdot (y + z) = (x + y) \cdot (\overline{x} + z)$

Théorème 8 : théorème de De Morgan

- $\forall x_0, \dots, x_n \in B \quad \overline{x_0 + \dots + x_n} = \overline{x_0} \cdot \dots \cdot \overline{x_n}$
- $\forall x_0, \dots, x_n \in B \quad \overline{\overline{x_0} \cdot \dots \cdot \overline{x_n}} = \overline{x_0} + \dots + \overline{x_n}$

3.2 FONCTIONS BOOLÉENNES ET REPRÉSENTATIONS

Définition : on appelle **fonction booléenne** f toute application de B^n dans B :

$$\begin{array}{ccc} f : & B^n & \longrightarrow B \\ & (x_1, x_2, \dots, x_n) & \longrightarrow f(x_1, x_2, \dots, x_n) \end{array}$$

Toute fonction booléenne peut être exprimée à l'aide des opérateurs ET, OU et Complément. Cela découle directement de la notion d'algèbre de Boole.

Définition : on appelle **groupe logique complet** tout ensemble d'opérateurs tels que toute fonction booléenne puisse être représentée uniquement à partir de ces opérateurs.

Compte tenu de ce qui précède, le groupe d'opérateurs {ET, OU, Complément} est un groupe logique complet.

Notons que parfois, une fonction n'est pas définie pour toutes les combinaisons possibles de ses entrées. C'est notamment le cas si certaines combinaisons des entrées correspondent à un état pour lequel la valeur de la fonction est indifférente, ou alors parce qu'une telle combinaison des entrées est impossible. La valeur de la fonction pour une telle combinaison est alors notée Φ (le symbole grec phi correspond à un 0 et un 1 superposés).

Définition : on appelle **fonction phi-booléenne** f toute application de $B^n = \{0, 1\}^n$ dans $\{0, 1, \Phi\}$:

$$\begin{array}{ccc} f : & B^n & \longrightarrow \{0, 1, \Phi\} \\ & (x_1, x_2, \dots, x_n) & \longrightarrow f(x_1, x_2, \dots, x_n) \end{array}$$

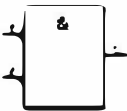
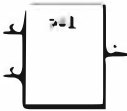
Représentation des fonctions booléennes

On distingue plusieurs représentations possibles pour les fonctions booléenne. Outre la représentation algébrique (qui définit la fonction à l'aide d'opérateurs booléens par le biais d'une équation) on peut définir une fonction par une représentation graphique (en utilisant des symboles logiques) et arithmétique (grâce à une table de vérité). Notons que la représentation algébrique pour une fonction n'étant pas unique, on aura aussi plusieurs représentations graphiques possibles, mais toujours une seule représentation arithmétique.

Représentation graphique

Toute fonction booléenne peut être représentée de manière graphique, en utilisant un ensemble de symboles prédéfinis pour les différents opérateurs booléens. Cette représentation s'appelle un **logigramme**. On distingue deux jeux de symboles, définis par la même norme : le jeu de symboles américain et le jeu de symboles européen. On préférera le jeu de symboles américain, l'identification des opérateurs étant plus rapide. Le tableau 3.1 résume les trois opérateurs que nous avons présentés à ce stade : le ET, le OU et le NON.

Tableau 3.1

Opérateur	Symbole américain	Symbole européen
ET		
OU		
NON		

Pour représenter une fonction de manière graphique à partir de son expression algébrique, il suffit de dessiner les symboles des opérateurs présents dans chaque terme de la fonction en reliant les entrées de ces symboles aux signaux représentant les variables ou aux sorties des opérateurs imbriqués.



REPRÉSENTATION GRAPHIQUE

On a représenté ci-après la fonction $f(x, y, z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$ de manière graphique en utilisant les symboles américains. Pour obtenir cette représentation, on commence par dessiner l'opérateur ET à trois entrées réalisant le terme $\bar{x} \cdot y \cdot z$. Puis on représente le terme $\bar{y} \cdot \bar{z}$ en utilisant un ET à deux entrées. On peut enfin dessiner un opérateur OU à trois entrées ; la première entrée est la sortie du premier ET, la deuxième entrée est la sortie du second ET et la troisième entrée est la variable x .

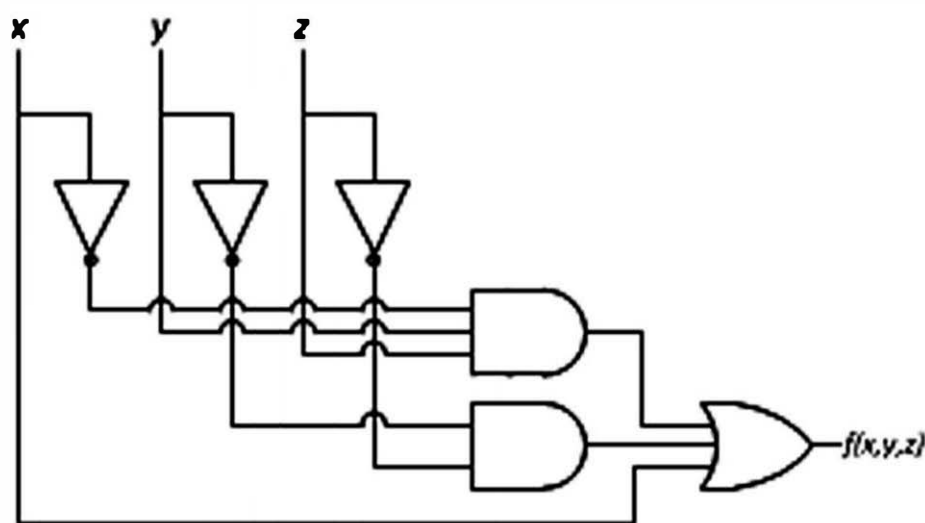


Figure 3.1

Représentation à l'aide d'une table de vérité

Une fonction booléenne f à n variables prend ses valeurs dans B^n et fournit un résultat dans B . Sachant qu'il y a deux éléments dans B , le nombre de combinaisons possibles pour les n variables est 2^n . Chacun de ces combinaisons s'appelle une **combinaison binaire**.

Si on donne la valeur de f pour chaque combinaison binaire, alors la fonction f sera complètement définie. Le tableau récapitulant les valeurs d'une fonction pour chacune de ses entrées s'appelle une **table de vérité**.

Pour établir la table de vérité d'une fonction à partir de son expression algébrique, il suffit de substituer dans l'expression de la fonction chaque variable par la valeur que prend cette variable pour chaque combinaison possible.



TABLE DE VÉRITÉ

On a représenté ci-après la fonction $f(x, y, z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$ en utilisant une table de vérité. Pour l'obtenir, on remplace x , y et z par 0, 0 et 0. On obtient ainsi la valeur de $f(x, y, z)$ pour la première ligne. Puis on remplace x , y et z par 0, 0 et 1, obtenant ainsi la valeur de $f(x, y, z)$ pour la deuxième ligne et ainsi de suite...

Notons que la fonction étant complètement définie par sa table de vérité, cette représentation de la fonction est unique.

x	y	z	$f(x,y,z)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Forme canonique disjonctive (première forme de Lagrange)

Définition : on appelle **littéral** toute variable complémentée ou non complémentée apparaissant dans une expression algébrique.

Définition : on appelle **monôme** la combinaison par l'opérateur ET de n littéraux.



MONÔMES

Les monômes apparaissant dans l'expression de la fonction $f(x,y,z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$ sont $\bar{x} \cdot y \cdot z$ et $\bar{y} \cdot \bar{z}$.

Définition : on appelle **monôme canonique** ou **minterme** pour une fonction un monôme faisant apparaître toutes les variables de cette fonction.



MINTERMES

Un seul monôme canonique apparaît dans l'expression de la fonction $f(x,y,z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$: il s'agit de $\bar{x} \cdot y \cdot z$.

Une fonction à n variables permet de définir 2^n mintermes. Chacun de ces mintermes correspond à une ligne de la table de vérité. Il s'agit de la ligne pour laquelle :

- les variables apparaissant complémentées dans l'expression du min-terme sont à 0 dans la table de vérité ;
- les variables apparaissant non complémentées dans l'expression du minterme sont à 1 dans la table de vérité.

Chaque minterme vaut 0 pour toutes les combinaisons des variables, sauf une : celle qui lui correspond.

Définition : soient n variables binaires $x_1, x_2, \dots, x_n$. On associe un symbole à chaque minterme : m_i , où i représente le nombre décimal correspondant à la combinaison des variables associées au minterme.

Théorème 9 : toute fonction booléenne $f(x_1, x_2, \dots, x_n)$ de B^n dans B peut s'écrire sous la forme d'un OU de mintermes. Cette forme s'appelle la **première forme de Lagrange** ou **forme canonique disjonctive**.

L'écriture abrégée d'une fonction sous forme canonique disjonctive est : $f(x_1, x_2, \dots, x_n) = \sum m(i_1, i_2, \dots, i_p)$ où les nombres i_x correspondent aux mintermes apparaissant dans la fonction.

Définition : la forme $f(x_1, x_2, \dots, x_n) = \sum m(i_1, i_2, \dots, i_p)$ est appelée forme $\sum \prod$.



FORME CANONIQUE DISJONCTIVE

Considérons toujours la fonction $f(x, y, z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$. À partir de la table de vérité donnée précédemment, on peut écrire :

$$\begin{aligned} f(x, y, z) &= \bar{x} \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} \\ &\quad + x \cdot y \cdot z \\ &= \sum m(0, 4, 3, 4, 4, 5, 6, 7) \end{aligned}$$

À partir d'une expression algébrique, il faut commencer par mettre la fonction sous forme disjonctive, c'est-à-dire sous la forme d'un OU de monômes. Il suffit ensuite d'utiliser les postulats 9 et 4 pour obtenir cette forme sans passer par la table de vérité.



FORME CANONIQUE DISJONCTIVE

Toujours en utilisant la fonction (déjà sous forme disjonctive)
 $f(x,y,z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$, on a :

$$\begin{aligned}
 f(x,y,z) &= \bar{x} \cdot y \cdot z + (x + \bar{x}) \cdot \bar{y} \cdot \bar{z} + x \\
 &= \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot \bar{y} \cdot \bar{z} + x \\
 &= \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot \bar{y} \cdot \bar{z} + x \cdot (y + \bar{y}) \cdot (z + \bar{z}) \\
 &= \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot \bar{y} \cdot \bar{z} + (x \cdot y + x \cdot \bar{y}) \cdot (z + \bar{z}) \\
 &= \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot z + x \cdot y \cdot \bar{z} \\
 &\quad + x \cdot \bar{y} \cdot z + x \cdot \bar{y} \cdot \bar{z}
 \end{aligned}$$

L'algèbre de Boole étant duale, on va définir similairement les formes canoniques conjonctives.

Forme canonique conjonctive (deuxième forme de Lagrange)

Définition : on appelle **monal** la combinaison par l'opérateur OU de n littéraux.



MONAUX

Les monaux apparaissant dans l'expression de la fonction
 $f(x,y,z) = (x + y + \bar{z}) \cdot (x + \bar{y} + z)$ sont $x + y + \bar{z}$ et $x + \bar{y} + z$.

Définition : on appelle **monal canonique** ou **maxterme** pour une fonction une combinaison de littéraux par l'opérateur OU faisant apparaître toutes les variables de cette fonction.

Une fonction à n variables permet de définir 2^n maxtermes. Chacun de ces maxtermes correspond à une ligne de la table de vérité. Il s'agit de la ligne pour laquelle :

- les variables apparaissant complémentées dans l'expression du maxterme sont à 1 dans la table de vérité ;
- les variables apparaissant non complémentées dans l'expression du maxterme sont à 0 dans la table de vérité.

Chaque maxterme vaut 1 pour toutes les combinaisons des variables, sauf une : celle qui lui correspond.

Définition : soient n variables binaires $x_1, x_2, \dots, x_n$. On associe un symbole à chaque maxterme : M_i , où i représente le nombre décimal correspondant à la combinaison des variables associées au maxterme.

Théorème 10 : toute fonction booléenne $f(x_1, x_2, \dots, x_n)$ de B^n dans B peut s'écrire sous la forme d'un ET de maxtermes. Cette forme s'appelle la **deuxième forme de Lagrange** ou **forme canonique conjonctive**.

L'écriture abrégée d'une fonction sous forme canonique conjonctive est :

$f(x_1, x_2, \dots, x_n) = \prod M(i_1, i_2, \dots, i_p)$ où les nombres i_x correspondent aux maxtermes apparaissant dans la fonction.

Définition : la forme $f(x_1, x_2, \dots, x_n) = \prod M(i_1, i_2, \dots, i_p)$ est appelée forme $\prod \sum$.



FORME CANONIQUE CONJONCTIVE

Considérons la fonction $f(x, y, z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$. À partir de la table de vérité donnée précédemment, on peut écrire : $f(x, y, z) = (x + y + \bar{z}) \cdot (x + \bar{y} + z) = \prod M(1, 2)$.

À partir d'une expression algébrique, il faut en premier lieu utiliser la distributivité de $+$ sur $\cdot$ pour mettre la fonction sous forme conjonctive. Il suffit ensuite d'utiliser les postulats 9 et 3 pour obtenir cette forme sans passer par la table de vérité.

Opérateurs courants

Les opérateurs ET, OU et NON ne sont pas les seuls à être couramment manipulés en logique. Nous allons présenter maintenant quelques opérateurs répandus et montrer leurs caractéristiques remarquables.

L'opérateur Nand (Non-et)

Définition : l'opérateur **Nand** réalise le complément du ET logique des termes auxquels il est appliqué. Il se note par une barre oblique : x/y , et son symbole graphique est celui du ET avec une bulle d'inversion en sortie de l'opérateur.



Figure 3.2

On peut déduire la table de vérité du Nand à deux entrées de celle de l'opérateur ET :

x	x	x/y
0	0	1
0	1	1
1	0	1
1	1	0

L'opérateur Nand est commutatif, mais il n'est pas associatif (on peut définir des opérateurs Nand à n entrées, mais pas en chaînant $n - 1$ opérateurs Nand à deux entrées).

L'opérateur Nand forme à lui seul un groupe logique complet. On peut prouver cette dernière propriété en montrant que les opérateurs ET, OU et NON peuvent être réalisés avec des Nand. De fait, on a $\bar{x} = x/x$, $x + y = (x/x)/(y/y)$ et $x \cdot y = (x/y)/(x/y)$

Cette propriété est remarquable : elle stipule que toute fonction booléenne, aussi complexe soit-elle, peut être exprimée à l'aide d'opérateurs Nand uniquement. Pour ce faire, il suffit de mettre la fonction sous forme disjonctive (c'est-à-dire sous la forme d'un OU de monômes), de complémenter deux fois l'expression précédente puis d'utiliser une fois le théorème de De Morgan.



EXPRESSION À L'AIDE D'OPÉRATEURS NAND UNIQUEMENT

Considérons la fonction $f(x,y,z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$. Cette expression étant déjà sous forme disjonctive, on peut écrire, en suivant les étapes énoncées :

$$\begin{aligned}
 f(x,y,z) &= \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x \\
 &= \overline{\bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x} \\
 &= \overline{\bar{x} \cdot y \cdot z} / \overline{\bar{y} \cdot \bar{z}} / \bar{x} \\
 &= (\bar{x}/y/z) / (\bar{y}/\bar{z}) / (x/x) \\
 &= ((x/x)/y/z) / ((y/y)/(z/z)) / (x/x)
 \end{aligned}$$

L'opérateur Nor (Non-ou)

Définition : l'opérateur **Nor** réalise le complément du OU logique des termes auxquels il est appliqué. Il se note par une flèche verticale : $x \downarrow y$, et son symbole graphique est celui du OU avec une bulle d'inversion en sortie de l'opérateur.



Figure 3.3

On peut déduire la table de vérité du Nor à deux entrées de celle de l'opérateur OU :

x	x	$x \downarrow y$
0	0	1
0	1	0
1	0	0
1	1	0

L'opérateur Nor est commutatif, mais il n'est pas associatif (on peut définir des opérateurs Nor à n entrées, mais pas en chaînant $n - 1$ opérateurs Nor à deux entrées).

L'opérateur Nor forme à lui seul un groupe logique complet. De même que pour le NAND, on a $\bar{x} = x \downarrow x$, $x + y(x \downarrow y) \downarrow (x \downarrow y)$ et $x \cdot y(x \downarrow x) \downarrow (y \downarrow y)$.

Pour exprimer une fonction à l'aide d'opérateurs Nor uniquement, il faut mettre la fonction sous forme conjonctive (c'est-à-dire sous la forme d'un ET de monaux), compléter deux fois l'expression précédente, puis utiliser une fois le théorème de De Morgan.



EXPRESSION À L'AIDE D'OPÉRATEURS NOR UNIQUEMENT

Considérons la fonction $f(x, y, z) = \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x$. On a déjà exprimé cette fonction sous forme conjonctive ; nous ne reprendrons pas le détail ici. On peut donc écrire, en suivant les étapes énoncées :

$$\begin{aligned}
 f(x, y, z) &= \bar{x} \cdot y \cdot z + \bar{y} \cdot \bar{z} + x \\
 &= (x + y + \bar{z}) \cdot (x + \bar{y} + z) \\
 &= \overline{(x + y + \bar{z}) \cdot (x + \bar{y} + z)} \\
 &= \overline{(x + y + \bar{z})} + \overline{(x + \bar{y} + z)} \\
 &= (x \downarrow y \downarrow \bar{z}) \downarrow (x \downarrow \bar{y} \downarrow z) \\
 &= (x \downarrow y \downarrow (z \downarrow z)) \downarrow (x \downarrow (y \downarrow y) \downarrow z)
 \end{aligned}$$

L'opérateur Xor (Ou-exclusif)

Définition : l'opérateur **Xor** réalise le OU exclusif des termes auxquels il est appliqué. Il se note par un signe plus entouré d'un cercle : $x \oplus y$, et son symbole graphique est celui du OU précédé d'un second arc de cercle au niveau des entrées.



Figure 3.4

On a représenté ci-dessous la table de vérité du Xor à deux entrées :

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

L'opérateur Xor possède certaines propriétés particulières (que nous ne démontrerons pas ici, les preuves étant aisées) : il est commutatif, associatif et distributif par rapport au Xor.

L'opérateur Coïncidence

Définition : l'opérateur **Coïncidence** réalise le complément du OU exclusif des termes auxquels il est appliqué. Il se note par un signe point entouré d'un cercle et son symbole graphique est celui du OU exclusif avec une bulle d'inversion en sortie de l'opérateur.

3.3 SIMPLIFICATIONS DES FONCTIONS BOOLÉENNES

Bien que la table de vérité d'une fonction booléenne soit unique, de multiples formes algébriques peuvent être données. Il existe, parmi toutes ces formes possibles, certaines qui sont optimales par rapport au nombre d'opérateurs et au nombre de leurs entrées.

Définition : le passage d'une forme non optimale à une forme optimale s'appelle la **simplification d'une fonction Booléenne**.

La simplification d'une fonction booléenne peut se faire de manière algébrique, en utilisant des algorithmes ou de manière graphique. Dans tous les cas, on va utiliser des principes similaires – seules changent les techniques de mise en œuvre.

Définition : on appelle **monômes adjacents** (respectivement **monaux adjacents**) deux monômes (respectivement monaux) qui ne diffèrent que par un littéral : sous forme normale dans l'un, sous forme complémenté dans l'autre.

Compte tenu du postulat 9, deux monômes adjacents (respectivement monaux adjacents) peuvent être remplacés par un seul monôme (respectivement monal) constitué uniquement des littéraux qui ne changent pas dans les monômes (respectivement monaux). C'est le principe de base exploité dans la simplification des fonctions booléennes.

Simplification par Karnaugh

La méthode de Karnaugh est une méthode graphique pour simplifier les fonctions booléennes. Elle repose sur l'utilisation de tables éponymes pour visualiser les cases adjacentes. Ces tables de Karnaugh peuvent être vues comme des tables de vérités bidimensionnelles, dont les cases sont numérotées en utilisant un code adjacent : le **code de Gray**.

Code de Gray

Le **code de Gray** (ou **code réfléchi**) est un code tel qu'un seul bit change entre deux termes successifs. Pour l'obtenir, on change le bit de plus faible poids permettant d'obtenir un nouveau terme.



CODE DE GRAY À DEUX BITS

Sur deux bits, on a : **00**, puis **01** (le bit de plus faible poids peut être changé sans répétition d'un terme déjà énuméré), puis **11** (car si on passe de 01 à 00 en changeant le bit de rang 0, on retombe sur un terme déjà énuméré ; il faut donc changer le bit de rang 1) puis enfin **10** (on peut changer le bit de rang 0 car le terme 10 n'a pas encore été donné).

Tables de Karnaugh

Une **table de Karnaugh** est similaire à une table de vérité, à deux différences près :

- ▶ elle n'est pas disposée linéairement en colonne (les valeurs de la fonction étant les unes en dessous des autres) mais dans un tableau à deux dimensions (lignes-colonnes) ;
- ▶ les lignes et les colonnes sont numérotées en utilisant le code de Gray.

Ainsi, une table de Karnaugh à trois variables ressemble à :

yz x	00	01	11	10
0				
1				

De même, une table de Karnaugh à quatre variables se construit de la sorte :

yz wx	00	01	11	10
00				
01				
11				
10				

Chaque case d'une table de Karnaugh correspond à une ligne de la table de vérité, et donc à un minterme de la forme canonique disjonctive et à un maxterme de la forme canonique conjonctive. L'objectif de la simplification va être de déterminer les monômes les plus simples permettant de définir une expression minimale pour la fonction.

L'utilisation du code de Gray permet de placer côte à côte les cases qui correspondent à des monômes (respectivement monaux) adjacents. Notons que cette adjacence est vérifiée pour des cases situées sur les bordures inférieure et supérieure d'une part, sur les bordures gauche et droite d'autre part.



CASES ADJACENTES

On a représenté ci-après deux exemples de cases adjacentes : verticalement (cases gris clair) et horizontalement (cases gris foncé).

yz wx	00	01	11	10
00				
01				
11				
10				

Remplissage des tables de Karnaugh

Le remplissage d'une table de Karnaugh se fait de la même manière que le remplissage d'une table de vérité. Chaque monôme correspond à un groupe de cases adjacentes ayant la valeur 1. Si on considère une fonction booléenne $f(x_1, x_2, \dots, x_n)$ de B^n dans B , alors un monôme comprenant p littéraux consistera en un regroupement de 2^{n-p} cases à 1. Notons que si la fonction est définie sous forme conjonctive, chaque monal comportant p littéraux consistera en un regroupement de 2^{n-p} cases à 0.



REPLISSAGE D'UNE TABLE DE KARNAUGH

Nous allons montrer le remplissage d'une table de Karnaugh à quatre variables pour la fonction

$$f(w, x, y, z) = \overline{w} \cdot \overline{x} \cdot z + \overline{w} \cdot x + \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot y \cdot z.$$

Le premier monôme, $\overline{w} \cdot \overline{x} \cdot z$, comporte trois littéraux. Il va donc permettre de remplir deux cases à 1. Ces deux cases sont celles pour lesquelles $w = 0$, $x = 0$ et $z = 1$.

Le deuxième monôme $(\overline{w} \cdot x)$ ayant deux littéraux, il va correspondre à un groupement de quatre cases à 1. Ce sont les cases qui correspondent à $w = 0$ et $x = 1$, c'est-à-dire la deuxième ligne.

On procède de la même manière pour les deux monômes suivants.

Le troisième monôme $(\overline{x} \cdot \overline{y} \cdot z)$ correspond à deux cases : celles pour lesquelles $x = 0$, $y = 0$ et $z = 1$. On note qu'une de ces deux cases est déjà à 1. Le dernier monôme ($w = 0$, $x = 0$ et $z = 1$) correspond lui aussi à deux cases.

Au final, on obtient la table suivante :

yz wx	00	01	11	10
00		1	1	
01	1	1	1	1
11				
10		1	1	

Détermination d'une forme disjonctive simplifiée (base minimale)

La détermination d'une forme simplifiée comporte deux étapes : la détermination des monômes les plus élémentaires intervenant dans la fonction, puis la recherche d'une forme simplifiée à partir de cette liste de monômes.

Pour déterminer les monômes les plus élémentaires de la fonction, on réalise des regroupements de cases portant la valeur 1 dans la table de Karnaugh. Ces regroupements doivent être tels que :

- Le nombre de cases dans le regroupement est une puissance de 2 (1, 2, 4, 8, etc.). Cela garantit que ce regroupement correspond à un monôme.
- Ce regroupement doit être le plus grand possible. Cela garantit que le monôme correspondant est premier.

Pour déterminer l'expression algébrique du monôme correspondant à un regroupement, il suffit de constituer un monôme ne faisant apparaître

que les variables qui gardent une valeur constante pour toutes les cases du regroupement, en complétant celles qui valent zéro.

À partir de la liste de ces monômes élémentaires, on va déterminer une expression minimale de la fonction. On va utiliser deux étapes pour cela :

- On détermine d'abord tous les monômes essentiels. Il s'agit des monômes qui correspondent à des regroupements dont une case au moins n'est contenue dans aucun autre regroupement.
- On détermine ensuite les monômes non essentiels qui complètent la liste précédemment établie pour couvrir la fonction, en cherchant à minimiser leur nombre, puis leur taille.



BASE MINIMALE D'UNE FONCTION

Soit $f(w, x, y, z) = \overline{w} \cdot \overline{x} \cdot z + \overline{w} \cdot x + \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot y \cdot z$. Les regroupements permettant de dresser la liste des monômes simplifiés de la fonction sont :

yz wx	00	01	11	10
00		1	1	
01	1	1	1	1
11				
10		1	1	

Figure 3-5

La liste des monômes élémentaires est : $\overline{w} \cdot x$ (deuxième ligne du tableau, w est constant et égal à zéro, x est constant et égal à un, y et z changent de valeur selon la case du regroupement), $\overline{w} \cdot z$ (regroupement à cheval sur les deux premières lignes du tableau) et $\overline{x} \cdot z$ (regroupement à cheval sur la première et la quatrième ligne).

Le monôme $\overline{w} \cdot x$ est obligatoire, car la case située à l'extrême gauche et celle située à l'extrême droite ne figurent que dans celui-ci.

De même, le monôme $\overline{x} \cdot z$ est obligatoire car les deux cases situées sur la dernière ligne n'apparaissent que dans ce monôme.

Les monômes obligatoires $\overline{w} \cdot x$ et $\overline{x} \cdot z$ couvrant la fonction, ils constituent une expression minimale. L'expression simplifiée de la fonction est donc :

$$f(w, x, y, z) = \overline{w} \cdot x + \overline{x} \cdot z$$

Détermination d'une forme conjonctive simplifiée

L'algèbre de Boole étant duale, on va pouvoir reproduire le principe mis en lumière au paragraphe précédent pour obtenir une forme conjonctive simplifiée. On procédera de la même manière :

- On déterminera les monaux élémentaires de la fonction en constituant des regroupements de cases portant la valeur 0 tels que :
 - Le nombre de cases dans ces regroupements soit une puissance de 2 (1, 2, 4, 8, etc.). Cela garantit que ces regroupements correspondent à des monaux.
 - Ces regroupements soient les plus grands possibles. Cela garantit que les monaux correspondant soient les plus simples possibles.

Pour déterminer l'expression algébrique du monal correspondant à un regroupement, il suffira de constituer un monal ne faisant apparaître que les variables qui gardent une valeur constante pour toutes les cases du regroupement, en complétant celles qui valent un.

- On déterminera la forme conjonctive simplifiée en deux étapes :
 - On choisira dans un premier temps les monaux élémentaires obligatoires, c'est-à-dire ceux qui correspondent à des regroupements dont une case au moins n'appartient à aucun autre regroupement.
 - On choisira ensuite les monaux élémentaires non obligatoires permettant de couvrir la fonction, en cherchant à minimiser leur nombre et leur taille.



FORME CONJONCTIVE SIMPLIFIÉE

Reprenons $f(w, x, y, z) = \overline{w} \cdot \overline{x} \cdot z + \overline{w} \cdot x + \overline{x} \cdot \overline{y} \cdot z + \overline{x} \cdot y \cdot z$. La fonction étant complètement définie, elle vaut zéro pour les cases ne comprenant pas de 1. La table de Karnaugh faisant apparaître les monaux est donc :

yz wx	00	01	11	10
00	0			0
01				
11	0	0	0	0
10	0			0

On peut procéder aux regroupements pour faire apparaître les monaux élémentaires. On a :

yz wx	00	01	11	10
00	0			0
01				
11	0	0	0	0
10	0			0

Figure 3.6

Les monaux élémentaires sont : $\bar{w} + \bar{x}$ (troisième ligne du tableau, w est constant et égal à un, x est constant et égal à un, y et z changent de valeur selon la case du regroupement), $x + z$ (les quatre coins du tableau, qui sont adjacents !) et $\bar{w} + z$ (le regroupement à cheval sur la première et la quatrième colonne).

Les deux premiers monaux sont obligatoires : $\bar{w} + \bar{x}$ car il est le seul à couvrir les deux cases du centre de la troisième ligne, et $x + z$ car il est le seul à couvrir la première et la dernière case de la première ligne.

Ces deux monaux obligatoires couvrant la fonction, une forme simplifiée conjonctive est donc :

$$f(w, x, y, z) = (\bar{w} + \bar{x}) \cdot (x + z)$$

Cas des fonctions phi-booléennes

Lors de la simplification d'une fonction phi-booléenne par une table de Karnaugh, on remplira la table avec les 1 (pour une forme disjonctive simplifiée) ou les 0 (pour une forme conjonctive simplifiée) de la fonction, et on précisera aussi les valeurs Φ correspondant aux combinaisons pour lesquelles la valeur de la fonction est indifférente.

Lors de la constitution des regroupements, on choisira la valeur la plus adaptée dans $\{0, 1\}$ pour remplacer les symboles Φ , en essayant de maximiser la taille des regroupements (car plus le regroupement est grand, moins il faut de littéraux pour le décrire).

EXERCICES

3-1 Transformations

1) Exprimez chacune des fonctions suivantes à l'aide d'opérateurs NAND uniquement.

a) $f(w, x, y, z) = \overline{w} \cdot x \cdot \overline{z} + x \cdot \overline{y} + w \cdot \overline{y} \cdot z$

b) $f(w, x, y, z) = (w + x) \cdot (\overline{y} + \overline{z}) \cdot (\overline{w} + \overline{x} + \overline{y})$

c) $f(w, x, y, z) = (w + y) \cdot (\overline{x} + \overline{z}) \cdot (\overline{w} + \overline{x} + \overline{y}) \cdot (\overline{y} + z + x + \overline{x} \cdot \overline{y})$

2) Reprenez les fonctions de l'exercice précédant et exprimez-les à l'aide d'opérateurs NOR uniquement.

3-2 Simplifications algébriques

1) Donnez la table de vérité de la fonction $f(x, y, z)$ représentée par le logigramme suivant, puis simplifiez cette fonction de manière algébrique.

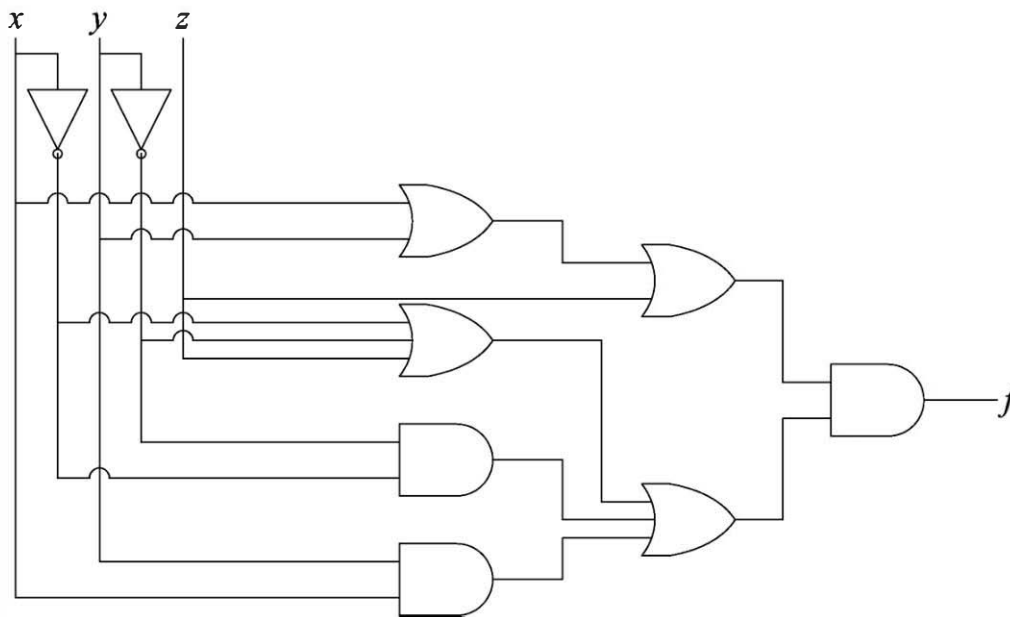


Figure 3.7

2) Simplifiez de manière algébrique la fonction suivante :

$$f(w, x, y, z) = \overline{((\overline{w} + \overline{y}) + x \cdot y) \cdot (w \cdot \overline{x} + (y + \overline{w})) \cdot \overline{x}}$$

3-3 Simplifications par Karnaugh

1) À l'aide de tables de Karnaugh, simplifiez les fonctions booléennes suivantes :

$$\text{a)} f(w, x, y, z) = \sum m(1, 3, 4, 7, 9, 11, 12, 15)$$

$$\text{b)} f(w, x, y, z) = \prod M(3, 5, 6, 7, 12, 14, 15)$$

$$\text{c)} f(w, x, y, z) = y \cdot z + \bar{w} \cdot z + \bar{w} \cdot \bar{x} + w \cdot \bar{x} \cdot \bar{z} + w \cdot x \cdot (y + z)$$

$$\text{d)} f(w, x, y, z) = (\bar{y} + \bar{z}) \cdot (w + x + \bar{y}) \cdot (\bar{x} + y + \bar{z})$$

2) À l'aide de tables de Karnaugh, simplifiez les fonctions phi-booléennes suivantes :

$$\text{a)} f(w, x, y, z) = \prod M(1, 3, 4, 9, 11, 14)$$

$$\text{avec } CI(w, x, y, z) = (6, 12, 13)$$

$$\text{b)} f(w, x, y, z) = \sum m(0, 7, 8, 10, 12, 13)$$

$$\text{avec } CI(w, x, y, z) = (1, 2, 5)$$

$$\text{c)} f(w, x, y, z) = \sum m(0, 1, 3, 5, 6, 7, 11, 13, 14, 15)$$

$$\text{avec } CI(w, x, y, z) = (4)$$

SOLUTIONS

3-1 Transformations

1) On va, dans un premier temps, écrire la fonction sous forme disjonctive – si elle n'est pas déjà sous cette forme. On en profitera alors pour la simplifier lorsque c'est possible. Ensuite, on complémentera deux fois et on appliquera une fois le théorème de De Morgan.

$$\text{a)} f(w, x, y, z) = \bar{w} \cdot x \cdot \bar{z} + x \cdot \bar{y} + w \cdot \bar{y} \cdot z$$

$$\begin{aligned} f(w, x, y, z) &= \overline{\bar{w} \cdot x \cdot \bar{z} + x \cdot \bar{y} + w \cdot \bar{y} \cdot z} \\ &= \overline{\bar{w} \cdot x \cdot \bar{z} \cdot x \cdot \bar{y} \cdot w \cdot \bar{y} \cdot z} \\ &= \overline{\bar{w} \cdot x \cdot \bar{z} / x \cdot \bar{y} / w \cdot \bar{y} \cdot z} \\ &= (\bar{w} / x / \bar{z}) / (x / \bar{y}) / (w / \bar{y} / z) \\ &= ((w / w) / x / (z / z)) / (x / (y / y)) / (w / (y / y) / z) \end{aligned}$$

$$\text{b)} f(w, x, y, z) = (w + x) \cdot (\bar{y} + \bar{z}) \cdot (\bar{w} + \bar{x} + \bar{y})$$

$$\begin{aligned} f(w, x, y, z) &= (w \cdot \bar{y} + w \cdot \bar{z} + x \cdot \bar{y} + x \cdot \bar{z}) \cdot (\bar{w} + \bar{x} + \bar{y}) \\ &= w \cdot \bar{y} \cdot \bar{w} + w \cdot \bar{z} \cdot \bar{w} + x \cdot \bar{y} \cdot \bar{w} + x \cdot \bar{z} \cdot \bar{w} + w \cdot \bar{y} \cdot \bar{x} \\ &\quad + w \cdot \bar{z} \cdot \bar{x} + x \cdot \bar{y} \cdot \bar{x} + x \cdot \bar{z} \cdot \bar{x} + w \cdot \bar{y} \cdot \bar{y} + w \cdot \bar{z} \cdot \bar{y} \\ &\quad + x \cdot \bar{y} \cdot \bar{y} + x \cdot \bar{z} \cdot \bar{y} \\ &= \bar{w} \cdot x \cdot \bar{y} + \bar{w} \cdot x \cdot \bar{z} + w \cdot \bar{x} \cdot \bar{y} + w \cdot \bar{x} \cdot \bar{z} + w \cdot \bar{y} \\ &\quad + w \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} + x \cdot \bar{y} \cdot \bar{z} \end{aligned}$$

$$\begin{aligned}
&= \overline{w} \cdot x \cdot \overline{z} + w \cdot \overline{x} \cdot \overline{z} + w \cdot \overline{y} + x \cdot \overline{y} \\
&= \overline{\overline{w} \cdot x \cdot \overline{z} + w \cdot \overline{x} \cdot \overline{z} + w \cdot \overline{y} + x \cdot \overline{y}} \\
&= \overline{\overline{w} \cdot x \cdot \overline{z}} \cdot \overline{w \cdot \overline{x} \cdot \overline{z}} \cdot \overline{w \cdot \overline{y}} \cdot \overline{x \cdot \overline{y}} \\
&= \overline{\overline{w} \cdot x \cdot \overline{z}} / \overline{w \cdot \overline{x} \cdot \overline{z}} / \overline{w \cdot \overline{y}} / \overline{x \cdot \overline{y}} \\
&= (\overline{w}/x/\overline{z}) / (w/\overline{x}/\overline{z}) / (w/\overline{y}) / (x/\overline{y}) \\
&= ((w/w)/x/(z/z)) / (w/(x/x)/(z/z)) / (w/(y/y)) / (x/(y/y))
\end{aligned}$$

$$\text{c) } f(w, x, y, z) = (w + y) \cdot (\overline{x} + \overline{z}) \cdot (\overline{w} + \overline{x} + \overline{y}) \cdot (\overline{y} + z + \overline{x} \cdot \overline{y})$$

$$\begin{aligned}
f(w, x, y, z) &= (w + y) \cdot (\overline{x} + \overline{z}) \cdot (\overline{w} + \overline{x} + \overline{y}) \cdot (\overline{y} + z + \overline{x} \cdot \overline{y}) \\
&= (w + y) \cdot x \cdot z \cdot (\overline{w} + \overline{x} + \overline{y}) \cdot (\overline{y} + z + x + y) \\
&= (w + y) \cdot x \cdot z \cdot (\overline{w} + \overline{y}) \\
&= (w \cdot \overline{w} + y \cdot \overline{w} + w \cdot \overline{y} + y \cdot \overline{y}) \cdot x \cdot z \\
&= (\overline{w} \cdot y + w \cdot \overline{y}) \cdot x \cdot z \\
&= \overline{w} \cdot x \cdot y \cdot z + w \cdot x \cdot \overline{y} \cdot z \\
&= \overline{\overline{w} \cdot x \cdot y \cdot z + w \cdot x \cdot \overline{y} \cdot z} \\
&= \overline{\overline{w} \cdot x \cdot y \cdot z} \cdot \overline{w \cdot x \cdot \overline{y} \cdot z} \\
&= \overline{\overline{w} \cdot x \cdot y \cdot z} / \overline{w \cdot x \cdot \overline{y} \cdot z} \\
&= (\overline{w}/x/y/z) / (w/x/\overline{y}/z) \\
&= ((w/w)/x/y/z) / (w/x/(y/y)/z)
\end{aligned}$$

2) On va, dans un premier temps, écrire la fonction sous forme conjonctive – si elle n'est pas déjà sous cette forme – en appliquant la distributivité de la loi + sur la loi ·. On en profitera alors pour la simplifier lorsque c'est possible. Ensuite, on complétera deux fois et on appliquera une fois le théorème de De Morgan.

$$\text{a) } f(w, x, y, z) = \overline{w} \cdot x \cdot \overline{z} + x \cdot \overline{y} + w \cdot \overline{y} \cdot z$$

$$\begin{aligned}
f(w, x, y, z) &= (\overline{w} \cdot x \cdot \overline{z} + x) \cdot (\overline{w} \cdot x \cdot \overline{z} + \overline{y}) + w \cdot \overline{y} \cdot z \\
&= x \cdot (\overline{w} + \overline{y}) \cdot (x + \overline{y}) \cdot (\overline{z} + \overline{y}) + w \cdot \overline{y} \cdot z \\
&= x \cdot (\overline{w} + \overline{y}) \cdot (\overline{z} + \overline{y}) + w \cdot \overline{y} \cdot z \\
&= (x \cdot (\overline{w} + \overline{y}) \cdot (\overline{z} + \overline{y}) + w) \cdot (x \cdot (\overline{w} + \overline{y}) \cdot (\overline{z} + \overline{y}) + \overline{y}) \\
&\quad \cdot (x \cdot (\overline{w} + \overline{y}) \cdot (\overline{z} + \overline{y}) + z) \\
&= ((x + w) \cdot (\overline{w} + \overline{y} + w) \cdot (\overline{z} + \overline{y} + w)) \\
&\quad \cdot ((x + \overline{y}) \cdot (\overline{w} + \overline{y} + \overline{y}) \cdot (\overline{z} + \overline{y} + \overline{y}))
\end{aligned}$$

$$\begin{aligned}
& \cdot ((x + z) \cdot (\overline{w} + \overline{y} + z) \cdot (\overline{z} + \overline{y} + z)) \\
&= (w + x) \cdot (w + \overline{z} + \overline{y}) \cdot (x + \overline{y}) \cdot (\overline{w} + \overline{y}) \cdot (\overline{y} + \overline{z}) \cdot (x + z) \\
& \quad \cdot (\overline{w} + \overline{y} + z) \\
&= (w + x) \cdot (\overline{w} + \overline{y}) \cdot (\overline{y} + \overline{z}) \cdot (x + z) \\
&= \overline{\overline{(w + x)} \cdot \overline{(\overline{w} + \overline{y})} \cdot \overline{(\overline{y} + \overline{z})} \cdot \overline{(x + z)}} \\
&= \overline{\overline{(w + x)} + \overline{(\overline{w} + \overline{y})} + \overline{(\overline{y} + \overline{z})} + \overline{(x + z)}} \\
&= (\overline{w + x}) \downarrow (\overline{\overline{w} + \overline{y}}) \downarrow (\overline{\overline{y} + \overline{z}}) \downarrow (\overline{x + z}) \\
&= (w \downarrow x) \downarrow (\overline{w} \downarrow \overline{y}) \downarrow (\overline{y} \downarrow \overline{z}) \downarrow (x \downarrow z) \\
&= (w \downarrow x) \downarrow ((w \downarrow w) \downarrow (y \downarrow y)) \downarrow ((y \downarrow y) \downarrow (z \downarrow z)) \downarrow (x \downarrow z)
\end{aligned}$$

b) $f(w, x, y, z) = (w + x) \cdot (\overline{y} + \overline{z}) \cdot (\overline{w} + \overline{x} + \overline{y})$

$$\begin{aligned}
f(w, x, y, z) &= \overline{\overline{(w + x)} \cdot \overline{(\overline{y} + \overline{z})} \cdot \overline{(\overline{w} + \overline{x} + \overline{y})}} \\
&= \overline{\overline{(w + x)} + \overline{(\overline{y} + \overline{z})} + \overline{(\overline{w} + \overline{x} + \overline{y})}} \\
&= (\overline{w + x}) \downarrow (\overline{\overline{y} + \overline{z}}) \downarrow (\overline{\overline{w} + \overline{x} + \overline{y}}) \\
&= (w \downarrow x) \downarrow (\overline{y} \downarrow \overline{z}) \downarrow (\overline{w} \downarrow \overline{x} \downarrow \overline{y}) \\
&= (w \downarrow x) \downarrow ((y \downarrow y) \downarrow (z \downarrow z)) \\
& \quad \downarrow ((w \downarrow w) \downarrow (x \downarrow x) \downarrow (y \downarrow y))
\end{aligned}$$

c) $f(w, x, y, z) = (w + y) \cdot (\overline{x} + \overline{z}) \cdot (\overline{w} + \overline{x} + \overline{y}) \cdot (\overline{y} + z + \overline{x} \cdot \overline{y})$

$$\begin{aligned}
f(w, x, y, z) &= (w + y) \cdot x \cdot z \cdot (\overline{w} + \overline{x} + \overline{y}) \cdot (\overline{y} + z + x + y) \\
&= (w + y) \cdot x \cdot z \cdot (\overline{w} + \overline{y}) \\
&= \overline{\overline{(w + y)} \cdot \overline{x} \cdot \overline{z} \cdot \overline{(\overline{w} + \overline{y})}} \\
&= \overline{\overline{(w + y)} + \overline{x} + \overline{z} + \overline{(\overline{w} + \overline{y})}} \\
&= (\overline{w + y}) \downarrow \overline{x} \downarrow \overline{z} \downarrow (\overline{\overline{w} + \overline{y}}) \\
&= (w \downarrow y) \downarrow (x \downarrow x) \downarrow (z \downarrow z) \downarrow (\overline{w} \downarrow \overline{y}) \\
&= (w \downarrow y) \downarrow (x \downarrow x) \downarrow (z \downarrow z) \downarrow ((w \downarrow w) \downarrow (y \downarrow y))
\end{aligned}$$

3-2 Simplifications algébriques

1) Donnez la table de vérité de la fonction $f(x, y, z)$ représentée par le logigramme, puis simplifiez cette fonction de manière algébrique.

