1 SYNTHÈSE DE FONCTIONS COMBINATOIRES

La synthèse de fonctions combinatoires consiste, à partir d'une table de vérité ou d'une expression booléenne, à spécifier les opérateurs matériels permettant l'implémentation de la table ou de l'expression correspondante. Il existe 3 grandes méthodes de synthèses de fonctions combinatoires, correspondant au niveau de complexité des opérateurs logiques utilisés comme éléments de base.

1.1 LA LOGIQUE “ANARCHIQUE”

La logique dite "anarchique" consiste à "implanter" la fonction booléenne à l'aide d'un ensemble minimum de portes de base : Et, Ou, inverseurs, ou NAND ou NOR....Cette méthode a été développée à l'époque des circuits logiques à faible niveau ou moyen niveau d'intégration (circuit SSI et MSI).
On commence par simplifier l'expression complète déduite de la forme disjonctive normale pour obtenir un nombre minimum de portes, avec un nombre minimum d'entrées pour ces portes.
Pour un nombre réduit d’entrées, on peut utiliser le diagramme de Karnaugh (voir cours précédent). Au-delà de 4 entrées, la représentation graphique devient complexe, il est difficile de mettre en évidence les symétries, et la méthode devient inutilisable. Dans ce cas, il fait utiliser des méthodes plus élaborées, comme celle de Quine - Mc Cluskey, qui est la base des heuristiques utilisées dans un certain nombre de logiciels spécialisés (Espresso, Mc Boole). D'autres logiciels utilisent des méthodes de réécriture d'expressions.
Il faut souligner que le problème de simplification d'expressions booléennes se pose, soit pour des expressions très simples à très peu de variables pour lesquelles le diagramme de Karnaugh est amplement suffisant, soit pour des expressions complexes à grand nombre de variables pour lesquelles les logiciels spécialisés sont inévitables.

1.2 LA LOGIQUE “STRUCTURÉE”

Toute fonction booléenne, aussi compliquée soit-elle, peut s'exprimer sous la forme d'une union de termes produit. Il est donc possible de spécifier une structure universelle, capable de réaliser toutes les unions possibles de tous les termes produit nécessaires.
La logique structurée consiste donc à "implanter" la fonction dans une structure régulière, prédéfinie à l'avance, et dont la surface ne dépendra pas de la configuration particulière des 0 et des 1 propre à une fonction, mais uniquement du nombre de variables d'entrées (structure ROM), ou du nombre de variables d'entrée et de termes produit de la fonction (structure PLA ou PAL).
Contrairement à la logique aléatoire, qui est caractéristique des circuits intégrés à très faible niveau d'intégration (SSI ou MSI), la logique structurée est caractéristiques des circuits intégrés à très forte densité d'intégration (VLSI). Pour ces circuits, le critères essentiel est la surface minimale : pour un nombre donné d'entrées et de sorties, la surface dépend essentiellement des facilités de connexions globales entre les différents points du circuits. C'est donc le nombre de connexions d'entrée et de sortie utilisables qui est le facteur essentiel, et non le nombre de termes produit et le nombre de variables par terme produit.

1.2.1 Structure ROM.

Elle se décompose en 2 parties :
- un décodeur (ou générateur complet de termes produit) dont le schéma fonctionnel est donné en Figure 1. La Figure 2 montre l'implantation d'un décodeur deux entrées quatre sorties avec des portes élémentaires.
- un Ou logique des termes produit pour lesquels la fonction a pour valeur 1.