La table de vérité montre que la fonction est définie par le maxterme M_0 .

On a donc $f(x, y, z) = x + y + z$.

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

2) Simplifiez de manière algébrique la fonction suivante :

$$\begin{aligned}
 f(w,x,y,z) &= \overline{((\overline{w+y}) + x \cdot y) \cdot (w \cdot \bar{x} + (y + \overline{w}))} \cdot \bar{x} \\
 &= \overline{((\overline{w+y}) + x \cdot y) + (w \cdot \bar{x} + (y + \overline{w}))} \cdot \bar{x} \\
 &= ((\overline{w+y}) \cdot \bar{x} \cdot y) + (\overline{w \cdot \bar{x}} \cdot (y + \overline{w})) \cdot \bar{x} \\
 &= ((w+y) \cdot (\bar{x} + \bar{y})) + ((\overline{w \cdot \bar{x}}) \cdot (\bar{y} \cdot w)) \cdot \bar{x} \\
 &= (w \cdot \bar{x} + y \cdot \bar{x} + w \cdot \bar{y} + y \cdot \bar{y}) + (\overline{w \cdot \bar{x}} \cdot w + x \cdot \bar{y} \cdot w) \cdot \bar{x} \\
 &= (w \cdot \bar{x} + y \cdot \bar{x} + w \cdot \bar{y}) + (x \cdot \bar{y} \cdot w) \cdot \bar{x} \\
 &= ((y \cdot \bar{x} + w \cdot \bar{y}) + (x \cdot \bar{y} \cdot w)) \cdot \bar{x} \\
 &= (y \cdot \bar{x} + w \cdot \bar{y} + x \cdot \bar{y} \cdot w) \cdot \bar{x} \\
 &= (y \cdot \bar{x} + w \cdot \bar{y}) \cdot \bar{x} \\
 &= y \cdot \bar{x} + w \cdot \bar{x} \cdot \bar{y}
 \end{aligned}$$

3-3 Simplifications par Karnaugh

1) À l'aide de tables de Karnaugh, simplifiez les fonctions booléennes suivantes :

a) $f(w,x,y,z) = \sum m(1,3,4,7,9,11,12,15)$

Table de Karnaugh remplie					Regroupements				
yz wx	00	01	11	10	yz wx	00	01	11	10
00		1	1		00		1	1	
01	1		1		01	1		1	
11	1		1		11	1		1	
10		1	1		10		1	1	

$$f(w, x, y, z) = x \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot z + y \cdot z$$

b) $f(w, x, y, z) = \prod M(3, 5, 6, 7, 12, 14, 15)$

Table de Karnaugh remplie					Regroupements				
yz wx	00	01	11	10	yz wx	00	01	11	10
00			0		00			0	
01		0	0	0	01		0	0	0
11	0		0	0	11	0		0	0
10					10				

$$f(w, x, y, z) = (w + \bar{x} + \bar{z}) \cdot (\bar{w} + \bar{x} + z) \cdot (w + \bar{y} + \bar{z}) \cdot (\bar{x} + \bar{y})$$

c) $f(w, x, y, z) = y \cdot z + \bar{w} \cdot z + \bar{w} \cdot \bar{x} + w \cdot \bar{x} \cdot \bar{z} + w \cdot x \cdot (y + z)$

Table de Karnaugh remplie					Regroupements				
yz wx	00	01	11	10	yz wx	00	01	11	10
00	1	1	1	1	00	1	1	1	1
01		1	1		01		1	1	
11		1	1	1	11		1	1	1
10	1		1	1	10	1		1	1

$$f(w, x, y, z) = \bar{x} \cdot \bar{z} + \bar{w} \cdot z + x \cdot z + w \cdot y$$

d) $f(w, x, y, z) = (\bar{y} + \bar{z}) \cdot (w + x + \bar{y}) \cdot (\bar{x} + y + \bar{z})$

Table de Karnaugh remplie					Regroupements				
yz wx	00	01	11	10	yz wx	00	01	11	10
00			0	0	00			0	0
01		0	0		01		0	0	
11		0	0		11		0	0	
10			0		10			0	

$$f(w, x, y, z) = (w + x + \bar{y}) \cdot (\bar{y} + \bar{z}) \cdot (\bar{x} + \bar{z})$$

2) À l'aide de tables de Karnaugh, simplifiez les fonctions phi-booléennes suivantes :

a) $f(w,x,y,z) = \prod M(1,3,4,9,11,14)$ avec $CI(w,x,y,z) = (6,12,13)$

Table de Karnaugh remplie					Regroupements				
yz wx	00	01	11	10	yz wx	00	01	11	10
00		0	0		00		0	0	
01	0			Φ	01	0			Φ
11	Φ	Φ		0	11	Φ	Φ		0
10		0	0		10		0	0	

$$f(w,x,y,z) = (\bar{x} + z) \cdot (x + \bar{z})$$

b) $f(w,x,y,z) = \sum m(0,7,8,10,12,13)$ avec $CI(w,x,y,z) = (1,2,5)$

Table de Karnaugh remplie					Regroupements				
yz wx	00	01	11	10	yz wx	00	01	11	10
00	1	Φ		Φ	00	1	Φ		Φ
01		Φ	1		01		Φ	1	
11	1	1			11	1	1		
10	1			1	10	1			1

On voit ici que le deuxième regroupement horizontal (correspondant au monôme $w \cdot x \cdot \bar{y}$) n'est pas nécessaire. La table minimale est donc :

yz wx	00	01	11	10
00	1	Φ		Φ
01		Φ	1	
11	1	1		
10	1			1

et l'expression de la fonction simplifiée est :

$$f(w,x,y,z) = \bar{w} \cdot x \cdot z + w \cdot x \cdot \bar{y} + \bar{x} \cdot \bar{z}$$

c) $f(w, x, y, z) = \prod M(0, 1, 3, 5, 6, 7, 11, 13, 14, 15)$
avec $CI(w, x, y, z) = (4)$

Table de Karnaugh remplie					Regroupements				
yz wx	00	01	11	10	yz wx	00	01	11	10
00	0	0	0		00	0	0	0	
01	Φ	0	0	0	01	Φ	0	0	0
11		0	0	0	11		0	0	0
10			0		10			0	

On voit ici que le regroupement carré situé en haut à droite (correspondant au monôme $w + \bar{z}$) n'est pas nécessaire. La table minimale est donc :

yz wx	00	01	11	10
00	0	0	0	
01	Φ	0	0	0
11		0	0	0
10			0	

et l'expression de la fonction simplifiée est :

$$f(w, x, y, z) = (w + y) \cdot (\bar{x} + \bar{z}) \cdot (\bar{x} + \bar{y}) \cdot (\bar{y} + \bar{z})$$

Logique combinatoire

PLAN

- 4.1 Définitions
- 4.2 La fonction de codage
- 4.3 La fonction de multiplexage
- 4.4 L'additionneur

OBJECTIFS

- Comprendre ce qu'est un circuit combinatoire, et la méthode de synthèse d'un tel circuit
- Appréhender les différents critères d'optimisation des circuits combinatoires
- Connaître les circuits classiques utilisés en logique combinatoire : codeurs, multiplexeurs, additionneurs

4.1 DÉFINITION

On a présenté, au chapitre précédent, quelques opérateurs logiques fondamentaux : le ET, le OU, le NON, le NAND, le NOR, le XOR et l'identité. Grâce à ces opérateurs, nous allons pouvoir concevoir des circuits réalisant des fonctions plus ou moins complexes.

Circuit combinatoire vs circuit séquentiel

Un **circuit combinatoire** est un assemblage de portes logiques implémentant une fonction booléenne, tel qu'une combinaison particulière de ses entrées donne toujours la même sortie.

Si la sortie d'un circuit dépend non seulement de la combinaison des entrées à l'instant t mais aussi de la succession des combinaisons présentées en entrée du circuit avant t , le circuit n'est plus combinatoire ; on dit qu'il est séquentiel. Dans ce cas, la sortie dépendant d'une succession

d'événements, le temps intervient dans l'établissement de la valeur de sortie – ce qui n'est pas le cas pour un circuit combinatoire. Il sera donc nécessaire d'utiliser des éléments de mémorisation. Ces techniques seront étudiées dans les chapitres 5 et 6.

Conception d'un circuit combinatoire

Notre problématique générale dans ce chapitre sera de concevoir un circuit combinatoire à partir de la description d'une fonction logique. Deux étapes seront nécessaires pour cela : établir la fonction logique décrite dans l'énoncé (en essayant le plus souvent d'optimiser cette fonction) puis câbler cette fonction logique en utilisant les opérateurs logiques à notre disposition.

Ces opérateurs peuvent être de simples portes logiques ou des blocs prédéfinis plus complexes. Nous allons, tout au long de ce chapitre, décrire des circuits combinatoires usuels qui auront vocation à être utilisés tels quels dans des circuits plus complexes – par exemple dans le schéma logique d'un processeur ou d'un ordinateur.

Optimisation d'un circuit combinatoire

La première étape de conception d'un circuit combinatoire consiste à établir une fonction logique *optimisée*. Cette optimisation de fonction peut se faire selon plusieurs critères, souvent antagonistes : le coût, la rapidité, la consommation, etc.

Nous allons préciser, dans les deux premiers cas, ce que recouvre cette optimisation et comment la mettre en œuvre.

Optimisation d'une fonction – critère de coût

Le coût de l'implémentation d'une fonction logique est directement lié au nombre de portes logiques et au nombre de leurs entrées. Nous avons présenté, dans le chapitre précédent, quelques méthodes de simplifications, notamment la simplification algébrique et la simplification par les tables de Karnaugh.

Optimisation d'une fonction – critère de rapidité

- Les portes logiques utilisées pour concevoir les circuits combinatoires sont des dispositifs électroniques à base de transistors. Ces derniers fonctionnent en régime non linéaire, c'est-à-dire qu'ils sont soit bloqués, soit saturés. Le passage d'un état à l'autre ne se fait

pas instantanément : un « certain temps » est nécessaire, qui dépend principalement de la technologie utilisée. Il en résulte que chaque porte logique, constituée de transistors, va présenter une latence.

- La latence d'une porte est généralement comprise entre deux valeurs (une valeur minimale et une valeur maximale) qui sont données par le fabricant de la porte.
- À partir d'un circuit combinatoire, on peut donc dresser la liste de tous les chemins existant entre chaque entrée et chaque sortie. En utilisant les valeurs minimales et maximales de chaque porte traversée par un chemin, on peut déterminer les **temps de traversée** minimum et maximum pour ce chemin.

Définition : le chemin d'un circuit dont le temps de traversée maximum est le plus grand s'appelle le **chemin critique** du circuit.

- Optimiser un circuit selon le critère de rapidité consiste à diminuer le temps de traversée du chemin critique. Cela nécessite généralement un nombre plus élevé de portes logiques. Nous verrons un tel exemple d'optimisation pour l'addition binaire.

4.2 LA FONCTION DE CODAGE

Un **décodeur** n vers N est un circuit possédant n entrées et $N = 2^n$ sorties. Il active la sortie dont le numéro est donné en binaire sur ses entrées. Il possède généralement une **entrée de validation** qui permet de valider le fonctionnement du circuit ou non. Dans ce dernier cas, toutes les sorties sont inactives.

Notons que ce circuit fonctionne généralement en **logique négative**, tant au niveau de l'entrée de validation qu'au niveau des sorties. Cela signifie qu'un signal en entrée sera actif lorsqu'il sera à 0 et inactif à 1. De même, la sortie sera considérée comme active lorsqu'elle sera à 0, et inactive lorsqu'elle sera à 1.

À partir de l'énoncé précédent, on peut dresser la table de vérité d'un décodeur 2 vers 4 muni d'une entrée de validation E . Les entrées seront nommées A_1 et A_0 , les sorties S_3 , S_2 , S_1 et S_0 . On a représenté ci-dessous le symbole graphique de ce décodeur.

La présence d'une bulle sur l'entrée de validation et les sorties indique que ces signaux fonctionnent en logique négative.

La table de vérité est la suivante.

E	A_1	A_0	S_3	S_2	S_1	S_0
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

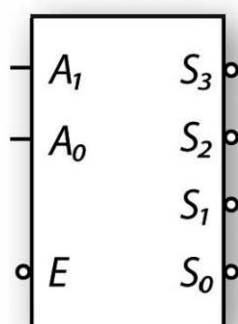


Figure 4.1 Représentation graphique d'un décodeur.

Lorsque $E = 1$, le décodeur est inactif. Quelles que soient les valeurs des entrées A_1 et A_0 , les quatre sorties restent inactives – donc à 1.

Lorsque $E = 0$, le décodeur est actif. La sortie $\bar{S}_i$ qui est activée (c'est-à-dire qui est mise à 0) est celle dont le numéro correspond au nombre formé par A_1A_0 . Ainsi, si $A_1 = 1$ et $A_0 = 1$, le nombre formé par A_1A_0 est 11 soit 3 en décimal. C'est donc S_3 qui est activée (c'est-à-dire mise à 0), les trois autres sorties restant inactives (donc à 1).

On peut déduire les équations logiques des sorties à partir de la table de vérité. On a :

- $S_3 = E + \bar{A}_1 + \bar{A}_0$
- $S_2 = E + \bar{A}_1 + A_0$
- $S_1 = E + A_1 + \bar{A}_0$
- $S_0 = E + A_1 + A_0$

Le schéma logique de ce décodeur est donné ci-après.

Grâce au signal de validation, il est possible d'associer deux décodeurs 2 vers 4 pour former un décodeur 3 vers 8.

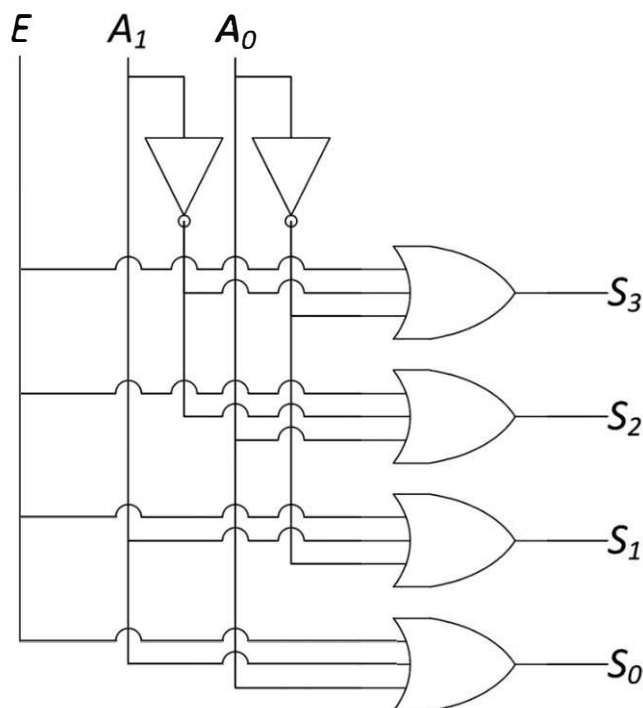


Figure 4.2 Schéma logique d'un décodeur 2 vers 4.

4.3 LA FONCTION DE MULTIPLEXAGE

Le multiplexage consiste à aiguiser vers une sortie unique le signal issu d'une entrée choisie parmi les N du circuit. Le démultiplexage réalise la fonction inverse : le signal issu d'une entrée est envoyé vers une des sorties parmi les N que comporte le circuit.

Un **multiplexeur** N vers 1 (avec $N = 2^n$) est un circuit qui possède une sortie, N entrées de données et n entrées de contrôle. Les entrées de contrôle permettent de déterminer le numéro de l'entrée de donnée dont la valeur sera transmise à la sortie. Le symbole graphique d'un multiplexeur 4 vers 1 est représenté ci-dessous.

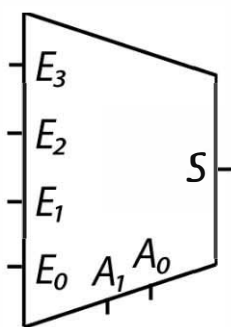


Figure 4.3 Représentation graphique d'un multiplexeur.

La table de vérité d'un multiplexeur 4 vers 1, si elle devait être exhaustive, comporterait 6 entrées (les 4 entrées de données et les 2 entrées de contrôle) et une sortie. Il faudrait donc $2^6 = 64$ lignes pour décrire complètement la fonction de multiplexage dans ce cas. Cela n'est pas envisageable.

Une solution consiste à constater que les valeurs des entrées de données sont recopiées sur la sortie, l'état de cette dernière ne dépendant finalement que des entrées de contrôle. On peut donc dresser une table de vérité indiquant l'état de la sortie uniquement en fonction des entrées de commande, sa valeur étant indiquée par le signal d'entrée de donnée connecté. On obtient :

A_1	A_0	S
0	0	E_0
0	1	E_1
1	0	E_2
1	1	E_3

Pour obtenir l'équation de la sortie S , on va exprimer que l'obtention en sortie de la valeur d'une entrée de donnée est conditionnée par le min-terme correspondant. On obtient :

$$S = \bar{A}_1 \cdot \bar{A}_0 \cdot E_0 + \bar{A}_1 \cdot A_0 \cdot E_1 + A_1 \cdot \bar{A}_0 \cdot E_2 + A_1 \cdot A_0 \cdot E_3$$

Le schéma logique est donné ci-dessous :

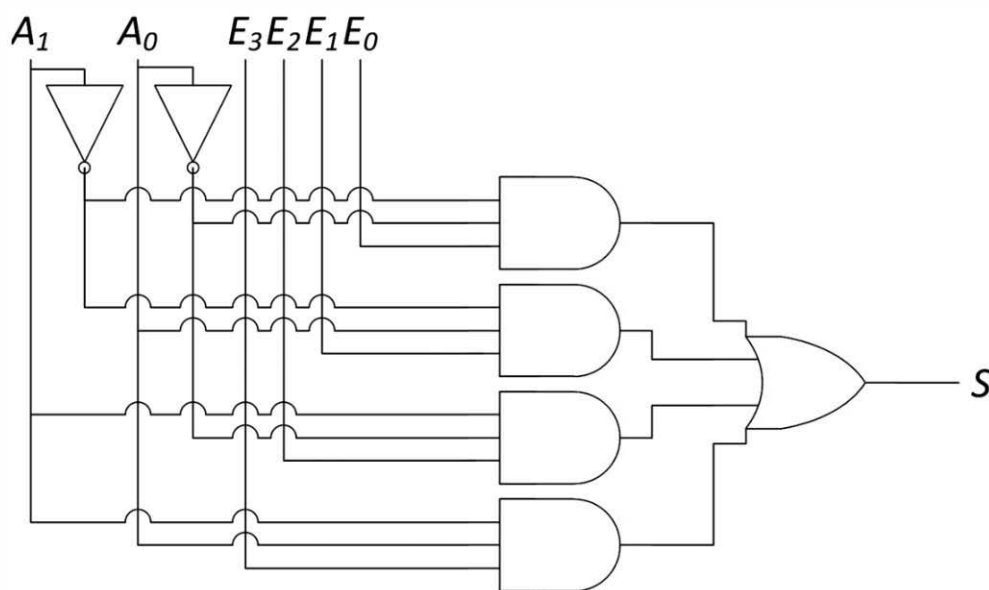


Figure 4.4 Schéma logique d'un multiplexeur 4 vers 1.

On notera que, contrairement aux décodeurs, il n'existe pas d'entrée de validation pour les multiplexeurs. Une telle entrée ne pourrait être intéressante qu'en logique trois états, pour permettre la déconnexion de la sortie du multiplexeur d'un bus ; nous en reparlerons plus tard.

4.4 L'ADDITIONNEUR BINAIRE

L'addition est une opération très fréquemment réalisée : comme nous le verrons, la séquentialité des instructions d'un programme nécessite une addition à chaque cycle pour déterminer l'adresse de l'instruction suivante. De même, les modes d'adressage utilisent massivement l'addition pour des opérations d'indexation. Il s'agit donc d'une opération particulièrement critique, qu'il faut optimiser au maximum en termes de rapidité d'exécution.

De l'opération arithmétique à la fonction logique

La première idée qui vient lorsqu'un circuit combinatoire réalisant l'addition est requis, c'est d'écrire la table de vérité de l'opération, puis d'en déduire les équations logiques des sorties en fonction des entrées. Si cette solution est possible pour l'addition de deux nombres de deux bits chacun, elle n'est guère envisageable au-delà...

On a vu, au chapitre 1, comment était réalisée l'opération d'addition en base B donc notamment en base 2 : on procède par étapes successives, chaque étape permettant d'additionner entre eux les bits de rang courant de chaque nombre, en tenant compte de la retenue (aussi appelée **report**) le cas échéant. On peut calquer notre circuit combinatoire sur ce mécanisme : il suffit d'implémenter un additionneur qui travaille sur 1 bit, puis de chaîner plusieurs de ces modules pour obtenir un opérateur sur le nombre de bits souhaité.

Additionneurs 1 bit

Commençons par additionner deux bits. On cherche à implémenter un circuit tel que celui représenté figure 4.5, qui additionne deux bits a et b , et fournit la somme S et la retenue sortante R_s .

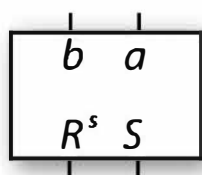


Figure 4.5 Représentation graphique d'un additionneur partiel.

Un tel circuit est appelé **demi-additionneur** ou additionneur partiel. Sa table de vérité est la suivante :

a	b	R^s	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

On en déduit les équations suivantes :

$$R^s = a \cdot b$$

$$S = a \oplus b$$

On peut donc proposer le schéma logique de la figure 4.6.

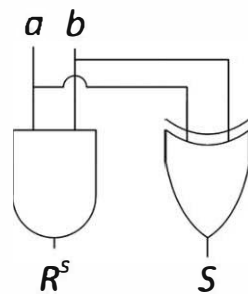


Figure 4.6 Schéma logique d'un additionneur partiel.

Cet additionneur partiel ne peut pas être utilisé tel quel pour réaliser une addition sur n bits, car il ne tient pas compte de la retenue entrante. Un additionneur 1 bit prenant en compte la retenue entrante est appelé **additionneur complet**. Sa représentation graphique est donnée figure 4.7.

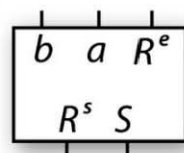


Figure 4.7 Représentation graphique d'un additionneur complet.

On peut dresser la table de vérité de ce circuit :

a	b	R^e	R^s	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

On en déduit les équations suivantes :

$$\begin{aligned}
 R^s &= \bar{a} \cdot b \cdot R^e + a \cdot \bar{b} \cdot R^e + a \cdot b \cdot \overline{R^e} + a \cdot b \cdot R^e \\
 &= (\bar{a} \cdot b + a \cdot \bar{b}) \cdot R^e + a \cdot b \cdot (\overline{R^e} + R^e) \\
 &= (a \oplus b) \cdot R^e + a \cdot b
 \end{aligned}$$

$$\begin{aligned}
 S &= \bar{a} \cdot \bar{b} \cdot R^e + \bar{a} \cdot b \cdot \overline{R^e} + a \cdot \bar{b} \cdot \overline{R^e} + a \cdot b \cdot R^e \\
 &= (\bar{a} \cdot \bar{b} + a \cdot b) \cdot R^e + (\bar{a} \cdot b + a \cdot \bar{b}) \cdot \overline{R^e} \\
 &= (\overline{a \oplus b}) \cdot R^e + (a \oplus b) \cdot \overline{R^e} \\
 &= (a \oplus b) \oplus R^e
 \end{aligned}$$

On peut donc construire un additionneur complet à partir de deux demi-additionneurs (figure 4.8) :

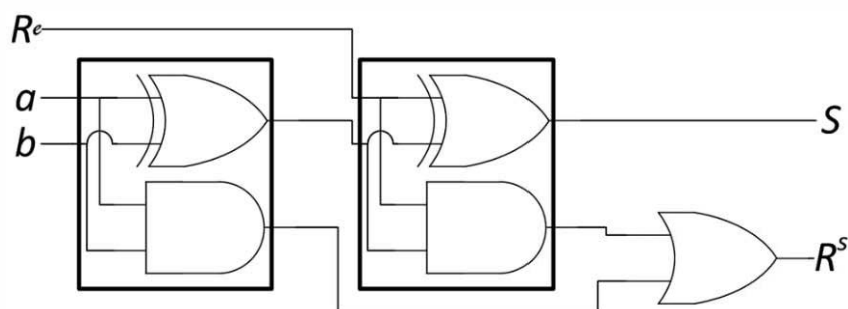


Figure 4.8 Additionneur complet en utilisant des demi-additionneurs.

Dans ce cas, le chemin critique est celui qui permet d'aller de a ou b jusqu'à R^s , puisqu'il faut traverser trois couches logiques.

Notons qu'on peut chercher à simplifier ces équations pour obtenir une forme plus compacte :

Pour la retenue sortante R^s :

bR^e a	00	01	11	10
0			1	
1		1	1	1

$$R^s = a \cdot b + a \cdot R^e + b \cdot R^e$$

Pour la somme S :

bR^e a	00	01	11	10
0		1		1
1	1		1	

$$S = \bar{a} \cdot \bar{b} \cdot R^e + \bar{a} \cdot b \cdot \bar{R}^e + a \cdot \bar{b} \cdot \bar{R}^e + a \cdot b \cdot R^e = a \oplus b \oplus R^e$$

On en déduit le schéma suivant (figure 4.9).

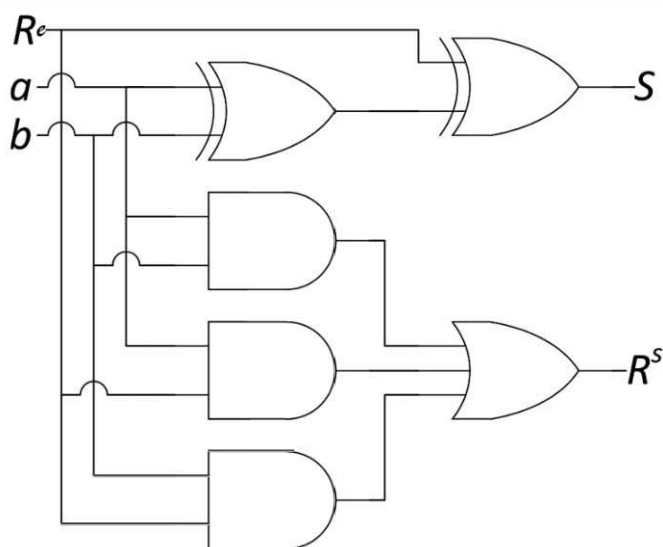


Figure 4.9 Schéma logique d'un additionneur complet.

Ici, le chemin critique nécessite la traversée de deux couches logiques seulement.

Additionneur à report en cascade

Pour concevoir un additionneur n bits à report en cascade, il suffit de chaîner n additionneurs 1 bit complets. On obtient alors le schéma ci-dessous :

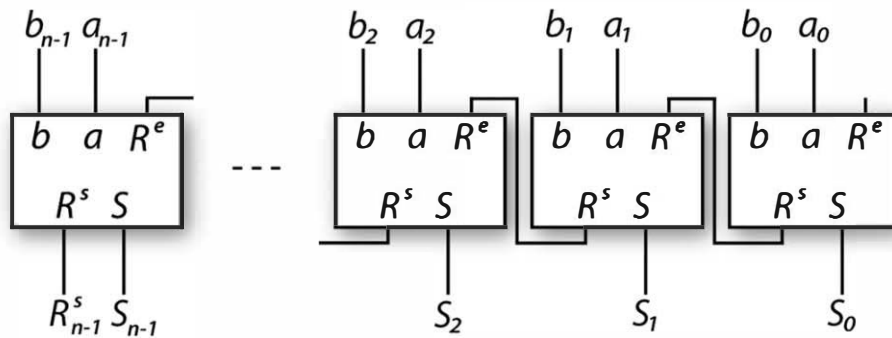


Figure 4.10 Additionneur n bits à report en cascade.

Pour calculer le résultat de l'addition de deux nombres codés sur 32 bits, il faut traverser :

- 65 couches logiques pour obtenir la dernière retenue sortante (et donc le résultat complet) avec l'additionneur complet implémenté sous forme de deux demi-additionneurs ;
- 64 couches logiques pour obtenir le résultat avec l'additionneur complet implémenté de manière directe.

Ces deux solutions sont comparables en termes de coût et de vitesse d'exécution de l'opération. Néanmoins, il ne s'agit pas de la solution la plus rapide...

EXERCICES

4-1 Encodeur de priorité

Un encodeur 4 vers 2 code, sur ses deux sorties S_1 et S_0 , le numéro de l'entrée E_i qui est active (niveau haut).

Si plusieurs entrées sont actives simultanément, c'est le plus grand des numéros des entrées actives qui est codé. On suppose qu'il y a toujours au moins une entrée active.

- 1) Établissez sa table de vérité.
- 2) Déduisez-en les équations de ses sorties.

3) Simplifiez ces équations à l'aide des théorèmes de l'algèbre de Boole (si nécessaire).

4) Donnez le schéma du circuit, réalisé à l'aide de portes NON, ET et OU.

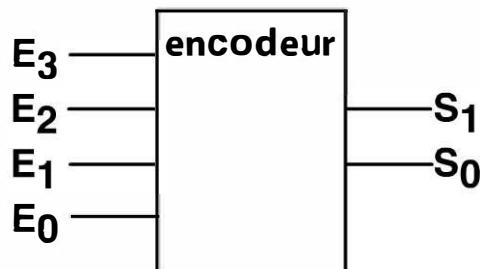


Figure 4.11

4-2 Comparateur

Un comparateur 4 bits accepte, en entrée, deux nombres entiers codés sur 4 bits ($A_3 - A_0$ et $B_3 - B_0$) et génère, en sortie, 3 signaux actifs (niveau haut) si, respectivement, $A > B$, $A < B$, $A = B$.

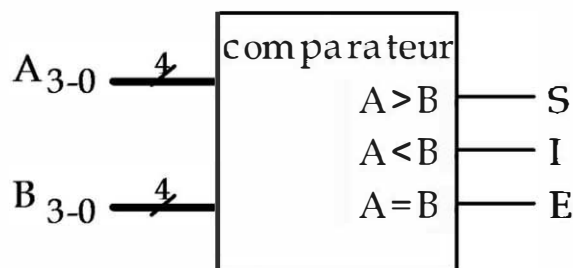


Figure 4.12

1) Établissez sa table de vérité.

2) Déduisez-en les équations de ses sorties. On pourra utiliser l'opérateur coïncidence (le complément du ou exclusif) pour simplifier les équations.

3) Donnez le schéma de S et E avec les opérateurs de votre choix.

4-3 Circuit décaleur

On veut concevoir un circuit combinatoire qui répercute sur ses sorties $S_3 - S_0$ la valeur codée sur ses entrées $E_3 - E_0$ décalée de 0, 1, 2 ou 3 positions à droite (avec introduction de 0). Le nombre de positions de décalage est codé sur les entrées de commande $C_1 C_0$.

1) Établissez les équations de sortie de ce circuit et en faire le schéma logique à l'aide de portes NON, ET et OU.

2) On souhaite pouvoir associer plusieurs de ces circuits pour réaliser des décalages de 0 à 3 positions sur des valeurs codées sur un multiple de 4 bits. Apportez les modifications nécessaires au circuit conçu dans la question précédente.

3) Montrez comment on peut associer deux circuits 4 bits pour réaliser un circuit permettant de décaler une valeur codée sur 8 bits de 0, 1, 2 ou 3 positions vers la droite avec introduction de 0.

4) Montrez comment on peut associer deux circuits 4 bits pour réaliser un circuit permettant de décaler de manière cyclique vers la droite une valeur codée sur 8 bits.

SOLUTIONS

4-1 Encodeur de priorité

1) Table de vérité

E_3	E_2	E_1	E_0	S_1	S_0
0	0	0	0	Φ	Φ
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

2) Équations des sorties

$$\begin{aligned}
 S_1 &= \overline{E_3} \cdot E_2 \cdot \overline{E_1} \cdot \overline{E_0} + \overline{E_3} \cdot E_2 \cdot \overline{E_1} \cdot E_0 + \overline{E_3} \cdot E_2 \cdot E_1 \cdot \overline{E_0} \\
 &\quad + \overline{E_3} \cdot E_2 \cdot E_1 \cdot E_0 + E_3 \cdot \overline{E_2} \cdot \overline{E_1} \cdot \overline{E_0} + E_3 \cdot \overline{E_2} \cdot \overline{E_1} \cdot E_0 \\
 &\quad + E_3 \cdot \overline{E_2} \cdot E_1 \cdot \overline{E_0} + E_3 \cdot \overline{E_2} \cdot E_1 \cdot E_0 + E_3 \cdot E_2 \cdot \overline{E_1} \cdot \overline{E_0} \\
 &\quad + E_3 \cdot E_2 \cdot \overline{E_1} \cdot E_0 + E_3 \cdot E_2 \cdot E_1 \cdot \overline{E_0} + E_3 \cdot E_2 \cdot E_1 \cdot E_0 \\
 S_0 &= \overline{E_3} \cdot E_2 \cdot \overline{E_1} \cdot E_0 + \overline{E_3} \cdot E_2 \cdot E_1 \cdot \overline{E_0} + E_3 \cdot \overline{E_2} \cdot \overline{E_1} \cdot \overline{E_0} \\
 &\quad + E_3 \cdot \overline{E_2} \cdot \overline{E_1} \cdot E_0 + E_3 \cdot \overline{E_2} \cdot E_1 \cdot \overline{E_0} + E_3 \cdot \overline{E_2} \cdot E_1 \cdot E_0 \\
 &\quad + E_3 \cdot E_2 \cdot \overline{E_1} \cdot \overline{E_0} + E_3 \cdot E_2 \cdot \overline{E_1} \cdot E_0 + E_3 \cdot E_2 \cdot E_1 \cdot \overline{E_0} \\
 &\quad + E_3 \cdot E_2 \cdot E_1 \cdot E_0
 \end{aligned}$$

3) Équations simplifiées

$$\begin{aligned}
 S_1 &= \overline{E_3} \cdot E_2 \cdot (\overline{E_1} \cdot (\overline{E_0} + E_0) + E_1 \cdot (\overline{E_0} + E_0)) \\
 &\quad + E_3 \cdot \overline{E_2} \cdot (\overline{E_1} \cdot (\overline{E_0} + E_0) + E_1 \cdot (\overline{E_0} + E_0)) \\
 &\quad + E_3 \cdot E_2 \cdot (\overline{E_1} \cdot (\overline{E_0} + E_0) + E_1 \cdot (\overline{E_0} + E_0))
 \end{aligned}$$

Or $x + \overline{x} = 1$ et $x \cdot 1 = x$. Donc :

$$\begin{aligned}
 S_1 &= \overline{E_3} \cdot E_2 \cdot (\overline{E_1} + E_1) + E_3 \cdot \overline{E_2} \cdot (\overline{E_1} + E_1) \\
 &\quad + E_3 \cdot E_2 \cdot (\overline{E_1} + E_1) \\
 &= \overline{E_3} \cdot E_2 + E_3 \cdot \overline{E_2} + E_3 \cdot E_2 \\
 &= \overline{E_3} \cdot E_2 + E_3 \cdot (E_2 + \overline{E_2}) \\
 &= \overline{E_3} \cdot E_2 + E_3
 \end{aligned}$$

Or $x + \overline{x} \cdot y = x + y$. Donc :

$$S_1 = E_3 + E_2$$

De la même manière, pour les 8 derniers termes, on a :

$$S_0 = \overline{E_3} \cdot E_2 \cdot \overline{E_1} \cdot E_0 + \overline{E_3} \cdot E_2 \cdot E_1 \cdot \overline{E_0} + E_3$$

On en déduit :

$$S_0 = \overline{E_3} \cdot (E_2 \cdot \overline{E_1} \cdot E_0 + E_2 \cdot E_1 \cdot \overline{E_0}) + E_3$$

D'où :

$$S_0 = E_2 \cdot \overline{E_1} \cdot E_0 + E_2 \cdot E_1 \cdot \overline{E_0} + E_3$$

4) Schéma logique.

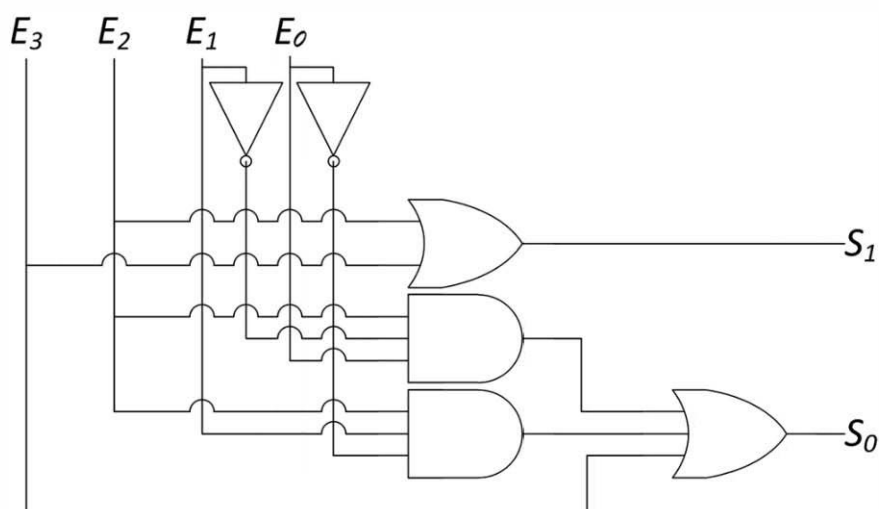


Figure 4.13

4-2 Comparateur

1) Une table de vérité exhaustive n'est pas envisageable, car avec 8 entrées, il y a 256 combinaisons différentes – et il faudrait donc 256 lignes. On peut établir un parallèle avec la manière de comparer des nombres en base 10 pour en déduire la fonction booléenne de comparaison arithmétique.

En base 10, 174 est plus petit que 512 car le chiffre des centaines du premier nombre (1) est plus petit que le chiffre des centaines du second nombre (5). De même, 174 est plus petit que 195 car les chiffres des centaines étant identiques pour les deux nombres (1), le chiffre des dizaines du premier nombre (7) est plus petit que le chiffre des dizaines du second nombre (9).

En notant x dans la table de vérité pour n'importe quelle valeur d'une entrée, on peut dresser la table de vérité partielle suivante :

A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	S	E	I
0	x	x	x	1	x	x	x	0	0	1
1	x	x	x	0	x	x	x	1	0	0
0	0	x	x	0	1	x	x	0	0	1
0	1	x	x	0	0	x	x	1	0	0
1	0	x	x	1	1	x	x	0	0	1
1	1	x	x	1	0	x	x	1	0	0
0	0	0	x	0	0	1	x	0	0	1
0	0	1	x	0	0	0	x	1	0	0

0	1	0	x	0	1	1	x	0	0	1
0	1	1	x	0	1	0	x	1	0	0
1	0	0	x	1	0	1	x	0	0	1
1	0	1	x	1	0	0	x	1	0	0
1	1	0	x	1	1	1	x	0	0	1
1	1	1	x	1	1	0	x	1	0	0
0	0	0	0	0	0	0	1	0	0	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	1	1	0	0	1
0	0	1	1	0	0	1	0	1	0	0
0	1	0	0	0	1	0	1	0	0	1
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	1	1	0	0	1
0	1	1	1	0	1	1	0	1	0	0
1	0	0	0	1	0	0	1	0	0	1
1	0	0	1	1	0	0	0	1	0	0
1	0	1	0	1	0	1	1	0	0	1
1	0	1	1	1	0	1	0	1	0	0
1	1	0	0	1	1	0	1	0	0	1
1	1	0	1	1	1	0	0	1	0	0
1	1	1	0	1	1	1	1	0	0	1
1	1	1	1	1	1	1	0	1		0

Toutes les autres combinaisons correspondent à $E = 1$.

2) À partir de la table de vérité partielle, on déduit :

$$\begin{aligned}
 S &= A_3 \cdot \overline{B_3} + \overline{A_3} \oplus \overline{B_3} \cdot (A_2 \cdot \overline{B_2}) \\
 &\quad + \overline{A_3} \oplus \overline{B_3} \cdot \overline{A_2} \oplus \overline{B_2} \cdot (A_1 \cdot \overline{B_1}) \\
 &\quad + \overline{A_3} \oplus \overline{B_3} \cdot \overline{A_2} \oplus \overline{B_2} \cdot \overline{A_1} \oplus \overline{B_1} \cdot (A_0 \cdot \overline{B_0}) \\
 I &= \overline{A_3} \cdot B_3 + \overline{A_3} \oplus \overline{B_3} \cdot (\overline{A_2} \cdot B_2) \\
 &\quad + \overline{A_3} \oplus \overline{B_3} \cdot \overline{A_2} \oplus \overline{B_2} \cdot (\overline{A_1} \cdot B_1) \\
 &\quad + \overline{A_3} \oplus \overline{B_3} \cdot \overline{A_2} \oplus \overline{B_2} \cdot \overline{A_1} \oplus \overline{B_1} \cdot (\overline{A_0} \cdot B_0)
 \end{aligned}$$

$$E = \overline{A_3 \oplus B_3} \cdot \overline{A_2 \oplus B_2} \cdot \overline{A_1 \oplus B_1} \cdot \overline{A_0 \oplus B_0}$$

3) Schéma logique

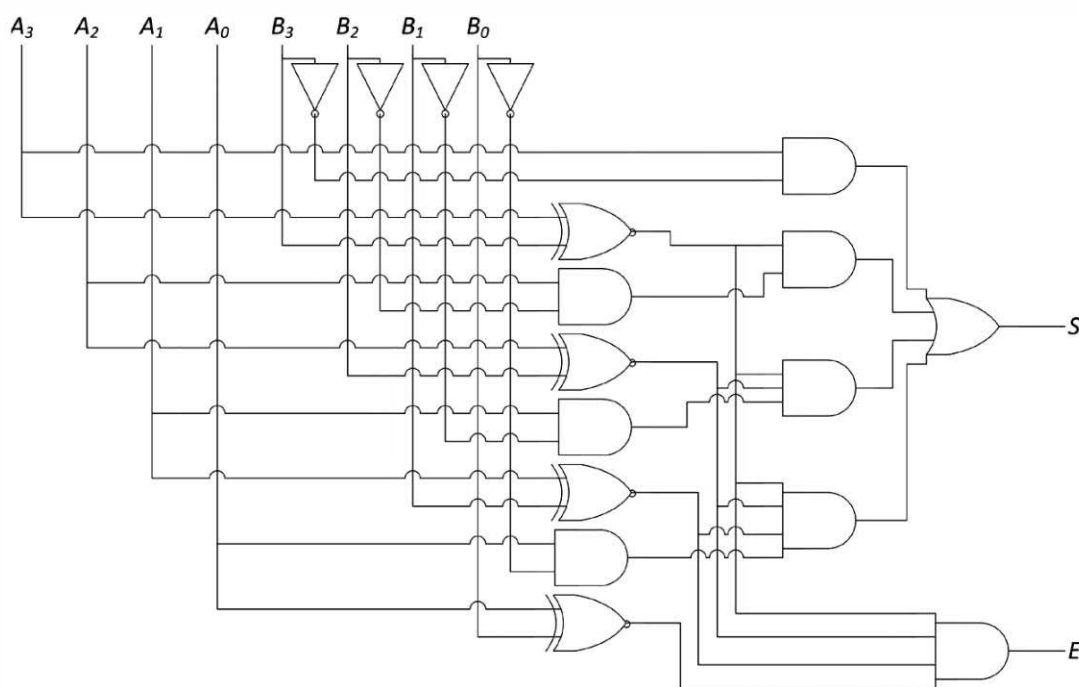


Figure 4.14

4-3 Circuit décaleur

1) Équations des sorties et schéma logique.

C_1	C_0	S_3	S_2	S_1	S_0
0	0	E_3	E_2	E_1	E_0
0	1	0	E_3	E_2	E_1
1	0	0	0	E_3	E_2
1	1	0	0	0	E_3

$$S_3 = \overline{C_1} \cdot \overline{C_0} \cdot E_3$$

$$S_2 = \overline{C_1} \cdot \overline{C_0} \cdot E_2 + \overline{C_1} \cdot C_0 \cdot E_3$$

$$S_1 = \overline{C_1} \cdot \overline{C_0} \cdot E_1 + \overline{C_1} \cdot C_0 \cdot E_2 + C_1 \cdot \overline{C_0} \cdot E_3$$

$$S_0 = \overline{C_1} \cdot \overline{C_0} \cdot E_0 + \overline{C_1} \cdot C_0 \cdot E_1 + C_1 \cdot \overline{C_0} \cdot E_2 + C_1 \cdot C_0 \cdot E_3$$

Le schéma logique du circuit est le suivant :

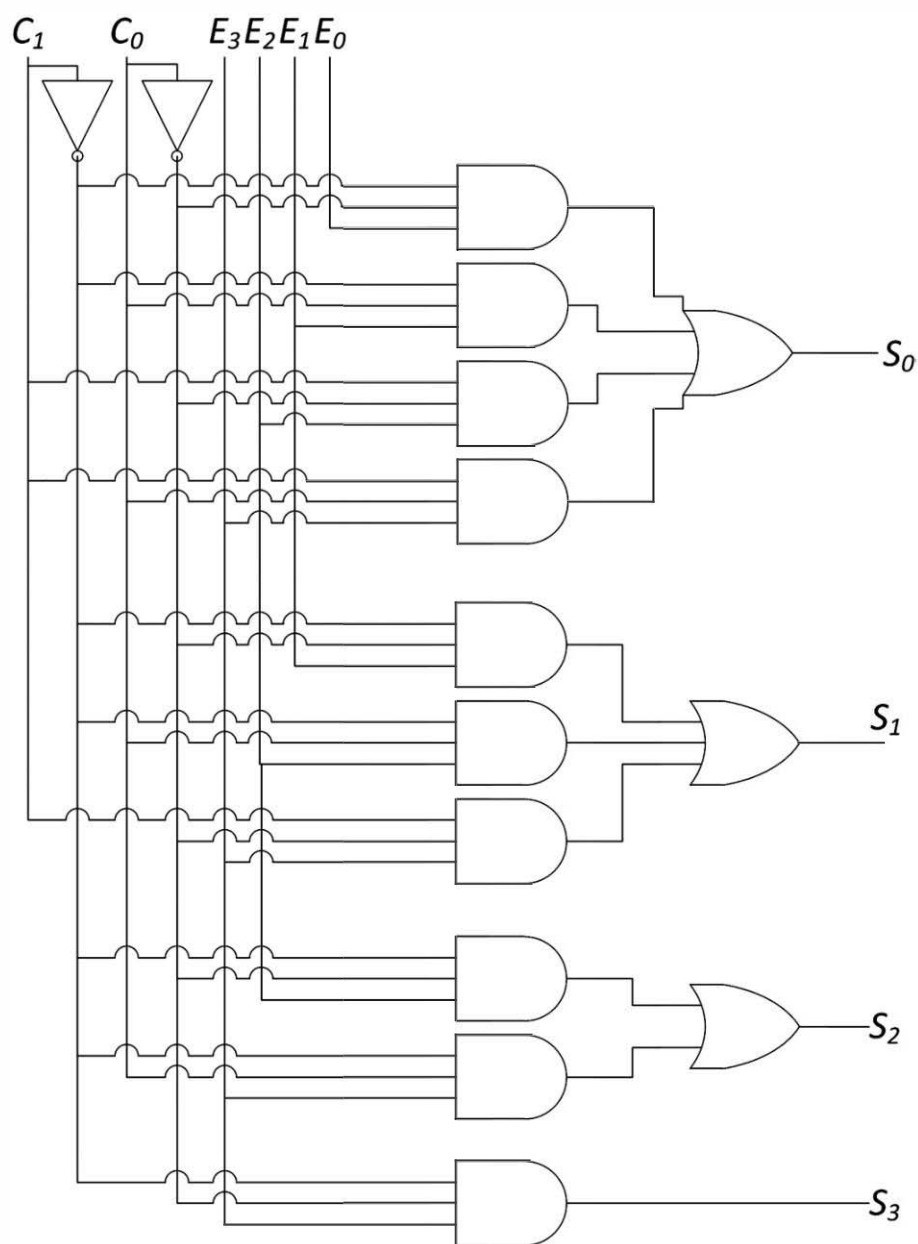


Figure 4.15

2) Modifications du circuit. Pour pouvoir associer plusieurs circuits, il est nécessaire d'ajouter des entrées au circuit. En effet, en utilisant deux circuits de la première question pour décaler un mot de 8 bits $E_7E_6E_5E_4E_3E_2E_1E_0$ de deux positions vers la droite, on obtient : $00E_7E_600E_3E_2$. Les quatre premiers bits $00E_7E_6$, fourni par le premier circuit, sont corrects, mais les quatre derniers bits $00E_3E_2$, fourni par le deuxième circuit, ne conviennent (il faudrait $E_5E_4E_3E_2$).

Par conséquent, en considérant l'hypothèse la pire (décalage de trois positions), il faut pouvoir considérer trois entrées supplémentaires. En nommant ces entrées X , Y et Z , on obtient la table de vérité partielle suivante :

C_1	C_0	S_3	S_2	S_1	S_0
0	0	E_3	E_2	E_1	E_0
0	1	Z	E_3	E_2	E_1
1	0	Y	Z	E_3	E_2
1	1	X	Y	Z	E_3

On en déduit les équations suivantes :

$$S_3 = \overline{C_1} \cdot \overline{C_0} \cdot E_3 + \overline{C_1} \cdot C_0 \cdot Z + C_1 \cdot \overline{C_0} \cdot Y + C_1 \cdot C_0 \cdot X$$

$$S_2 = \overline{C_1} \cdot \overline{C_0} \cdot E_2 + \overline{C_1} \cdot C_0 \cdot E_3 + C_1 \cdot \overline{C_0} \cdot Z + C_1 \cdot C_0 \cdot Y$$

$$S_1 = \overline{C_1} \cdot \overline{C_0} \cdot E_1 + \overline{C_1} \cdot C_0 \cdot E_2 + C_1 \cdot \overline{C_0} \cdot E_3 + C_1 \cdot C_0 \cdot Z$$

$$S_0 = \overline{C_1} \cdot \overline{C_0} \cdot E_0 + \overline{C_1} \cdot C_0 \cdot E_1 + C_1 \cdot \overline{C_0} \cdot E_2 + C_1 \cdot C_0 \cdot E_3$$

3) Schéma logique – décalage avec introduction de 0.

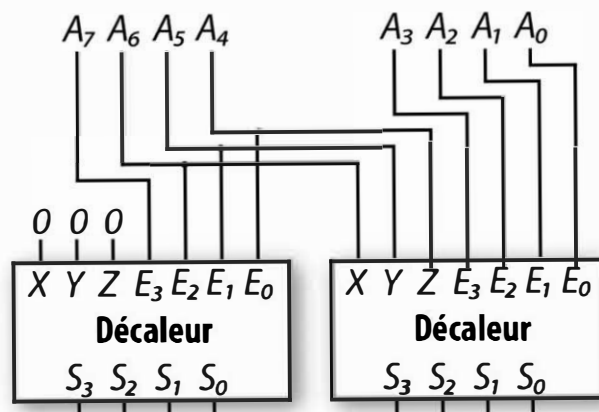


Figure 4.16

4) Schéma logique – décalage cyclique.

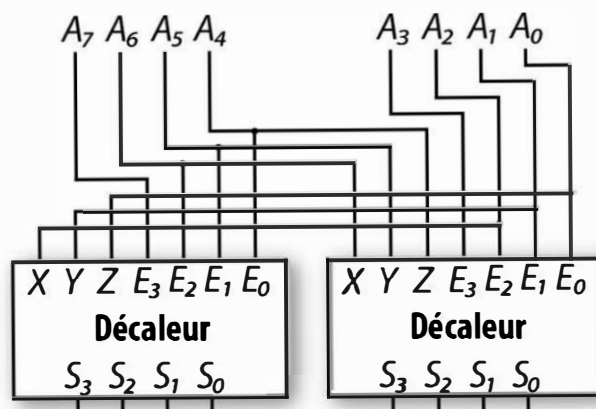


Figure 4.17



Logique séquentielle

PLAN

- 5.1 Introduction
- 5.2 Éléments de mémorisation élémentaire
- 5.3 Les verrous
- 5.4 Les bascules
- 5.5 Comportement des entrées/sorties des verrous et des bascules
- 5.6 Classification des verrous et des bascules sous la dénomination « BASCULE »
- 5.7 Association de bascules

OBJECTIFS

- Comprendre la logique séquentielle et ses opérateurs
- Examiner le fonctionnement et le comportement des verrous et des bascules
- Examiner des structures plus complexes s'appuyant sur ces briques de base

5.1 INTRODUCTION

Jusqu'ici, nous nous sommes concentrés sur les circuits logiques dont l'état des sorties est seulement une fonction de l'état de leurs entrées. Ces circuits sont des circuits logiques **combinatoires**. Nous allons étendre maintenant l'étude vers des circuits logiques dont l'état des sorties est une fonction aussi bien des séquences de l'état courant que de l'état précédent de leurs entrées. De tels circuits stockent l'information concernant l'historique de l'état précédent de leurs entrées : ces circuits sont appelés des *éléments de stockage ou de mémorisation*. Ils constituent le fondement des circuits logiques **séquentiels**. Nous allons nous

intéresser dans un premier temps à la structure et au comportement des éléments de base de ces circuits logiques. Ces circuits forment des briques élémentaires de construction de circuits séquentiels plus complexes que nous allons aussi introduire dans ce chapitre.

5.2 ÉLÉMENTS DE MÉMORISATION ÉLÉMENTAIRE

Un **élément de mémorisation élémentaire** peut être implémenté par deux inverseurs montés en cascade avec rebouclage ou rétroaction de la sortie sur l'entrée. C'est la structure de base utilisée dans la conception des mémoires statiques. La figure 5.1 montre cette structure de mémoire élémentaire unitaire (mémorisation d'un bit). Le rôle du multiplexeur de 2:1, muni de son signal de contrôle *Ctrl* est de permettre :

- Lorsque $Ctrl = 1$ de rompre le rebouclage et par conséquent de permettre le stockage de la valeur du bit présent sur l'entrée *E*.
- Lorsque $Ctrl = 0$ de rompre l'entrée *E* et de rétablir le rebouclage permettant ainsi le maintien de la valeur stockée.

Une fois que cette valeur du bit présent sur l'entrée *E* est insérée, elle peut être maintenue indéfiniment par le circuit. Cette valeur est disponible à la sortie *S* du circuit.

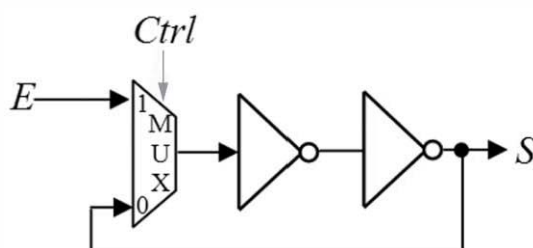


Figure 5.1 Éléments de mémorisation 1 bit avec 2 inverseurs en cascade.

Une alternative pour concevoir un élément de mémoire élémentaire unitaire est d'utiliser deux portes NON-OU ou NON-ET montées en couplage croisé, c'est-à-dire en connectant les sorties sur les entrées (**rétroaction** ou **rebouclage** des sorties sur des entrées). Le circuit ainsi constitué par ces portes forme aussi un bloc de base de mémorisation élémentaire qui est appelé, selon les cas, **verrou** ou **bascule**. Verrous et bascules forment les portes logiques de base de la logique séquentielle. Ils sont à la logique séquentielle ce que sont les portes ET, OU, etc. à la logique combinatoire. Nous allons donc commencer par examiner le fonctionnement et le comportement de ces structures élémentaires.

5.3 LES VERROUS

Le verrou R-S

Le schéma logique du circuit constitué à partir de deux portes NON-OU montées en couplage croisé donné sur la figure 5.2 est appelé **verrou**. Nous verrons plus loin que deux portes NON-ET peuvent être utilisées de la même façon.

Un verrou (*LATCH* en anglais) est un élément de mémorisation unitaire. Il permet de mémoriser l'état d'une variable booléenne pour un temps donné.

Une porte NON-OU avec une de ses entrées à 0, agit comme un inverseur vis-à-vis de son autre entrée. Par contre si une des entrées est à 1, le NON-OU fournira toujours 0 en sortie quel que soit l'état de son autre entrée. Donc en fonction de l'état des entrées R et S , un des NON-OU de la figure 5.2 agit comme inverseur alors que l'autre injecte un 0.

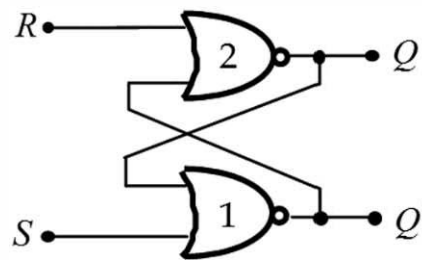


Figure 5-2 Schéma logique du verrou R-S.

Fonctionnement et comportement temporel du verrou

Si, au départ, $S = R = 0$ et qu'on considère que la sortie $Q = 0$, alors cette dernière reste dans cet état.

Étudions le comportement de ce verrou, lors de l'arrivée de signaux sur S et R selon le chronogramme figure 5.3.

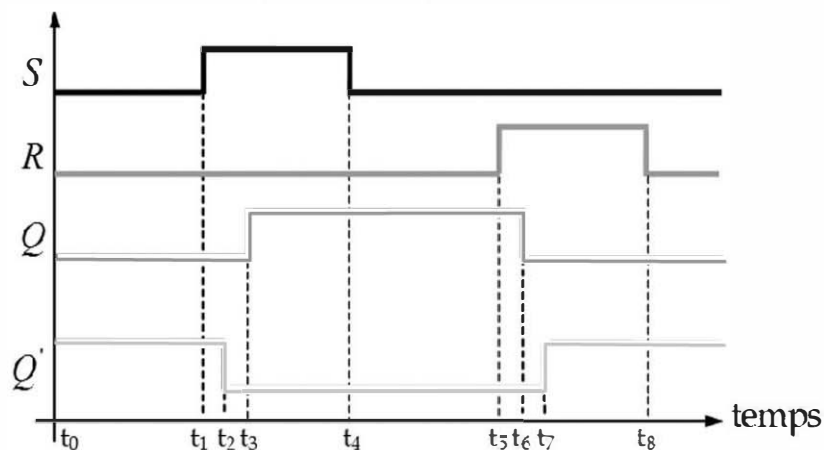


Figure 5-3 Chronogramme de comportement du verrou R-S.

➤ **État initial**

- En t_0 , on a $S = R = Q = 0$ et $Q' = 1$.

➤ **Action sur l'entrée S**

- En t_1 , on fait passer S de 0 à 1 ce qui a pour conséquence :
 - Sur le NON-OU 1 : le passage, en t_2 , de la sortie Q' de 1 à 0.
 - Sur le NON-OU 2 : le passage, en t_3 , de la sortie Q de 0 à 1.

Le maintien de S à 1 et R à 0 est sans effet : Q reste à 1 et Q' reste à 0.

- En t_4 , si on fait repasser S de 1 à 0, ceci n'a pas d'effet sur l'état des sorties $Q = 1$ et $Q' = 0$.

L'entrée S permet donc de mettre à 1 la sortie Q , et à 0 la sortie Q' . Le raisonnement est identique avec l'entrée R , qui permet la mise à 0 de la sortie Q et la mise à 1 de la sortie Q' .

Ce type de verrou est connu sous le nom de R-S (*Reset-Set*). Les sorties Q et Q' sont inverses l'une de l'autre et traditionnellement appelées Q et $\overline{Q}$.

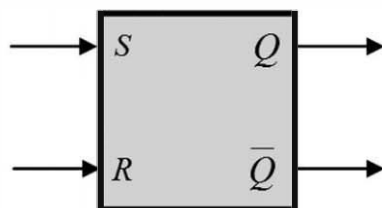
L'état $R = S = 1$ est interdit pour deux raisons :

- Si on fait passer R et S simultanément au niveau logique 1 les deux NON-OU auront leur sortie à 0, c'est-à-dire que Q et $\overline{Q}$ seront en même temps à 0, ce qui est contraire à l'hypothèse de complémentarité des deux sorties. Ce cas constitue la première raison.
- Si R et S repassent simultanément à 0, les sorties Q et $\overline{Q}$ vont rentrer dans un état oscillatoire qui peut durer aussi longtemps que R et S sont à 0. Cependant **les délais de propagation** dans les deux portes NON-OU ne sont jamais parfaitement identiques. Ceci provoque l'amortissement du phénomène d'oscillation et les sorties Q et $\overline{Q}$ finissent par se stabiliser. Malheureusement, l'état final est imprévisible (non déterministe) : Q peut valoir 1 ou 0 ! C'est la seconde raison d'interdiction.

Table de transitions et équation du verrou R-S

Toutes les combinaisons possibles des valeurs de R , S et l'état de la sortie Q peuvent se représenter sur une table dite **table de transitions**, analogue à une table de vérité quant à la présentation, mais différente quant au raisonnement nécessaire pour l'établir. Il faut en effet donner la valeur future de la sortie en fonction de sa valeur présente et de celle des variables R et S .

La représentation symbolique du verrou R-S et la table de transitions sont détaillées ci-après.

Représentation symbolique du verrou R-S (figure 5.4)**Figure 5.4** Représentation symbolique du verrou R-S conçu par des NON-OU.**Table de transitions (tableau 5.1)**

La notation X^n et X^{n+1} : signifie X à l'instant n (t_n) et X à l'instant $n + 1$ (t_{n+1}). L'instant n (t_n) représente un temps arbitraire jusqu'auquel les entrées et les sorties ont une valeur stable donnée. L'instant $n + 1$ (t_{n+1}) représente l'instant où la valeur d'une sortie est stable suite à une transition sur l'une des entrées à l'instant n (t_n).

Tableau 5.1 TABLE DE TRANSITIONS.

S^n	R^n	Q^n	Q^{n+1}	Remarques
0	0	0	0	Maintien de l'état précédent c'est l'opération (HOLD)
0	0	1	1	
0	1	0	0	Mise à 0 de la bascule c'est l'opération (RESET)
0	1	1	0	
1	0	0	1	Mise à 1 de la bascule c'est l'opération (SET)
1	0	1	1	
1	1	0	X	Combinaisons interdites
1	1	1	X	

Équation du verrou R-S

La table de transitions (tableau 5.1) décrit le comportement du verrou. On peut en tirer, comme en logique combinatoire, une forme algébrique de la fonction séquentielle représentée. L'équation du verrou R-S s'écrit :

$$Q^{n+1} = \overline{S}^n \cdot \overline{R}^n \cdot Q^n + S^n \cdot \overline{R}^n \cdot \overline{Q}^n + S^n \cdot \overline{R}^n \cdot Q^n$$

Si l'on tient compte des deux combinaisons interdites :

$S^n R^n$	00	01	11	10
Q^n				
0	0	0	Φ	1
1	1	0	Φ	1

On obtient :

$$Q^{n+1} = S^n + \overline{R^n} \cdot Q^n$$

On peut écrire la table de transitions sous une forme réduite (tableau 5.2).

Tableau 5.2 TABLE DE TRANSITIONS RÉDUITE DU VERROU R-S.

S^n	R^n	Q^{n+1}
0	0	Q^n
0	1	0
1	0	1
1	1	X

Conditions pour un fonctionnement correct du verrou R-S

La figure 5.5, montre les caractéristiques temporelles que doivent avoir les signaux actifs sur les entrées R et S pour un fonctionnement correct du verrou.

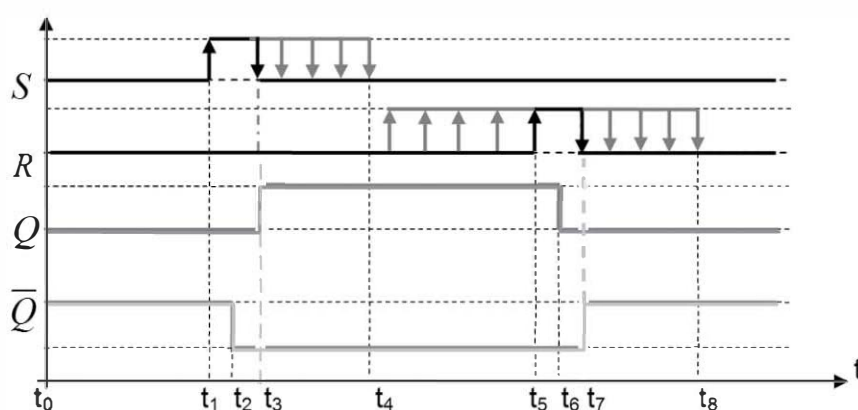


Figure 5.5 Caractéristiques des signaux sur les entrées R et S .

Transition, impulsion, niveau

En logique combinatoire on ne considère que les états stables, les signaux étant représentés par des niveaux. Les états **transitoires** asso-

ciés aux changements de niveaux sont sans intérêt. En logique séquentielle le problème n'est pas le même.

En effet, comme le montre la figure 5.6, à l'instant t_1 on fait passer S de 0 à 1 ($0 \rightarrow 1$) et R reste à 0 jusqu'à un instant arbitraire (le raisonnement est identique avec R). Si on était en logique combinatoire, S peut repasser à 0 n'importe quand. Ici S doit rester stable « un certain temps » afin que le changement d'état du verrou s'effectue correctement. Si on fait repasser S de 1 à 0 avant l'instant t_3 , Q n'est pas encore affirmé à 1 et on repasse par l'état $S = Q = 0$ qui a tendance à faire repasser $\overline{Q}$ à 0. Donc, il est raisonnable que S reste à 1 pour une « courte » période de temps, c'est-à-dire un laps de temps $\Delta t \geq t_3 - t_1$ correspondant au délai de propagation des signaux dans les couches logiques du verrou. Ensuite S peut repasser S de 1 à 0 n'importe quand, puisque cette transition ($1 \rightarrow 0$) est sans effet. Mais il faut que ceci soit réalisé avant la prochaine transition sur R pour éviter l'état interdit. Or nous avons vu précédemment que la transition (passage de S de 0 à 1) est une condition nécessaire pour un changement d'état du verrou, et pour que celui-ci se fasse correctement il suffit que l'entrée S reste au niveau logique 1 pendant au moins un laps de temps correspondant au délai de propagation dans le verrou. Mais ce délai peut être beaucoup plus long avant que S ne repasse à 0 et peut être également considéré comme une impulsion. N'est-ce pas alors un niveau ? La réponse est donnée par la façon dont les circuits répondent.

Impulsion

Une **impulsion**, quelle que soit sa durée, est constituée de deux transitions ; mais elle représente un seul **évènement** c'est-à-dire ne conduisant qu'à une seule réponse. La réponse peut se produire soit sur le **front montant** soit sur le **front descendant** de l'impulsion. Une impulsion peut être positive ou négative périodique ou apériodique.

Notion d'horloge, temps de pré-positionnement et de maintien

La sortie d'un circuit séquentiel est une fonction de l'état de ses entrées et de tous les signaux qui sont rebouclés sur les entrées. Nous appelons ces signaux de rebouclages ou rétroactions **l'état courant** du circuit. Un évènement externe périodique, appelé une **horloge**, détermine quand le circuit remplacera l'état courant par un nouvel état. Quand cet évènement de l'horloge arrive, le circuit **échantillonne** l'état de ses entrées et son état courant et calcule son nouvel état.

L'évènement d'horloge peut être considéré soit comme le front montant (transition $0 \rightarrow 1$) soit comme le front descendant (transition $1 \rightarrow 0$) de

l'horloge. Il est important que les signaux sur les entrées d'un circuit séquentiel, lorsque celui-ci est en cours de calcul de son nouvel état, restent stables autour de l'évènement d'horloge. On définit une fenêtre de temps par deux contraintes de temps : le **temps de pré-positionnement** avant l'évènement d'horloge, et le **temps de maintien** après l'évènement d'horloge. Si les signaux sur les entrées ne changent pas à l'intérieur de cette fenêtre de temps, l'état peut être mis à jour de manière correcte et non ambiguë. Si au contraire ils changent à l'intérieur de cette fenêtre le nouvel état résultant est indéfini.

Définition : le **temps de pré-positionnement** est l'intervalle de temps minimum précédant l'évènement d'horloge durant lequel les signaux sur les entrées doivent être stables pour être reconnus comme valides. Nous appellerons t_s ce temps de pré-positionnement.

Définition : le **temps de maintien** est l'intervalle de temps minimum après le front de l'évènement d'horloge durant lequel les signaux sur les entrées doivent être encore stables pour être reconnus comme valides. Nous appellerons t_h ce temps de maintien.

La figure 5.6 montre ces concepts pour un front montant comme évènement d'horloge. Le signal « Entrée » doit être stable dans la **fenêtre de temps** $t_s + t_h$ c'est-à-dire qu'« Entrée » doit être stable avant le front montant de l'horloge pendant un temps au moins égal à t_s et maintenu stable après le front d'horloge pendant un temps au moins égal à t_h . Si ces conditions sont respectées alors le signal « Entrée » est reconnu comme un niveau logique haut.

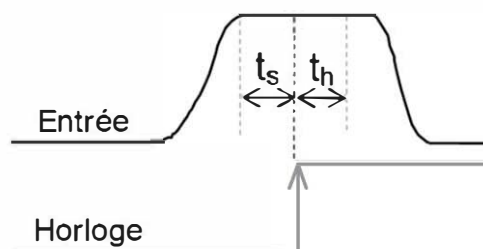


Figure 5.6 Concepts de temps de pré-positionnement et de maintien et fenêtre de temps $t_s + t_h$ autour de l'évènement d'horloge.

Si on revient au concept d'évènement d'échantillonnage, on peut imaginer cet évènement comme un signal de validation qui ordonne à l'élément de mémorisation d'échantillonner son entrée pour changer son état.

On peut utiliser une seule horloge pour **synchroniser** par exemple la mise à jour de plusieurs éléments mémoire.

Quand les sorties des éléments de mémorisation changent immédiatement en réponse aux changements sur les entrées, ces sorties sont appelées sorties **transparentes**. Le verrou R-S a des sorties transparentes.

Si un verrou n'a que les entrées de données, comme R et S , c'est-à-dire sans entrée d'horloge, il est dit **verrou asynchrone**. Le verrou R-S de la figure 5.2 est donc asynchrone.

Les verrous **synchrones** ont une entrée supplémentaire de validation, souvent appelée **horloge**. Ils sont sensibles au niveau de l'horloge et non aux transitions de cette dernière.

Le verrou R-S synchrone

La figure 5.7 (a) montre un verrou R-S dont les entrées R et S sont conditionnées par le signal de validation ou d'horloge C . Ce verrou synchrone échantillonne de manière continue ses entrées tant que C est au niveau logique haut. Tout changement sur les entrées est propagé sur les sorties. L'état de maintien arrive quand $R = S = 0$.

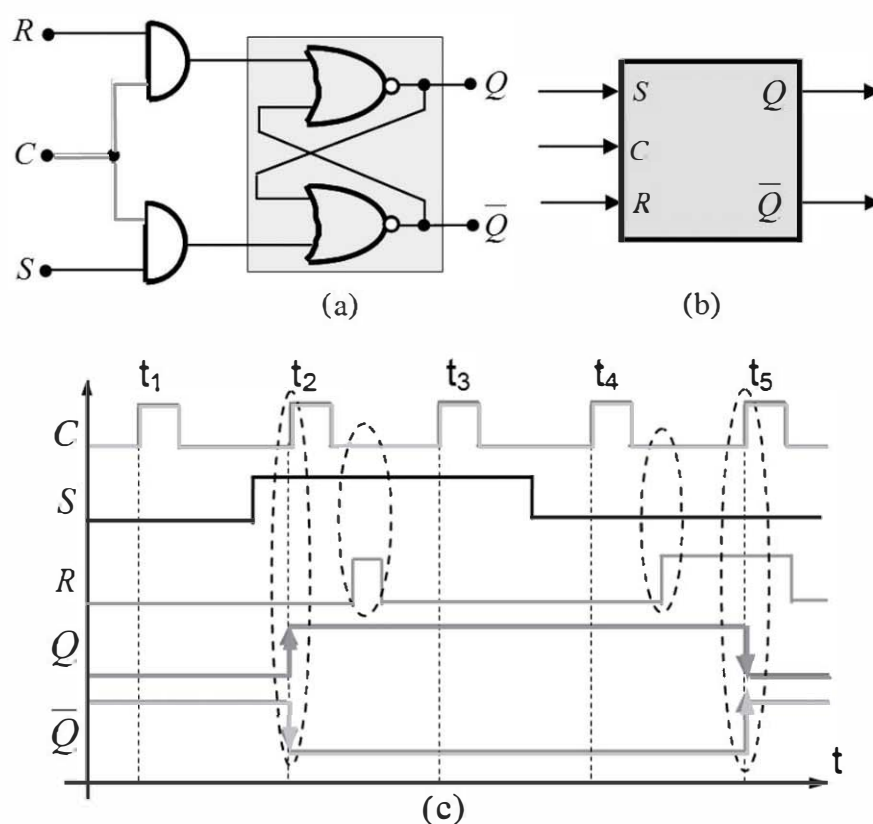


Figure 5.7 (a) Verrou R-S synchrone, (b) Sa représentation symbolique, (c) Son chronogramme de comportement temporel.

Quand C est à l'état bas, chacune des deux portes ET a sa sortie forcée à 0, plaçant ainsi les deux portes NON-OU dans l'état de maintien. Quand le signal de validation C est désactivé (passage niveau haut $\rightarrow$ niveau bas), la dernière valeur sur les entrées détermine l'état mémorisé (maintenu) par le verrou. La valeur « **verrouillée** » est déterminée par la fenêtre de temps formée par les temps de pré-positionnement et de maintien autour du front descendant (niveau haut $\rightarrow$ niveau bas) du signal C .

Les figures 5.7 (b) et (c) montrent respectivement la représentation symbolique du verrou R-S synchrone et le comportement temporel de ce dernier.

Le verrou D

Le **verrou D** (*Delay*) est obtenu à partir du verrou R-S synchrone dans lequel il n'y a plus qu'une seule entrée D comme le montre la figure 5.8 (a). L'équation du verrou D peut être facilement obtenue en remplaçant R et S respectivement par $\overline{D}$ et D dans l'équation du verrou R-S.

On obtient donc :

$$Q^{n+1} = D^n + \overline{\overline{D}^n} \cdot Q^n = D^n(1 + Q^n) = D^n$$

Tant que l'entrée de validation C est au niveau logique 1, le verrou D est « transparent » : la sortie Q recopie continuellement l'état de l'entrée D . Quand C est au niveau logique 0, chacune des deux portes ET a sa sortie forcée à 0, plaçant le verrou R-S dans l'état de maintien. La valeur maintenue est celle présente sur l'entrée lors de la transition de l'état haut à l'état bas de C . Il faut tenir compte des contraintes de temps (pré-positionnement et maintien).

Le verrou D est toujours synchrone.

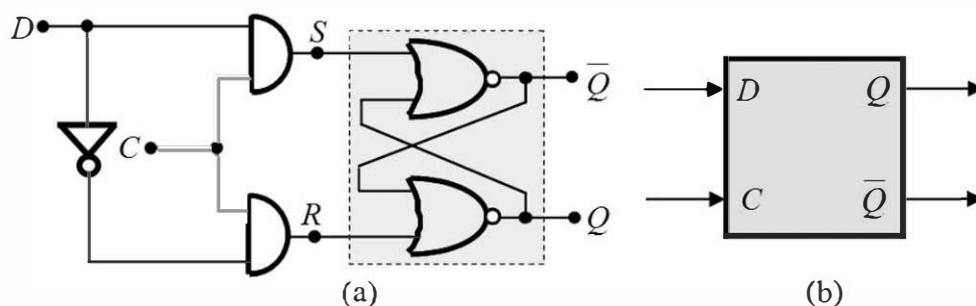


Figure 5.8 (a) Schéma logique du verrou D,
(b) sa représentation symbolique

Le verrou J-K

Un **verrou J-K** est un verrou R-S dans lequel la combinaison interdite a été éliminée au profit du complément de l'état courant. Son schéma logique est donnée en figure 5.9.

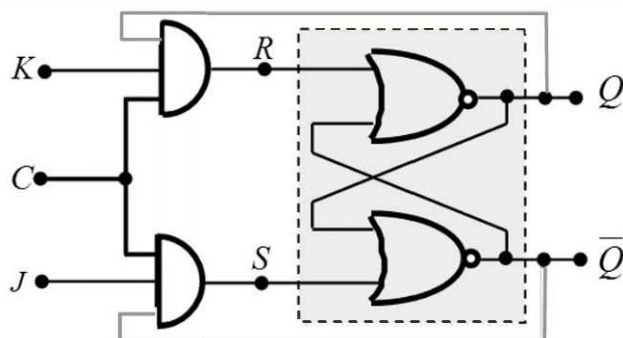


Figure 5.9 Schéma logique du verrou J-K.

En rebouclant les sorties Q et $\bar{Q}$ sur les entrées et en les conditionnant avec des entrées de contrôle *Set* et *Reset* externes (que nous appellerons maintenant respectivement J et K), nous garantissons que les entrées R et S internes ne seront jamais simultanément à 1 (ceci suppose que Q et $\bar{Q}$ ne sont jamais à 1 en même temps).

Le verrou R-S est peu utilisé, supplanté par le verrou J-K qui est toujours synchrone et dont la représentation symbolique et la table de transitions sont données ci-après.

Tableau 5.3 TABLE DE TRANSITIONS DU VERROU J-K.

J^n	K^n	Q^n	Q^{n+1}	Remarques
0	0	0	0	Maintien de l'état précédent c'est l'opération (HOLD)
0	0	1	1	
0	1	0	0	Mise à 0 de la bascule c'est l'opération (RESET)
0	1	1	0	
1	0	0	1	Mise à 1 de la bascule c'est l'opération (SET)
1	0	1	1	
1	1	0	1	Complémentation c'est l'opération (TOGGLE)
1	1	1	0	

La table réduite est donnée dans le tableau 5.4.

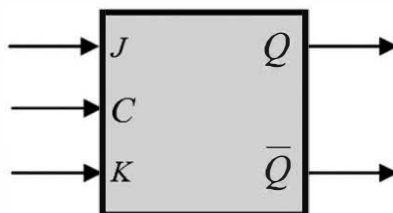


Figure 5.10 Représentation symbolique du verrou J-K.

Tableau 5.4 TABLE DE TRANSITIONS RÉDUITE DU VERROU J-K.

J^n	K^n	Q^{n+1}
0	0	Q^n
0	1	0
1	0	1
1	1	Q^n



J est analogue à S et K est analogue à R.

Équation du verrou J-K

$J^n K^n$	00	01	11	10
Q^n				
0	0	0	1	1
1	1	0	0	1

$$Q^{n+1} = \overline{K}^n \cdot Q^n + J^n \cdot \overline{Q}^n$$

Le verrou T

Le **verrou T** (*Trigger* ou *Toggle*) est peu utilisé ; on le rencontre surtout dans les compteurs. Habituellement, ce verrou n'existe pas dans les catalogues en tant que circuit intégré standard car il est facile à construire à partir du verrou J-K : les entrées J et K sont reliées ensemble et forment une entrée T . À chaque fois que $T = 1$, l'état de la sortie Q du verrou est complémenté.

La figure 5.11 donne sa représentation symbolique.

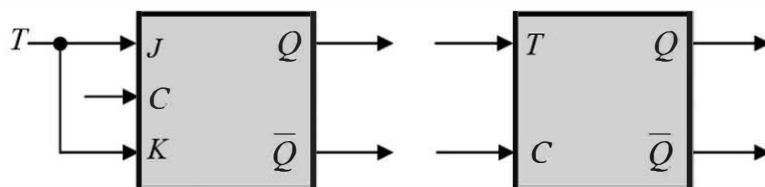


Figure 5.11 Représentation symbolique du verrou T.

On a $J = K = T$. En remplaçant J et K par T dans l'équation du verrou J-K, on obtient l'équation du verrou T :

$$\begin{aligned} Q^{n+1} &= \overline{T}^n \cdot Q^n + T^n \cdot \overline{Q}^n \\ &= T^n \oplus Q^n \end{aligned}$$

- Si $T^n = 0$ alors $Q^{n+1} = Q^n$: pas de changement d'état, donc maintien de l'état précédent.
- Si $T^n = 1$ alors $Q^{n+1} = \overline{Q}^n$: l'état courant ou présent est le complément de l'état précédent.



Le verrou D que nous avons déjà vu obtenu à partir du verrou R-S synchrone peut être obtenu aussi à partir du verrou J-K. Il suffit pour cela de relier D à J et $\overline{D}$ à K .

Problème du verrou J-K

Le mode de mise en complément ou basculement ($J = K = 1$) de l'état de la sortie Q provoque un phénomène d'oscillations. En effet, dans ce mode, durant l'état haut du signal d'horloge C , la sortie Q va basculer entre 0 et 1 jusqu'au passage à l'état bas de C – à moins qu'une des entrées J ou K repasse à 0 (figure 5.12). Le problème est que durant l'état haut de l'horloge C , les signaux J et K sont maintenus à 1 pendant le changement d'état des sorties. Ceci fait permuter les valeurs de R et S internes du verrou R-S causant le basculement de la sortie Q de manière continue entre 0 et 1 jusqu'à ce que la condition de basculement ou de mise en complément soit levée.

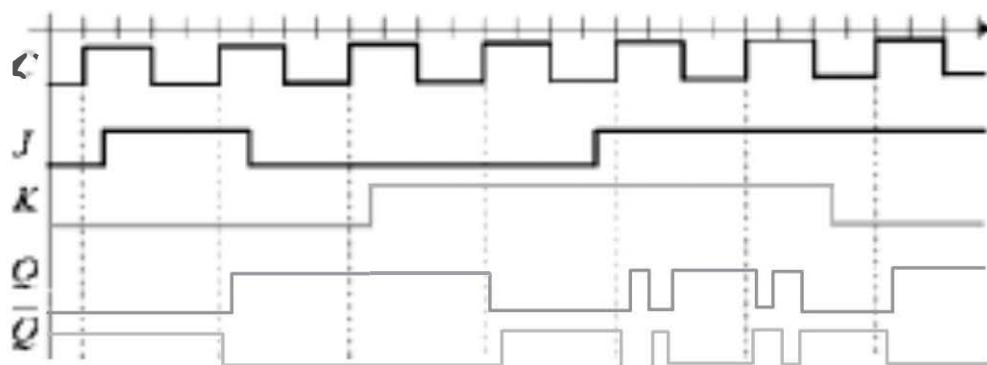


Figure 5.12 Chronogramme de fonctionnement du verrou J-K.

5.4 LES BASCULES

Les éléments de mémorisation élémentaires tombent dans deux grandes classes de circuits séquentiels élémentaires : les verrous et les bascules.

Les bascules diffèrent des verrous dans le fait que leurs sorties changent d'état seulement en fonction de l'horloge. Elles sont donc toujours synchrones. On peut caractériser le changement d'état des sorties des bascules sur la base des transitions de l'horloge : ces bascules sont à **déclenchement sur front montant, à déclenchement sur front descendant et maître/esclave**.

Une bascule à déclenchement sur front montant échantillonne ses entrées sur la transition positive ($0 \rightarrow 1$) de l'horloge. Pour être correctement reconnue, l'entrée doit être stable à l'intérieur de la fenêtre des temps de pré-positionnement et de maintien autour du front d'horloge. Le changement des sorties a lieu avec un délai de propagation après le front montant de l'horloge.

Une bascule à déclenchement sur front descendant fonctionne de manière similaire. Elle échantillonne ses entrées sur la transition négative ($1 \rightarrow 0$) de l'horloge. Le changement des sorties a lieu avec un délai de propagation après le front descendant de l'horloge. Avec cette classification une bascule maître/esclave est indistinguable de celle à déclenchement sur front descendant, sauf qu'elle exhibe un comportement étrange appelé « piège des 1 ». À présent nous allons examiner les différents types de bascules.

La bascule J-K maître/esclave

La solution pour résoudre le problème lié à l'oscillation de la sortie lorsque le verrou J-K se retrouve dans l'état de mise en complément ($J = K = 1$), nous amène vers une voie alternative pour concevoir un nouvel élément de mémorisation élémentaire appelé bascule J-K maître/esclave. L'idée de base est de construire cet élément de mémorisation en deux étages (figure 5.13). Le premier étage, le maître, accepte de nouvelles entrées R et S et génère en sorties Z et $\bar{Z}$ sur le front montant du signal d'horloge C . Le second étage, l'esclave, échantillonne en entrées Z et $\bar{Z}$ sur le front descendant du signal d'horloge C puis met à jour ses sorties Q et $\bar{Q}$ après ce front descendant de l'horloge. Le temps que les signaux Q et $\bar{Q}$ se propagent (délai de propagation) par le rebouclage sur le premier l'étage (le maître), le signal d'horloge a déjà disparu, garantissant que la sortie ne peut pas commuter de nouveau en complémentant sa valeur précédente (du moins pas avant le prochain front montant de l'horloge).

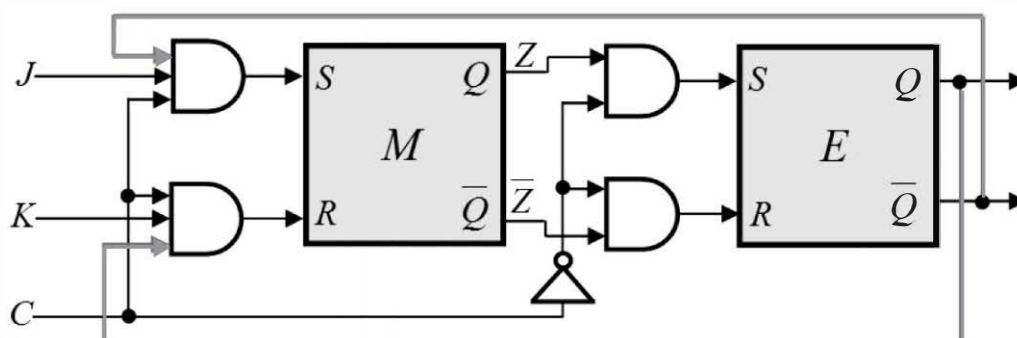


Figure 5.13 Schéma logique de la bascule J-K maître/esclave.

Fonctionnement et problème de la bascule J-K

Le chronogramme de fonctionnement de la bascule J-K maître/esclave (figure 5.14) illustre quelques caractéristiques importantes :

- Les sorties Z et $\bar{Z}$ du premier étage **suivent** les sorties Q et $\bar{Q}$ du second étage. Le changement d'état de ces dernières s'effectue avec un retard (correspondant au délai de propagation) **après** le front descendant de l'horloge.
- Une propriété générale des bascules maître/esclave appelée « piège des uns » arrive à chaque fois qu'une entrée J ou K est à l'état haut n'importe quand durant la période d'horloge, provoquant systématiquement une mise à 1 ou une mise à 0 de la bascule suivant que l'état haut a été piégé sur J ou sur K .

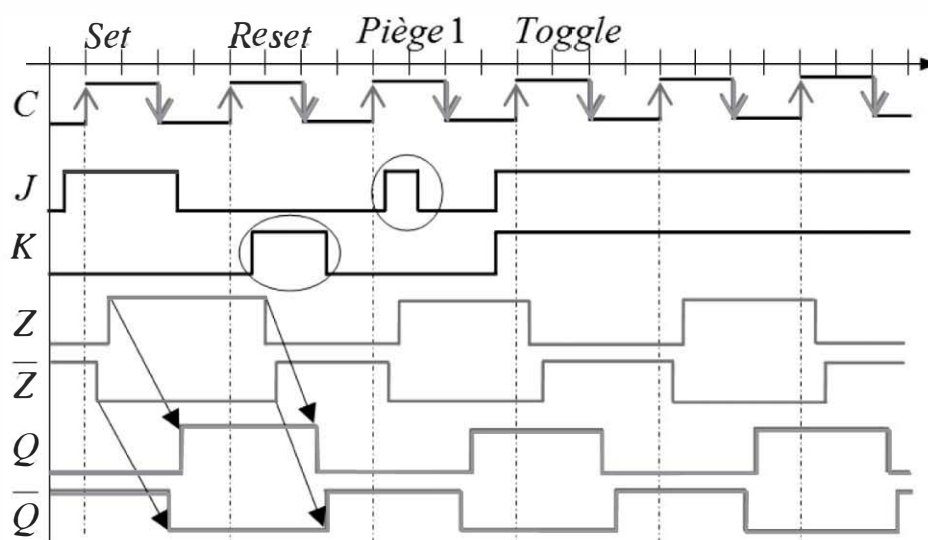


Figure 5.14 Chronogramme de fonctionnement de la bascule J-K maître/esclave.