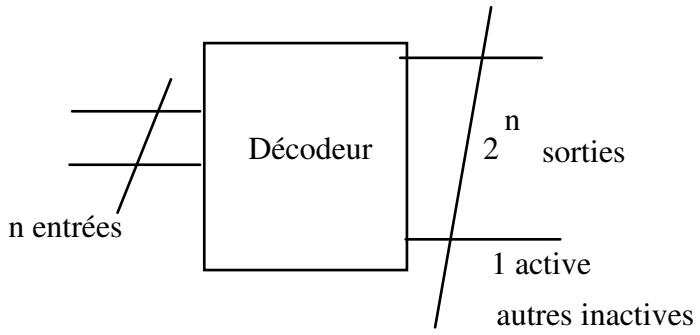

Figure 1 : Schéma fonctionnel du décodeur

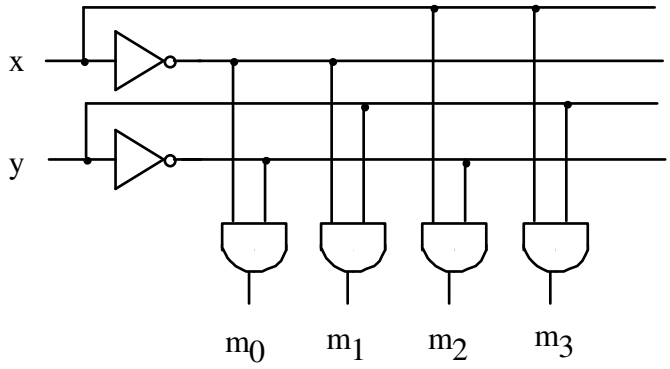

Figure 2 : Réalisation du décodeur 2 entrées 4 sorties avec des portes logiques

La ROM est connue généralement comme une mémoire à lecture seulement. C'est en fait un opérateur combinatoire. Le nombre de bits en sortie correspond au nombre de fonctions logiques différentes implantées. Le nombre de bits d'adresse correspond au nombre de variables des fonctions logiques.
Pour une table de vérité, la ROM associe à chaque sortie du décodeur une valeur (1 ou 0). Lorsqu'il y a n sorties, un mot de n bits est associé à chaque sortie du décodeur, ce qui permet d'implanter n tables de vérité.
Pour l'implantation de fonctions logiques, la ROM présente l'inconvénient d'utiliser un décodeur complet alors que les fonctions logiques à grand nombre d'entrées n'utilisent généralement qu'un nombre réduit de termes produit : en d'autres termes, la fonction a beaucoup plus de 0 que de 1. Il est possible d'utiliser uniquement un décodeur partiel, ce qui est fait dans les réseaux logiques programmables (ou PLA)

1.2.2 Structures PLA et PAL.

Un PLA est constitué de 2 demi-PLA
- Le demi-PLA Et qui est un générateur partiel de termes produit. On ne génère que ceux qui sont nécessaires pour implanter la fonction.
- Le demi-PLA Ou qui réalise le Ou logique des termes produit pour lesquels la fonction a pour valeur 1.

La Figure 3 donne un exemple de PLA. Le demi-PLA des ET implante les termes produits nécessaires pour la réalisation des sorties. Les termes produit sont réalisés par des ET distribués, un point sur une ligne signifiant la participation de la variable (complémentée ou non) à la réalisation du terme produit. Les sorties sont réalisées par le demi-PLA des OU, une croix sur une ligne signifiant que le terme produit correspondant participe à la réalisation de la sortie.
Un PLA est donc caractérisé par son nombre d'entrées, son nombre de sorties et le nombre de termes produit utilisables. Les demi-PLA Et et Ou sont programmables.
La structure appelée PAL est un PLA simplifié, pour lequel seul la partie Et est programmable (choix des termes produit) alors que le Ou logique des termes produit est précâblé.
PLA et PAL sont largement utilisés dans les composants logiques programmables complexes (CPLD pour Complex Programmable Logic Devices) qui sont l'une des classes de composan ts logiques programmables.