Pour observer ce comportement nous allons suivre sur le chronogramme de la figure 5.14, l'évolution des entrées et l'effet sur les sorties :

- Initialement la bascule a $Q = 0$.
- On fait $J = 1$. Ensuite l'horloge passe de $(0 \rightarrow 1)$ et un délai après le front d'horloge $P = 1$. Quand C passe de $(1 \rightarrow 0)$, Z et $\bar{Z}$ sont échantillonnées par le second étage entraînant $Q = 1$. C'est l'opération mise à 1 (Set).
- Sur le front montant de d'horloge suivant, on a $J = K = 0$ le premier étage maintient son état précédent. C'est l'opération de maintien (*Hold*)
- Cependant au milieu de la période d'horloge, K passe à 1 provoquant la mise à 0 du premier l'étage. Cette valeur 0 est passée ensuite au second étage causant sa mise à 0 après le font descendant de l'horloge. Ceci est le phénomène du piège des uns. Ce phénomène est encore bien clairement identifié sur la période d'horloge suivante lorsque sur $J(0 \rightarrow 1 \rightarrow 0)$ une impulsion apparaît durant l'état haut de l'horloge C provoquant une mise à 1 de la bascule même si l'entrée J est revenue à 0 au moment où l'horloge passe à l'état bas.
- Le reste du chronogramme montre les configurations de l'opération de mise en complément (*Toggle*) où la sortie Q passe de 1 à 0 et de 0 à 1 avec un seul changement par période d'horloge.



Si les verrous sont faits à partir des portes logiques NON-ET, il y a un problème analogue mais avec le piège des zéros, du fait que les entrées R et S fonctionnent en logique négative.

La bascule J-K maître/esclave doit être utilisée avec précaution. N'importe quelle impulsion parasite sur les entrées peut provoquer un changement d'état non désiré.

On peut aussi à partir de la bascule J-K maître/esclave obtenir des bascules T.

La bascule D

La **bascule D** (à ne pas confondre avec le verrou D) échantillonne son entrée sur un front d'horloge uniquement (*Edge Triggered*). C'est la plus utilisée dans les calculateurs. Son fonctionnement interne est complexe, mais son utilisation est celle qui présente les caractéristiques les plus simples, car l'entrée de donnée est mémorisée par l'un des deux fronts de l'horloge.

Les figures 5.15 (a) et (b) montrent respectivement la représentation symbolique et le chronogramme de temporel de la bascule D à déclenchement sur un front montant.

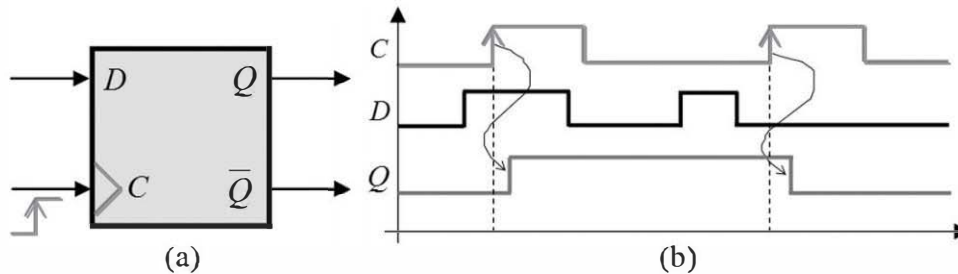


Figure 5.15 (a) Représentation symbolique de la bascule D à déclenchement sur front.
(b) Chronogramme de comportement temporel.

Pour que le fonctionnement soit correct, il faut que l'entrée de donnée soit stable un certain temps avant et après le front actif de l'horloge (figure 5.16).

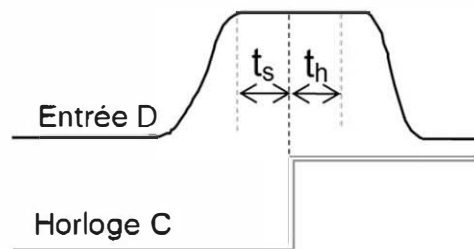


Figure 5.16 t_s : temps de pré-positionnement (*setup Time*).
 t_h : temps de maintien (*Hold Time*).

Bascule D augmentée d'entrées asynchrones

Certaines bascules D à déclenchement sur front sont disponibles avec des entrées de mise à 1 (SET, parfois appelée PRESET) et de mise à 0 (RESET) utilisées pour définir l'état initial si la bascule est utilisée en fonctionnement normal synchrone. Le schéma logique d'une telle bascule est donné sur la figure 5.17 (a) ; la figure 5.17 (b) montre sa représentation symbolique.

Le tableau 5.5, montre la table de transitions de cette bascule. On remarque bien que le mode asynchrone prime sur le mode synchrone.

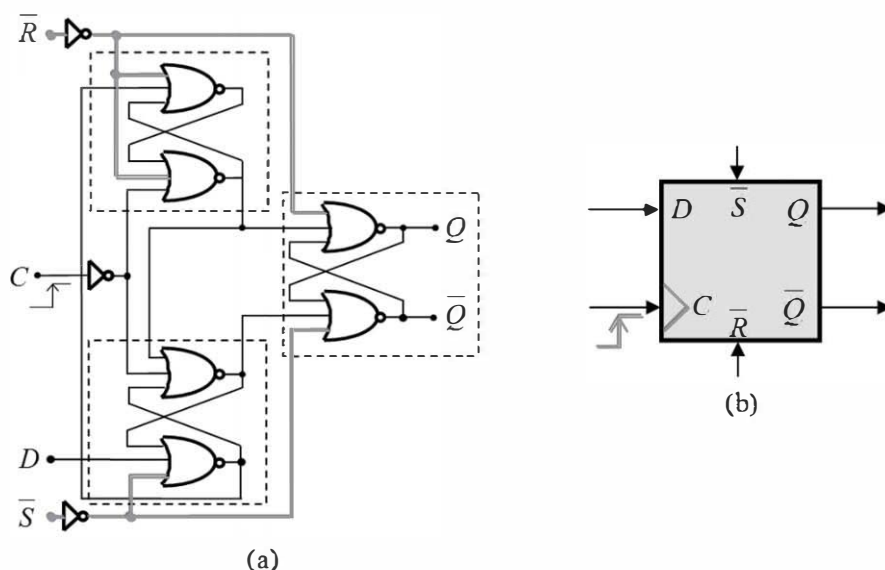


Figure 5.17 (a) Schéma logique d'une bascule D à déclenchement sur front montant munie d'entrées asynchrones. (b) Sa représentation symbolique.

Tableau 5.5 TABLE DE TRANSITIONS DE LA BASCULE D À FONCTIONNEMENT SYNCHRONES ET ASYNCHRONES.

R^n	S^n	D^n	C^n	Q^{n+1}	Remarques
1	0	X	X	1	Mise à 1 (PRESET)
0	1	X	X	0	Mise à 0 (RESET)
1	1	X	0	Q^n	Effet de mémorisation
1	1	0	↑	0	Écriture de 0
1	1	1	↑	1	Écriture de 1
0	0	X	X	X	Combinaisons interdites

5.5 COMPORTEMENT DES E/S DES VERROUS ET DES BASCULES

Le tableau 5.6 récapitule le comportement des entrées/sorties (E/S) des verrous et des bascules. Il s'agit de donner, pour chaque type verrou ou bascule, le moment où l'état des entrées peut être échantillonné et reconnu correctement, ainsi que le moment à partir duquel on peut considérer que l'état des sorties est valide.



Il très est fréquent de trouver dans la littérature les verrous et les bascules classés sous le terme générique « Bascules ». Dans ce cas il faut distinguer deux sous-classes : Bascules synchrones et asynchrones. Dans la sous-classe Bascules synchrones il faut encore distinguer deux groupes : les bascules à fonctionnement sensible au niveau et celles à fonctionnement sensible au front de l'horloge.

Tableau 5.6 COMPORTEMENT DES E/S DES VERROUS ET DES BASCULES.

Type	Échantillonnage des entrées	Sorties valides
Verrou asynchrone	Continu	Délai de propagation à partir du changement des entrées
Verrou sensible au niveau	État haut de l'horloge (t_s, t_h autour front descendant)	Délai de propagation à partir du changement des entrées
Bascule à front montant	Transition ($0 \rightarrow 1$) de l'horloge (t_s, t_h autour front montant)	Délai de propagation à partir du front montant de l'horloge
Bascule à front descendant	Transition ($1 \rightarrow 0$) de l'horloge (t_s, t_h autour front descendant)	Délai de propagation à partir du front descendant
Bascule maître/esclave	Transition ($1 \rightarrow 0$) de l'horloge (t_s, t_h autour front descendant)	Délai de propagation à partir du front descendant

5.6 CLASSIFICATION DES VERROUS ET BASCULES SOUS LA DÉNOMINATION « BASCULE »

Bascules		
Asynchrone	Synchrones	
	Sensibles à un niveau d'horloge	Sensibles à un front d'horloge
Bascule R-S	R-S, D, J-K, T	J-K maître/esclave, D
Verrous		Bascules

5.7 ASSOCIATION DE BASCULES

Dans le reste du document on considère que les bascules sont des bascules sensibles au front d'horloge.

Les compteurs

Fonction des compteurs

Le terme général de **compteur** englobe tous les montages constitués par association de bascules synchrones et qui permettent de numérotter les

impulsions qui leur sont appliquées, selon un code binaire lu sur les sorties Q des bascules.

On pourra ainsi repérer, par exemple, la 3^e et la 7^e impulsion d'un train de 10 impulsions successives ; ou encore identifier et extraire une impulsion toutes les N impulsions. Dans ce dernier cas on lui donne plutôt le nom de « diviseur par N ».

Code de comptage

Après chaque impulsion, les valeurs logiques des sorties Q des bascules constituent une des combinaisons ou un des ETATS du compteur. La suite de ces états, tous différents, constitue le **code de comptage** qui permet la numérotation des impulsions.

En général, il se répète cycliquement toutes les N impulsions et comprend alors N états différents, apparaissant toujours dans le même ordre. On dit qu'il s'agit d'un compteur à N états (dans le cas particulier d'un compteur à 10 états, on lui donne souvent le nom de « **décade** »). Le code de comptage peut être absolument quelconque.

Compteurs synchrones ou asynchrones

Un compteur est dit **synchrone** si toutes ses bascules sont commandées simultanément par les impulsions issues d'une source d'horloge unique. La figure 5.16 montre le principe d'un tel compteur.

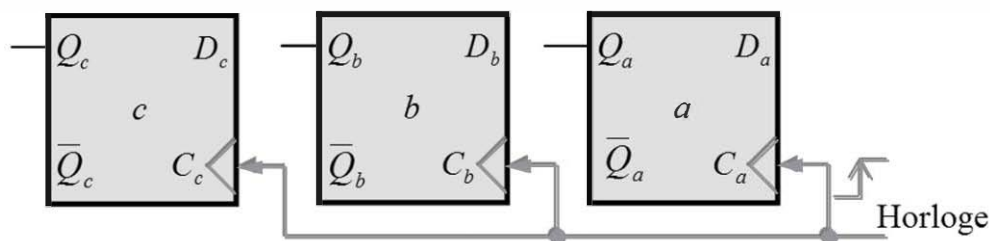


Figure 5.18 Principe d'un compteur synchrone.

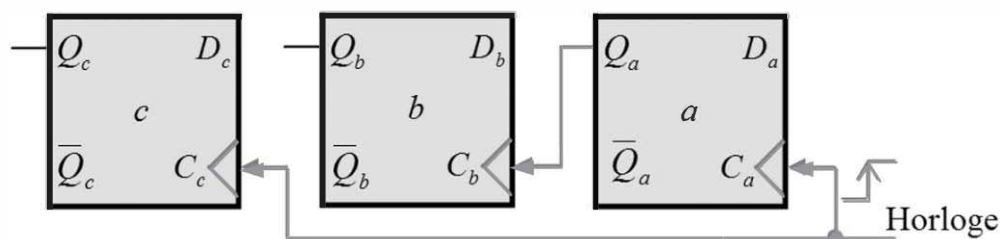


Figure 5.19 Principe d'un compteur asynchrone.

Un compteur est dit asynchrone si une ou plusieurs de ses bascules sont commandées par des transitions apparaissant sur les sorties d'autres bascules. En d'autres termes toutes les bascules ne sont pas synchrones

d'une même horloge. La figure 5.19 montre le principe d'un tel compteur. Dans la suite nous ne considérons que des compteurs synchrones.

Méthode d'analyse d'un circuit

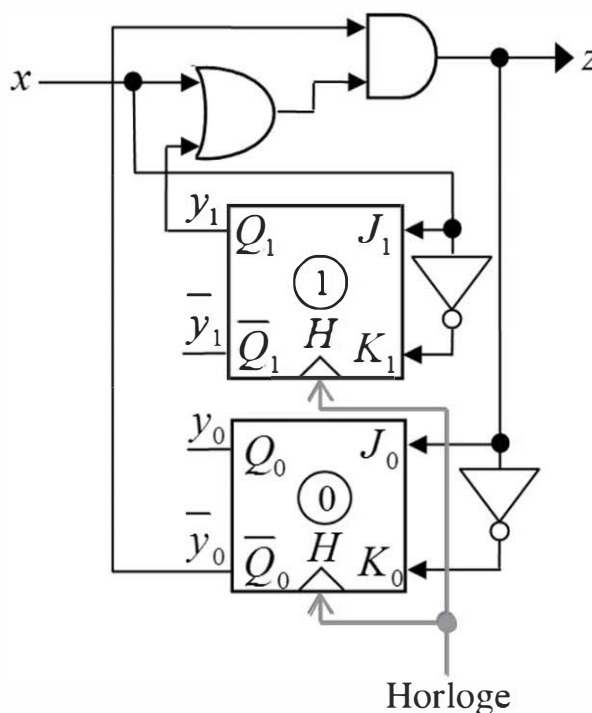
L'analyse d'un circuit consiste à déterminer la fonction réalisée par un circuit à partir de son schéma. Différentes étapes sont nécessaires pour analyser un circuit :

- **Détermination des équations d'entrées des bascules et des équations de sortie.** Ces équations peuvent facilement être déterminées à partir du schéma du circuit.
- **Calcul des équations des états suivants.** Ces équations sont obtenues à partir des équations d'entrées des bascules et des équations d'états suivants des bascules utilisées.
- **Constitution de la table de transition.** On la remplit grâce aux équations d'états suivants.
- **Table d'état.** La table de transition peut être remplacée par une table dans laquelle chaque état est remplacé par un symbole. On a donc, pour chaque état et selon les valeurs des entrées, l'état suivant et la valeur des sorties.
- Enfin, réalisation du **diagramme d'état**. Ce diagramme représente de manière graphique les informations contenues dans la table d'état.



EXEMPLE 1 : ANALYSE D'UN CIRCUIT

Considérons le circuit suivant :



Équations d'entrée des bascules

$$\begin{aligned} J_1 &= x & K_1 &= \bar{x} \\ J_0 &= z = (x + y_1) \cdot \bar{y}_0 & K_0 &= \overline{(x + y_1) \cdot \bar{y}_0} \end{aligned}$$

Équation de sortie

$$z = (x + y_1) \cdot \bar{y}_0$$

Équations des états suivants

Pour connaître l'évolution du système on va se servir de l'équation d'états suivants de la bascule J-K :

$$Q^{n+1} = J^n \cdot \bar{Q}^n + \bar{K}^n \cdot Q^n$$

D'où :

$$\begin{aligned} y_1^{n+1} &= J_1^n \cdot \bar{Q}_1^n + \bar{K}_1^n \cdot Q_1^n = x^n \cdot \bar{y}_1^n + x^n \cdot y_1^n \\ &= x^n (y_1^n + \bar{y}_1^n) = x^n \end{aligned}$$

$$\begin{aligned} y_0^{n+1} &= J_0^n \cdot \bar{Q}_0^n + \bar{K}_0^n \cdot Q_0^n \\ &= ((x^n + y_1^n) \cdot \bar{y}_0^n) \cdot \bar{y}_0^n + ((x^n + y_1^n) \cdot \bar{y}_0^n) \cdot y_0^n \\ &= (x^n + y_1^n) \cdot \bar{y}_0^n \end{aligned}$$

Table de transitions

		X = 0		X = 1		X = 0	X = 1
y_1^n	y_0^n	y_1^{n+1}	y_0^{n+1}	y_1^{n+1}	y_0^{n+1}	Z^n	
0	0	0	0	1	1	0	1
0	1	0	0	1	0	0	0
1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	0

La table devient vite complexe. Sa lecture devient un peu plus simple et claire si l'on remplace les valeurs des états par une lettre indicée :

$$\begin{aligned} y_1 y_0 = 00 &\rightarrow e_0, y_1 y_0 = 01 \rightarrow e_1, y_1 y_0 = 10 \\ &\rightarrow e_2, y_1 y_0 = 11 \rightarrow e_3 \end{aligned}$$

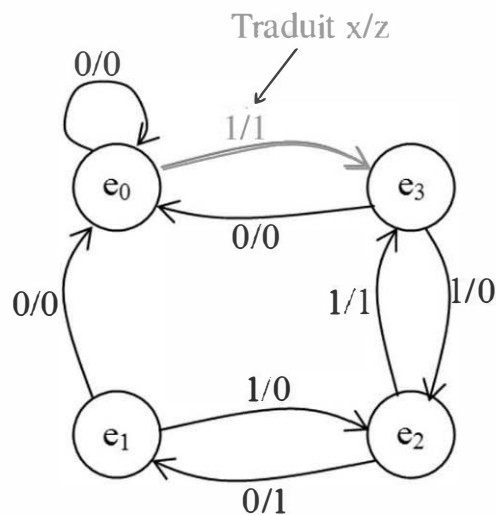
On obtient alors la table d'état.

Table d'états

Etat présent	Etat suivant	
	$x^n=0$	$x^n=1$
e_0	$e_0, 0$	$e_3, 1$
e_1	$e_0, 0$	$e_2, 0$
e_2	$e_1, 1$	$e_3, 1$
e_3	$e_0, 0$	$e_2, 0$

Etat suivant
Valeur de la sortie

Diagramme d'états



La lecture des valeurs au-dessus des arcs se fait de la manière suivante. Prenons l'exemple de l'arc en couleur rouge : lorsque le système est dans l'état e_0 et qu'il reçoit sur son entrée la valeur logique 1, il passe dans l'état e_3 et sa sortie prend la valeur logique 1.

Méthode de synthèse des compteurs synchrones

La synthèse d'un compteur synchrone consiste à déterminer les valeurs logiques à donner aux entrées de chaque bascule, c'est-à-dire trouver les équations des entrées. Ces équations dépendent :

- de la nature et du nombre d'états du code comptage que l'on désire obtenir ;
- du type de bascule choisi : D, J-K, ou T. Le plus souvent toutes les bascules d'un compteur sont identiques.

Un compteur est constitué d'autant de bascules qu'il y a de bit dans chaque état du comptage.

Les différentes étapes de synthèse d'un compteur sont :

- **Mise en équation des entrées** : à l'aide de la table de transition du type de bascule choisi, on dresse le tableau de toutes les valeurs logiques que doivent prendre les entrées de chaque bascule pour donner tous les changements d'états imposés par le code de comptage. Il ne faut pas oublier l'enchaînement du dernier état avec le premier si le code doit se répéter cycliquement. Si l'on souhaite au contraire que le compteur se bloque dans le dernier état, il faut prévoir l'enchaînement de ce dernier état avec lui-même.
- **Simplification par la méthode de Karnaugh** afin d'obtenir l'expression minimale de chaque équation d'entrée, à partir des états présents des sorties Q des bascules.
- **Enfin réalisation du circuit.**



EXEMPLE 2 : SYNTHÈSE D'UN COMPTEUR SYNCHRONES 3 BITS AVEC DES BASCULES J-K

Le mode de comptage est le binaire naturel. Le comptage est cyclique. On commence par établir la table de transitions :

États présents (EP)			États suivants (ES)			Entrées					
Q_c	Q_b	Q_a	Q_c	Q_b	Q_a	J_c	K_c	J_b	K_b	J_a	K_a
0	0	0	0	0	1	0	Φ	0	Φ	1	Φ
0	0	1	0	1	0	0	Φ	1	Φ	Φ	1
0	1	0	0	1	1	0	Φ	Φ	0	1	Φ
0	1	1	1	0	0	1	Φ	Φ	1	Φ	1
1	0	0	1	0	1	Φ	0	0	Φ	1	Φ
1	0	1	1	1	0	Φ	0	1	Φ	Φ	
1	1	0	1	1	1	Φ	0	Φ	0	1	Φ
1	1	1	0	0	0	Φ	1	Φ	1	Φ	1

On cherche ensuite les équations.

J_a et K_a sont triviales, en choisissant $\Phi = 1$, on obtient :

$$J_a = K_a = 1$$

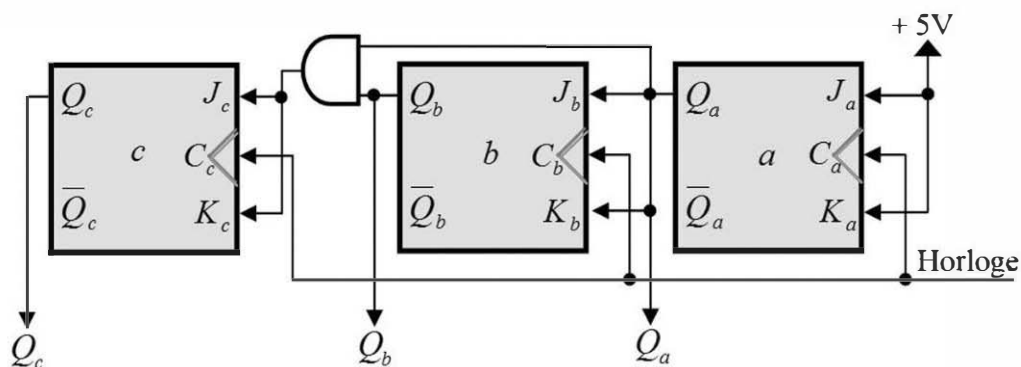
J_b					K_b				
$Q_b Q_a$	00	01	11	10	$Q_b Q_a$	00	01	11	10
Q_c					Q_c				
0	Φ	Φ	1	0	0	0	1	Φ	Φ
1	Φ	Φ	1	0	1	0	1	Φ	Φ

$$J_b = K_b = Q_a$$

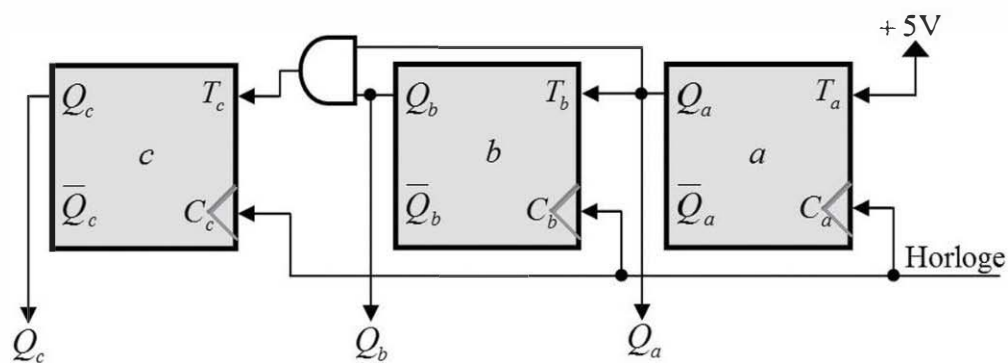
J_c					K_c				
$Q_b Q_a$	00	01	11	10	$Q_b Q_a$	00	01	11	10
Q_c					Q_c				
0	Φ	Φ	Φ	Φ	0	0	0	1	0
1	0	0	1	0	1	Φ	Φ	Φ	Φ

$$J_c = K_c = Q_b \cdot Q_a$$

Le schéma logique est donc le suivant :



On remarque que, d'après les équations (et donc aussi sur le schéma), les bascules ont $J = K$. On peut alors les remplacer par des bascules T. On aura le schéma suivant :



Les registres

Registre parallèle

Considérons l'association **parallèle** de bascules D (figure 5.20).

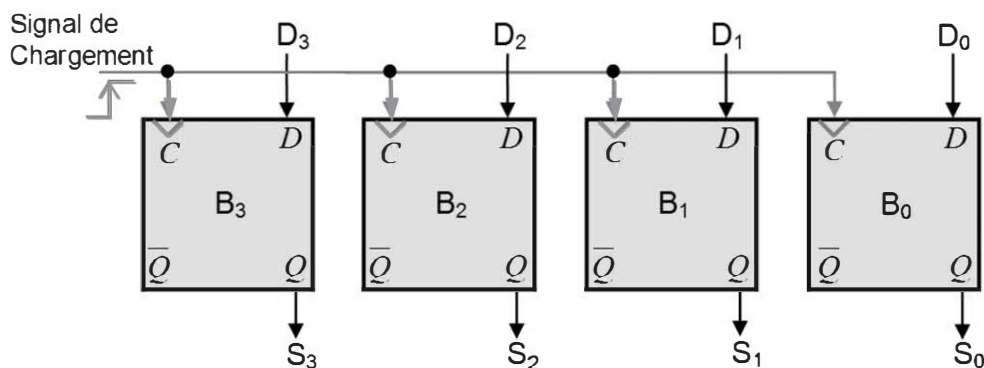


Figure 5.20 Registre parallèle 4 bits.

On obtient ici un registre de 4 bits : les 4 bascules sont chargées en parallèle par la transition $0 \rightarrow 1$ du signal d'échantillonnage (signal de chargement). Les entrées sont censées véhiculer des informations de même nature : les lignes d'entrées constituent le **BUS** d'entrée. Les sorties Q constituent de même le bus de sortie. L'évolution de ce registre peut-être symbolisée par le chronogramme de la figure 5.21.

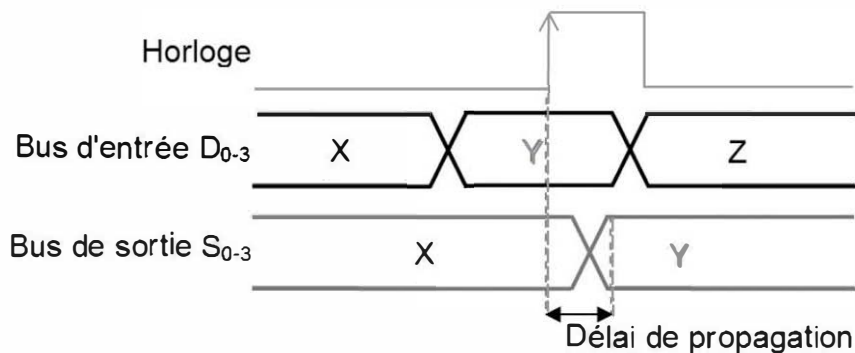
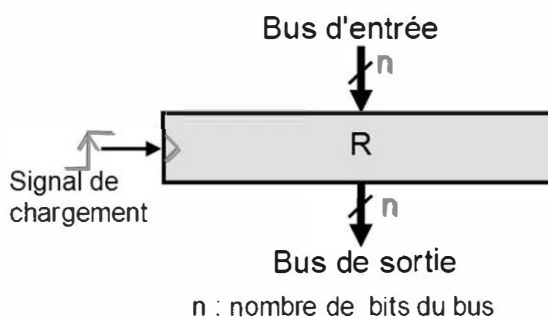


Figure 5.21 Chronogramme du comportement du registre parallèle.

La figure 5.22 montre une façon synthétique pour représenter l'ensemble bus et registre.



n : nombre de bits du bus

Figure 5.22 Représentation symbolique d'un registre parallèle de n bits.

Registre à lecture et écriture parallèle

La figure 5.23, montre un schéma d'un registre 4 bits à écriture et lecture parallèle. On remarque que l'écriture est séquentielle alors que la lecture est totalement combinatoire.

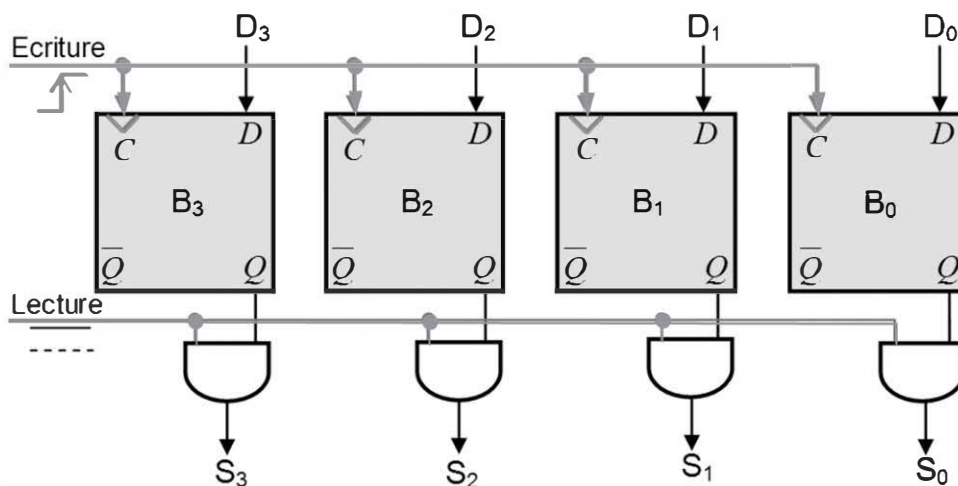


Figure 5.23 Registre à écriture (chargement) et lecture parallèle.

Registres série (registres à décalage)

Considérons l'association **série** de bascules D de la figure 5.24.

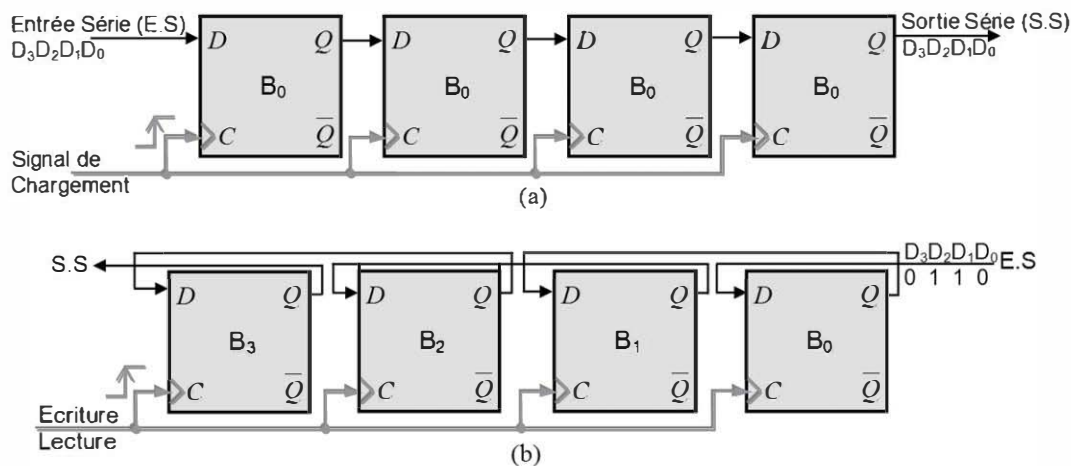


Figure 5.24 Registre Série 4 bits : (a) à décalage à droite, (b) à décalage à gauche.

Ici 4 transitions (*tops*) d'horloge sont nécessaires pour charger ou écrire le registre et 4 tops sont aussi nécessaires pour lire les données par décalage. Le signal de chargement (Écriture) sert aussi de signal de lecture.

Registres à écriture série/lecture série et parallèle

La figure 5.25 montre deux exemples de câblage de registres à décalage à droite et à gauche en mode lecture/écriture série et lecture parallèle. Il est facilement envisageable de concevoir un **registre universel programmable** pouvant réaliser toutes combinaisons des opérations de lecture et d'écriture.

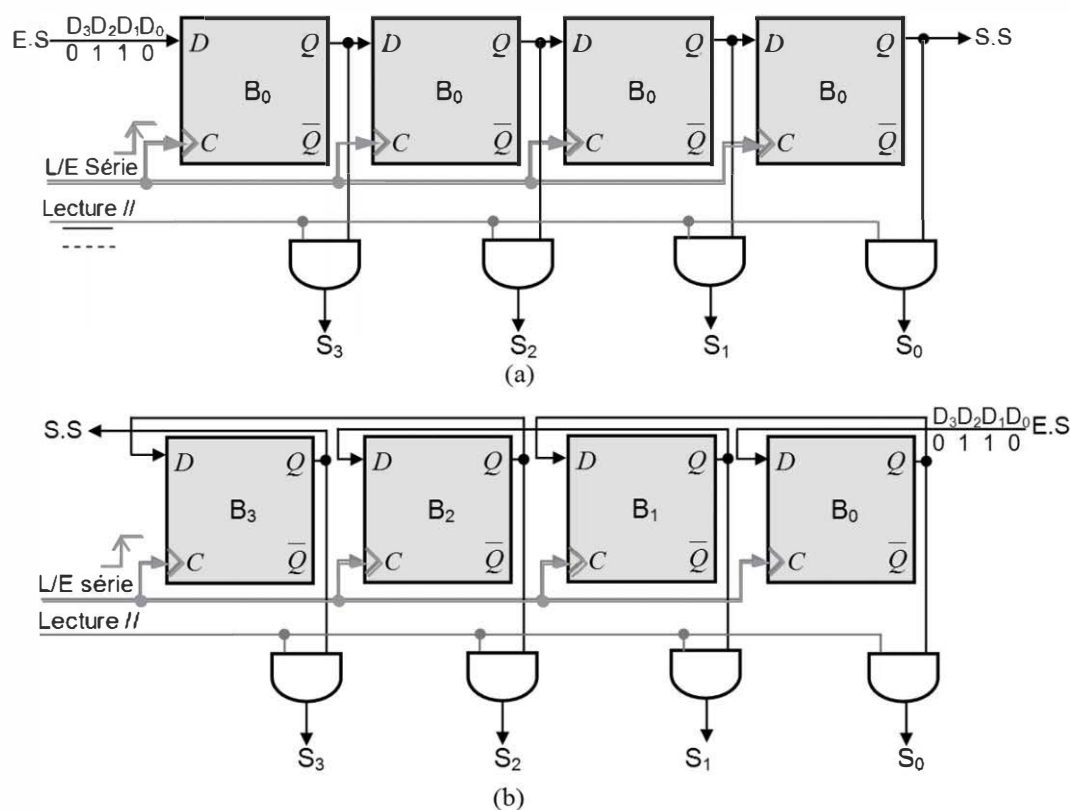


Figure 5.25 Registre à écriture série/lecture série et parallèle : (a) registre à décalage à droite, (b) registre à décalage à gauche.

Association de registres

Cas d'un seul registre source

Considérons le cas d'un registre source en communication avec plusieurs registres destination. Les registres peuvent être reliés par des bus de communication (figure 5.26)

Pour transférer le contenu de R_1 dans R_3 par exemple il suffit d'activer C_3 . Le chargement en parallèle de plusieurs registres est possible, par exemple en activant C_2 et C_3 simultanément. Le problème est ici simple car il n'y a qu'une seule source d'information. Ajouter des registres en destination ne pose pas de problèmes majeurs mis à part la limitation du

courant que peut fournir le bus de sortie du registre source R_1 . Par contre si l'on rajoute un registre en source d'information, de la logique supplémentaire est nécessaire.

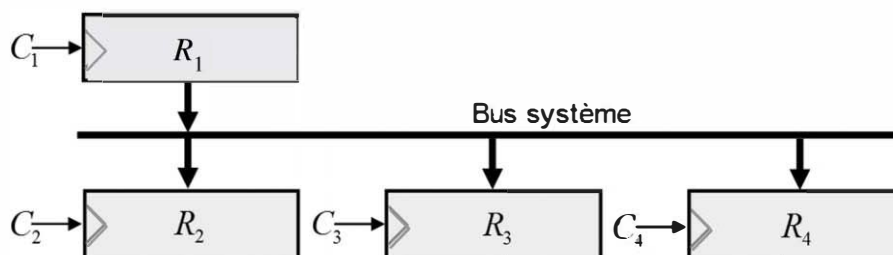


Figure 5.26 Communication entre un registre source et plusieurs registres destination.

Cas de plusieurs registres sources

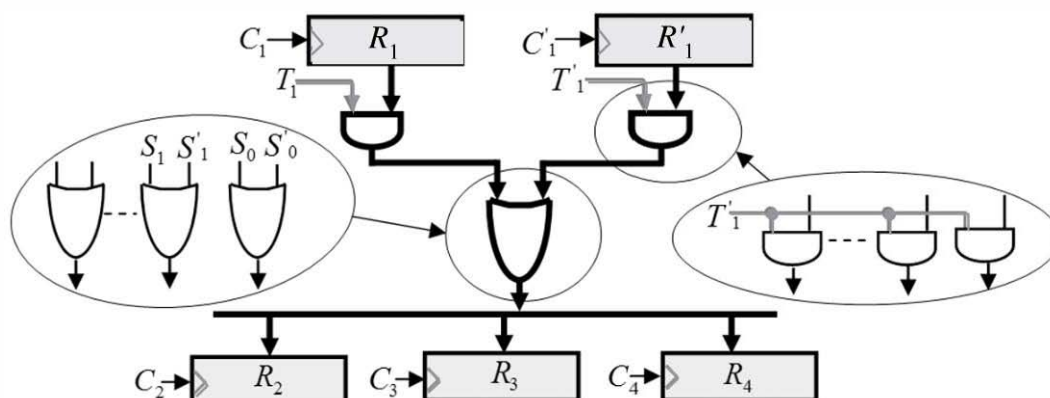


Figure 5.27 Communication entre plusieurs registres sources et registres destination.

Pour transférer le contenu de R_1 dans R_3 il faut :

- activer le signal T_1 de commande des portes ET ;
- laisser l'information se propager et se stabiliser sur le bus de communication ;
- activer le signal de chargement C_3 ;
- désactiver le signal T_1 afin de libérer le bus.

T_1 et T'_1 ne doivent jamais être actifs simultanément sinon la fonction de transfert n'est pas correctement assurée.

Problème d'ajout de registre source

Un registre et ses portes (ET logique) de commande associées constituent un bloc modulaire. L'adjonction d'un registre supplémentaire

revient à connecter au bus un bloc registre et ses portes de commande. Malheureusement il aurait fallu avoir prévu 8 entrées supplémentaires sur la porte OU ; l'expansion (l'adjonction) ne semble donc pas simple.

Solution à base de OU câblé

La technologie pallie à cet inconvénient en permettant de réaliser un OU câblé (*wired OR*). Ceci est possible à condition que l'impédance de sortie de la porte de commande soit très élevée lorsque cette dernière n'est pas commandée : dans ce cas elle peut être considérée comme électriquement déconnectée (retirée) du bus.

Cette porte spéciale ne se comporte plus tout à fait comme un ET logique, mais plutôt comme une barrière :

- Lorsqu'elle est ouverte ($C(ommande)=1$) elle présente en sortie l'état de son entrée : 0 ou 1.
- Lorsqu'elle est fermée ($C(ommande)=0$) elle présente en sortie un état haute impédance.

D'où son nom de **barrière « trois états »** (*tri-state*). La figure 5.28, montre la différence de comportement entre une porte ET logique et une porte trois états.

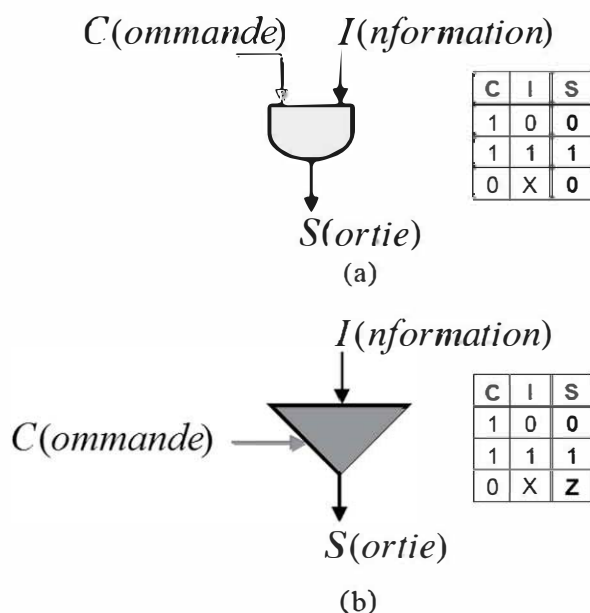


Figure 5.28 (a) Porte ET logique et sa table de vérité (b) porte trois états et sa table de vérité.

Ainsi, il est possible de considérer un bus comme une voie de communication entre registres, ces derniers pouvant être en nombre quelconque. Un seul bus ne permet qu'un transfert à la fois. En rebouclant le bus sur

les entrées des registres R_1 , R'_1 , R''_1 et en connectant les sorties des registres R_2 , R_3 , R_4 et sur ce bus tous les transferts deviennent possibles, d'une seule source vers une ou plusieurs destinations (figure 5.29).

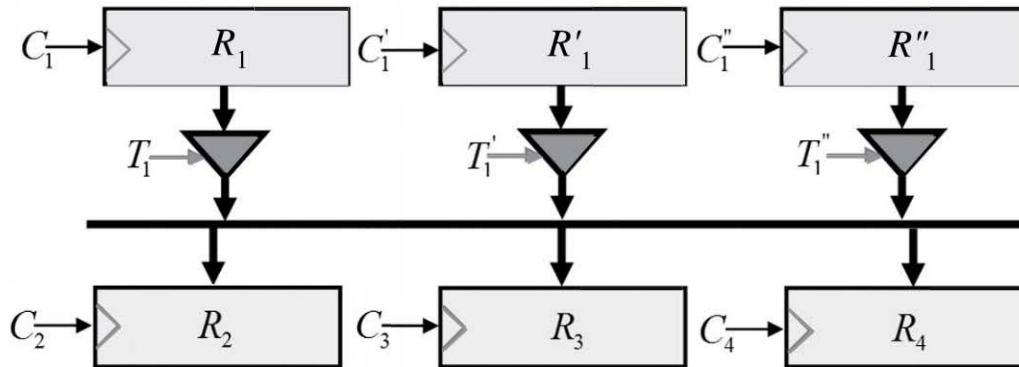


Figure 5.29 Le bloc (registre+portes 3 états) simplifie la communication entre registres et l'ajout de registres sources et destination.

EXERCICES

5-1 Registre parallèle

En utilisant des bascules D à déclenchement sur front, réaliser un registre de 3 bits à écriture et lecture parallèle, possédant les signaux suivants :

- $E < 2..0 >$: bus de données en entrée sur 3 bits.
- $O < 2..0 >$: bus de données en sortie sur 3 bits.
- WR : signal permettant l'écriture dans le registre.
- RD : signal permettant la lecture du registre.

Pour écrire les données on utilisera les entrées de forçage à 1 (SET) et à 0 ($RESET$) c'est-à-dire les entrées $\bar{S}$ et $\bar{R}$ de chaque bascule. Donnez les équations des entrées $\bar{S}$ et $\bar{R}$ pour réaliser une écriture et les équations du bus de sortie pour une lecture. Donnez le schéma logique de ce registre.

5-2 Registre universel programmable

On utilisera dans cet exercice des bascules D à déclenchement sur front. On désire réaliser un registre universel programmable à écriture/lecture

parallèle, écriture/lecture série à droite ou à gauche, le registre possédant les signaux suivants :

- $E < 2..0 >$: bus de données d'entrée sur 3 bits en utilisant l'entrée D de chaque bascule.
- $O < 2..0 >$: bus de sorties sur 3 bits en utilisant la sortie Q de chaque bascule.
- E_{SD} : entrée série droite.
- E_{SG} : entrée série gauche.
- E_{SD} : sortie série droite. Il s'agira de Q_0 .
- E_{SG} : sortie série gauche. Il s'agira de Q_2 .
- WR : signal permettant l'écriture dans le registre.
- RD : signal permettant la lecture sérielle du registre.
- $\overline{CLR}$: signal permettant la mise à 0 du registre.
- $\overline{SET}$: signal permettant la mise à 1 du registre.
- $P_1 P_0$: 2 signaux permettant la programmation du registre (00 écriture parallèle, 01 décalage à droite, 10 décalage à gauche, la combinaison 11 est non utilisée).

Donnez les équations des entrées des bascules (D_2 , D_1 et D_0). Donnez le schéma logique permettant de réaliser l'équation de D_2 par exemple. Quelle est la fonction assurée par ce schéma ? Donnez le schéma logique du registre universel en ne considérant que la représentation symbolique de la fonction réalisant les équations des entrées.

5-3 Gestion d'une file de type FIFO (*First In First Out*)

On utilisera dans cet exercice des bascules D à déclenchement sur front. On ne tiendra pas compte dans tout l'exercice des problèmes liés à l'initialisation.

1) Registre parallèle

Soit un registre de 4 bits à lecture et écriture parallèle, possédant les signaux suivants :

- $D < 3..0 >$: bus de données d'entrée sur 4 bits ;
- $S < 3..0 >$: bus de sorties sur 4 bits ;
- WR : signal permettant l'écriture dans le registre sur chaque front montant.

Donnez le schéma logique de ce registre.

2) Gestion du défilement

On considère maintenant un circuit possédant 4 registres de 4 bits chacun : R_0 , R_1 , R_2 et R_3 . Il possède les signaux suivants :

- CLK : signal réalisant le défilement ;
- $S < 3..0 >$: bus de sorties sur 4 bits.

Son fonctionnement est le suivant :

- à chaque front montant sur l'entrée CLK , le contenu de R_2 (respectivement R_1 , R_0) passe dans R_3 (respectivement R_2 , R_1) et reçoit 0 ;
- $S < 3..0 >$: le bus S fournit le mot contenu dans le registre R_3 .

Donnez le schéma logique de ce circuit.