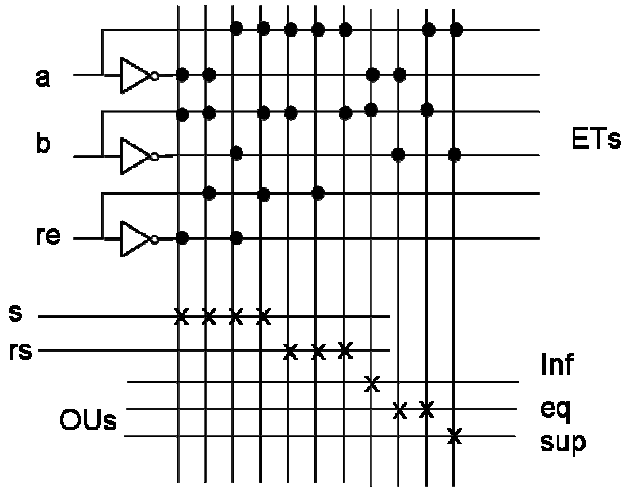

Figure 3 : Exemple de PLA

1.2.3 Multiplexeurs

Un multiplexeur est un opérateur logique à 2^n entrées et une sortie contrôlé par n fils de commande. L'entrée n° i est reliée à la sortie si la commande correspond à i codé sur n bits. Les schémas logiques des multiplexeurs à 2 et 4 entrées sont montrés en Figure 4.
Les équations logiques des multiplexeurs 2 entrées et 4 entrées sont respectivement

$$s = e_0.\bar{c} + e_1.c$$
$$s = e_0.\bar{c_1}.\bar{c_0} + e_1.\bar{c_1}.c_0 + e_2.c_1.\bar{c_0} + e_3.c_1.c_0 \quad \textbf{(F1)}$$

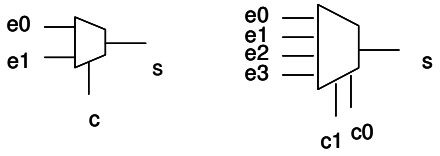

Figure 4 : Schémas logiques des multiplexeurs 2 et 4 entrées

Un multiplexeur 4 entrées peut être implanté directement à l'aide de portes ET-OU-INV ou NAND en utilisant l'équation ci-dessus ou avec des portes multiplexeurs 2 entrées comme indiqué en Figure 5. Ceci peut bien évidemment être étendu à 8, 16… 2^N entrées.

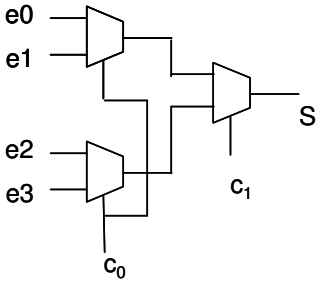

Figure 5 : Multiplexeur 4 entrées à partir de mux 2 entrées

Un multiplexeur N entrées peut être utilisé comme un LUT-N (voir 1.2.4). En utilisant les entrées de commande comme entrées de la table de vérité, et les entrées du multiplexeur comme les valeurs de la fonction pour chaque terme produit, on peut implanter n'importe quelle table de vérité. Par exemple, la Figure 6 montre l'implémentation de la fonction Ou exclusif avec un multiplexeur 4 entrées.

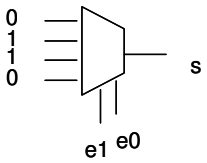

Figure 6 : Ou exclusif implanté avec un multiplexeur

On peut des fonctions logiques quelconques en utilisant des multiplexeurs et des portes logiques de base. Par exemple, la fonction $f(e_2,e_1,e_0) = \sum m(3,5,6)$ peut être implémentée avec un multiplexeur 4 entrées et un inverseur, comme le montre la Figure 7.

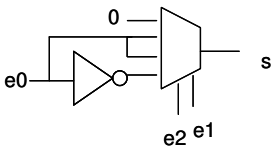

Figure 7 : Implémentation de $f(e_2,e_1,e_0) = \sum m(3,5,6)$

1.2.4 Les tables préprogrammées

Les tables préprogrammées (LUT pour Look-up Tables) sont des blocs logiques de base implantant à l'aide de mémoires SRAM des tables de vérité de 2, 3 ou 4 variables. Un LUT-2 (table de recherche à 2 entrées) peut donc implanter n'importe laquelle des 16 différentes fonctions logiques de 2 entrées. La Figure 8 montre l'implémentation d'une table de recherche pour une fonction à trois entrées. La mémoire SRAM implante une table de vérité à trois entrées, où chaque cellule mémoire contient la valeur de la fonction pour un des termes produit.
Les LUT sont utilisés dans les réseaux de portes programmables, (FPGA pour Field Programmable Gate Arrays), qui sont la seconde grande classes de réseaux logiques programmables. Dans les FPGA, les LUT sont interconnectés par des connexions programmables verticales et horizontales.