3) Écriture sélective

On s'intéresse maintenant à un nouveau circuit, possédant toujours 4 registres de 4 bits chacun, et pourvu des signaux suivants :

- $D < 3..0 >$: bus de données d'entrée sur 4 bits ;
- CLK : signal réalisant l'écriture ;
- S_1 et S_0 : signaux indiquant le numéro du registre dans lequel il faut écrire la donnée présente sur D .

Lors d'un front montant sur CLK , la donnée présente sur D est écrite dans le registre dont le numéro correspond au nombre formé par S_1 et S_0 . Donnez le schéma logique de ce circuit.

4) File complète

On désire maintenant réaliser une file de type FIFO (*First In First Out*). Ce circuit possède les signaux suivants : $D < 3..0 >$, $S < 3..0 >$, CLK , $E_{cr} \overline{D_{ef}}$, S_1 et S_0 . Le signal $E_{cr} \overline{D_{ef}}$ permet de faire la différence entre une demande d'écriture (signal à 1) et une demande de défilement (signal à 0). Le rôle des autres signaux a été expliqué dans les questions précédentes. Le fonctionnement est le suivant :

- lors d'un front montant sur CLK avec $E_{cr} \overline{D_{ef}}$ à 1, la donnée présente sur D est écrite dans le registre dont le numéro correspond au nombre formé par S_1 et S_0 (les valeurs de ces derniers étant gérées de manière extérieure au circuit) ;
- à chaque front montant sur l'entrée CLK avec $E_{cr} \overline{D_{ef}}$ à 0, le contenu de R_2 (respectivement R_1 , R_0) passe dans R_3 (respectivement R_2 , R_1) et R_0 reçoit 0 ; (le contenu de R_3 est produit sur le bus de sortie S).

Donnez le schéma logique de ce circuit.

5-4 Compteur pseudo-aléatoire

1) En utilisant des bascules D puis des bascules J-K, et des opérateurs de votre choix, donnez les équations simplifiées par Karnaugh et le schéma d'un compteur synchrone à 7 états ayant le code de comptage cyclique suivant :

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

2) Que se passe-t-il si, à la mise sous tension, si le compteur utilisant les bascules D se trouve dans l'état : 000 ?

3) Utilisez le principe de la rétroaction sur les entrées de forçage (*Set/Reset*) des bascules (rebouclage des sorties sur des entrées), pour éviter cet inconvénient. Quelle autre solution aurait-on pu utiliser ?

5-5 Gestion de l'index d'écriture d'une file de type FIFO (*First In First Out*)

On reprend la file de type FIFO vue à l'exercice. On cherche à concevoir le compteur de gestion de l'index d'écriture.

Donnez le schéma logique du circuit de gestion de S_1 et S_0 (on suppose qu'un défilement ne peut pas se produire lorsque la file est vide ; de la même manière, une écriture ne peut pas se produire lorsque la file est pleine).

SOLUTIONS

5-1 Registre parallèle

Pour concevoir un registre trois bits, nous allons utiliser trois bascules. Établissons la table de vérité des entrées de forçage de la bascule i .

WR	E_i	S_i	R_i
0	0	1	1
0	1	1	1
1	0	1	0
1	1	0	1

Lorsque WR est à 0, les entrées de forçage doivent rester inactives ; elles sont donc positionnées à 1. Par contre, lorsque WR est à 1, l'entrée de forçage correspondant à la valeur du bit à écrire ($\overline{S_i}$ si on veut écrire 1, $\overline{R_i}$ si on veut écrire 0) est sollicitée (donc mise à 0), l'autre entrée restant inactive (à 1).

On en déduit les équations suivantes :

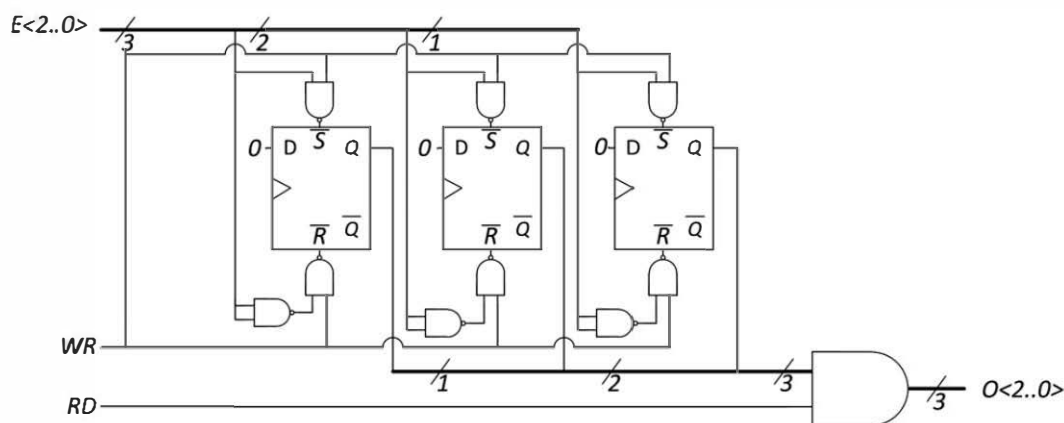
$$\overline{S_i} = \overline{WR} + \overline{E_i} = \overline{WR \cdot E_i} = \overline{WR/E_i}$$

$$\overline{R_i} = \overline{WR} + E_i = \overline{WR \cdot \overline{E_i}} = \overline{WR/(E_i/E_i)}$$

Le bus de sortie recevra le mot contenu dans le registre lorsque le signal de lecture RD sera à 1. On en déduit, pour chaque bascule i :

$$O_i = Q_i \cdot RD$$

On en déduit le schéma suivant :



5-2 Registre universel programmable

Les entrées des bascules dépendent de la configuration du registre, donc de $P_1 P_0$. Nous allons donc donner les équations des entrées des bascules en fonction de ces deux variables. La table de vérité est la suivante :

P_{10}	P_0	D_2	D_1	D_0
0	0	E_2	E_1	E_0
0	1	E_{SG}	Q_2	Q_1
1	0	Q_1	Q_0	E_{SD}
1	1	Φ	Φ	Φ

On en déduit les équations des entrées des bascules :

$$D_2 = P_1 \cdot P_0 \cdot E_2 + P_1 \cdot P_0 \cdot E_{SG} + P_1 \cdot P_0 \cdot Q_1$$

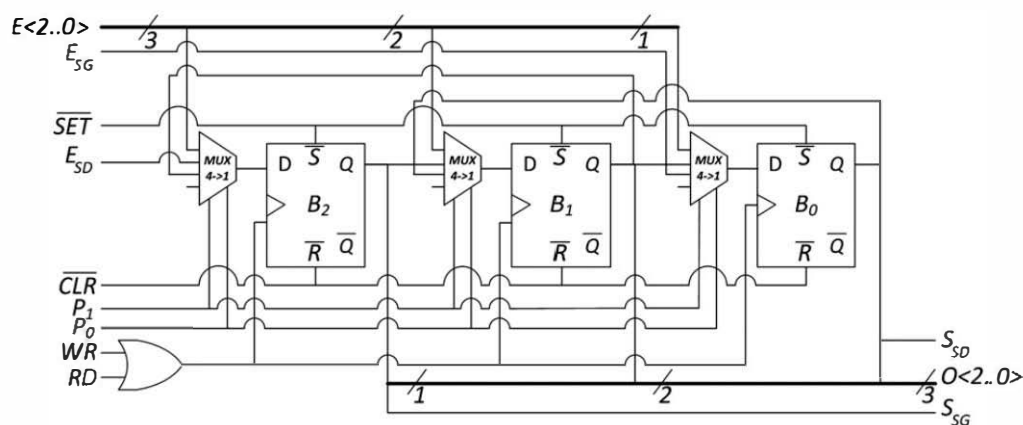
$$D_1 = P_1 \cdot P_0 \cdot E_1 + P_1 \cdot P_0 \cdot Q_2 + P_1 \cdot P_0 \cdot Q_0$$

$$D_0 = P_1 \cdot P_0 \cdot E_0 + P_1 \cdot P_0 \cdot Q_1 + P_1 \cdot P_0 \cdot E_{SD}$$

Il s'agit, pour chaque entrée, de l'équation d'un multiplexeur 4 vers 1, piloté par $P_1 P_0$, et dont la quatrième entrée est non utilisée.

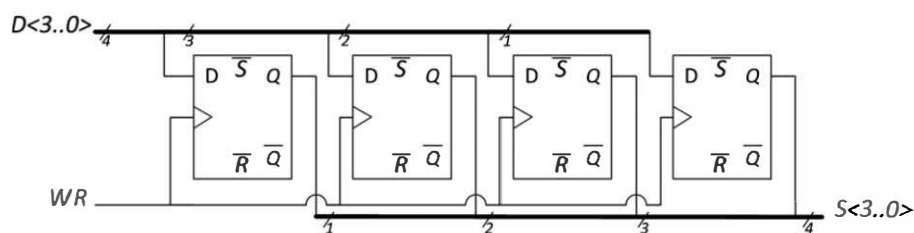
Notons enfin que l'entrée d'horloge de chaque bascule va réagir soit sur une demande d'enregistrement (signal WR), soit sur une demande de lecture série (signal RD) puisque la lecture série nécessite le décalage des informations contenues dans le registre.

On en déduit le schéma suivant :



5-3 Gestion d'une file de type FIFO (First In First Out)

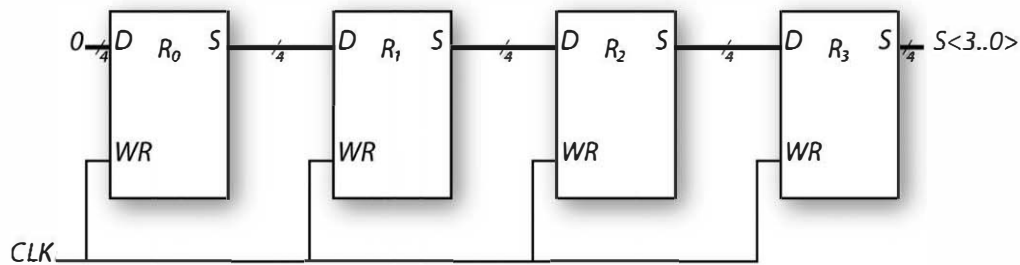
1) Schéma logique du registre parallèle



2) Gestion du défilement

Lors d'un front montant sur l'entrée CLK , le contenu de R_2 passe dans R_3 ; l'entrée de R_3 est donc connectée à la sortie de R_2 . On procède de même avec R_2 et R_1 ainsi qu'avec R_1 et R_0 . R_0 reçoit 0, donc il suffit de présenter cette valeur en entrée de ce registre.

Enfin, le bus fournit le mot contenu dans le registre R_3 ; c'est donc la sortie de ce dernier qui est connectée à S . On en déduit le schéma suivant :

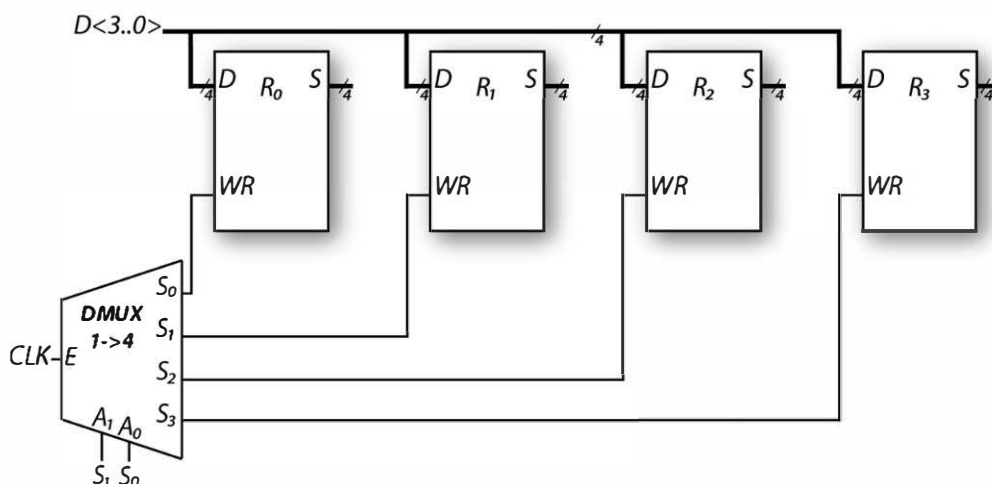


3) Écriture sélective

Lors d'un front montant sur CLK , seul un registre mémorise la donnée : celui qui est désigné par les signaux S_1 et S_0 . Le signal d'horloge CLK ne doit donc être transmis qu'à l'entrée WR de ce seul registre (les entrées WR des autres registres restant à 0). Il faut donc un circuit capable de ne transmettre son entrée (signal CLK) qu'à une seule de ses quatre sorties (une sortie pour le signal WR de chaque registre). Un tel circuit a déjà été étudié : il s'agit d'un démultiplexeur. Ici, il sera commandé par les signaux S_1 et S_0 . Nous ne considérerons que la représentation graphique de ce démultiplexeur, son schéma ayant déjà été étudié dans le chapitre 4.

Quel que soit le registre qui est désigné par S_1 et S_0 pour enregistrer une donnée, l'information mémorisée est toujours celle présente sur le bus D . Ce bus est donc connecté en entrée de chaque registre.

On en déduit le schéma suivant :



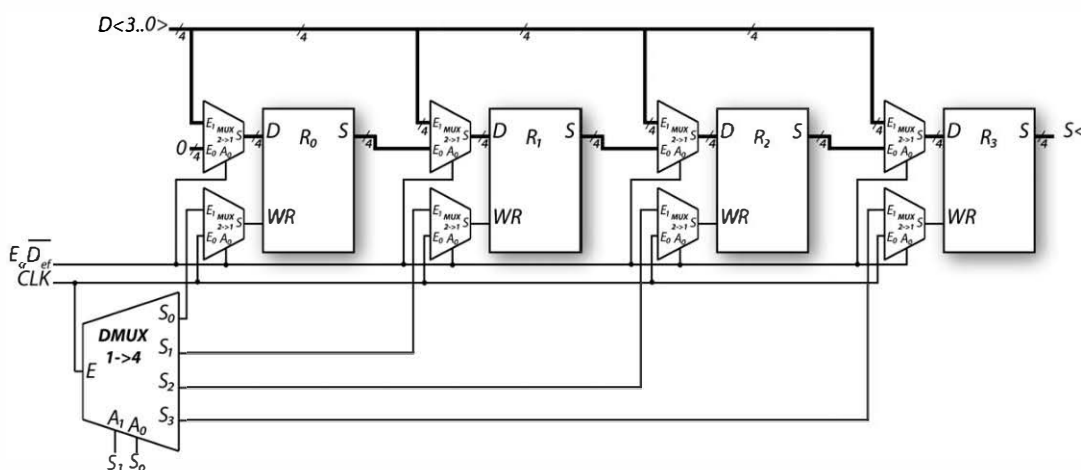
4) File complète

Il faut faire la synthèse des deux questions précédentes, en utilisant le signal $E_{cr} \overline{D_{ef}}$ pour choisir les signaux à connecter aux différentes entrées.

Au niveau de l'entrée de donnée de chaque registre, il faut choisir entre le signal issu du registre précédent ou 0 (dans le cas d'un défilement, donc lorsque $E_{cr} \overline{D_{ef}}$ est à 0) ou la donnée présente sur D (dans le cas d'une écriture, c'est-à-dire quand $E_{cr} \overline{D_{ef}}$ est à 1). On peut utiliser un multiplexeur 2 vers 1, piloté par $E_{cr} \overline{D_{ef}}$, pour effectuer cette sélection.

Au niveau de l'entrée d'horloge de chaque registre, il faut là aussi choisir entre CLK (lors d'un défilement, donc lorsque $E_{cr} \overline{D_{ef}}$ est à 0) ou la sortie du démultiplexeur 1 vers 4 (lors d'une écriture, c'est-à-dire quand $E_{cr} \overline{D_{ef}}$ est à 1). Là encore, un multiplexeur 2 vers 1 peut être utilisé.

On en déduit le schéma suivant :



Notons que ce schéma nécessite, pour fonctionner correctement, de respecter les contraintes temporelles des bascules D. Il faut donc, dans l'ordre, positionner le signal $E_{cr} \overline{D_{ef}}$, puis positionner les données sur le bus le cas échéant, et enfin, lorsque ces données sont stabilisées, activer le signal d'horloge CLK .

5-4 Compteur pseudo-aléatoire

1) Commençons par dresser la table de transition en considérant des bascules D :

Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}	D_2^n	D_1^n	D_0^n
0	0	0	Φ	Φ	Φ	Φ	Φ	Φ
0	0	1	1	0	0	1	0	0
0	1	0	1	0	1	1	0	1
0	1	1	0	0	1	0	0	1
1	0	0	0	1	0	0	1	0
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	1	1	0	1	1

On en déduit les tables de Karnaugh et les équations simplifiées suivantes :

Pour D_2^n :

Table de Karnaugh remplie					Regroupements				
$Q_0^n \quad E_{CR} \quad \overline{D_{EF}}$	00	01	11	10	$Q_0^n \quad E_{CR} \quad \overline{D_{EF}}$	00	01	11	10
$Q_2^n Q_1^n$					$Q_2^n Q_1^n$				
00	Φ				00	Φ			
01			1		01			1	
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10		Φ	Φ	Φ	10		Φ	Φ	Φ

$$D_2^n = \overline{Q_0^n} \cdot Q_1^n + Q_0^n \cdot \overline{Q_1^n} = Q_0^n \oplus Q_1^n$$

Pour D_1^n :

Table de Karnaugh remplie					Regroupements				
$Q_0^n \quad E_{CR} \quad \overline{D_{EF}}$	00	01	11	10	$Q_0^n \quad E_{CR} \quad \overline{D_{EF}}$	00	01	11	10
$Q_2^n Q_1^n$					$Q_2^n Q_1^n$				
00	Φ		1		00	Φ		1	
01		1		1	01		1		1
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10	1	Φ	Φ	Φ	10	1	Φ	Φ	Φ

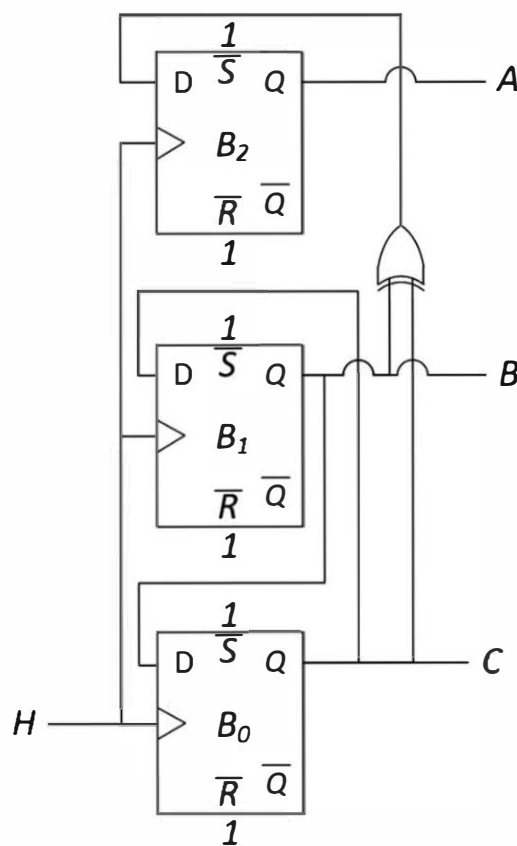
$$D_1^n = Q_0^n$$

Pour D_0'' :

Table de Karnaugh remplie					Regroupements				
$Q_2^n \quad Q_1^n$	00	01	11	10	$Q_2^n \quad Q_1^n$	00	01	11	10
00	Φ	1			00	Φ	1		
01	1	1			01	1	1		
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10	1		Φ	Φ	10	1		Φ	Φ

$$D_0'' = Q_1^n$$

Le schéma correspondant est donc :



Pour dresser la table de transition relative aux bascules JK, commençons par rappeler la table de transition de la bascule JK :

J_2^n	K_2^n	Q_2^{n+1}
0	0	Q_2^n
0	1	0
0	1	1
1	1	$\overline{Q_2^n}$

On peut en déduire la table précisant les valeurs à affecter aux entrées J_2^n et K_2^n en fonction de l'état courant Q_2^n et de l'état futur Q_2^{n+1} :

Q_2^n	Q_2^{n+1}	J_2^n	K_2^n
0	0	0	Φ
0	1	1	Φ
1	0	Φ	1
1	1	Φ	0

En utilisant des bascules JK, on a la table de transition suivante :

Q_2^n	Q_1^n	Q_0^n	Q_2^{n+1}	Q_1^{n+1}	Q_0^{n+1}	J_2^n	K_2^n	J_1^n	K_1^n	J_0^n	K_0^n
0	0	0	Φ	Φ	Φ	Φ	Φ	Φ	Φ	Φ	Φ
0	0	1	1	0	0	1	Φ	0	Φ	Φ	1
0	1	0	1	0	1	1	Φ	Φ	1	1	Φ
0	1	1	0	0	1	0	Φ	Φ	1	Φ	0
1	0	0	0	1	0	Φ	1	1	Φ	0	Φ
1	0	1	1	1	0	Φ	0	1	Φ	Φ	1
1	1	0	1	1	1	Φ	0	Φ	0	1	Φ
1	1	1	0	1	1	Φ	1	Φ	0	Φ	0

On en déduit les tables de Karnaugh et les équations simplifiées suivantes :

Pour J_2^n :

Table de Karnaugh remplie					Regroupements				
$Q_1^n Q_0^n$	00	01	11	10	$Q_1^n Q_0^n$	00	01	11	10
Q_2^n					Q_2^n				
0	Φ	1		1	0	Φ	1		1
1	Φ	Φ	Φ	Φ	1	Φ	Φ	Φ	Φ

$$J_2^n = \overline{Q_1^n} + \overline{Q_0^n}$$

Pour K_2^n :

Table de Karnaugh remplie					Regroupements				
$Q_1^n Q_0^n$	00	01	11	10	$Q_1^n Q_0^n$	00	01	11	10
Q_2^n					Q_2^n				
0	Φ	Φ	Φ	Φ	0	Φ	Φ	Φ	Φ
1	1		1		1	1		1	

$$K_2^n = \overline{Q_1^n} \cdot \overline{Q_0^n} + Q_1^n \cdot Q_0^n = \overline{Q_1^n} \oplus Q_0^n$$

Pour J_1^n :

Table de Karnaugh remplie					Regroupements				
$Q_1^n Q_0^n$	00	01	11	10	$Q_1^n Q_0^n$	00	01	11	10
Q_2^n					Q_2^n				
0	Φ		Φ	Φ	0	Φ		Φ	Φ
1	1	1	Φ	Φ	1	1	1	Φ	Φ

$$J_1^n = Q_2^n$$

Pour K_1^n :

Table de Karnaugh remplie					Regroupements				
$Q_1^n Q_0^n$	00	01	11	10	$Q_1^n Q_0^n$	00	01	11	10
Q_2^n					Q_2^n				
0	Φ	Φ	1	1	0	Φ	Φ	1	1
1	Φ	Φ			1	Φ	Φ		

$$K_1^n = \overline{Q_2^n}$$

Pour J_0^n :

Table de Karnaugh remplie					Regroupements				
$Q_1^n Q_0^n$	00	01	11	10	$Q_1^n Q_0^n$	00	01	11	10
Q_2^n					Q_2^n				
0	Φ	Φ	Φ	1	0	Φ	Φ	Φ	1
1		Φ	Φ	1	1		Φ	Φ	1

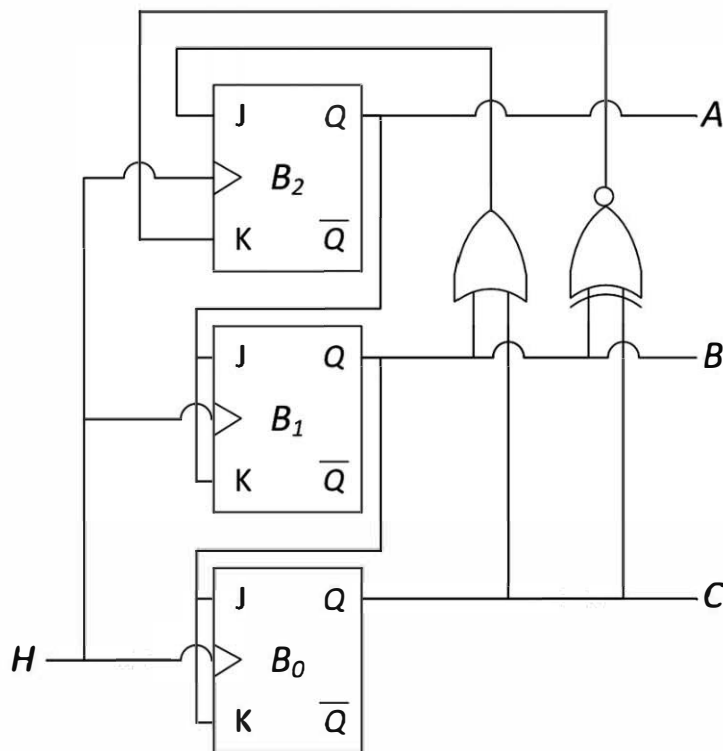
$$J_0^n = Q_1^n$$

Pour K_0^n :

Table de Karnaugh remplie					Regroupements				
$Q_1^n Q_0^n$	00	01	11	10	$Q_1^n Q_0^n$	00	01	11	10
Q_2^n					Q_2^n				
0	Φ	1		Φ	0	Φ	1		Φ
1	Φ	1		Φ	1	Φ	1		Φ

$$K_0^n = \overline{Q_1^n}$$

On en déduit le schéma suivant :



2) Lorsqu'on se trouve dans l'état 000 avec le compteur utilisant les bascules D, on a :

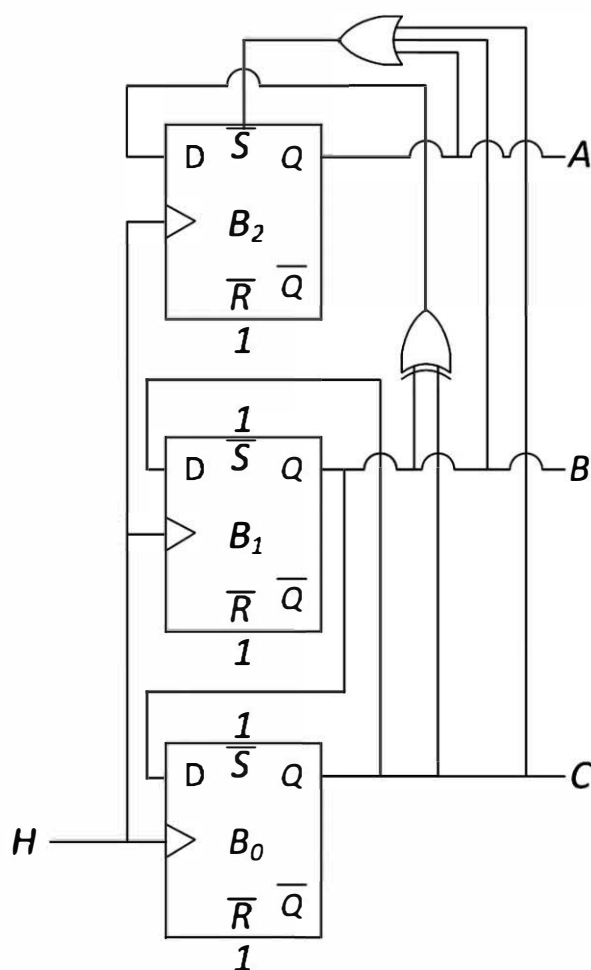
$$D_2^n = Q_0^n \oplus Q_1^n = 0 \oplus 0 = 0$$

$$D_1^n = Q_0^n = 0$$

$$D_0^n = Q_1^n = 0$$

On reste donc dans ce même état.

3) Si on décide, par exemple, de passer dans l'état 100 lorsque l'état 000 est détecté, il faut forcer l'entrée $\overline{S}$ de la bascule B_2 à 0 dans ce cas. On obtient alors le schéma suivant :



5-5 Gestion de l'index d'écriture d'une file de type FIFO (First In First Out)

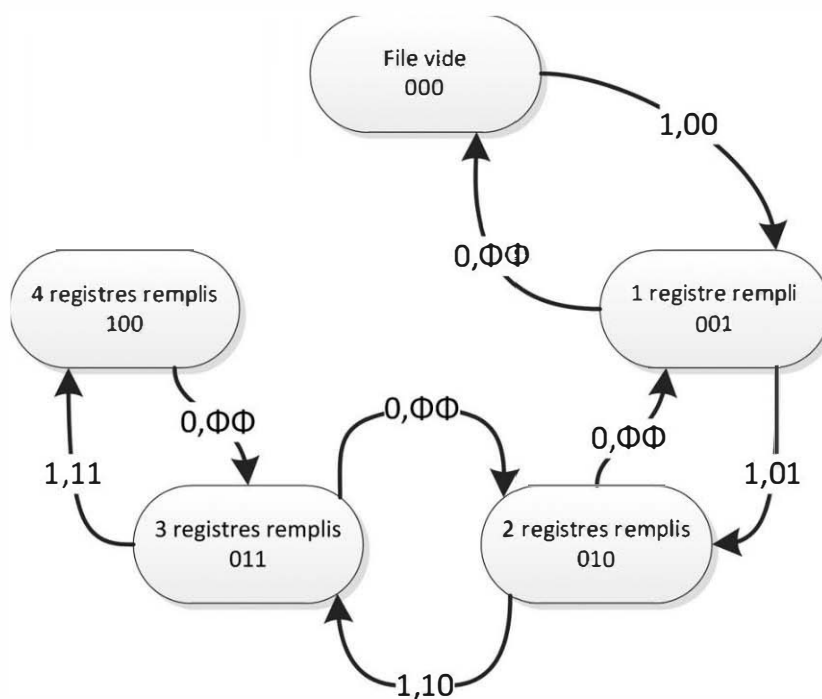
- File vide (état initial)
- 1 registre rempli
- 2 registres remplis
- 3 registres remplis
- 4 registres remplis (file pleine).

Il nous faudra donc 3 bascules pour représenter ces cinq états. Choisissons une numérotation directe : l'état « file vide » sera représenté 000, etc. jusqu'à l'état « 4 registres remplis (file pleine) » représenté 100.

Notre circuit va évoluer à chaque front du signal d'horloge CLK . Le nouvel état du circuit va dépendre non seulement de l'état courant, mais aussi de l'état de l'entrée $E_{cr} \overline{D_{ef}}$. Par exemple, si l'état courant est 001 (1 registre rempli), alors l'état suivant sera :

- 010 si le signal $E_{cr} \overline{D_{ef}}$ est à 1 (on passe de 1 registre rempli à 2 registres remplis lors d'une écriture dans la file) ;
- 000 si le signal $E_{cr} \overline{D_{ef}}$ est à 0 (on passe de 1 registre rempli à une file vide lors d'un défilement).

Le diagramme d'état de notre circuit est donc :



Nous allons utiliser des bascules D pour implémenter notre compteur. Comme nous l'avons déjà vu, les entrées des bascules seront identiques aux états futurs. On peut donc écrire la table de transition suivante.

Q_2^n	Q_1^n	Q_0^n	$E_{cr} \overline{D_{ef}}$	Q_2^{n+1} D_2^n	Q_1^{n+1} D_1^n	Q_0^{n+1} D_0^n	S_1	S_0
0	0	0	0	Φ	Φ	Φ	Φ	Φ
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	0	Φ	Φ
0	0	1	1	0	1	0	0	1
0	1	0	0	0	0	1	Φ	Φ
0	1	0	1	0	1	1	1	0
0	1	1	0	0	1	0	Φ	Φ
0	1	1	1	1	0	0	1	1
1	0	0	0	0	1	1	Φ	Φ
1	0	0	1	Φ	Φ	Φ	Φ	Φ
1	0	1	0	Φ	Φ	Φ	Φ	Φ
1	0	1	1	Φ	Φ	Φ	Φ	Φ
1	1	0	0	Φ	Φ	Φ	Φ	Φ
1	1	0	1	Φ	Φ	Φ	Φ	Φ
1	1	1	0	Φ	Φ	Φ	Φ	Φ
1	1	1	1	Φ	Φ	Φ	Φ	Φ

On en déduit les tables de Karnaugh suivantes pour les bascules et les sorties :

Pour D_2^n :

Table de Karnaugh remplie					Regroupements				
$Q_0^n \quad E_{cr} \overline{D_{ef}}$					$Q_0^n \quad E_{cr} \overline{D_{ef}}$				
$Q_2^n Q_1^n$	00	01	11	10	$Q_2^n Q_1^n$	00	01	11	10
00	Φ				00	Φ			
01			1		01			1	
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10		Φ	Φ	Φ	10		Φ	Φ	Φ

$$D_2^n = Q_1^n \cdot Q_0^n \cdot E_{CR} \overline{D_{EF}}$$

Pour D_1^n :

Table de Karnaugh remplie					Regroupements				
$Q_0^n \ E_{CR} \overline{D_{EF}}$ $Q_2^n Q_1^n$	00	01	11	10	$Q_0^n \ E_{CR} \overline{D_{EF}}$ $Q_2^n Q_1^n$	00	01	11	10
00	Φ		1		00	Φ		1	
01		1		1	01		1		1
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10	1	Φ	Φ	Φ	10	1	Φ	Φ	Φ

$$D_1^n = Q_2^n + Q_1^n \cdot \overline{Q_0^n} \cdot E_{CR} \overline{D_{EF}} + Q_1^n \cdot Q_0^n \cdot \overline{E_{CR} \overline{D_{EF}}}$$

Pour D_0^n :

Table de Karnaugh remplie					Regroupements				
$Q_0^n \ E_{CR} \overline{D_{EF}}$ $Q_2^n Q_1^n$	00	01	11	10	$Q_0^n \ E_{CR} \overline{D_{EF}}$ $Q_2^n Q_1^n$	00	01	11	10
00	Φ	1			00	Φ	1		
01	1	1			01	1	1		
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10	1		Φ	Φ	10	1		Φ	Φ

$$D_0^n = \overline{Q_0^n} \cdot \overline{E_{CR} \overline{D_{EF}}} + \overline{Q_2^n} \cdot \overline{Q_0^n}$$

Pour S_1 :

Table de Karnaugh remplie					Regroupements				
$Q_0^n \ E_{CR} \overline{D_{EF}}$ $Q_2^n Q_1^n$	00	01	11	10	$Q_0^n \ E_{CR} \overline{D_{EF}}$ $Q_2^n Q_1^n$	00	01	11	10
00	Φ			Φ	00	Φ			Φ
01	Φ	1	1	Φ	01	Φ	1	1	Φ
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10	Φ	Φ	Φ	Φ	10	Φ	Φ	Φ	Φ

$$S_1 = Q_1^n$$

Pour S_0 :

Table de Karnaugh remplie					Regroupements				
Q_0^n E_{CR} $\overline{D_{EF}}$	00	01	11	10	Q_0^n E_{CR} $\overline{D_{EF}}$	00	01	11	10
$Q_2^n Q_1^n$					$Q_2^n Q_1^n$				
00	Φ		1	Φ	00	Φ		1	Φ
01	Φ		1	Φ	01	Φ		1	Φ
11	Φ	Φ	Φ	Φ	11	Φ	Φ	Φ	Φ
10	Φ	Φ	Φ	Φ	10	Φ	Φ	Φ	Φ

$$S_0 = Q_0^n$$

CHAPITRE 6

Les mémoires

PLAN

- 6.1 Introduction
- 6.2 Classification des mémoires
- 6.3 Les mémoires vives
- 6.4 Les mémoires mortes

OBJECTIFS

- Découvrir les différents types de mémoires à semi-conducteurs et leur rôle au sein d'un ordinateur
- Connaître leurs organisations physique et logique ainsi que leur structure afin de bien appréhender leurs caractéristiques et leur fonctionnement
- Comprendre la nécessité d'une hiérarchie de mémoire dans un ordinateur

6.1 INTRODUCTION

Dans un ordinateur la fonction mémorisation est organisée sous forme d'une **hiérarchie** de plusieurs niveaux : le niveau le plus rapide d'accès et de plus faible capacité est le plus proche du processeur. Le niveau le moins rapide d'accès (le plus lent) et de plus grande capacité est le plus éloigné du processeur. Le coût au bit décroît du niveau le plus rapide au niveau le plus lent. Cette hiérarchie a pour but, grâce à certaines propriétés intrinsèques des programmes (localités temporelles et spatiales), d'avoir un temps d'accès moyen à la hiérarchie qui se rapproche de celui du niveau le plus rapide (mémoire la plus rapide du marché à un moment donné) et d'un coût au bit qui se rapproche de celui du niveau le moins onéreux (mémoire la moins chère du marché au même moment). Cette hiérarchie peut être classée en deux grandes catégories :

- **Les mémoires de travail** : elles sont à base de semi-conducteurs. Elles sont actives lors l'exécution des programmes. Elles forment aussi deux classes :
 - **Les mémoires vives** : on y trouve en premier lieu les registres (cf. § 6.5) internes au processeur qui ont généralement la taille d'un mot, la mémoire cache, la mémoire centrale.
 - **Les mémoires mortes** : ce sont des mémoires qui contiennent un programme résident. Ces mémoires sont accédées au démarrage de l'ordinateur par le processeur pour exécuter le programme résident afin de charger par exemple dans la mémoire centrale le noyau du système d'exploitation à partir du disque dur.
- **Les mémoires de stockage** : ce sont des mémoires de masse. Elles servent à stocker de manière permanente de grandes quantités d'informations (programmes et données). À l'inverse des mémoires de travail, elles ne permettent pas l'exécution directe des programmes. En effet les programmes, pour être exécutés, doivent être chargés au préalable dans la mémoire centrale qui constitue la mémoire de travail. Elles sont formées par des disques magnétiques, (disques durs) optiques (CD, DVD) et les bandes magnétiques.

Dans ce chapitre nous allons nous intéresser exclusivement à la **mémoire de travail**. Nous n'allons pas nous intéresser ni à la gestion de la mémoire cache ni à celle de la mémoire centrale. L'objectif ici est d'étudier la composition, la structure et l'organisation interne des différents types de composants ou circuits mémoires. Il s'agit aussi d'étudier les opérations de lecture et d'écriture et les chronogrammes qui y sont associés, ainsi que la manière d'associer des composants mémoires pour réaliser des mémoires de grandes capacités.

6.2 CLASSIFICATION DES MÉMOIRES

La figure 6.1 montre une classification des différents types de mémoires à semi-conducteurs.

Cette classification distingue tout d'abord deux grandes familles : les mémoires vives et les mémoires mortes.

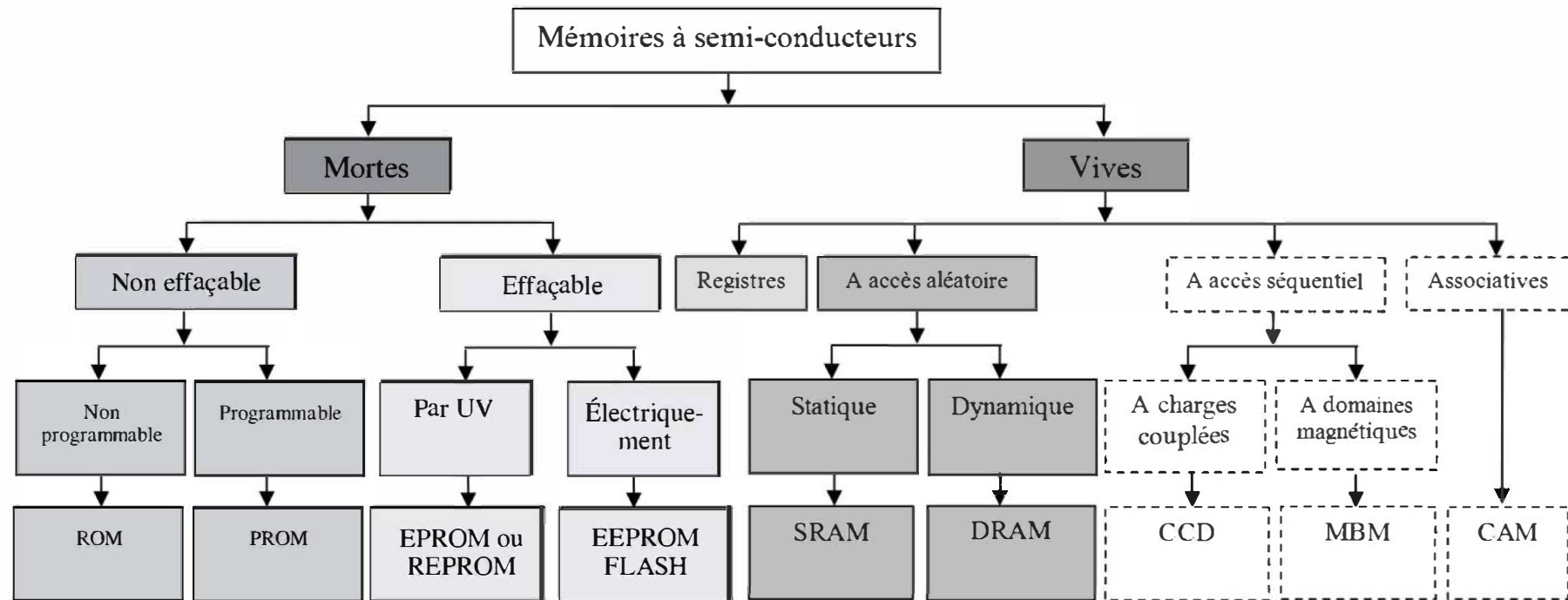


Figure 6.1 Classification des mémoires à semi-conducteurs.

6.3 LES MÉMOIRES VIVES

Ce sont les mémoires sur lesquelles **les opérations de lectures ou d'écritures sont possibles**. On distingue dans cette classe trois sous-classes : les mémoires à **accès aléatoires**, les mémoires à **accès séquentiels** et les mémoires **associatives**. Les deux dernières sous-classes sont données à titre d'information ; nous les présenterons succinctement. Toute notre attention sera portée sur les mémoires à accès aléatoires.

- Les mémoires à **accès séquentiels** ont pour particularité d'avoir un temps d'accès à l'information qui dépend de la position de l'information dans la mémoire. Pour accéder par exemple à une information à une position α on doit passer par les positions intermédiaires de 0 à $\alpha - 1$. Dans cette catégorie on retrouve :
 - Les **mémoires à charges couplées** ou à transfert de charges : les mémoires CCD dans les caméras et les appareils photos.
 - Les **mémoires à domaines magnétiques** : les mémoires à bulles magnétiques (MBM) utilisés parfois dans le domaine spatial.
- Les **mémoires associatives** sont accessibles par le contenu : CAM (*Content Addressable Memory*). Elles sont utilisées comme cache de pages dans les mémoires virtuelles.
- Les **mémoires à accès aléatoires**, dites **RAM** (*Random Access Memory*), sont telles que le temps d'accès à l'information est indépendant de la place (c'est-à-dire la position) de l'information dans la mémoire. Les mémoires RAM sont **volatiles**. On trouve deux types de mémoires RAM : les RAM **statiques** et les RAM **dynamiques**. Elles sont utilisées en général respectivement en tant que mémoire cache et mémoire centrale.

RAM Statique

Les **mémoires statiques SRAM** (*Static RAM*) sont construites à partir de bascules (cf. § 5.5). En effet, comme nous l'avons vu, la **bascule** est un élément de mémoire unitaire (la mémorisation de l'état d'un bit nécessite de 4 à 6 transistors). Nous avons vu que l'association en parallèles de bascules nous permet de construire un registre. L'association de plusieurs registres parallèles identiques nous permet de concevoir une SRAM. Ces mémoires sont très rapides : elles ont un **temps d'accès** de 5 à 10 nanosecondes. Mais elles ont un coût et une consommation au bit plus importants que la mémoire dynamique. Elles sont de faibles capa-

cités du fait qu'elles ont une **densité d'intégration** au bit plus faible que la RAM dynamique. Elles sont réservées pour la mise en œuvre des mémoires caches. L'information dans la mémoire peut être conservée aussi longtemps que l'alimentation est maintenue.

Structure d'une mémoire SRAM

Considérons l'organisation d'une SRAM d'une capacité de 512 Kilo-octets = 2^{19} octets = 512 Ko. La structure de cette mémoire SRAM est donnée sur la figure 6.2 où nous avons mis en parallèles 524 288 registres de 8 bits chacun. Les mots mémoire sont donc de taille 8 bits. Pour accéder à un registre ou un mot de la mémoire on doit le sélectionner. Il est impensable d'imaginer 524 288 broches sur le circuit mémoire pour sélectionner individuellement chaque mot ou registre.