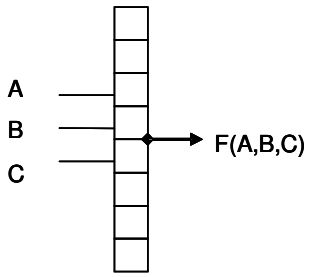

Figure 8 : Implémentation d'un LUT-3

La synthèse avec LUT introduit différents critères d'optimisation. Alors qu'avec la logique «anarchique», l'objectif est de minimiser le nombre de portes et le nombre d'entrées, dans la synthèse avec LUT, l'objectif est de minimiser le nombre de LUT. Il faut souligner qu'avec des LUT-n (à n entrées), toutes les fonctions logiques à n entrées sont équivalentes et s'implantent avec un seul LUT. Notamment, les fonctions de type Ou exclusif, difficiles à implanter sous forme ET-OU ou sous forme NAND-NAND sont strictement équivalentes à toutes les autres fonctions avec des LUT.
Un exemple est fourni pour l'implantation simultanée des fonctions logiques $f(e_3,e_2,e_1,e_0) = \sum m(1,5,8,9,10,12)$ et $g(e_3,e_2,e_1,e_0) = \sum m(0,1,2,5,6,9,14)$ avec le nombre minimal de tables de correspondance à 3 entrées (LUT-3).
Les fonctions f et g ont 1,5,9 en commun.
On utilise
A $= \sum m(1,5,9)$: partie commune à f et g.
B $= \sum m(8,10,12)$: partie de f sans les termes de A
C $= \sum m(0,2,6,14)$: partie de g sans les termes de A.

Soit $A = \overline{e_3}.\overline{e_2}.\overline{e_1}.e_0 + \overline{e_3}.e_2.\overline{e_1}.e_0 + e_3.\overline{e_2}.\overline{e_1}.e_0 = (\overline{e_3}.\overline{e_2}.e_1 + \overline{e_3}.e_2.\overline{e_1} + e_3.\overline{e_2}.\overline{e_1}).e_0 = a.e_0$

$B = e_3.\overline{e_2}.\overline{e_1}.\overline{e_0} + e_3.\overline{e_2}.e_1.\overline{e_0} + e_3.e_2.\overline{e_1}.\overline{e_0} = (e_3.\overline{e_2}.e_1 + e_3.\overline{e_2}.e_1 + e_3.e_2.\overline{e_1}).\overline{e_0} = b.\overline{e_0}$

$C = \overline{e_3}.\overline{e_2}.\overline{e_1}.\overline{e_0} + \overline{e_3}.\overline{e_2}.e_1.\overline{e_0} + \overline{e_3}.e_2.\overline{e_1}.\overline{e_0} + e_3.e_2.\overline{e_1}.\overline{e_0} = (\overline{e_3}.\overline{e_2}.e_1 + \overline{e_3}.\overline{e_2}.e_1 + \overline{e_3}.e_2.e_1 + e_3.e_2.e_1).\overline{e_0} = c.\overline{e_0}$

$f = A + B = a.e_0 + b.\overline{e_0}$

$g = A + C = a.e_0 + c.\overline{e_0}$

L'implantation est montrée en Figure 9

1.3 LA LOGIQUE EN TRANCHES

Elle consiste à réaliser des opérateurs n bits à l'aide d'opérateurs 1 bit en définissant les règles d'assemblage.
Un cas typique est celui des opérateurs arithmétiques. Leur caractéristique essentielle est le traitement par bit (ou bloc de bits) et la propagation des retenues. Ils sont le domaine type d'utilisation de la logique en tranches.
Nous traitons le cas des additionneurs.

1.3.1 Le traitement d'une tranche de 1 bit

L'addition de deux bits a_i et b_i avec la retenue entrante r_{i-1} fournit une somme S_i et une retenue r_i, selon la Table 1. Le schéma logique de l'additionneur correspondant, appelé additionneur 1 bit, est donné en Figure 10.