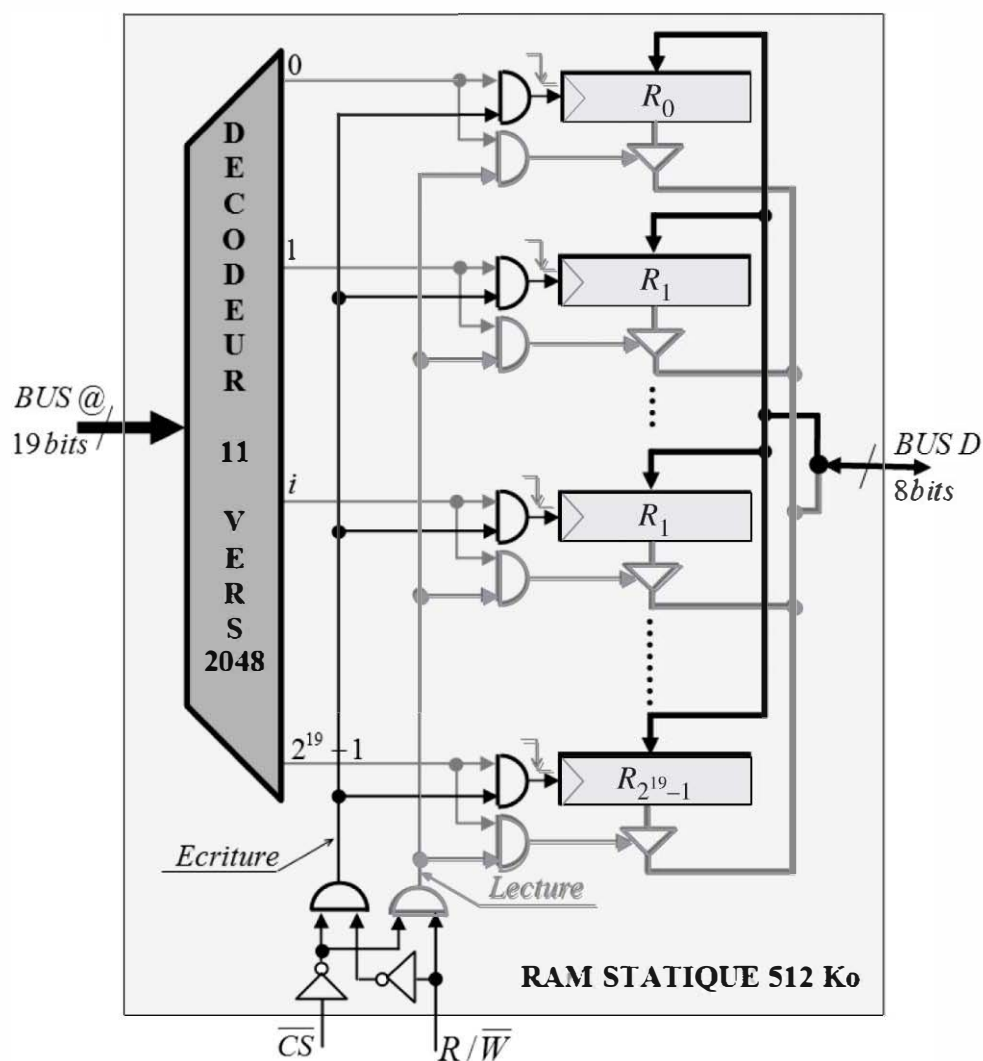


Figure 6.2 Structure logique d'une SRAM.

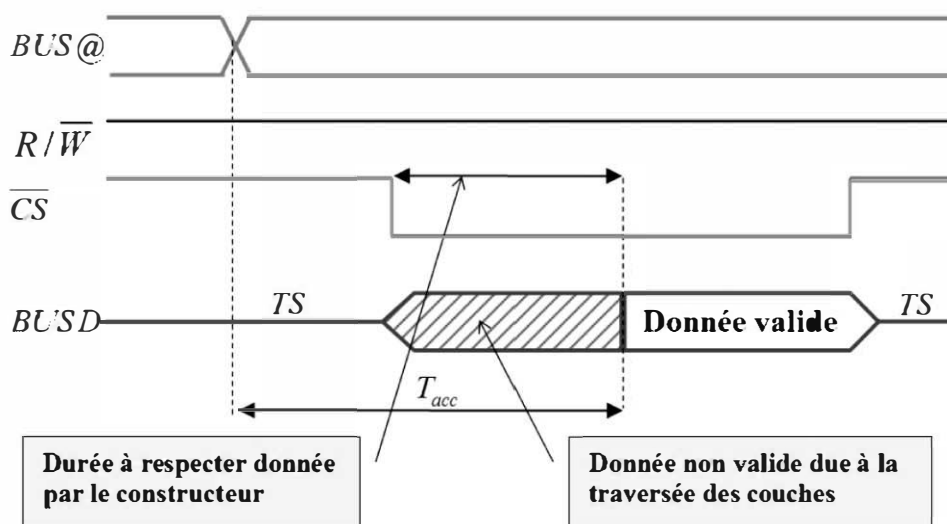
Afin de limiter le nombre de signaux de commande des restrictions ont été faites :

- Un **seul registre** (mot) est accessible à la fois. Les signaux de sélection du registre concerné peuvent ainsi être encodés mais il faut rajouter une logique de décodage de ces signaux encodés. Ils forment le bus d'« adresse » : le bus sur lequel on doit fournir le numéro du mot ou du registre, c'est-à-dire son **adresse**.
- Le registre a **deux bus** (entrée et sortie (cf. § 6.5)). Afin d'éviter d'avoir sur le boîtier du circuit mémoire 2×8 broches pour les deux bus (entrée et sortie), l'opération sur un registre est **soit** une lecture **soit** une écriture : un signal unique de Lecture/Écriture (*Read/Write*) $R/\overline{W}$ garantit l'exclusion mutuelle de ces deux opérations. Il s'ensuit que les deux bus (entrée et sortie) du registre peuvent être reliés pour constituer un bus unique bidirectionnel dont le sens est fourni par ce signal $R/\overline{W}$.

Un signal général $\overline{CS}$ (*Chip Select*) de validation (ou sélection) du boîtier (ou du circuit) est ajouté. Il va permettre la mise en parallèle d'ensembles logiques de ce type, qui constituent des circuits de mémoire vive de type SRAM. On remarquera que ce signal $\overline{CS}$ agit sur le signal $R/\overline{W}$ et non sur une entrée de validation du décodeur.

Les opérations possibles sur ce circuit mémoire sont la lecture ou l'écriture d'une donnée.

Lecture



$$TS = Tri - State \text{ (trois états)}$$

Figure 6.3 Chronogramme d'un cycle de lecture.

On remarquera que le temps d'accès est calculé depuis la stabilisation de l'adresse. Le temps d'accès peut se décomposer en :

- Un temps de traversée de décodeur. C'est en fait le temps le plus long.
- Un temps de traversée des portes logiques pour les signaux de commande, dès l'activation du signal $\overline{CS}$.
- Un temps de traversée de la barrière trois états pour la donnée.

Si l'arrivée de $\overline{CS}$ se fait de telle façon que le signal d'activation associé arrive au plus tard en même temps que la sortie adressée du décodeur, on obtient le temps d'accès minimum.

La lecture est combinatoire : il suffit de présenter l'adresse du mot à lire sur le bus d'adresse, indiquer l'opération de lecture et sélectionner le circuit. Au bout d'un certain temps (le temps d'accès, fixé par le constructeur du circuit) la donnée est disponible et le reste jusqu'à ce que le circuit soit désélectionné ou que l'adresse change.

Écriture

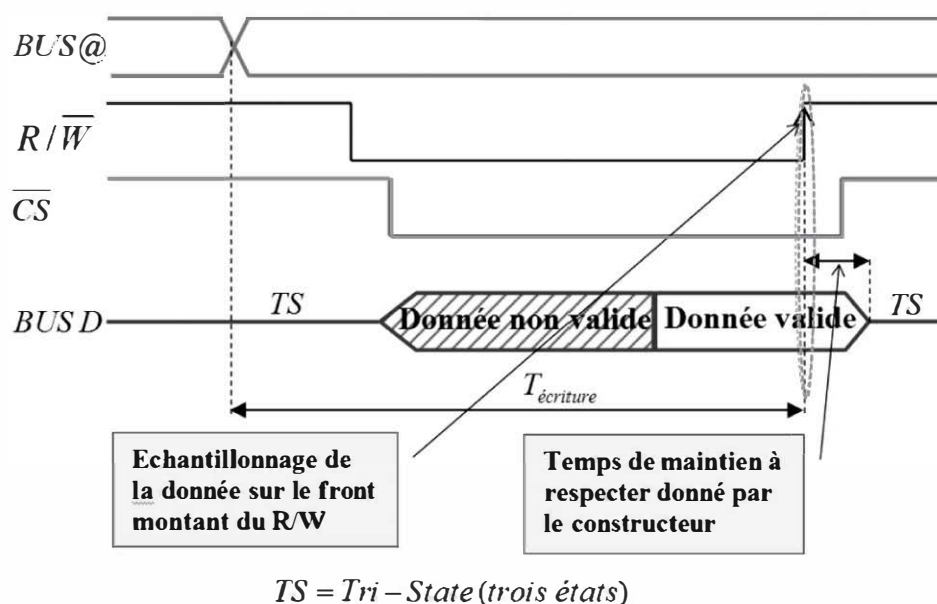


Figure 6.4 Chronogramme d'un cycle d'écriture.

Il est préférable que le signal $R/\overline{W}$ soit rendu actif avant $\overline{CS}$: le changement de sens du bus de données se fait avec la sortie de l'état « flottant ». Pour que l'écriture se fasse correctement il faut que :

- l'adresse ait traversé le décodeur, ce qui représente toujours le temps le plus long ;
- le signal de chargement du registre adressé soit « armé » (présence de la sortie du décodeur, de $R/\overline{W}$ et $\overline{CS}$) ;
- la donnée soit stable sur le bus d'entrée du registre : c'est elle qui a le moins de couches logiques à traverser.

Dès que ces conditions sont remplies, la première transition active de $R/\overline{W}$ ou $\overline{CS}$ (qui conditionne $R/\overline{W}$) va provoquer le chargement de la donnée dans le registre adressé et terminer ainsi le cycle d'écriture.

L'écriture est séquentielle : il faut donner l'adresse, indiquer l'opération d'écriture (ce qui change le sens du bus D), sélectionner le circuit, fournir la donnée, attendre que tout soit stabilisé (temps fixé par le constructeur) puis terminer en désactivant le signal de lecture/écriture $R/\overline{W}$ ou bien celui de sélection $\overline{CS}$ qui produit le même effet, c'est-à-dire l'écriture de la donnée dans le registre concerné.



Le temps de cycle et temps d'accès sont identiques dans une SRAM.

La représentation symbolique d'une mémoire SRAM est montrée sur la figure 6.5.

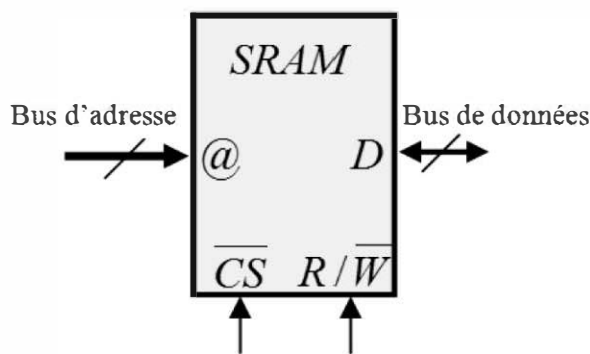


Figure 6.5 Représentation symbolique d'une mémoire SRAM.

Association de circuits de mémoire SRAM

Le but est, par exemple, de réaliser une mémoire SRAM de 2 Méga-octets = 2^{21} octets = 2 Mo à partir de circuits de 512 Ko. Pour ce faire il faut quatre circuits de 512 Ko.

Pour adresser 512 Ko il faut 11 bits (ou fils pour le bus d'adresse), et pour 2 Mo on a besoin de 21 bits (ou 21 fils pour le bus d'adresse). Si on note les bits d'adresse $A_{20}A_{19} \dots A_1A_0$, les deux bits A_{20} et A_{19} indiqueront en fait le numéro du boîtier dans lequel se trouve le mot recherché et dont l'adresse est donnée par les bits restants $A_{18} \dots A_1A_0$ (tableau 6.1).

La figure 6.6 montre l'association physique de ces circuits mémoires. Les deux bits qui codent le numéro du circuit mémoire sont connectés à l'entrée d'un décodeur de 2 vers 4, dont la fonction logique permet de commander le circuit mémoire sélectionné sur son entrée $\overline{CS}$. Les trois autres circuits mémoire ne réagissent pas, restant inactifs. Si l'entrée

Tableau 6.1 RÉPARTITION DES ADRESSES POUR ASSOCIER DES BOÎTIERS SRAM DE 512K
POUR CONSTITUER UN ESPACE SRAM DE 2M.

N° du boîtier		Adresse du mot (octet) mémoire dans le boîtier										Bus adresse
A ₂₀	A ₁₉	A ₁₈	A ₁₇	A ₁₆		A ₅	A ₄	A ₃	A ₂	A ₁	A ₀	
0	0	0	0	0	0.....0	0	0	0	0	0	0	Adresses de 0 à 512K-1
0	0	0	0	0	0.....0	0	0	0	0	0	1	
.	.	.	.	.	.	.	.	.	.	.	.	
0	0	1	1	1	1.....1	1	1	1	1	1	1	
0	1	0	0	0	0.....0	0	0	0	0	0	0	Adresses de 512K A 1M-1
0	1	0	0	0	0.....0	0	0	0	0	0	1	
.	.	.	.	.	.	.	.	.	.	.	.	
0	1	1	1	1	1.....1	1	1	1	1	1	1	
1	0	0	0	0	0.....0	0	0	0	0	0	0	Adresses de 1M A 1,5M-1
.	.	.	.	.	.	.	.	.	.	.	.	
1	0	1	1	1	1.....1	1	1	1	1	1	1	
.	.	.	.	.	.	.	.	.	.	.	.	
1	1	0	0	0	0.....0	0	0	0	0	0	0	Adresses de 1,5M A 2M-1
.	.	.	.	.	.	.	.	.	.	.	.	
1	1	1	1	1	1.....1	1	1	1	1	1	1	
.	.	.	.	.	.	.	.	.	.	.	.	

$\overline{E}$ (Enable) de validation du décodeur n'est pas active, alors aucun circuit mémoire n'est sélectionné, et l'ensemble apparaît comme un unique circuit avec le bus D fonctionnant en logique trois états (*tri-state*). En effet, l'utilisation de l'entrée de validation du décodeur $\overline{E}$ permet d'introduire un « $\overline{CS}$ » externe ; on retrouve ainsi la même structure que le celui du circuit initial de 512 Ko. Il est donc possible de construire des ensembles plus grands, soit récursivement, soit en prenant un décodeur à plus d'entrées.



Lorsque le nombre de circuits associés est important, on doit interposer des « amplificateurs de courant » (*buffers*) sur les signaux qui ont plus d'une entrée à commander pour retrouver une « charge électrique d'entrée » identique. Mais dans ce cas, le temps d'accès est augmenté du temps de traversée de ces circuits. La fonction d'amplification de courant peut être assurée par des barrières trois états. La figure 6.6 montre, lorsque cela est nécessaire, où il faut placer ces barrières pour amplifier certains signaux et bus. Les barrières sont montées de telle sorte qu'elles soient toujours passantes.



Ce mécanisme d'association de circuits permet de réaliser des mémoires de taille quelconque. Si les circuits à associer n'ont pas la même capacité, l'association est un peu plus difficile et nécessite l'utilisation de mémoires mortes

de transcodages ou de réseaux logiques programmables (RLP). Les RLP sont des circuits contenant des matrices de ET et de OU permettant de câbler par programmation juste les mintermes nécessaires au codage des circuits en fonction de leur capacité. Il est ainsi possible d'avoir une vision logique de la mémoire considérée comme un espace linéaire de mots.

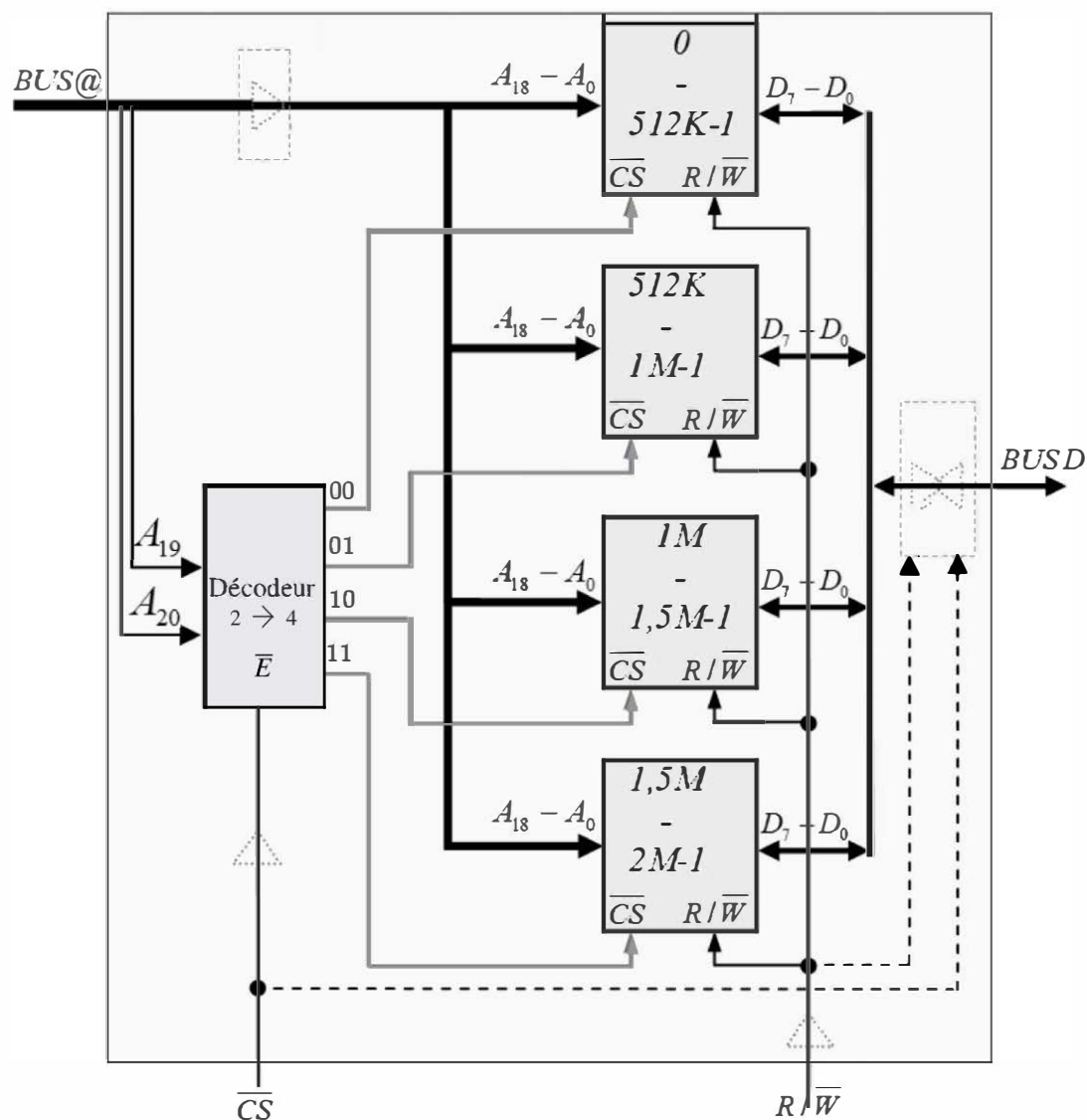


Figure 6.6 Schéma logique d'une SRAM de 2 Mo obtenue par association de circuits de 512 Ko.

RAM dynamique

Les mémoires dynamiques DRAM (*Dynamic RAM*) ont l'élément de mémorisation élémentaire (mémorisation de l'état d'un bit) fondé sur la base d'un effet capacitif d'un seul transistor unipolaire MOS (Métal Oxyde Silicium) (figure 6.7). En effet le bit mémoire est lié à l'état de la charge du condensateur (représentant l'effet capacitif) associé à ce trans-

istor (le bit est à 1 lorsque le condensateur est chargé, il est à 0 lorsque le condensateur est déchargé). La DRAM présente une densité d'intégration au bit 4 à 6 fois plus importante qu'une SRAM ayant le même nombre de transistors. Le coût et la consommation au bit des DRAM sont faibles par rapport à la SRAM. Elle constitue la base des mémoires centrales. Les DRAM de base ont un temps d'accès qui se situe entre 50 et 60 nanosecondes.

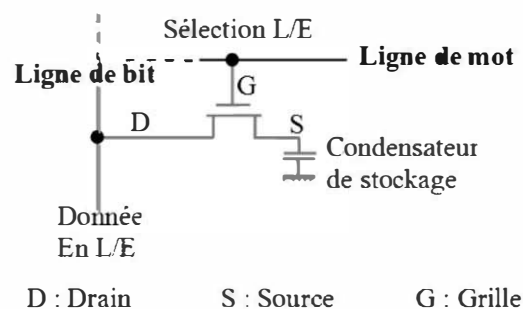


Figure 6.7 Élément de mémorisation unitaire d'une mémoire DRAM.

Lorsque la tension grille V_G est au niveau logique haut, le transistor est saturé : un courant circule entre la source et le drain. Le transistor se comporte comme un circuit fermé.

Lorsque la tension de la grille V_G est au niveau logique bas, le transistor est bloqué : aucun courant ne circule entre la source et le drain. Le transistor se comporte comme un circuit ouvert.

En conclusion le transistor est similaire à un interrupteur piloté par V_G . Pour **lire** l'état d'un bit on **charge** (on active) la ligne de mot et on lit la valeur de la charge du condensateur sur la ligne bit. Pour **écrire** on **charge** toujours la ligne de mots et on applique le niveau logique 1 ou 0 sur la ligne bit pour écrire 1 ou 0 (charge ou décharge du condensateur). L'inconvénient majeur de la DRAM est la décharge du condensateur engendrée par les courants de fuite en quelques millisecondes. La DRAM est donc volatile même sous alimentation, et nécessite donc un **rafraîchissement** régulier. L'opération de rafraîchissement consiste à effectuer des opérations de lectures de mots pour recharger les condensateurs toutes les quelques dizaines de millisecondes (paramètre fixé par le constructeur).

Structure d'une mémoire DRAM

La mémoire DRAM est organisée sous forme d'une matrice **carrée** de lignes et de colonnes. Les lignes sont dites **lignes de mot** et les colonnes sont dites **lignes de bits**. À l'intersection d'une ligne mot et d'une ligne de bit il existe un transistor MOS (figure 6.7).

Pour minimiser le nombre de broches sur un circuit DRAM, le bus d'adresse est multiplexé : l'adresse est fournie en deux temps (d'abord l'adresse ligne, puis l'adresse colonne) pour accéder sur un bit. La figure 6.8 montre la structure d'une DRAM de 16 Mbits organisée en 4 096 lignes de mots $\times$ 4 096 lignes de bits.

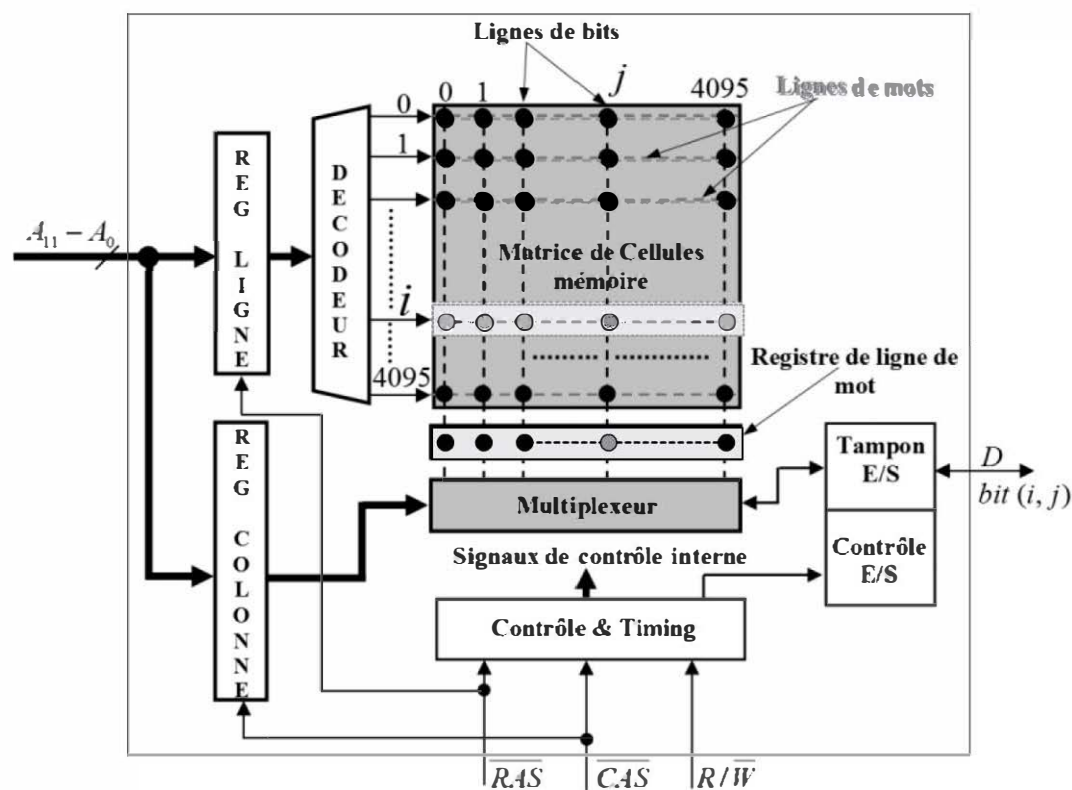


Figure 6.8 Structure logique d'une DRAM.

Lecture

Une opération de lecture se décompose en plusieurs étapes :

- **Étape 1** : mise de l'adresse ligne sur le bus d'adresse puis mémorisation de cette adresse dans le registre ligne (REG LIGNE) en activant le signal sélection de ligne $\overline{RAS}$ (Row Address Strobe). Cette étape décharge les condensateurs de la ligne de mots c'est-à-dire lecture de tous les bits de la ligne mots (dans notre exemple 4 096 bits) et chargement dans le registre ligne de mot (ce registre joue le rôle d'amplificateur). Depuis l'activation du signal $\overline{RAS}$ jusqu'au chargement du registre de ligne, un temps d'environ 40 ns est nécessaire c'est le **temps de décharge**.
- **Étape 2** : mise sur le bus de l'adresse colonne puis mémorisation de cette dernière dans le registre (REG COLONNE) en activant le

signal de sélection de colonne $\overline{CAS}$ (*Column Address Strobe*). Cette étape sélectionne le bit à lire à partir du registre ligne. Elle nécessite depuis l'activation du signal $\overline{CAS}$ environ 10 ns.

- **Étape 3** : réécriture du registre de ligne dans la ligne de mot lue (opération réalisée de manière interne par la mémoire). En effet, la lecture décharge une ligne de mot (elle est destructive) donc une réécriture du registre de ligne dans la ligne de mot lue est nécessaire.



Cette réécriture fait que le temps d'accès est de 50 ns alors que le temps de cycle est de 90 ns environ. Le temps de cycle dans une DRAM est donc presque le double du temps d'accès !

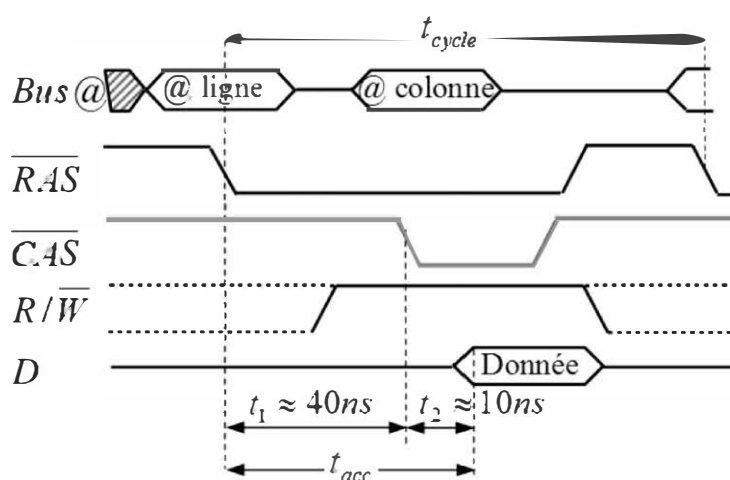


Figure 6.9 Chronogramme d'un cycle de lecture d'une DRAM.

Écriture

Une opération d'écriture se décompose en deux étapes :

- **Étape 1** : elle est identique à l'étape 1 de la lecture. Elle consiste donc à charger le registre de ligne.
- **Étape 2** : elle est aussi identique à l'étape 2 de la lecture sauf qu'on sélectionne un bit dans le registre ligne en vue de le modifier, puis on réécrit le registre de ligne dans la ligne de mot déchargée.

La figure 6.11 montre la représentation symbolique d'une DRAM.

Pour améliorer en particulier la performance des DRAM, plusieurs techniques ont permis le développement de versions dérivées de la DRAM classique. Actuellement, les versions les plus importantes au niveau performance sont :

- **La DRAM synchrone** : SDRAM (*Synchronous DRAM*), permettant des échanges synchronisés avec le processeur évitant les cycles d'at-

Association de circuits mémoire DRAM

En général la mémoire DRAM se présente sous formes de plusieurs circuits regroupés sur des barrettes : **SIMM** (*Single Inline Memory Module*) ou **DIMM** (*Double Inline Memory Module*) qui offrent deux fois plus de contacts de connexion que la SIMM. Ces modules sont vendus actuellement selon plusieurs formats de 1 Go, 2 Go et 4 Go. Une barrette spécifique existe pour la mémoire Rambus c'est **RIMM** (*Rambus Inline Memory Module*). La barrette constitue un banc de mémoire associant en parallèle plusieurs circuits DRAM à l'instar de l'association des circuits SRAM. La différence est que cette association (barrette ou banc) est vue par le microprocesseur à travers un contrôleur de DRAM qui se charge de générer tous les signaux de service nécessaires pour une lecture ou une écriture. Ce contrôleur comporte aussi une logique permettant d'assurer le rafraîchissement des circuits DRAM. Si un accès est fait par le processeur en plein milieu d'un cycle de rafraîchissement, cet accès est retardé et ce retard est signalé au processeur par un signal d'attente dit WAIT.

Le rafraîchissement est en fait une lecture de toutes les lignes des matrices de tous les circuits dans un temps fixé par le constructeur : de quelques millisecondes à quelques dizaines de millisecondes. Ces lectures sont réparties sur la durée du cycle, soit environ une lecture ligne (quelques dizaines à une centaine de nanosecondes) toutes les quelques dizaines de microsecondes.

6.4 LES MÉMOIRES MORTES

À l'inverse des mémoires vives qui sont accessibles en lecture et en écriture (mais qui sont volatiles), les **mémoires mortes** (en dehors de leur programmation) ne sont accessibles qu'en lecture seulement, et ne sont pas volatiles même en l'absence d'alimentation. Elles peuvent, par exemple dans certaines applications de contrôle d'un processus industriel ou d'un système embarqué, contenir des programmes et données stockés de manière permanente. Dans les ordinateurs, elles contiennent le programme de boot permettant de charger au démarrage le noyau du système d'exploitation dans la mémoire centrale. Le nom générique des différents types de mémoire morte est : **ROM** (*Read Only Memory*). Elles sont aussi des RAM (!) au sens où le temps d'accès est indépendant de la position (de l'adresse) du mot. On distingue deux types de ROM : les **non effaçables** et les **effaçables**.

Les mémoires mortes non effaçables

Non programmable : ROM

Elles se présentent sous une forme de matrice de transistors comme dans les DRAM. Chaque transistor se trouve à l'intersection d'une ligne de mot et d'une ligne de bits. Elles sont programmées à la fabrication : le codage des informations est assuré par le fabricant, d'après les instructions de l'utilisateur, au moment de la dernière opération de fabrication. En effet, à l'aide du masque de métallisation, le fabricant écrit des « 1 » ou des « 0 » en raccordant ou non les transistors de chaque ligne de mot à la ligne de bits (figure 6.12). Le prix de revient de ce type de mémoire est élevé ; elle est réservée aux grandes séries.

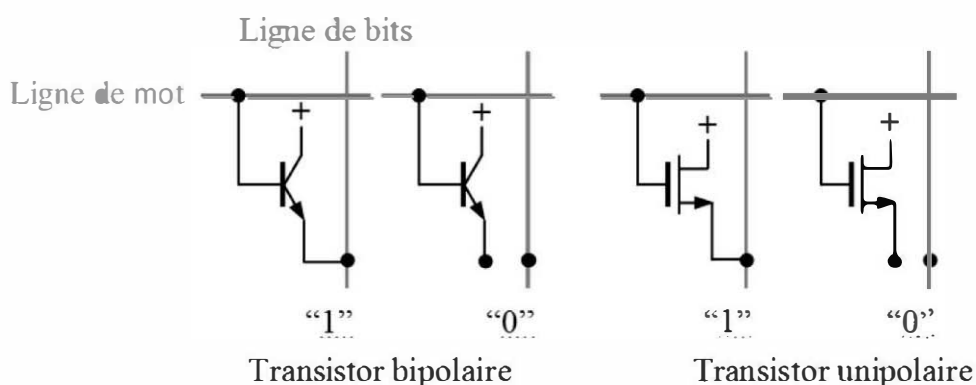


Figure 6.12 Structure des bits d'une mémoire ROM.

Programmable : PROM

La PROM (*Programmable ROM*) est programmable une seule fois par l'utilisateur. Ensuite elle est seulement lue.

Elle est aussi similaire dans son organisation interne à la ROM, sauf que chaque transistor couplant une ligne de mot et une ligne de bit est mis en série avec un fusible (figure 6.13). La présence d'un fusible représente un « 1 » et son absence un « 0 ». Une PROM vierge a tous ses fusibles intacts. La programmation nécessite un équipement spécifique et elle consiste à « claquer » (détruire) les fusibles là où les bits doivent être à 0.

Les transistors peuvent être remplacés par des diodes. On obtient le même principe de fonctionnement.

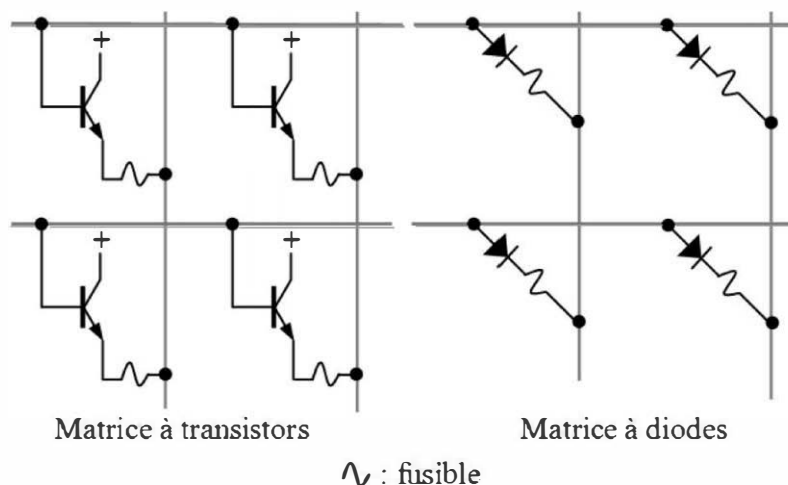


Figure 6.13 Structure des matrices de PROM à transistors et à diodes.

Les mémoires mortes effaçables

Effaçable par rayonnement ultraviolet

EPROM ou REEPROM

L'EPROM (*Erasable PROM* = PROM effaçable) est dite aussi REEPROM (*PROM reprogrammable*). Le bit mémoire est fondé sur un transistor MOS particulier. La figure 6.14, montre un transistor FAMOS (*Floating gate Avalanche injection MOS*) qui est à la base des mémoires effaçables. La procédure de programmation est réalisée par un phénomène d'avalanche, crée volontairement et momentanément entre le drain et le substrat. Ce phénomène d'avalanche permet d'obtenir des électrons d'énergie suffisante pour qu'ils traversent une faible couche de silice (10×10^{-9} mètre = 10 nm). Après être arrivés sur la grille flottante, les électrons, ayant perdu leur énergie cinétique y restent piégés. La grille flottante chargée rend bloqué le transistor préalablement passant en supprimant son canal *N*. Une grille de sélection du bit est superposée à la grille flottante et sa tension V_G agit ou n'agit pas selon que le bit a été effacé « 1 » (la grille flottante est déchargée) ou programmé « 0 » (la grille flottante est chargée).

Pour reprogrammer la mémoire il faut au préalable supprimer les données présentes. Pour ce faire, on expose la mémoire à une source de rayons ultra-violets (UV) (c'est pour cela qu'en général le boîtier mémoire est muni d'une fenêtre en quartz transparente aux UV). L'opération d'effacement dure environ 30 minutes, alors que l'écriture dure quelques secondes. Lorsque la mémoire est effacée (vierge), tous les bits sont à un c'est-à-dire tous les transistors deviennent passants. La programmation s'effectue aussi sur un équipement spécifique.

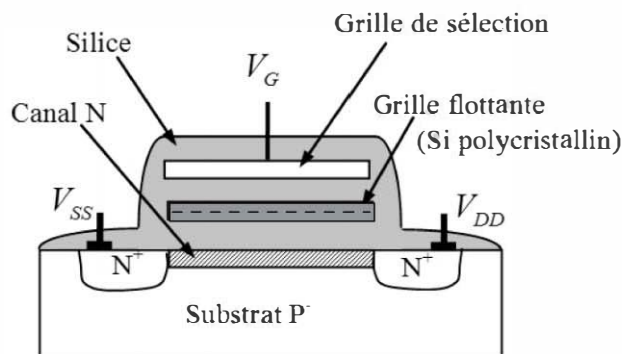


Figure 6.14 Transistor FAMOS servant d'un bit mémorisation de type EPROM/REPROGRAMMABLE.

Effaçable électriquement

EEPROM

L'**EEPROM** (*Electrically Erasable PROM*), elle est similaire à l'EPROM ou REPROGRAMMABLE, à ceci près qu'elle est effaçable électriquement. Le procédé d'effacement est plus simple (pas besoin de rayons UV) et surtout il est très rapide (quelques millisecondes). De plus on n'a pas besoin de retirer la mémoire de son support : elle peut être effacée et écrite sur son site. Un autre avantage est que l'effacement et l'écriture peuvent affecter seulement une zone sans modifier le reste de la mémoire.

FLASH

C'est une mémoire de type EEPROM programmable électriquement par bloc (comme pour les secteurs des disques durs) mais de manière assez rapide (Flash). On les retrouve dans les appareils photos, téléphones mobiles, disque flash (clé USB), etc.

Association des circuits mémoires ROM

L'association en parallèle de plusieurs boîtiers mémoires (ROM, PROM, EPROM, EEPROM) reste identique au cas des SRAM. La seule différence par rapport à la SRAM c'est qu'il n'y a plus de signal de lecture/écriture.



L'unité centrale

PLAN

- 7.1 Organisation d'un ordinateur
- 7.2 Architecture générale d'une unité centrale
- 7.3 Introduction à la programmation en assembleur de l'ARM
- 7.4 Instructions de traitement des données
- 7.5 Instructions de référence mémoire
- 7.6 Instructions de contrôle de programme
- 7.7 Syntaxe d'un fichier source en langage d'assemblage

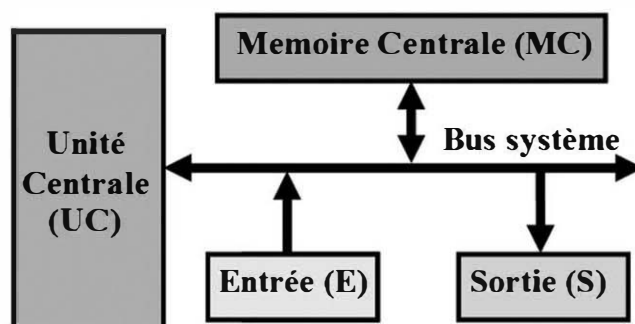
OBJECTIFS

- Appréhender le rôle et le fonctionnement de l'**unité centrale**
- Comprendre les mécanismes de programmation en **langage d'assemblage** en prenant comme support un processeur ARM

7.1 ORGANISATION D'UN ORDINATEUR

Avant de présenter l'architecture de l'unité centrale de traitement, nous allons donner très brièvement les quatre composantes classiques d'un ordinateur que nous pouvons qualifier aussi d'unités fonctionnelles (figure 7.1).

- **L'unité centrale (UC)** : c'est l'unité fonctionnelle qui est responsable de l'exécution des programmes du système d'exploitation et des programmes utilisateurs. Nous allons présenter rapidement dans un premier temps l'architecture générale d'une unité centrale hypothétique. Dans un deuxième nous ferons une introduction à l'architecture et la programmation en **langage d'assemblage** du processeur



Bus système = Bus (Adresse, Données, Contrôle)

Figure 7.1 Les composantes classiques d'un ordinateur.

ARM. L'accent sera mis sur les ressources accessibles par le programmeur à travers les instructions dans le mode utilisateur.

- **La mémoire centrale (MC)** : les programmes et les données du système d'exploitation et de l'utilisateur sont chargés au préalable à partir du disque dans la MC en vue de leur exécution. Les problèmes liés à la gestion de la MC où aux transferts entre MC et disque ne seront pas traités.
- **Les unités d'Entrées/Sorties (E/S)** : ils permettent à l'UC de communiquer avec le monde extérieur. La « communication » entre l'utilisateur et l'ordinateur se fait par l'intermédiaire de **périphériques**.
 - **Entrée (E)** : clavier, souris, etc.
 - **Sortie (S)** : écran, imprimante, afficheurs, etc.
 - **Entrées/Sorties (E/S)** : disque dur, écran tactile, etc.

Ces périphériques communiquent avec le processeur à travers des **circuits d'interface** : contrôleur d'E/S parallèle et/ou série, contrôleur de disque dur, contrôleur d'écran (ou processeur graphique), etc. (figure 7.2).

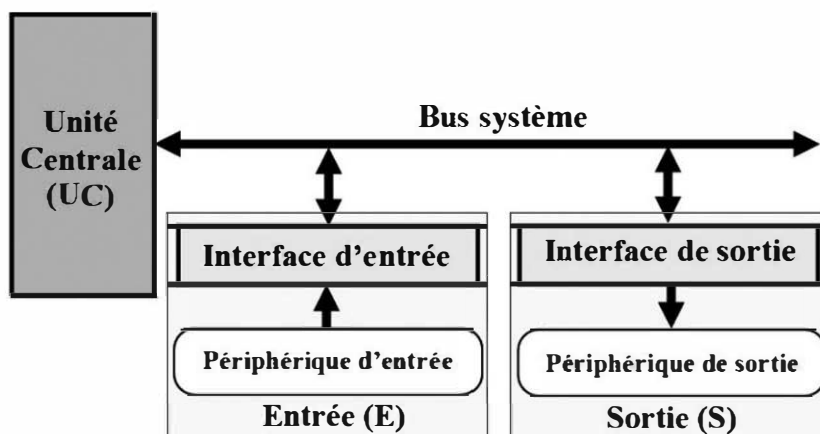


Figure 7.2 Composantes des unités d'entrée et de sortie.

Ces circuits d'interfaces sont généralement programmables. Ils permettent entre autres d'adapter le format des données entre processeur et périphériques (par exemple parallèle vers série et vice versa), de recevoir et de fournir les signaux de et vers les périphériques. Les circuits d'interfaces possèdent au moins les registres internes suivants : un ou plusieurs **registres de données** servant à stocker les données venant du processeur en vue de les envoyer vers le périphérique et inversement, un **registre de commande** permettant de paramétrer le mode fonctionnement du circuit, et un **registre d'état** permettant de connaître l'état du périphérique (disponible ou non). Grâce à ces derniers, on peut savoir par exemple si un registre de données est vide ou non avant d'écrire une nouvelle donnée, ou si un registre de données est plein ou non avant de lire. Les programmes permettant de gérer ces circuits d'interfaces sont dits : **pilotes de périphériques**.

7.2 ARCHITECTURE GÉNÉRALE D'UNE UNITÉ CENTRALE

La plupart des ordinateurs actuels suivent les principes énoncés par **Von Neumann** en 1946. La mémoire de l'ordinateur contient à la fois les données et le programme qui va modifier ces données.

La mémoire étant un organe passif, les données doivent au préalable en être extraites et amenées dans le « **bloc chemin de données** » plus précisément dans le « **bloc de registres** » où elles vont être présentées plus tard au « **bloc de calcul** » pour subir la modification demandée avant d'être ramenées dans le bloc de registres, afin de libérer le bloc de calcul pour d'autres opérations sur d'autres données.

Les opérations effectuées sur les données se font sous le contrôle du programme. Un programme est une suite d'ordres élémentaires, les **instructions**, exécutées en séquence sauf indication contraire.

Comme les données, les instructions doivent être extraites de la mémoire et amenées dans le « **bloc chemin de contrôle** », et leur exécution consiste à mettre en relation les circuits adéquats pour réaliser la fonction demandée.

Le bloc chemin de contrôle et le bloc chemin de données constituent « **l'unité centrale** » de l'ordinateur. Le premier en est le cerveau, le second le cœur (figure 7.3).