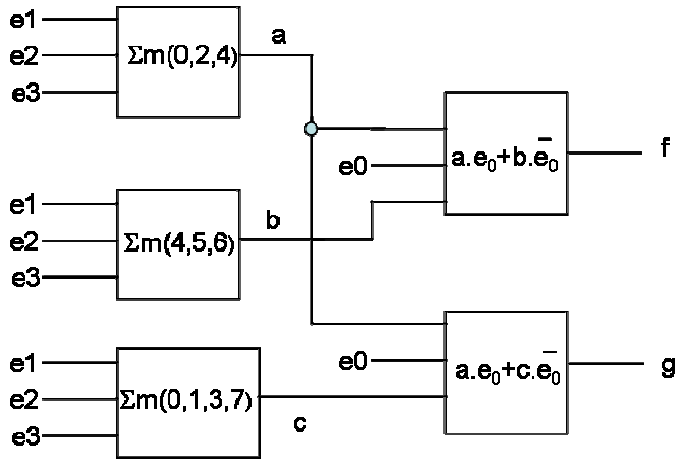

Figure 9 : Exemple d'implantation de fonction avec des LUT à trois entrées

r_{i-1}	a_i	b_i	r_i	S_i
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

Table 1 : additionneur 1 bit

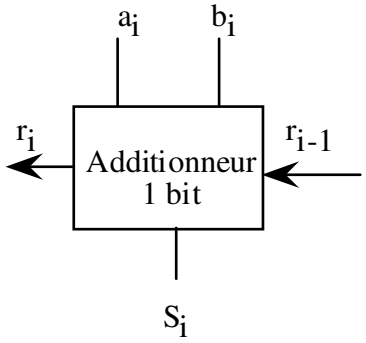

Figure 10 : Additionneur 1 bit

L'additionneur 1 bit peut être implanté avec des portes logiques, par exemple de type NAND. D'après la Table 1, on a les relations :

$$S_i = \overline{r_{i-1}}.\overline{a_i}.b_i + \overline{r_{i-1}}.a_i.\overline{b_i} + r_{i-1}.\overline{a_i}.\overline{b_i} + r_{i-1}.a_i.b_i$$

$$r_i = a_i.b_i + r_{i-1}.a_i + r_{i-1}.b_i$$

Un schéma d'implantation avec portes NAND est donné en Figure 11. Si le temps de propagation de la porte NAND est t_p, les temps de retard sont respectivement 3 t_p pour S_i et 2 t_p pour r_i.

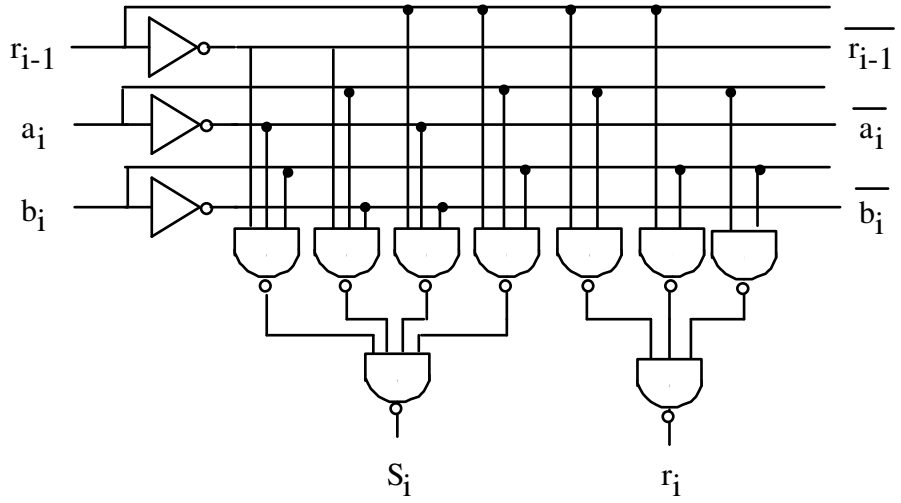

Figure 11 : Réalisation de l'additionneur 1 bit avec des portes NAND

Le schéma de la Figure 11 correspond à l'implémentation directe de la table de vérité de l'additionneur 1 bit.
Une autre implémentation utilise des portes XOR et des portes NAND. Elle découle des équations logiques suivantes

$$S_i = r_{i-1}.\overline{a_i}.\overline{b_i} + \overline{r_{i-1}}.a_i.\overline{b_i} + \overline{r_{i-1}}.\overline{a_i}.b_i + r_{i-1}.a_i.b_i = \overline{r_{i-1}}.(\overline{a_i}.b_i + a_i.\overline{b_i}) + r_{i-1}(.\overline{a_i}.\overline{b_i} + .a_i.b_i)$$

$$S_i = \overline{r_{i-1}}.(a_i \oplus b_i) + r_{i-1}(\overline{a_i \oplus b_i}) = r_{i-1} \oplus a_i \oplus b_i)$$

$$r_i = a_i.b_i + r_{i-1}.a_i + r_{i-1}.b_i = a_i.b_i + r_{i-1}(.a_i + b_i) = a_i.b_i + r_{i-1}(.a_i \oplus b_i)$$

$$r_i = \overline{\overline{a_i.b_i} + \overline{r_{i-1}(.a_i \oplus b_i)}}$$

L'implantation est donnée par la figure Figure 12

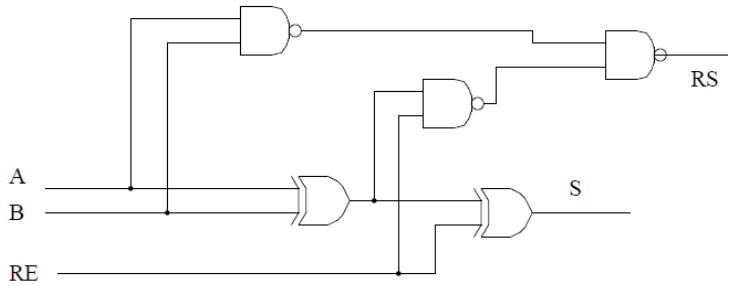

Figure 12 : Réalisation de l'additionneur 1 bit avec portes XOR et NAND

Certaines implémentations tiennent compte de la manière de réaliser des portes complexes en technologie MOS, notamment des portes Et-NOR, Ou-Et-NOR, etc..
Par exemple, le complément de la retenue peut s'écrire

$\overline{r_i} = \overline{a_i b_i + r_{i-1} a_i + r_{i-1} b_i}$

D'autre part, d'après la Table 1, on constate que

$S_i = \overline{r_i}(a_i+b_i+r_{i-1}) + a_i b_i r_{i-1}$

L'implémentation correspondante est donnée en Figure 13.

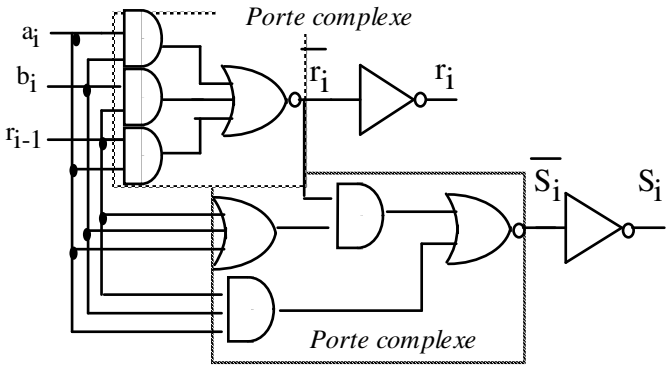

Figure 13 : Implémentation de l'additionneur 1 bit avec portes complexes

La manière de réaliser des portes complexes avec la technologie CMOS sera vue dans un cours ultérieur.
L'additionneur 1 bit est la brique de base pour la constitution d'un additionneur n bits. Les différentes réalisations diffèrent selon la manière de propager la retenue.