L'unité centrale utilise trois bus pour communiquer avec les autres composantes de l'ordinateur : un bus d'adresse unidirectionnel, un bus de données bidirectionnel et un « bus » de contrôle en fait il n'est véritablement pas un vrai bus. En effet, c'est la « glue » des signaux autres que

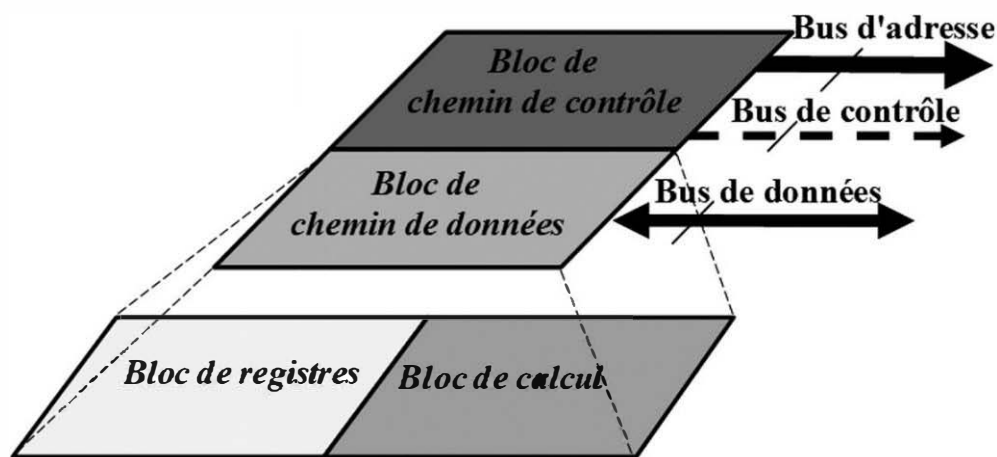


Figure 7.3 Blocs fonctionnels de l'unité centrale.

ceux des bus d'adresse et données. Ils n'ont ni la même fonction, ni la même direction, ni la même polarité.

Toute la logique qui compose l'unité centrale est intégrable, depuis 1972, dans un seul circuit : **le microprocesseur ou processeur**.

Comme nous l'avons vu au chapitre précédent, toute mémoire peut être vue comme un composant logique avec un bus adresse, un bus données, et un bus contrôle (le signal *Lecture/Ecriture*). La façon de relier le microprocesseur à la mémoire est d'utiliser ces mêmes bus, comme indiqué à la figure 7.4.

Avec l'évolution de la technologie, le bus adresse est passé de 16 à 32 et même 40 bits et le bus de données de 8 à 32 et même 64 bits. Les fonctions réalisées par le bloc de calcul se sont un peu diversifiées, mais sont surtout exécutées plus rapidement. Cependant sur le principe rien n'a été changé.

Le bloc chemin de contrôle

Les instructions sont de nature très simple et il faut en exécuter plusieurs pour arriver à réaliser une opération, elle aussi élémentaire, sur la ou les données concernées (par exemple $C := A + B$ où A , B et C représentent des variables rangées en mémoire). C'est le grand nombre de ces petites opérations réalisées dans un temps très court qui donne l'impression d'un travail colossal.

Les instructions sont rangées en mémoire et donc repérées par leur position c'est-à-dire l'adresse du mot mémoire qui la contient. Comme ces instructions s'enchaînent en séquence, leur adresse est suivie en permanence par le « **bloc de contrôle** », à l'aide d'un registre dit **compteur**

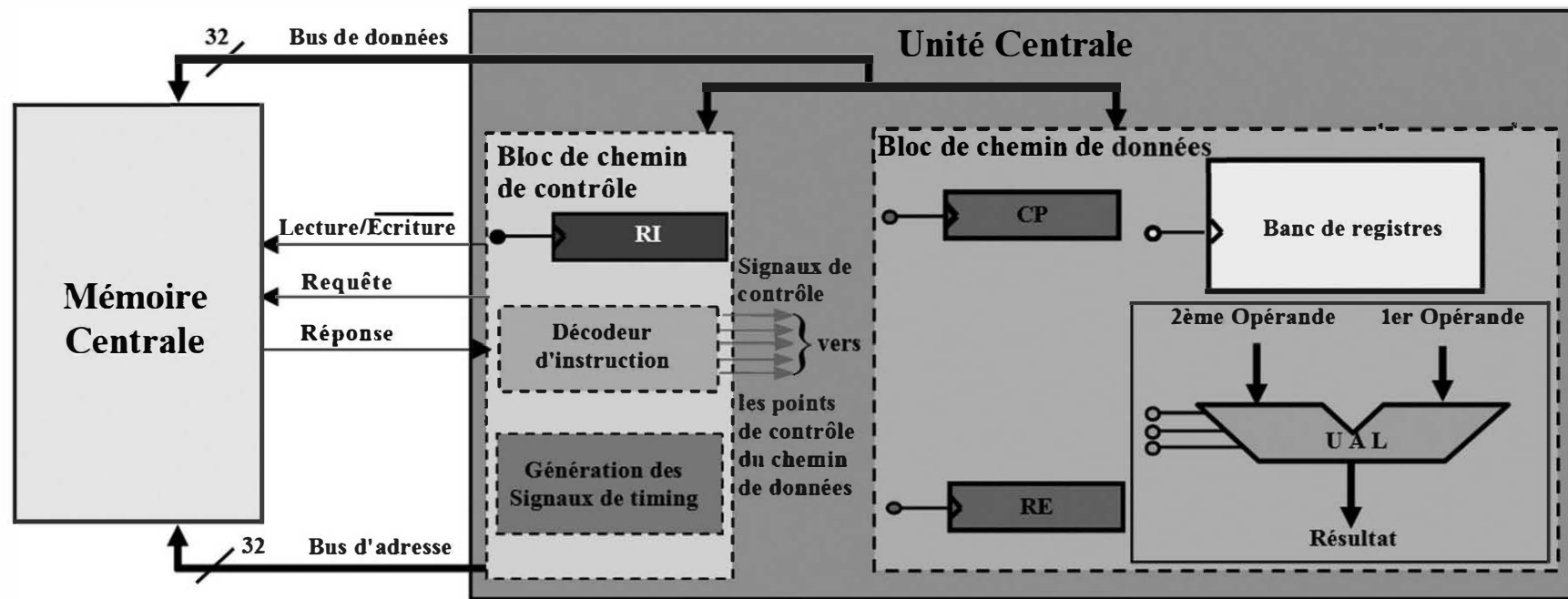


Figure 7.4 Couplage unité centrale-mémoire centrale (UC-MC).

ordinal ou compteur de programme (*Program Counter*, PC). Le bloc de contrôle qui interprète l'instruction assure la mise à jour du PC.

Une instruction est un ordre élémentaire à exécuter. Cet ordre est en général codé sur un mot que le bloc ou l'organe de contrôle interprète comme une association de plusieurs « champs » de bits. L'un d'entre eux représente la nature de l'ordre à exécuter : le « **code opération** » (COP). Le nombre de bits du champ code opération est fonction du nombre d'ordres que l'organe de contrôle est capable d'exécuter (le « **jeu d'instructions** »). Ce nombre d'ordres varie de quelques dizaines à quelques centaines selon les microprocesseurs.

Lorsqu'une instruction est lue en mémoire et amenée dans le bloc de contrôle, l'ensemble des champs constituant le code de l'instruction est mémorisé dans un registre du bloc de contrôle, dit **registre instruction** (RI), jusqu'à la fin de l'exécution de cette dernière. La partie code opération contenue dans le RI est exploitée par le bloc de contrôle à travers le **décodeur d'instructions** qui génère les signaux nécessaires pour l'exécution de l'instruction. Une instruction s'exécute en plusieurs étapes dont le séquençage est assuré par le séquenceur. Le RI n'est pas accessible par le programmeur.

La structure du bloc de contrôle est complexe. Il est constitué d'un ensemble de circuits séquentiels activés en fonction de la nature de l'ordre à réaliser. Son fonctionnement nécessite une horloge. Le microprocesseur possède un « **oscillateur** » interne qui fournit cette horloge mais dont la stabilité doit être assurée par un cristal de quartz externe au processeur.

Le bloc chemin de données

C'est la partie dans laquelle les traitements sur les données sont effectués. Ce bloc lui-même est composé de deux blocs : le bloc de registres et le bloc de calcul.

Toute opération sur les données doit se faire dans le bloc de calcul. Ce bloc contient au moins une **unité arithmétique et logique** (UAL) dont le rôle est de réaliser les opérations arithmétiques et logiques sous les ordres de l'organe de contrôle. Comme il n'existe qu'un seul chemin d'accès à la mémoire, les données doivent être lues une par une et stockées temporairement dans des registres du bloc de registres proches du bloc de calcul. Ils constituent le niveau le plus rapide de la hiérarchie mémoire. Ils ont un temps d'accès de l'ordre d'un à deux cycles processeur. Le bloc de registres est appelé **banc de registres** du fait de sa particularité : il permet l'accès simultané à deux registres en lecture et un

registre en écriture. Le nombre de ces registres varie en général de 1 à 32 selon les processeurs. La présence de plusieurs registres permet d'enchaîner des opérations sur les données en limitant les accès mémoire qui sont très coûteux par rapport au temps de cycle processeur.

Un transfert de données entre le microprocesseur et la mémoire se fait à la demande d'une instruction. L'instruction qui demande ce transfert doit donc en plus du code opération fournir une information sur la donnée à aller chercher (son adresse en mémoire en général) et le numéro d'un des registres du banc de registres concerné par le transfert. On distingue deux types de transfert :

- Si le transfert d'une donnée se fait de la mémoire vers un registre du processeur on parle d'une **instruction de chargement** ou **instruction de lecture**.
- Si le transfert d'une donnée se fait d'un registre du processeur vers la mémoire on parle d'une **instruction de rangement** ou **instruction d'écriture**.

Outre le banc de registres, il existe d'autres registres très particuliers faisant partie de l'environnement du programmeur parmi lesquels (figure 7.4) :

- Le compteur de programme ou ordinal (PC) dont nous avons déjà vu le rôle. Nous verrons plus loin que le CP est accessible par le programmeur comme un registre.
- Un registre d'état (RE) dont les différents bits forment un ensemble d'indicateurs d'état pouvant être divisé en deux groupes :
 - Un groupe (**bits de contrôle**) concerne le contrôle de l'état du processeur. En général le rôle et le nombre des bits de ce groupe dépendent du microprocesseur.
 - L'autre groupe (**bits de code condition**) concerne l'état des opérations arithmétiques et logiques, réalisées par l'UAL. Il est constitué par quatre indicateurs dont le rôle est le même dans tous les microprocesseurs, avec parfois quelques différences sur le positionnement de ces indicateurs. Ils sont connus sous leur nom en anglais : *Carry* (C), *overflow* (V), *Negative* (N), *Zero* (Z).

Rôle des indicateurs d'états

- **C** : indicateur de débordement de capacité/anomalie dans le cas des additions/soustractions sur **des entiers naturels** (nombres non signés). Il joue le rôle de Report (Carry) dans le cas de l'addition et

de retenue (Borrow) dans le cas de la soustraction. Par abus de langage on parle toujours de retenue pour les deux opérations.

• $C = 0$: résultat correct, $C = 1$: résultat incorrect.

- **N** : indicateur du signe d'un opérande ou du résultat d'une opération arithmétique ou logique.

• $N = 0$: positif, $N = 1$: négatif.

- **Z** : indique si un opérande ou le résultat d'une opération arithmétique ou logique est nul.

• $Z = 0$ résultat/opérande non nul, $Z = 1$ résultat/opérande nul.

- **V** : indicateur de débordement de capacité dans le cas des d'additions/soustractions sur des **entiers relatifs** (nombres signés).

• $V = 0$ résultat correct, $V = 1$ résultat incorrect.



À la suite des opérations arithmétiques, le processeur met à jour la fois C et V . Le programmeur doit tester C s'il considère les nombres comme des entiers naturels. Par contre, s'il considère les nombres comme des entiers relatifs il doit tester V .



CAS DES NOMBRES ENTIERS NATURELS

Considérons des nombres (N) codés sur 8 bits donc les valeurs N sont comprises entre : $0 \leq N \leq 255$.

$$\begin{array}{r}
 \begin{array}{cccccccc}
 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
 + & 1 & 0 & 0 & 0 & 0 & 0 & 1
 \end{array} \\
 \hline
 \begin{array}{cccccccc}
 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0
 \end{array}
 \end{array}
 \begin{array}{l}
 (67) \\
 (129) \\
 (196)
 \end{array}$$

$C = 0$ le résultat est correct car compris entre 0 et 255.

$$\begin{array}{r}
 \begin{array}{cccccccc}
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 + & 0 & 1 & 0 & 0 & 0 & 0 & 1
 \end{array} \\
 \hline
 \begin{array}{cccccccc}
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1
 \end{array}
 \end{array}
 \begin{array}{l}
 (64) \\
 (65) \\
 (129)
 \end{array}$$

$C = 0$ le résultat est correct car compris entre 0 et 255.

On ne tient pas compte de V car les nombres sont considérés comme des entiers naturels.

$$\begin{array}{r}
 \begin{array}{cccccccc}
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 + & 1 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array} \\
 \hline
 \begin{array}{cccccccc}
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}
 \end{array}
 \begin{array}{l}
 (128) \\
 (128) \\
 (256)
 \end{array}$$

$C = 1$ le résultat n'est pas correct car est > 255 . Il faut 9 bits pour coder le résultat.

Considérons des nombres (N) codés sur 8 bits donc les valeurs sont comprises entre : $128 \leq N \leq +127$.

$$\begin{array}{rcccccccccccl}
& & & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & (+67) \\
& + & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & (-127) \\
\hline
\cancel{C=0} & V=0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & (-60)
\end{array}$$

$$\begin{array}{rcccccccc}
& & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & (+64) \\
& + & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & (+65) \\
\hline
\cancel{C=Q} & V=1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & (-127)
\end{array}$$
$$\begin{array}{rcccccccccl}
& & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & (-128) \\
& + & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & (-128) \\
\hline
\cancel{C=1} & V=1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (0)
\end{array}$$

Cycle d'exécution d'une instruction

- Les instructions arithmétiques et logiques sont codées avec quatre champs : un champ pour le code opération, deux champs pour les opérandes sources et un champ pour l'opérande destination.
- Les instructions de transfert de données entre le microprocesseur et la mémoire sont codées sur trois champs : un champ code opération, un champ pour l'adresse mémoire, et un champ pour indiquer un opérande destination (cas de lecture mémoire) ou un opérande source (cas d'écriture mémoire).

Les autres cas sont des cas particuliers de ces formats. La figure 7.5, montre le cycle d'exécution d'une instruction dans le cas général.

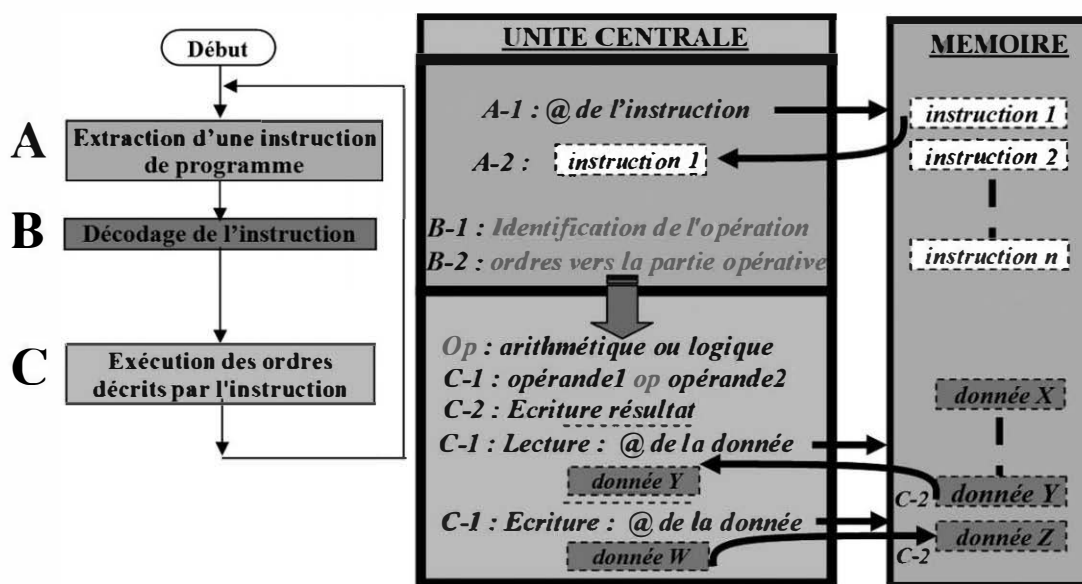


Figure 7.5 Cycle général d'exécution d'une instruction.

L'exécution d'une instruction se déroule en trois grandes étapes :

- **Étape A** : Extraction ou lecture d'une instruction à partir de la mémoire :
 - **Phase A-1** : le bloc de contrôle fournit sur le bus d'adresse l'adresse en mémoire de l'instruction 1.
 - **Phase A-2** : la mémoire retourne l'instruction 1 sur le bus de données.
- **Étape B** : Décodage de l'instruction
 - **Phase B-1** : identification par le bloc de contrôle de l'opération à réaliser
 - **Phase B-2** : émission des ordres (signaux de contrôle) vers la partie opérative (chemin des données) pour réaliser l'opération demandée par l'instruction.
- **Étape C** : Exécution de l'instruction

Si l'opération est du type arithmétique ou logique :

 - **Phase C-1** réalisation de l'opération entre les deux opérandes spécifiés dans l'instruction lus à partir du banc de registres.
 - **Phase C-2** écriture du résultat de l'opération dans le registre de destination spécifié dans l'instruction.

Si l'opération est de type transfert de donnée :

Cas d'une lecture en mémoire de la donnée Y :

 - **Phase C1** : fourniture à la mémoire de l'adresse de la donnée Y à lire.



Attention (Carry) fonctionne en logique négative dans le cas de la soustraction. C'est particulier au processeur ARM

Bits de contrôle – Les bits de 0 à 7 forment le groupe concernant l'état du processeur. Ils ne sont pas accessibles en mode utilisateur donc nous ne les détaillerons pas dans ce chapitre.

Organisation des données en mémoire

La mémoire est vue comme un tableau linéaire d'octets numérotés de 0 à $2^{32} - 1$. Les données manipulées sont des **octets** (8 bits), des **demi-mots** (16 bits) ou des **mots** (32 bits).



Les mots sont toujours alignés sur des limites de 4 octets (leur adresse est toujours multiple de 4). Les demi-mots sont alignés sur des limites de 2 octets (leur adresse est toujours multiple de 2). Les octets sont accessibles aux adresses paires et impaires (figure 7.7).

En supposant α une adresse multiple de 4 :

- Les accès autorisés :
 - mots : aux adresses α , $\alpha + 4$, $\alpha + 8$, $\alpha + 12$, etc.
 - demi-mots : aux adresses α , $\alpha + 2$, $\alpha + 4$, $\alpha + 6$, etc.
- Les accès non autorisés :
 - mots : aux adresses $\alpha + 1$, $\alpha + 2$, $\alpha + 3$, $\alpha + 5$, etc.
 - demi-mots : aux adresses $\alpha + 1$, $\alpha + 3$, $\alpha + 5$, $\alpha + 7$, etc.

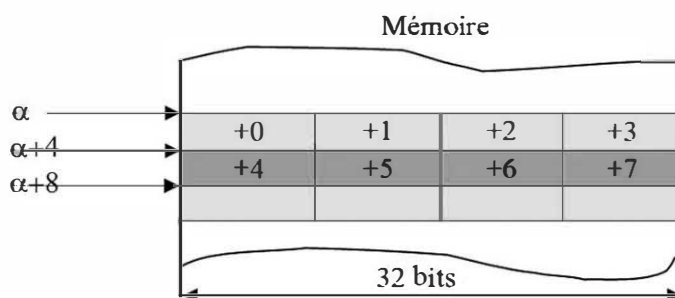


Figure 7.7 Alignement des mots et demi-mots en mémoire.

Le processeur ARM autorise l'utilisation d'une des deux organisations des données en mémoires suivantes.

Organisation big endian

Dans cette organisation les mots (les demi-mots) sont rangés en mémoire avec l'octet de fort poids à l'adresse basse et les octets suivants aux adresses supérieures (l'octet suivant à l'adresse supérieure) (figure 7.8).



EXEMPLE

Pour le mot à l'adresse α (α multiple de 4), on a : octet de fort poids à l'adresse α , octet suivant à $\alpha + 1$, encore octet suivant à $\alpha + 2$ et enfin octet de faible poids à l'adresse $\alpha + 3$. (figure 7.8).

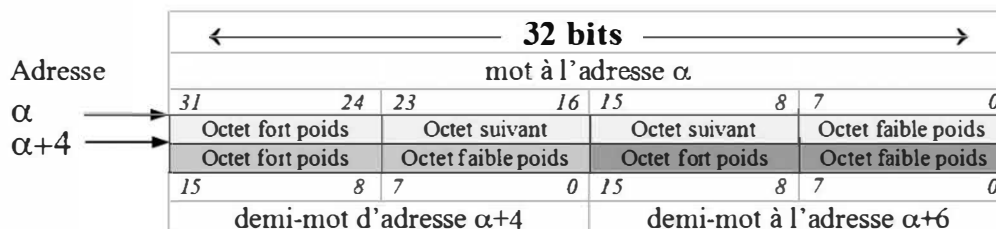


Figure 7.8 Organisation big endian.

Organisation little endian

Dans cette organisation les mots (demi-mots) sont rangés en mémoire avec l'octet de faible poids à l'adresse basse et les octets suivants aux adresses supérieures (l'octet suivant à l'adresse supérieure) (figure 7.9).



EXEMPLE

Pour le mot à l'adresse α (α multiple de 4), on a : octet de faible poids à l'adresse α , octet suivant à $\alpha + 1$, encore octet suivant à $\alpha + 2$ et enfin octet de fort poids à l'adresse $\alpha + 3$ (figure 7.9).

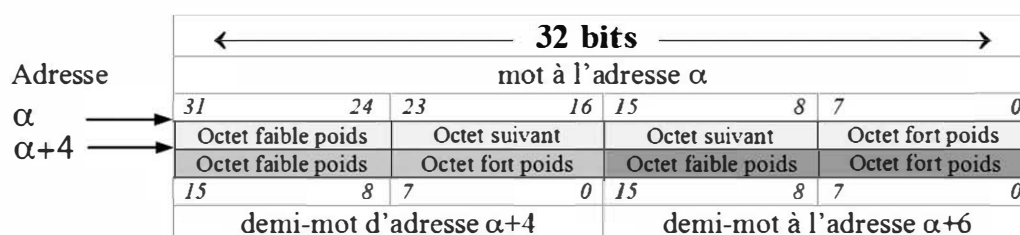


Figure 7.9 Organisation little endian.

Dans tout le reste du chapitre nous considérons l'organisation big endian.

Caractéristiques du jeu d'instructions de l'ARM

Les instructions de l'ARM ont un format fixe de 32 bits. L'architecture de l'ARM est dite du type LOAD/STORE c'est-à-dire seule les instruc-

tions de chargement (Lecture) et de rangement (Écriture) peuvent accéder à la mémoire.

Toutes les instructions peuvent être exécutées de manière conditionnelle. Nous considérons une exécution classique inconditionnelle. Nous allons aborder les différentes catégories d'instructions en présentant à chaque fois les modes d'adressage utilisés dans chaque catégorie.

7.4 INSTRUCTIONS DE TRAITEMENT DES DONNÉES

Les instructions de cette catégorie nécessitent généralement deux opérandes sources et fournissent éventuellement un résultat. Lorsque deux opérandes sources sont requis, l'un des opérandes est nécessairement un registre. Le second opérande peut être une valeur immédiate, ou un registre. Dans ce dernier cas, il est possible d'effectuer un décalage du contenu du registre avant exécution de l'instruction. L'opérande résultat, lorsqu'il est fourni par une instruction, est aussi un registre.

Cette catégorie englobe plusieurs classes d'instructions :

- les instructions arithmétiques et logiques ;
- les instructions d'assignation ou d'affectation ;
- les instructions de comparaisons et de test.

On va commencer par introduire les **modes d'adressage** utilisés par cette catégorie. Nous présenterons ensuite les instructions par classe.

Modes d'adressage des instructions de traitement des données

Tous les modes d'adressage cités ci-après utilisent sans restriction tous les registres (r0 à r14) du banc de registres. Le registre r15, qui joue le rôle de compteur, doit être utilisé avec précaution

Dans tout ce qui suit `<Rm>` indique le registre à utiliser. Si le contenu de `<Rm>` doit être décalé par décalage logique, arithmétique ou par rotation, Le nombre de décalage est indiqué soit par `<shift_imm_5>`, une constante codée sur 5 bits (Immédiat), soit par les 8 bits de faible poids d'un registre `<Rs>` (Par registre).

Adressage immédiat

Ce mode d'adressage permet de préciser une constante comme opérande codée dans l'instruction. Il est indiqué par le symbole `#`. Syntaxe :

| `#<immédiat>`

Comme il est impossible de coder une constante sur 32 bits dans l'instruction, cette dernière est codée par deux champs :

- Un de 8 bits permettant de coder une valeur notée `immédiat_8` avec ($0 \leq \text{immédiat_8} \leq 255$).
- Un de 4 bits permettant de coder une valeur notée `rotation_imm` avec ($0 \leq \text{rotation_imm} \leq 15$).

L'opérande immédiat est obtenu en appliquant sur la valeur `immédiat_8` un nombre de rotations à gauche égal à $2 \times \text{rotation_imm}$ au moment de l'exécution de l'instruction. Au final l'opérande résultat égal `immédiat_8` $\times 2^{2 \times \text{rotation_imm}}$ est étendu au format 32 bits. Pour un contrôle plus fin sur l'encodage de l'instruction, la valeur immédiate peut être remplacée par la syntaxe :

```
#<immédiat_8>, <rotate_imm>
```

Il est donc possible de vérifier que la valeur immédiate souhaitée respecte bien les limites de précision imposée par le codage. On remarque qu'on ne peut coder que des constantes particulières. Nous verrons plus tard une directive de l'assembleur qui peut nous affranchir du codage des constantes supérieures à 255.

La constante peut être codée en :

- **Décimal** : (par défaut) rien ne précède la constante.
- **Hexadécimal** : la constante doit être précédée de 0x.
- **Binaire** : la constante doit être précédée de 0b.

Adressage par registre

Ce mode exploite le contenu d'un registre. Syntaxe :

```
<Rm>
```

Adressages par registre avec décalage logique à gauche

L'opération de décalage est LSL (*Logical Shift Left*) **décalage logique à gauche**. Ce mode d'adressage permet d'utiliser le contenu d'un registre après décalage préalable à gauche d'un certain nombre de positions.



Figure 7.10 Décalage logique à gauche.

- Immédiat : permet de faire de 0 à 31 décalages. Syntaxe :

| <Rm>, LSL #<shift_imm_5>

- Par registre : permet de faire de 0 à 255 décalages. Syntaxe :

| <Rm>, LSL <Rs>



Lorsque <shift_imm_5> est égal à 0, on retrouve le mode d'adressage par registre. Il est un cas particulier de ce mode.

Adressages par registre avec décalage logique à droite

L'opération de décalage est LSR (*Logical Shift Right*) **décalage logique à droite**. Ce mode d'adressage permet d'utiliser le contenu d'un registre après décalage préalable à droite d'un certain nombre de positions.

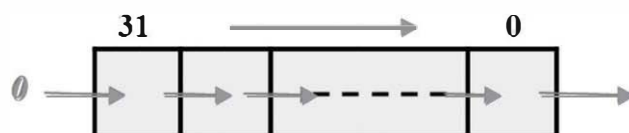


Figure 7.11 Décalage logique à droite.

- Immédiat : permet de faire de 1 à 32 décalages. Syntaxe :

| <Rm>, LSR #<shift_imm_5>

- Par registre : permet de faire de 0 à 255 décalages. Syntaxe :

| <Rm>, LSR <Rs>



Lorsque la constante est égale à 32, elle sera codée par la valeur binaire 00000 dans l'instruction. 0 représente donc un décalage de 32 positions. Dans ce cas, l'opérande est égal à 0 et le bit C du CPSR est mis à jour avec le bit de plus fort poids de Rm.

Adressages par registre avec décalage arithmétique à droite

L'opération de décalage est ASR (*Arithmetic Shift Right*) **décalage arithmétique à droite**. Il est identique au mode par registre avec décalage logique à droite à ceci près que les bits insérés sont égaux au bit de signe du registre décalé.

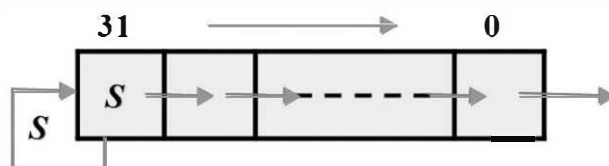


Figure 7.12 Décalage arithmétique à droite.

- Immédiat : permet de faire de 1 à 32 décalages. Syntaxe :

| <Rm>, ASR #<shift_imm_5>

- Par registre : permet de faire de 0 à 255 décalages. Syntaxe :

| <Rm>, ASR <Rs>

Adressages par registre avec rotation à droite

L'opération de **rotation à droite** est ROR (*ROtate Right*). Elle permet de décaler circulairement à droite, le contenu d'un de 1 à 31 positions avant son utilisation. Le décalage se fait par report en plus fort poids du bit éjecté en plus faible poids.

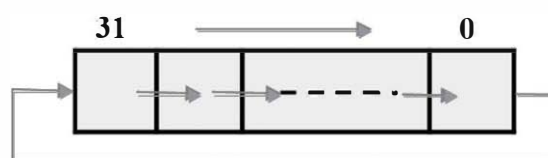


Figure 7.13 Rotation à droite avec extension.

- Immédiat : permet de faire de 1 à 31 rotations à droite. Syntaxe :

| <Rm>, ROR #<shift_imm_5>

- Par registre : permet de faire de 0 à 255 décalages. Syntaxe :

| <Rm>, ROR <Rs>

Adressage par registre avec rotation droite avec extension

L'opération de rotation à droite d'une *seule position* avec extension est RRX (*Rotate Right eXtended*). Ce mode d'adressage permet d'utiliser la valeur contenue dans un registre décalée circulairement à droite d'une position. Le décalage se fait par insertion de la valeur contenue dans le bit C (*Carry*) du CPSR.

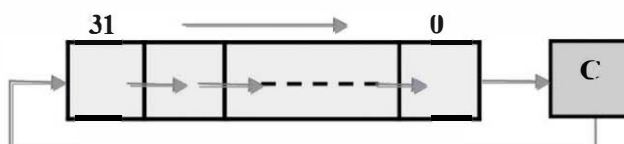


Figure 7.14 Rotation à droite.

Syntaxe :

| <Rm>, RRX



Dans tous les modes avec décalages ou des rotations, si le nombre de décalages est une valeur immédiate et si $\langle Rm \rangle$ est r15 alors la valeur exploitée de r15 est l'adresse de l'instruction courante plus 8. Si le nombre de décalages est dans un registre, r15 ne peut pas être utilisé ni comme $\langle Rm \rangle$ ni comme $\langle Rs \rangle$.

Nous allons maintenant étudier les différentes classes des instructions de traitement de données. Dans tout ce qui suit « COP » spécifie le code opération d'une instruction de la classe, « source1 », « source2 » et « destination » sont les opérandes sources et l'opérande destination du résultat de l'instruction.

La classe des instructions arithmétiques et logiques

Le format général de cette classe d'instructions est :

| COP destination, source1, source2

- **destination et source1** utilisent **toujours** le mode registre. Ils **sont toujours** des registres parmi r0 à r14.
- **source2** peut utiliser un des modes d'adressage suivant :
 - *Le mode registre* : source2 est un registre parmi r0 à r14.
 - *Le mode immédiat* : source2 est une constante.
 - Le mode registre avec décalage ou rotation : source2 est un registre parmi r0 à r14.

Les instructions arithmétiques

Ces instructions réalisent une opération entre « source1 » et « source2 » et rangent le résultat dans « destination ».

Mnémonique	Signification	Opération
ADD	ADDition	destination $\leftarrow$ source1+source2
ADC	Addition with Carry	destination $\leftarrow$ source1+source2+C
SUB	SUBstract	destination $\leftarrow$ source1-source2
SBC	SUBstract with Carry	destination $\leftarrow$ source1-source2+C-1
RSB	Reverse SUBstract	destination $\leftarrow$ source2-source1
RSC	Reverse Substract with Carry	destination $\leftarrow$ source2-source1+C-1

Les instructions logiques

Ces opérations sont appliquées entre les bits de même rang de source1 et source2.

Mnémonique	Signification	Opération
AND	AND	destination ← source1 ET logique source2
ORR	OR Register	destination ← source1 OU logique source2
EOR	Exclusive OR	destination ← source1 OU exclusif source2
BIC	Bit Clear	destination ← source1 ET logique complément source2



EXEMPLES

Source 2 : mode par registre

```
| ADD r0, r1, r2          @ r0 ← r1+r2
```

@ indique que tout ce qui suit est un commentaire

L'opération réalise l'addition de r1 (source1) avec r2 (source2) et range le résultat dans r0.

Source2 : mode immédiat

```
| ADD r0, r1, #2          @ r0 ← r1+2
| AND r8, r7, #0xFF       @ r8 ← r7(bits7..0)
```

copie des 8 bits de faible poids de r7 dans r8.

```
| ORR r2, r1, #0b00001111 @ r2 ← r1(bits31..4)
```

et bits (3..0) = 0b1111

On copie des bits de rang 31 à 4 de r1 dans r2 et les 4 bits de faibles poids de r2 prennent la valeur binaire 1111.

Source2 : mode par registre avec décalage ou rotation

Exemple 1 : nombre de décalages par valeur immédiate

```
| ADD r3, r2, r1, LSL #3  @ r3 ← r2+r1*8
```

r1 subit un décalage logique de 3 positions vers la gauche avant addition ceci revient à une multiplication de r1 par 2^3 .

Exemple 2 : nombre de décalages par registre (octet de faible de poids).

```
| ADD r5, r5, r3, LSL r2  @ r5 ← r5+r3*2r2
```



Dans les modes par décalage ou rotation, le contenu du registre (source2) sur lequel est appliquée l'opération de décalage ou de rotation n'est pas affecté après exécution de l'instruction sauf si ce même registre sert aussi de destination.

La classe des instructions d'assignation

Cette classe d'instructions concerne l'assignation ou d'affectation d'une valeur à un registre ou le transfert de données entre registres.

Le format général de cette classe est identique à la classe des instructions arithmétiques et logiques avec omission de l'opérande source1 :

COP destination, source2

MOV (MOVe) et **MVN** (MoVe Not) : réalise un transfert de source2 respectivement du complément à un de source2, dans destination.



EXEMPLES

Source 2 : mode par registre

```
MOV r0, r2    @ r0 ← r2 copie r2 dans r0
MVN r0, r2    @ r0 ← r2 copie r2 dans r0
```

Source2 : mode immédiat

```
MOV r0, #2    @ r0 ← 2 assigne à r0 la valeur 2
MVN r0, #2    @ r0 ← 2 | assigne à r0 le complément
                @ de la valeur 2 c'est-à-dire 0xFFFFFDD.
```

Source2 : mode par registre avec décalage ou rotation

```
MOV r0, r2, LSL #2 @ r0 ← r2*4 copie r2 multiplié
                    @ par 4 dans r0.
MVN r0, r2, LSL #2 @ r0 ← r2*4 copie (r2*4) dans r0
```

La classe des instructions de comparaison et de test

Cette classe d'instructions est un cas particulier des instructions arithmétiques (pour les comparaisons) et logiques (pour les tests). Le format général de cette classe est identique à celui de la classe des instructions arithmétiques et logiques avec omission de l'opérande destination :

COP source1, source2.

Ces instructions ne produisent pas de résultat et ne font que mettre à jour le Code Condition (CC) (C, V, N, Z) du registre CPSR.

Les instructions de comparaisons

CMP (CoMPare) et **CMN** (CoMpare Negative) : réalise une comparaison respectivement par une soustraction (source1-source2) ou par une addition (source1+source2), et positionne le CC du registre CPSR en fonction du résultat (qui n'est pas sauvegardé).



EXEMPLES

Source 2 : mode par registre

CMP r2, r3	@CPSR ← CC(r2 - r3)
CMN r2, r3	@CPSR ← CC(r2 + r3)

Source2 : mode immédiat

CMP r2, #5	@CPSR ← CC(r2 - 5)
CMN r2, #5	@CPSR ← CC(r2 + 5)

Source2 : mode par registre avec décalage ou rotation

CMP r2, r3, LSL #2	@CPSR ← CC(r2 - r3*4)
CMN r2, r3, LSL #2	@CPSR ← CC(r2 + r3*4)

Les instructions de test

TST (TeST) et **TEQ** (Test EQuivalence) : réalise un test de champ de bits respectivement par une opération de ET logique ou par une opération de OU exclusif entre source1 et Source2, et met à jour les bits du CC du CPSR.



EXEMPLES

Source 2 : mode par registre

TST r2, r3	@CPSR ← CC(r2 ET logique r3)
TEQ r2, r3	@CPSR ← CC(r2 OU exclusif r3)

Source2 : mode immédiat

TST r2, #5	@CPSR ← CC(r2 ET logique 5)
TEQ r2, #0xF	@CPSR ← C(r2 OU exclusif 15)

Source2 : mode par registre avec décalage ou rotation

TST r2, r3, LSL #3	@CPSR ← CC(r2 ET r3*8)
TEQ r2, r3, LSL #3	@CPSR ← CC(r2 OU exclusif r3*8)

Mise à jour du code condition

Les instructions de comparaison et de test mettent **toujours systématiquement** à jour le code condition (CC) du registre CPSR : c'est leur seul rôle. Par contre, toutes les autres instructions de traitement de données ne mettent à jour le code condition que si le programmeur l'explicite spécifiquement dans chacune des instructions dont il souhaite qu'elle

affecte le code condition. Pour ce faire le programmeur doit ajouter la lettre S (*Set condition code*) au code opération de l'instruction.



Les instructions logiques et d'assignation ou de transfert ne mettent à jour que les bits Z (Zero) et N (Négative), exception faite du bit C qui est mis à jour par les décalages et la rotation étendue dont il reçoit le bit éjecté. Les instructions arithmétiques mettent à jour tous les bits du code condition : C, V, N, Z.



EXEMPLE

Addition de deux nombres N1 et N2 de 64 bits chacun N1 est dans les registres (r1-r0) et N2 dans les registres (r3-r2).

```
ADDS r2, r2, r0 @ r2 ← r2+r0 addition des 32 bits
                @ de faible poids et report (retenue)
                @ dans C
ADC r3, r3, r1  @ addition des 32 bits de fort poids
                @ et du report (retenue) C
```

7.5 INSTRUCTIONS DE RÉFÉRENCE MÉMOIRE

Le rôle de cette catégorie d'instructions est de **permettre uniquement de faire des accès à la mémoire** pour effectuer soit un chargement (lecture) soit un rangement (écriture). Les instructions de cette catégorie nécessitent généralement deux opérandes : un opérande source et un opérande destination. Un de ces deux opérandes servira toujours pour indiquer une adresse pour accéder à la mémoire. Cette adresse peut être :

- soit l'opérande source, et donc dans ce cas la destination est un registre du processeur : c'est le cas du chargement (ou lecture mémoire).
- soit l'opérande destination, et donc dans ce cas la source est un registre du processeur : c'est le cas du rangement (ou écriture mémoire).

Cette catégorie d'instructions se compose de deux classes : les instructions de transfert simple et les instructions de transfert multiple.

On va d'abord décrire les modes d'adressages qu'utilise cette catégorie d'instructions.

Modes d'adressages des instructions de référence mémoire

Adressage par registre indirect

L'adresse de l'opérande à charger ou ranger est spécifiée par un registre dit registre de base ou d'adresse (*<Rn>*). Syntaxe :

| [*<Rn>*]

Adressage par registre indirect avec déplacement signé

Déplacement simple

L'adresse de l'opérande à charger ou ranger est spécifiée par un registre (*<Rn>*). Le déplacement est toujours signé, et peut être soit immédiat codé sur 12 bits (*<offset_12>*), soit indiqué dans un second registre (*<Rm>*). Dans ce dernier cas, une éventuelle opération de décalage/rotation peut être appliquée au préalable. Le nombre de décalage ou rotation est toujours une constante codée sur 5 bits (*<Rm>*, {LSL|LSR|ASR|ROR|RRX}#*<shift_imm_5>*]).

➤ Syntaxe :

| [*<Rn>*, #+/-*<offset_12>*]

➤ Syntaxe :

| [*<Rn>*, +/−*<Rm>*]

➤ Syntaxe :

| [*<Rn>*, #+/-*<Rm>*, {LSL|LSR|ASR|ROR|RRX} #*<shift_imm_5>*]



L'adresse effective pour accéder à la mémoire est calculée de manière interne par le processeur, et est égale à la somme du contenu du registre d'adresse et de la valeur du déplacement signé.

Le registre de base ou d'adresse n'est pas modifié après l'accès.

Déplacement pré-indexé

Les modes d'adressages avec déplacement simple peuvent éventuellement être suivis d'un point d'exclamation (!). Dans ce cas, l'adresse effective calculée par le processeur est écrite dans le registre d'adresse (*<Rn>*) après l'accès à la mémoire.

Déplacement post-indexé

Dans ce mode, le crochet fermant de l'adressage indirect avec déplacement simple se positionne juste après le registre d'adresse (*<Rn>*). Cela indique que la valeur de ce registre est utilisée pour accéder à la

mémoire, et **qu'après l'accès** à la mémoire, le registre de base ou d'adresse ($\langle R_n \rangle$) est mis à jour par l'adresse effective qui est égale à la somme du contenu de registre d'adresse et de la valeur du déplacement signé.

Les instructions de transfert simple

Cette classe regroupe les instructions suivantes :

- **LDR** (*LoaD Register*) et **STR** (*STore Register*) : ce sont respectivement les instructions de chargement (lecture) et rangement (écriture) de mots (32 bits).
- **LDRH** (*LoaD Register Half word*) et **STRH** (*STore Register Half word*) : ce sont respectivement les instructions de chargement et rangement de demi-mots (16 bits).
- **LDRB** (*LoaD Register Byte*) et **STRB** (*STore Register Byte*) : ce sont respectivement les instructions de chargement et rangement d'octets (8 bits).



Quelle que soit l'instruction de chargement, les 32 bits du registre de destination sont toujours affectés.

Dans le cas de LDRH (respectivement LDRB), les 16 bits (respectivement 24 bits) de forts poids du registre de destination sont mis à zéro. Ce sont des instructions de chargement d'opérandes considérés comme des entiers naturels.

Il existe donc des instructions de chargement de demi-mots et d'octets pour des opérandes considérés comme des entiers relatifs : LDRSH (*LoaD Register Signed Half word*) et LDRSB (*LoaD Register Signed Byte*). Ces instructions étendent les 16 bits (respectivement 24 bits) de fort poids du registre de destination avec le bit signe situé au rang 15 (respectivement au rang 7) de la donnée chargée.



EXEMPLES

Adressage par registre indirect

```

LDR r0, [r1]    @ r0 ← Mémoire32[r1] chargement dans r0,
                  @ des 32 bits du mot mémoire
                  @ dont l'adresse est dans r1
STR r0, [r1]    @ Mémoire32[r1] ← r0 rangement
                  @ des 32 bits de r0 dans le mot mémoire
                  @ dont l'adresse est dans r1

```

Adressage par registre indirect et déplacement

```

LDR r0, [r1, #4]    @ r0 ← Mémoire32[r1+4]
STR r0, [r1, #4]    @ Mémoire32[r1+4] ← r0

```

Adressage par registre indirect et déplacement post-indexé

LDR r0, [r1, #4]! @ $r0 \leftarrow \text{Mémoire}_{32}[r1+4]$ et $r1 \leftarrow r1+4$

STR r0, [r1, #4]! @ $\text{Mémoire}_{32}[r1+4] \leftarrow r0$ et $r1 \leftarrow r1+4$

Adressage par registre indirect et déplacement pré-indexé

LDR r0, [r1], #4 @ $r0 \leftarrow \text{Mémoire}_{32}[r1]$ et $r1 \leftarrow r1+4$

STR r0, [r1], #4 @ $\text{Mémoire}_{32}[r1] \leftarrow r0$ et $r1 \leftarrow r1+4$

Les instructions de transfert multiple

Cette classe regroupe les instructions suivantes : **LDM** (*LoaD Multiple*) et **STM** (*STore Multiple*). Ce sont des instructions de chargement et de rangement multiple qui permettent d'effectuer des transferts de blocs de données mémoires vers plusieurs registres simultanément et inversement. L'accès se fait uniquement sur la base du mot. Le bloc mémoire est une zone ordinaire qui peut être allouée soit statiquement pour effectuer des chargements/rangements de blocs, soit dynamiquement et gérée comme une **pile** pour effectuer des dépilements/empilements. Cette classe utilise des modes d'adressages spécifiques : adressage bloc et adressage pile.

Modes d'adressage bloc

L'adresse de base utilisée pour accéder à la mémoire doit être incrémentée (*Increment*)/décrémentée (*Decrement*) à chaque opération de chargement (lecture) ou de rangement (écriture) de 4 octets. De plus, cette modification de l'adresse de base peut intervenir avant (*Before*) ou après (*After*) l'opération de lecture ou d'écriture, ce qui conduit à distinguer quatre modes recensés dans le tableau 7.1.