1.3.2 L'additionneur n bits à propagation simple de retenue.

L'additionneur n bits peut être réalisé par juxtaposition de n additionneurs 1 bit (Figure 11), avec propagation simple de la retenue, comme le montre la Figure 14. Pour le calcul de la retenue de sortie, il n'y a pas d'inverseur entre les entrées et la sortie et le temps de calcul de la retenue est $2t_p$ si tp est le temps de traversée d'un NAND. Le temps de propagation de la retenue est $2n\ t_p$ pour n étages. La sortie S_{n-1} est obtenue $(2n + 1)\ t_p$ après l'arrivée de la retenue d'entrée r_{-1}.

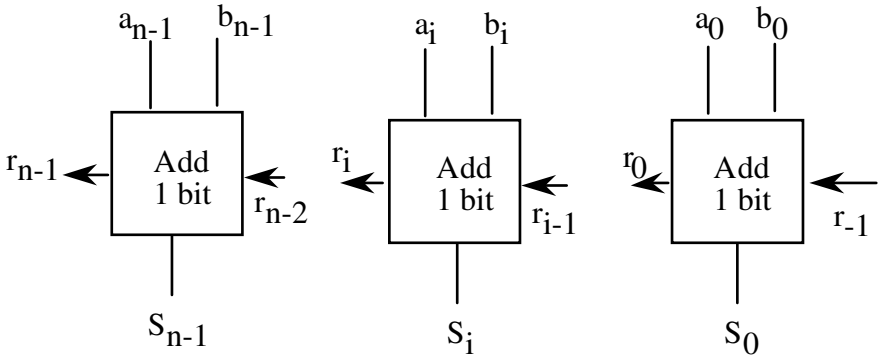

Figure 14 : Additionneur à propagation de retenue

1.3.3 Les propagations rapides de retenue.

La propagation de retenue constituant le chemin critique pour obtenir la retenue de sortie et les sorties de poids fort, des techniques permettant d'obtenir plus rapidement les retenues sont indispensables pour réaliser des additionneurs rapides

1.3.3.1 Le mécanisme de retenue anticipée

D'après la Table 1, l'expression de r_i en fonction de r_{i-1} peut être réécrite sous l'une des deux formes :

$$r_i = a_i.b_i + r_{i-1}\ (a_i+b_i) \quad (F2)$$

$$r_i = a_i.b_i + r_{i-1}\ (a_i \oplus b_i) \quad (F3)$$

En posant $G_i = a_i.b_i$ et $P_i = a_i+b_i$ ou $P_i = a_i \oplus b_i$ où G_i et P_i sont respectivement les fonctions génération et propagation de retenue, la retenue de sortie peut être réécrite sous la forme $r_i = G_i + r_{i-1}.P_i$. Cette formule a une interprétation immédiate : il y a une retenue en sortie si l'étage d'additionneur génère une retenue ($G_i = 1$) ou si l'étage propage la retenue d'entrée égale à 1 ($r_{i-1}.P_i = 1$). La Table 2 donne les fonctions G_i et P_i en fonction des entrées a_i et b_i, et montre la relation entre G_i, P_i et $a_i \oplus b_i$.

a_i	b_i	G_i	P_i	$a_i \oplus b_i$	$G_i \oplus P_i$
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

Table 2

Dans le cas où $P_i = a_i + b_i$, on constate que $a_i \oplus b_i = \overline{G_i}.P_i = G_i \oplus P_i$. Les fonctions G_i et P_i, utilisées pour calculer la retenue r_i, peuvent également être utilisées pour calculer la somme S_i

$S_i = a_i \oplus b_i \oplus r_{i-1} = G_i \oplus P_i \oplus r_{i-1} = \overline{G_i}.P_i \oplus r_{i-1}$

1.3.3.2 Circuits anticipateurs de retenue.

L'application de la formule F2 permet de calculer en 2 couches logiques (2 t_p) la retenue sortant de n étages d'additionneurs 1 bit. En effet, on peut calculer la retenue de sortie en fonction des retenues générées et propagées par chaque étage. Si l'on considère la retenue sortant d'un bloc de quatre bits, constitués d'indice i à i+3, on obtient la formule suivante, qui exprime la condition pour que l'étage i+3 génère une retenue, ou que l'étage i+2 génère une retenue propagée par l'étage i+3, etc.