Tableau 7.1 MODES D'ADRESSAGE BLOC.

	Increment (I) (incrémenter)	Decrement (D) (décrémenter)
After (A) (après)	IA	DA
Before (B) (avant)	IB	DB

Syntaxe :

LDM<mode_adr> <Rn>, {<liste_registres>}

Syntaxe :

STM<mode_adr> <Rn>, {<liste_registres>}

Où :

- `<mode_adr>` est l'un des quatre modes donnés dans le tableau 7.1 ;
- `<Rn>` est le registre de base ou d'adresse ;
- `<liste_registres>` une liste peut comprendre n'importe quel registre de r0 à r15.



L'ordre des registres dans la liste n'a pas d'importance. Le transfert se fait toujours dans l'ordre croissant des numéros de registres, le registre de numéro le plus bas étant situé à l'adresse la plus basse.



EXEMPLES

```

LDMIA r1, {r0, r2, r5}    @ r0 ← Mémoire32[r1] ;
                           @ r2 ← Mémoire32[r1+4] ;
                           @ r5 ← Mémoire32[r1+8].
STMIA r1, {r2, r0, r5}    @ Mémoire32[r1] ← r0 ;
                           @ Mémoire32[r1+4] ← r2 ;
                           @ Mémoire32[r1+8] ← r5.
  
```

Modes d'adressage pile

Une pile (*Stack*) est une zone de mémoire ordinaire allouée **dynamiquement** sur la base de mots de 32 bits et gérée en mode **LIFO** (*Last In First Out*) c'est-à-dire que le dernier élément à être entré est le premier élément à être sorti. La pile peut s'accroître lors de l'empilement vers les adresses croissantes (*Ascending stack*) ou décroissantes (*Descending stack*). Le registre de base ou d'adresse est dit **pointeur de pile** (*Stack Pointer*, SP). Il contient l'adresse du sommet de la pile : soit l'adresse du dernier élément rangé dans la pile (*Full stack pointer*), soit l'adresse de la position disponible où sera rangé le prochain élément empilé (*Empty stack pointer*). La valeur du pointeur de pile (SP) est incrémentée/décré-

Tableau 7.2 MODES D'ADRESSAGE PILE.

Pile (<i>Stack</i>) Pointeur de pile (<i>Stack Pointer</i>)	@ croissantes Ascending (A)	@ décroissantes Desscending (D)
Première position disponible Empty (E)	EA	ED
Dernière position occupée Full (F)	FA	FD

mentée à chaque opération d'empilement ou de dépilement de 4 octets. L'évolution des adresses de la pile et la position indiquée par le pointeur de pile conduit à quatre organisations de pile et un mode d'adressage par organisation (tableau 7.2).

Syntaxe :

```
LDM<mode_adr> <Rn> !, {<liste_registres>}
```

Syntaxe :

```
STM<mode_adr> <Rn> !, {<liste_registres>}
```

Où :

- <mode_adr> est l'un des modes du tableau 7.2 ;
- <Rn> est le registre SP. Le point d'exclamation (!) indique la mise à jour automatique du SP ;
- <liste_registres> une liste peut comprendre n'importe quel registre de r0 à r15.



Le processeur ARM peut être configuré pour fonctionner avec une des quatre organisations de pile ci-dessus. On considère pour tout le reste du chapitre une pile FD (Full stack pointer, Descending stack).

Le pointeur de pile SP peut être n'importe quel registre sauf r15 et r14. Habituellement cette fonction est réservée au registre r13.



EXEMPLES

Empiler dans une pile FD r2, r5, r1 pointée par r13

```
STMFD r13!, {r1, r2, r5} @ r13 ← r13-4, Mém32[r13] ← r1 ;
                        @ r13 ← r13-4, Mém32[r13] ← r2 ;
                        @ r13 ← r13-4, Mém32[r13] ← r5.
```

Dépiler d'une pile FD r0, r5, r1 pointée par r13

```
LDMFD r13!, {r1, r0, r5} @ r5 ← Mém32[r13], r13 ← r13+4;
                        @ r1 ← Mém32[r13], r13 ← r13+4;
                        @ r0 ← Mém32[r13], r13 ← r13+4.
```

7.6 INSTRUCTIONS DE CONTRÔLE DE PROGRAMME

Nous avons vu les instructions permettant de traiter les données et de les transférer de la mémoire vers les registres du processeur et inversement. Il nous manque la manière de contrôler le **flot d'exécution** des instructions d'un programme. En effet un programme s'exécute de manière séquentielle, mais cette exécution peut être rompue de manière **condi-**

tionnelle ou **inconditionnelle** pour atteindre une autre séquence d'instructions afin de satisfaire une alternative ou une répétition, ou encore pour faire appel à **un sous-programme**. Cette catégorie d'instructions utilise généralement un mode d'adressage implicite relatif au compteur de programme. Elle regroupe les instructions de branchements et d'appel de sous-programme.

Les instructions de branchement simple

B (*Branch*) : une instruction de branchement provoque la rupture de l'exécution séquentielle d'un programme :

- Soit de manière inconditionnelle (rupture systématique).
- Soit d'une manière conditionnelle (rupture si la condition est vraie).

Syntaxe :

```
B{<cond>} <adresse_cible>
```

Où **<cond>** précise la condition d'exécution de l'instruction, liée au code condition du registre CPSR. Par défaut, elle est considérée comme toujours vraie (cas du branchement inconditionnel).

<Adresse_cible> spécifie un déplacement, signé sur 24 bits, relatif au PC, pour atteindre l'adresse de l'instruction cible. Il est calculé automatiquement par l'assembleur.



Bien qu'il soit codé sur 24 bits, la cible du branchement peut se situer dans la plage d'adresse de ± 32 Mo autour de la valeur courante de PC : le déplacement est multiplié par 4 et ajouté à la valeur du PC pour le calcul de l'adresse cible.

Le tableau 7.3 donne la liste des instructions de branchement et les conditions associées.

Les instructions de branchement sont invoquées généralement après une instruction de comparaison ou de test pour comparer deux opérandes source.

Par exemple, **CMP A, B** positionne le code condition selon le résultat de $A - B$ (voir section 7.4). Ensuite il faut choisir l'instruction de branchement selon la condition souhaitée d'une part, et l'appartenance des opérandes soit à l'ensemble des entiers naturels, soit à l'ensemble des entiers relatifs d'autre part. Le tableau 7.4 indique l'instruction de branchement à choisir selon ces critères.

Tableau 7.3 LES INSTRUCTIONS DE BRANCHEMENT.

Instruction	Signification	Condition
BEQ	Branch if E Qual : si égal	Z
BNE	Branch if N ot E Qual : si différent	$\bar{Z}$
BCS/BHS	Branch if C arry S et/unsigned H igher or S ame : si supérieur ou égal non signé	C
BCC/BLO	Branch if C arry C lear/unsigned L ower : si inférieur non signé	$\bar{C}$
BMI	Branch if M inus/negative : si négatif	N
BPL	Branch if P lus/positive or zero : si positif ou nul	$\bar{N}$
BVS	Branch if o Verflow S et : si débordement	V
BVC	Branch if o Verflow C lear : si pas de débordement	$\bar{V}$
BHI	Branch if unsigned H igher : si supérieur non signé	$C \cdot \bar{Z}$
BLS	Branch if unsigned L ower or S ame : si inférieur ou égal non signé	$\bar{C} + Z$
BGE	Branch if signed G reater than or E qual : si supérieur ou égal signé	$(N \equiv V) + Z$
BLT	Branch if signed L ess T han : si inférieur signé	$(N \oplus V) \cdot \bar{Z}$
BGT	Branch if signed G reater T han : si supérieur signé	$\bar{Z} \cdot (N \equiv V)$
BLE	Branch if signed L ess than E qual : si inférieur ou égal signé	$Z + (N \oplus V)$
B	Branch A Lways : branchement inconditionnel	Toujours vraie

Tableau 7.4 CHOIX DES INSTRUCTIONS DE BRANCHEMENT EN FONCTION DE LA CONDITION ET DE L'ENSEMBLE AUQUEL APPARTIENNENT LES NOMBRES.

Condition	Nombres non signés	Nombres signés
$A = B$	BEQ	BEQ
$A \neq B$	BNE	BNE
$A > B$	BHI	BGT
$A \geq B$	BCS	BGE
$A < B$	BCC	BLT
$A \leq B$	BLS	BLE



EXEMPLES

Exemple 1 :

```

        MOV    r0, #0          @ initialisation
                                @ d'un compteur de boucle (r0)
boucle: CMP    r0, #10         @ comparaison du compteur à 10
        BHS    suite          @ si r0>=10 aller à suite
        ....          @ corps de boucle
        ....          @ "      "
        ADD    r0, r0, #1      @ incrémente compteur de boucle
        B      boucle          @ branchement à boucle
suite:   ....

```

Exemple 2 :

```

        MOV    r0, #0          @ initialisation
                                @ d'un compteur de boucle (r0)
boucle:   ....          @ corps de boucle
        ....          @ "      "
        ADD    r0, r0, #1      @ incrémente compteur de boucle
        CMP    r0, #10         @ comparaison du compteur à 10
        BLO    boucle          @ branchement à boucle si r0<10
suite:   ....

```

Les instructions de branchement avec lien

Dans un programme, on souhaite parfois faire un branchement vers une procédure ou une fonction (appel de sous-programme), l'exécuter, puis reprendre l'exécution du programme, ce qui nécessite de mémoriser l'adresse de retour (adresse de l'instruction qui suit le branchement). Ceci est réalisé par l'instruction suivante :

- **BL** (*Branch with Link*) : branchement avec lien de retour. Elle permet de faire un branchement à un sous-programme et de sauvegarder automatiquement l'adresse de retour dans le registre r14 ou *Link Register* (LR).

Syntaxe :

```
| BL{<cond>}    <adresse_cible>
```

<cond> et <adresse_cible> ont la même signification que précédemment.

Nous considérons pour tout le reste du chapitre que la condition pour l'instruction BL est toujours vraie.



EXEMPLE : APPEL DE SOUS-PROGRAMME

```

.....
BL sous_prof    @ r14 ← pc+4 (adr. suite) ;
                  @ pc ← sous_prog
suite:          .... (instruction suivante)
.....
sous_prog:      ....
.....
MOV pc, r14     @ pc ← r14

```

Rappel : $pc = r15 \rightarrow$ donc on peut écrire aussi `MOV r15, r14`

Déroulement de l'appel de sous-programme

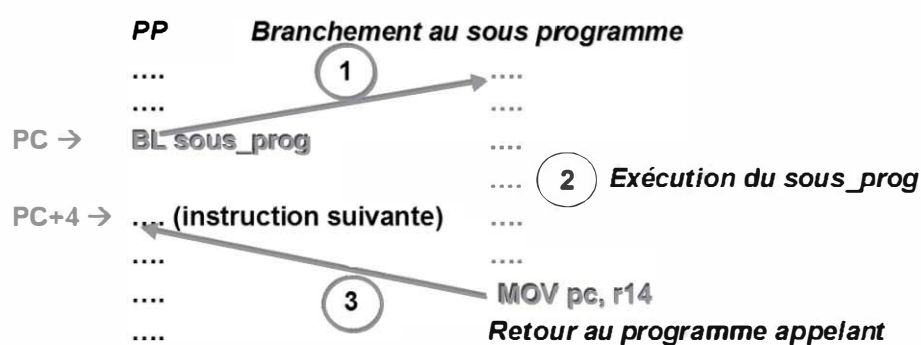


Figure 7.15



Remarque : comme l'adresse de retour est rangée dans r14, un sous-programme ne peut pas faire appel à un autre sous-programme sans avoir pris le soin de sauvegarder l'adresse de retour de son appelant. Cette sauvegarde se fait généralement dans la pile.



EXEMPLE : SOUS PROGRAMMES IMBRIQUÉS

```

.....
Programme Principal
.....
BL      sp1      @ appel de sp1 : r14 ← pc+4 ;
                  @ pc ← sp1
.....
sp1:    ....
.....
STMF    r13!,{r14} @ r13← r13-4 ; Mem32[r13]←r 14
BL      sp2      @ appel de sp2 : r14 ← pc+4 ;
                  @ pc ← sp2
.....
LMFD    r13!,{r15} @ pc ← Mem32[r13] ; r13 ← r13+4

```

```

sp2:  ....
      ....
      ....
      MOV pc, r14

```

Déroulement d'appel de sous-programmes imbriqués

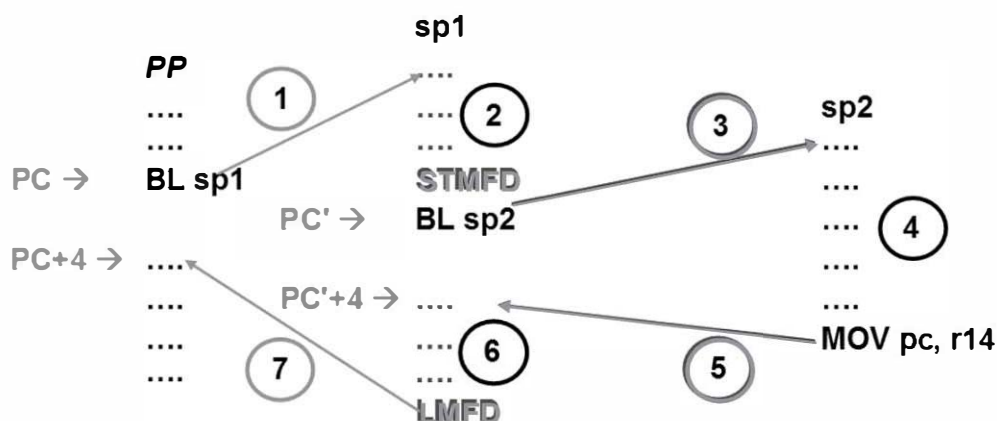


Figure 7.16

L'instruction **STMFD** (respectivement **LMFD**) sauvegarde (respectivement restaure) dans la pile de l'adresse de retour au programme principal.

7.7 SYNTAXE D'UN FICHIER SOURCE EN LANGAGE D'ASSEMBLAGE

Structure d'une ligne

Une ligne de code est composée par les différents champs suivants :

```

<étiquette>  <code  instruction>  <opde>,  <opde>, ...
@<commentaires>

```

- Les différents champs doivent être séparés par au moins un caractère blanc (espace, tabulation).
- Si le champ étiquette est absent l'espace avant le mnémonique doit être toujours présent.
- L'étiquette est un nom qui permet de représenter l'adresse de l'instruction en mémoire.

- Le champ code instruction représente le mnémonique de cette dernière.
- Les champs opérande doivent être séparés par des virgules.
- Le champ commentaire doit débiter par le caractère @. Une ligne de commentaire doit débiter par @.

Directives d'assemblage

Les directives ont un rôle essentiel de paramétrage du programme : définir symboliquement des constantes, faire des réservations de places mémoires initialisées ou non initialisées.

- **.equ** : définition de constante symbolique. Syntaxe :

```
| <étiquette> .equ <expression>
```

Exemple :

```
| N .equ 200 @ au moment de l'assemblage le symbole
| @ N se voit affecter la valeur 200
```

- **.byte** : réservation d'octets mémoire initialisés. Syntaxe :

```
| <étiquette>.byte <expression1>,<expression2>,...
```

Exemples :

```
| var .byte 55 @ réservation d'une variable
| @ initialisée avec 55
| Tab .byte 30,0xFF,50,...@ réservation d'un tableau
| @ d'octets initialisés
```

- **.hword** : réservation de demi-mots mémoire initialisés. Syntaxe :

```
| <étiquette> .hword <expression1>, <expression2>, ...
```

- **.int** : réservation de mots mémoire initialisés. Syntaxe :

```
| <étiquette> .hword <expression1>, <expression2>, ...
```

- **.fill** : réservation d'emplacements mémoire pour stocker des données. Syntaxe :

```
| <étiquette> .fill <nb>, <taille>, <valeur>
```

Elle permet de réserver **<nb>** emplacements de **<taille>** octets et initialiser chaque emplacement avec la valeur **<valeur>**. Par défaut **<taille>** = 1 et **<valeur>** = 0.

Pseudo-instructions

Le rôle des **pseudo-instructions** est de libérer le programmeur des tâches fastidieuses de calcul pour déterminer la valeur d'un déplacement pour atteindre l'adresse d'une variable par exemple, ou de ne pas se soucier du codage d'une constante supérieure à 255. Ces pseudo-instructions sont remplacées, par l'assembleur, par de vraies instructions.

- Pseudo-instruction **ADR** : elle permet d'initialiser un registre de base avec l'adresse d'une variable ou d'un tableau. Syntaxe :

```
ADR rd, <étiquette>
```

Où **rd** est un registre parmi (r0-r14) et **<étiquette>** indique une adresse d'une variable ou d'un tableau.



INITIALISATION DE R3 AVEC L'ADRESSE DE LA VARIABLE VAR

```
ADR r3, var @ r3 ← adresse var
Instruction 2
Instruction 3
Instruction 4
var: .fill 1,4
```

- Pseudo-instruction **LDR** : elle permet de ne pas se soucier du codage des constantes (voir mode d'adressage immédiat) et de choisir n'importe quelle valeur. Syntaxe :

```
LDR rd, =<valeur_32>
```

Où **rd** indique un registre parmi r0 à r14 et **<valeur_32>** une constante sur 32 bits.

- Si **valeur_32** est de la forme $\text{immédiat}_8 \cdot 2^{2 \cdot \text{rotation_imm}}$ elle est transformée par l'assembleur en :

```
MOV rd, #immédiat_8, rotation_imm
```

- Sinon elle est transformée en :

```
ETIQ1 : LDR rd, [r15, #dep] @ rd ← Mem32[pc+dep]
...
ETIQ2 : .int <valeur_32>
```

Le déplacement (dep) = ETIQ2-pc = ETIQ2 - ETIQ1 + 4



Attention ne pas confondre la pseudo-instruction LDR avec l'instruction de chargement de mot LDR.

EXERCICES

7-1 Structures algorithmiques classiques

1) Le choix : Si...Alors... On considère l'algorithme suivant :

```
Si R0 > 0 Alors  
    ...  
Sinon  
    ...  
Finsi
```

Traduire cette forme algorithmique en assembleur.

2) La répétition : TantQue ... Faire ... FinTantQue. On considère l'algorithme suivant :

```
TantQue R0>0 Faire  
    ...  
FinTantQue
```

Traduire cette forme algorithmique en assembleur.

7-2 Manipulation des instructions de traitement des données

1) Somme des N premiers entiers. Écrivez un programme qui calcule, dans R0, la somme des N premiers entiers (N étant une constante définie dans le programme par un .equ)

2) Calculez une multiplication. Écrivez un programme qui calcule la multiplication de R0 par R1 (supposés contenir des nombres codés sur 16 bits) et range le résultat dans R2, sans utiliser l'instruction de multiplication. On supposera dans un premier temps que les deux nombres sont positifs. Comment modifier le programme pour prendre en compte les multiplications de nombres négatifs ?

3) Calculez une division. Écrivez un programme qui calcule la division de R0 (dividende) par R1 (diviseur) et range le résultat dans R2 (quotient) et R3 (reste). On supposera que le dividende et le diviseur sont positifs.

7-3 Calcul d'une somme d'éléments

Écrivez un programme qui calcule, dans R0, la somme des éléments d'un tableau. Comparez le nombre d'instructions nécessaires pour un tableau contenant 256 valeurs.

7-4 Tri d'un tableau

Écrivez un programme qui trie un tableau par ordre décroissant de ses éléments. Le tableau sera constitué de N éléments entiers relatifs codés sur 32 bits.

7-5 Tri d'un tableau

Reprenez l'exercice précédent pour le résoudre à l'aide d'un sous-programme.

1) Le sous-programme RechMax. Dans un premier temps, écrire un sous-programme RechMax qui recherche l'élément le plus grand dans un sous-tableau ainsi que l'adresse de cet élément. Ce sous-programme utilisera les paramètres suivants :

- Registre R0 (en entrée) : adresse de début du tableau
- Registre R1 (en entrée) : adresse de fin du tableau
- Registre R2 (en sortie) : valeur de l'élément le plus grand
- Registre R3 (en sortie) : adresse de l'élément le plus grand

2) Le programme principal. Écrivez un programme utilisant le sous-programme RechMax pour trier par ordre croissant un tableau contenant des entiers relatifs codés sur 32 bits représentant cette valeur entière. Les paramètres de ce sous-programme seront, en entrée, la valeur entière (registre R0) et l'adresse de la zone mémoire où stocker la chaîne de caractères résultant de l'opération de conversion (registre R1).

On précisera comment R14 doit être géré pour permettre l'appel d'un sous-programme dans un sous-programme.

SOLUTIONS

7-1 Structures algorithmiques classiques

1) Le choix : Si...Alors...

	CMP	R0,#0
	BGT	ALORS
SINON	...	
	...	
	B	FINSI
ALORS:	...	
	...	

```

FINSI:      ...
           ...

```

On peut aussi proposer la forme suivante, plus compacte, qui se base sur la négation de la condition :

```

           CMP      R0,#0
           BLE      SINON
ALORS:     ...
           ...
           B        FINSI
SINON:     ...
           ...
FINSI:     ...
           ...

```

2) La répétition : TantQue ... Faire ... FinTantQue

```

TQ:        CMP      R0,#0
           BGT      SUITE
           B        FTQ
SUITE:     ...
           ...
           B        TQ
FTQ:       ...

```

On peut aussi proposer la forme suivante, plus compacte, qui se base sur la négation de la condition :

```

TQ:        CMP      R0,#0
           BLE      FTQ
           ...
           ...
           B        TQ
FTQ:       ...

```

7-2 Manipulation des instructions de traitement des données

1) On considère que N contient la valeur 20. On a l'algorithme suivant :

```

Début
    SOM <- 0;
    I <- 1;
    Tant que I <= N Faire
        SOM <- SOM+I;
        I <- I+1;
    FTQ;
FIN.

```

On en déduit le programme suivant :

```

N          .equ      20          @ (.equ N,20)
          MOV        R0,#0       @ SOM <- 0
          MOV        R1,#1       @ I <- 1
TQ:        CMP        R1,#N      @ Compare I à N
          BHI        FTQ        @ Si I>N aller à FTQ
          ADD        R0,R0,R1    @ SOM <- SOM+I
          ADD        R1,R1,#1    @ I <- I+1
          B          TQ         @ aller à TQ
FTQ:       B          FTQ        @ aller à FTQ

```

2) L'algorithme le plus simple pour réaliser $C \leftarrow A \times B$ se base sur une succession d'additions :

```

Début
    C <- 0;
    Si A <>0 Alors
        Tant que B > 0 Faire
            C <- C+A;
            B <- B-1;
        FTQ;
    FSI;
FIN.

```

En supposant que A (demi-mot) est dans le registre $R0$, B (demi-mot) dans le registre $R1$ et C (mot) dans le registre $R2$, l'algorithme se traduit de la manière suivante :

```

          MOV        R2,#0       @ C <- 0
          CMP        R0,#0       @ compare A à 0
          BEQ        FIN        @ si A=0 aller à FIN
TQ:       CMP        R1,#0       @ compare B à 0
          BLS        FIN        @ si B<= 0 aller à FIN
          ADD        R2,R2,R0    @ C <- C+A
          SUB        R1,R1,#1    @ B <- B-1
          B          TQ         @ aller à TQ
FIN:      B          FIN        @ aller à Fin

```

On peut aussi remarquer que, dans le cas d'une multiplication binaire, le résultat de la multiplication de A par B est l'addition successive de A décalé d'un nombre de fois vers la gauche égal au rang des bits à 1 dans B .

On en déduit l'algorithme suivant :

```

Début
  C <- 0;
  Si A <>0 Alors
    Si B <>0 Alors
      I <- 0 ;
      Tant que I < 16 Faire
        Si B0 (bit rang 0 de B) = 1 Alors
          C <- C+A;
          FSI;
          Décaler B à droite d'un bit;
          Décaler A à gauche d'un bit;
        FTQ;
      FSI;
    FSI;
  FIN.

```

Le programme correspondant est :

```

N      .equ      16      @ (.equ N,16)
      MOV      R2,#0      @ C <- 0
      CMP      R0,#0      @ compare A à 0
      BEQ      FIN      @ si A = 0 aller à FIN
      CMP      R1,#0      @ compare B à 0
      BEQ      FIN      @ si B = 0 aller FIN
      MOV      R3,#0      @ I <- 0
MULT:  CMP      R3,#N      @ compare I à N
      BHS      FIN      @ si I >= N aller à FIN
      TST      R1, #0X1    # Test du bit 0 de B
      BCC      SHIFT      @ si B0=0 aller à SHIFT
      ADD      R2,R2,R0    @ C <-C+A
SHIFT: MOV      R1,R1,LSL #1 @ déc à dte de B
      MOV      R0,R0,LSL #1 @ déc à gche de A
      ADD      R3,R3,#1    @ I <- I+1
      B        MULT      @ aller à MULT
FIN:   B        FIN      @ aller FIN

```

On peut utiliser l'instruction d'affectation avec décalage et mise à jour : `MOVS R1,R1,RRX`. Lors de l'utilisation de cette instruction, le bit C du CPSR est mis à jour avec le bit issu du décalage (bit de rang 0 de R1). On obtient alors la forme suivante :

```

N      .equ      16      @ (.equ N,16)
      MOV      R2,#0      @ C <- 0
      CMP      R0,#0      @ compare A à 0
      BEQ      FIN      @ si A = 0 aller à FIN
      CMP      R1,#0      @ compare B à 0
      BEQ      FIN      @ si B = 0 aller FIN

```

```

      MOV      R3,#0           @ I <- 0
MULT:  CMP      R3,#N          @ compare I à N
      BHS      FIN            @ si I >= N aller à FIN
      MOVS     R1,R1,RRX       @ déc à dte de B
      BCC      SHIFT          @ si B0=0 aller à SHIFT
      ADD      R2,R2,R0        @ C <- -C+A
SHIFT:  MOV      R0,R0,LSL #1   @ déc à gche de A
      ADD      R3,R3,#1        @ I <- I+1
      B        MULT           @ aller à MULT
FIN:    B        FIN           @ aller FIN

```

Considérons maintenant que A et B sont des entiers relatifs. Nous allons multiplier les valeurs absolues, et rectifier le signe du résultat le cas échéant (si les signes des deux opérandes sont différents). L'algorithme est le suivant :

```

Début
  c <- 0;
  Si a <>0 Alors
    Si a > 0 Alors
      signe_a <- 0;
    Sinon
      signe_a <- 1;
      a <- -a;
  FSI;
  Si b <>0 Alors
    Si b > 0 Alors
      signe_b <- 0;
    Sinon
      signe_b <- 1;
      b <- -b;
  FSI;
  I <- 0 ;
  Tant que I < 16 Faire
    Si B0 (bit rang 0 de B) = 1 Alors
      C <- C+A;
    FSI;
    Décaler B à droite d'un bit;
    Décaler A à gauche d'un bit;
  FTQ;
  Si signe_a <> signe_b Alors
    c <- -c;
  FSI;
FSI;
FSI;
FIN.

```

On en déduit le programme suivant :

```

N          .equ      16          @ (.equ N,16)
          MOV        R2,#0        @ C <- 0
          CMP        R0,#0        @ compare A à 0
          BEQ        FIN          @ si A = 0 aller à FIN
          BLT        A_NEG        @ si A<0 aller à A_NEG
A_POS:     MOV        R4,#0        @ sinon signe_a <-0
          B          SUITE1        @ aller à SUITE1
A_NEG:     MOV        R4,#1        @ signe_a <- 1
          RSB        R0,R0,#0      @ A <- 0-A
SUITE1:    CMP        R1,#0        @ compare B à 0
          BEQ        FIN          @ si B = 0 aller FIN
          BLT        B_NEG        @ si B<0 aller à B_NEG
B_POS:     MOV        R5,#0        @ sinon signe_b <-0
          B          SUITE2        @ aller à SUITE2
B_NEG:     MOV        R5,#1        @ signe_b <- 1
          RSB        R1,R1,#0      @ B <- 0-B
SUITE2:    MOV        R3,#0        @ I <- 0
MULT:      CMP        R3,#N        @ compare I à N
          BHS        FIN          @ si I >= N aller à FIN
          MOVS       R1,R1,RRX      @ déc à dte de B
          BCC        SHIFT        @ si B0=0 aller à SHIFT
          ADD        R2,R2,R0      @ C <-C+A
SHIFT:     MOV        R0,R0,LSL #1 @ déc à gche de A
          ADD        R3,R3,#1      @ I <- I+1
          B          MULT          @ aller à MULT
          CMP        R4,R5        @ compare signe A et B
          BEQ        FIN          @ si égaux aller à FIN
          RSB        R2,R2,#0      @ C <-0-C
FIN:       B          FIN          @ aller à FIN

```

3) On cherche Q et R réfifiant, pour A et B donnés : $A = B \times Q + R$.

La solution la plus simple pour traiter ce problème consiste à effectuer une succession de soustractions, comme le montre cet algorithme :

```

Début
    Q <- 0;
    Tant que A >= B Faire
        A <- A-B;
        Q <- Q+1;
    FTQ;
    R <- A;
FIN.

```

Le programme correspondant est le suivant :

```

      MOV      R2,#0          @ Q <- 0
TQ:   CMP      R0,R1          @ compare a à b
      BLO      FTQ            @ si b<0 aller à FIN
      SUB      R0,R0,R1       @ a <- a-b
      ADD      R2,R2,#1       @ Q <- Q+1
      B        TQ             @ aller à TQ
FIN:   MOV      R3,R0          @ R <- a
ATT:   B        ATT           @ aller à ATT (fin)

```

D'autres méthodes plus rapides existent (à base de recherche dichotomique) mais nous ne les présenterons pas ici faute de place.

7-3 Calcul d'une somme d'éléments

En utilisant un formalisme tableau, l'algorithme est le suivant :

```

Début
  SOM <- 0;
  I <- 0;
  Tant que I < N Faire
    SOM <- SOM+TAB[I];
    I <- I+1;
  FTQ;
FIN.

```

Cette forme algorithmique est indépendante de la taille des éléments du tableau. L'implémentation, par contre, n'est pas la même selon la taille des éléments. Dans le cas d'octets :

```

      ADR      R0, TAB         @ R0 <- adr TAB
      MOV      R2,#0          @ SOM <- 0
      MOV      R1,#0          @ I <- 0
TQ:   CMP      R1,#N          @ compare R1 à N
      BHS      FTQ            @ si >=0 aller à FTQ
      LDRB     R3,[R0,R1]      @ R3 <- TAB[I]
      ADD      R2,R2,R3        @ SOM <- SOM+TAB[I]
      ADD      R1,R1,#1        @ I <- I+1
      B        TQ             @ aller à TQ
FTQ:   B        FTQ           @ Fin
@ Définition des paramètres
N      .equ     ...           @ (.equ N,...) valeur de N
TAB     .byte   elt1, elt2,... @ N val. de 1 octet

```

Si on utilise des demi-mots, il faut indexer le tableau de 2 en 2 afin de respecter la taille des éléments du tableau :

```

                ADR      R0, TAB      @ R0 <- adr TAB
                MOV      R2,#0        @ SOM <- 0
                MOV      R1,#0        @ I <- 0
TQ:             CMP      R1,#2*N      @ compare R1 à 2*N
                BHS      FTQ          @ si>=0 aller à FTQ
                LDRH     R3,[R0,R1]   @ R3 <- TAB[I]
                ADD      R2,R2,R3     @ SOM <-SOM+TAB[I]
                ADD      R1,R1,#2     @ I <- I+2
                B        TQ          @ aller à TQ
FTQ:           B        FTQ          @ Fin
@ Définition des paramètres
N              .equ      ...         @ (.equ N,...) valeur de N
TAB            .hword    elt1, elt2,... @ N val. de 2 octets

```

Il faut exécuter 1539 instructions pour calculer la somme d'un tableau de 256 valeurs avec cette solution.

Cette méthode étant peu intuitive, on peut utiliser le mode d'adressage indexé avec décalage :

```

                ADR      R0, TAB      @ R0 <- adr TAB
                MOV      R2,#0        @ SOM <- 0
                MOV      R1,#0        @ I <- 0
TQ:             CMP      R1,#N        @ compare R1 à N
                BHS      FTQ          @ si>=0 aller à FTQ
                LDRH     R3,[R0,R1 LSL #1] @ R3 <- TAB[I]
                ADD      R2,R2,R3     @ SOM <-SOM+TAB[I]
                ADD      R1,R1,#1     @ I <- I+1
                B        TQ          @ aller à TQ
FTQ:           B        FTQ          @ Fin

```

Pour un tableau de mots, le principe est le même :

```

                ADR      R0, TAB      @ R0 <- adr TAB
                MOV      R2,#0        @ SOM <- 0
                MOV      R1,#0        @ I <- 0
TQ:             CMP      R1,#N        @ compare R1 à N
                BHS      FTQ          @ si>=0 aller à FTQ
                LDRH     R3,[R0,R1 LSL #2] @ R3 <- TAB[I]
                ADD      R2,R2,R3     @ SOM <-SOM+TAB[I]
                ADD      R1,R1,#1     @ I <- I+1
                B        TQ          @ aller à TQ
FTQ:           B        FTQ          @ Fin

```

Il faut aussi exécuter 1539 instructions pour calculer la somme d'un tableau de 256 valeurs avec cette solution.

En utilisant un formalisme pointeur, l'algorithme devient :

```

Début
  A ← TAB
  SOM ← 0;
  I ← 0;
  Tant que I < N Faire
    SOM ← SOM+(A);
    I ← I+1;
    A ← A+1
  FTQ;
FIN.

```

Le programme correspondant s'écrit :

	ADR	R0, TAB	@ A ← adr TAB
	MOV	R2,#0	@ SOM ← 0
	MOV	R1,#0	@ I ← 0
TQ:	CMP	R1,#N	@ compare R1 à N
	BHS	FTQ	@ si>=0 aller à FTQ
	LDR	R3,[R0]	@ R3 ← (A)
	ADD	R2,R2,R3	@ SOM ←SOM+(A)
	ADD	R1,R1,#1	@ I ← I+1
	ADD	R0,R0,#4	@ A ← A+4
	B	TQ	@ aller à TQ
FTQ:	B	FTQ	@ Fin

Il faut aussi exécuter 1795 instructions pour calculer la somme d'un tableau de 256 valeurs avec cette solution.

Notons qu'on peut utiliser un adressage post-incrémenté pour R0 :

	ADR	R0, TAB	@ A ← adr TAB
	MOV	R2,#0	@ SOM ← 0
	MOV	R1,#0	@ I ← 0
TQ:	CMP	R1,#N	@ compare R1 à N
	BHS	FTQ	@ si>=0 aller à FTQ
	LDR	R3,[R0], #4	@ R3 ← (A) et A ← A+4
	ADD	R2,R2,R3	@ SOM ←SOM+(A)
	ADD	R1,R1,#1	@ I ← I+1
	B	TQ	@ aller à TQ
FTQ:	B	FTQ	@ Fin

Ici encore, il faut aussi exécuter 1539 instructions pour calculer la somme d'un tableau de 256 valeurs avec cette solution.

Enfin, la condition de boucle peut être écrite en utilisant un pointeur vers la fin du tableau. On a alors l'algorithme suivant :

```

Début
  A <- TAB;
  SOM <- 0;
  Tant que A <= FTAB Faire
    SOM <- SOM+(A);
    A <- A+1
  FTQ;
FIN.

```

Qui peut être implémenté de la manière suivante :

```

          ADR      R0, TAB          @ A <- adresse TAB
          ADR      R1, FTAB         @ adresse fin boucle
          MOV      R2,#0            @ SOM <- 0
TQ:        CMP      R0,R1           @ compare R0 à FTAB
          BHI      FTQ              @ si>0 aller à FTQ
          LDR      R3,[R0],#4       @ R3 <- (A) et A <- A+4
          ADD      R2,R2,R3         @ SOM <-SOM+(A)
          B        TQ               @ aller à TQ
FTQ:       B        FTQ             @ Fin
@ Définition des paramètres
N          .equ     ...             @ (.equ N,...) valeur de N
TAB        .int     elt1, elt2,...  @ N val. de 2 octets
FTAB       .equ     *-4

```

Ici, l'étiquette FTAB pointe vers l'adresse courante (*) moins 4, soit vers le dernier élément du tableau. Lorsque cette adresse est atteinte, le programme est terminé. 1 283 instructions doivent être exécutées avec cette version pour calculer la somme des éléments d'un tableau de 256 valeurs, ce qui en fait la solution la plus efficace.

7-5 Tri d'un tableau

L'algorithme utilisé est le suivant :

```

Début
  I <- 0 ;
  Tant que I < N-1 Faire
    ValMax = TAB[I] ;
    IMax <- I;
    ICourant <- I+1;
    Tant que ICourant < N Faire
      ValCourant = TAB[ICourant];
      Si ValCourant > ValMax Alors
        ValMax <- ValCourant;
        IMax <- ICourant;
      FSi;
    ICourant <- ICourant + 1;

```

```

        FTQ;
        ValCourant <- TAB[I];
        TAB[I] <- ValMax;
        TAB[IMax] <- ValCourant;
        I <- I+1;
    FTQ;
FIN.

```

Le programme correspondant s'écrit :

	ADR	R5,TAB	@ R5 <- adr TAB
	MOV	R0,#0	@ I <- 0
TQ1:	CMP	R0,#N-1	@ compare I à N-1
	BHS	FTQ1	@ si >= aller FTQ1
	LDR	R1,[R5,R0,LSL #2]	@ ValMax <- TAB[I]
	MOV	R2,R0	@ IMax <- I
	ADD	R3,R0,#1	@ ICour <- I+1
TQ2:	CMP	R3,#N	@ compare ICour à N
	BHS	FTQ2	@ si >= aller FTQ2
	LDR	R4,[R5, R3,LSL #2]	@ ValCour <- TAB[ICour]
	CMP	R4,R1	@ Comp. ValCour à ValMax
	BLE	FSI	@ si <= aller FSI
	MOV	R1,R4	@ ValMax <- ValC
	MOV	R2,R3	@ IMax <- ICour
FSI:	ADD	R3,R3,#1	@ ICour <- ICour+1
	B	TQ2	@ aller à TQ2
FTQ2:	LDR	R4,[R5,R0,LSL #2]	@ ValCour <- TAB[I]
	STR	R1,[R5,R0,LSL #2]	@ TAB[I] <- ValMax
	STR	R4,[R5,R2,LSL #2]	@ TAB[IMax] <- ValCour
	ADD	R0,R0,#1	@ I <- I+1;
	B	TQ1	@ aller à TQ1
FTQ1:	B	FTQ1	@ fin

7-6 Tri d'un tableau

1) L'algorithme du sous-programme RechMax est le suivant :

```

Début
    AdrMax <- AdrFTab;
    ValMax <- (AdrMax);
    AdrFTab <- AdrFTab - 1;
    Tant que AdrFTab >= AdrDTab Faire
        ValCour <- (AdrFTab);
        Si ValCour > ValMax Alors
            ValMax <- ValCour;
            AdrMax <- AdrFTab;
        FSi;
        AdrFTab <- AdrFTab - 1;
    FTQ;
FIN.

```

En assembleur, nous obtenons :

```

RechMax:  STMFID  R13!, {R1,R4}    @ sauve contexte
          MOV     R3,R1            @ AdrMax <- AdrFTab
          LDR     R2,[R1]          @ ValMax <- (AdrMax)
          SUB     R1,R1,#4          @ AdrFTab <- AdrFTab-4
LoopMax:  CMP     R1,R0            @ comp. AdrFTab à AdrDTab
          BLO     FinRech          @ si < aller FTQ
          LDR     R4,[R1]          @ ValCourante <- (AdrFTab)
          CMP     R4,R2            @ Compare ValCour à ValMax
          BLE     Next             @ si <= aller à Next
          MOV     R2,R4            @ ValMax <- ValCour
          MOV     R3,R1            @ AdrMax <- AdrFTab
Next:     SUB     R1,R1,#4          @ AdrFTab <- AdrFTab-1
          B       LoopMax          @ aller à TQ
FinRech:  LDMFID  R13!, {R1,R4}    @ restitution contexte
          MOV     R15,R14          @ restitution adr retour

```

On peut aussi utiliser la sauvegarde / restauration du contexte pour implémenter le retour de sous-programme :

```

RechMax:  STMFID  R13!, {R1,R4, R14} @ sauve contexte
          MOV     R3,R1            @ AdrMax <- AdrFTab
          LDR     R2,[R1]          @ ValMax <- (AdrMax)
          SUB     R1,R1,#4          @ AdrFTab <- AdrFTab-4
LoopMax:  CMP     R1,R0            @ comp. AdrFTab à AdrDTab
          BLO     FinRech          @ si < aller FTQ
          LDR     R4,[R1]          @ ValCourante <- (AdrFTab)
          CMP     R4,R2            @ Compare ValCour à ValMax
          BLE     Next             @ si <= aller à Next
          MOV     R2,R4            @ ValMax <- ValCour
          MOV     R3,R1            @ AdrMax <- AdrFTab
Next:     SUB     R1,R1,#4          @ AdrFTab <- AdrFTab-1
          B       LoopMax          @ aller à TQ
FinRech:  LDMFID  R13!, {R1,R4, R15} @ rest. contexte

```

2) On utilise l'algorithme suivant :

```

Début
  AdrDTab <- TAB;
  AdrFTab <- FTAB;
  Tant que AdrFTab > AdrDTab Faire
    RechMax;
    Temp <- (AdrFTab);
    (AdrFTab) <- ValMax;
    (AdrMax) <- Temp;
    AdrFTab <- AdrFTab - 1;
  FTQ;
FIN.

```

Le programme correspondant est :

```

Main:      ADR      R0,TAB      @  AdrDTab<- TAB
           ADR      R1, FTAB    @  AdrFTab <- FTAB
TQ:        CMP      R1,R0      @  compare AdrFTab à AdrDTab
           BLS      FTQ        @  si < aller FTQ
           BL       RechMax     @  appel sous prog RechMax
           LDR      R4,[R1]     @  Temp <- (AdrFTab)
           STR      R2,[R1]     @  (AdrFTab) <- ValMax
           STR      R4,[R3]     @  (AdrMax) <- Temp
           SUB      R1,R1,#4    @  AdrFTab <- AdrFTab- 4
           B        TQ         @  aller à TQ
FTQ:       B        FTQ        @  fin
@ Définition des paramètres
N          .equ     ...        @ (.equ N,...) valeur de N
TAB        .int     12,-145,75,-56,-45896,12456,...
FFTAB:     @ fin TAB+4
FTAB       .equ     FFTAB-4    @ fin TAB

```



Index

A

absorption 45
addition 7
additionneur complet 78
adresse 144
algèbre de Boole 43
 dualité de l' 44
ASCII 31
associativité 45
asynchrone 99

B

barrière « trois états » 120
bascule 142
bit 26
 de parité 32
BUS 116

C

chemin critique 73
circuit
 combinatoire 71
 d'interface 158
codages optimaux 35
code
 à l'excès 30
 correcteur d'erreur 31
 de comptage 110
 de Gray 57
 de Hamming 32
 de Huffman 35
 détecteur d'erreur 31
 opération 162
 réfléchi 57

coïncidence 56
combinaison binaire 48
complément
 à deux 27
 logique 44
compteur 109
conditionnelle 183

D

décade 110
décalage
 arithmétique 172
 logique à droite 172
 logique à gauche 171
décodeur 73
 d'instructions 162
délai 97
 de propagation 94
demi-additionneur 78
densité d'intégration 143
dépassement de capacité 26
DIMM 153
directives d'assemblage 189
division 10

E

échantillonne 97
EEPROM 156
éléments absorbants 45
entrée de validation 73
ET logique 44
évènement 97
exposant 29

F

fenêtre 98
 flot d'exécution 183
 fonction booléenne 46
 forme
 canonique conjonctive 52
 canonique disjonctive 50
 front
 descendant 97
 montant 97

G

groupe logique complet 46

H

horloge 97
 hiérarchie 189

I

idempotence 45
 impulsion 96, 97
 inconditionnelle 184
 instructions 159
 traitement des données 170
 involution 45

J

jeu d'instructions 162

L

Lagrange 50
 deuxième forme de 52
 première forme de 50
 langage d'assemblage 157, 167
 LIFO 182
 littéral 49
 logigramme 47
 logique négative 73

M

maintien 98
 maître/esclave 103
 mantisse 29
 matrice de parité 32
 maxterme 51

mémoire

à accès aléatoires 142
 à accès séquentiels 142
 à bulles magnétiques 142
 à charges couplées 142
 à domaines magnétiques 142
 associatives 142
 flash 156
 volatiles 142
 minterme 49
 modes d'adressages 170
 monal 51
 adjacent 56
 canonique 51
 monôme 49
 adjacent 56
 canonique 49
 multiplexeur 75
 multiplication 7

N

Nand 53
 niveau 96
 nombre fractionnaire 4
 Nor 54
 norme IEEE754 30
 notation scientifique 29
 numération 3

O

octet 26
 organisation
 big endian 168
 little endian 169
 oscillateur 162
 OU
 câblé 120
 logique 44

P

parallèle 116
 parité
 impaire 31
 paire 31
 périphériques 158
 pile 181