$$r_{i+3} = G_{i+3} + P_{i+3}G_{i+2} + P_{i+3}P_{i+2}G_{i+1} + P_{i+3}P_{i+2}P_{i+1}G_i + P_{i+3}P_{i+2}P_{i+1}P_i r_{i-1}. \quad (F4)$$

On peut également déterminer les fonctions génération et propagation pour un bloc de quatre étages, qui traduisent les conditions pour que le bloc de quatre étages génère une retenue (c'est la fonction $G_{i+3,i}$) ou propage une retenue présente à l'entrée du bloc (c'est la fonction $P_{i+3,i}$). Ces fonctions sont respectivement

$$G_{i+3,i} = G_{i+3} + P_{i+3}G_{i+2} + P_{i+3}P_{i+2}G_{i+1} + P_{i+3}P_{i+2}P_{i+1}G_i \quad (F5)$$

$$P_{i+3,i} = P_{i+3}P_{i+2}P_{i+1}P_i \quad (F6)$$

Le circuit anticipateur de retenues pour un bloc de 4 étages est donné en Figure 15.

La limitation du nombre n d'étages pouvant constituer un bloc est lié au nombre maximal d'entrée (encore appelé entrance) des portes utilisées. Pour une anticipation sur n bits, l'entrance maximale est n + 1.

Le mécanisme d'anticipation de retenue peut s'appliquer par blocs de bits.

La formule F4 peut se réécrire en utilisant les fonctions $G_{i+3,i}$ et P_{i+3} définies en F5 et F6.

$$r_{i+3} = G_{i+3,i} + P_{i+3,i}r_{i-1} \quad (F7)$$

Les fonctions $G_{i+3,i}$ et $P_{i+3,i}$ permettent de calculer la retenue sur un bloc de 16 étages en 4 couches logiques selon la formule F8

$$r_{i+15} = G_{i+15,i+12} + P_{i+15,i+12}\,G_{i+11,i+8} + P_{i+15,i+12}\,P_{i+11,i+8}\,G_{i+7,i+4} + P_{i+15,i+12}\,P_{i+11,i+8}\,P_{i+7,i+4}\,G_{i+3,i} + P_{i+15,i+12}\,P_{i+11,i+8}\,P_{i+7,i+4}\,P_{i+3,i}\,r_{i-1} \quad (F8)$$

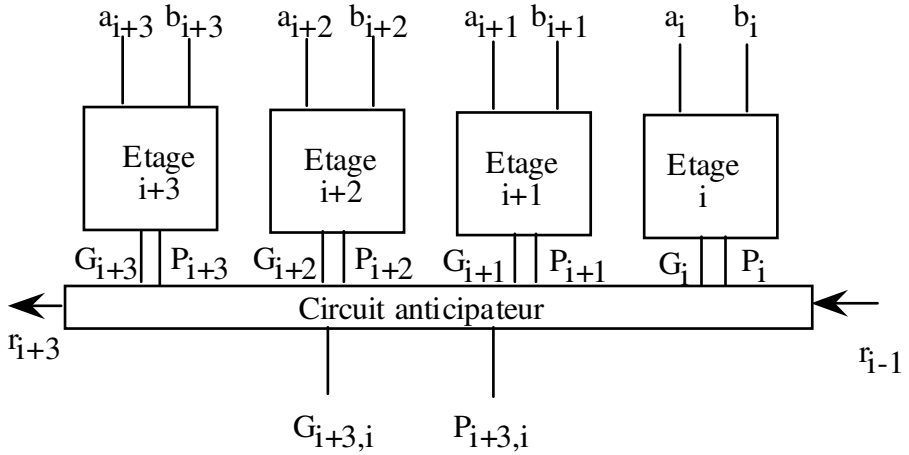

Figure 15 : Circuit d'anticipation de retenue

Avec des portes NAND, 2 t_p sont nécessaires pour obtenir les fonctions G_i et P_i, 2 t_p sont nécessaires pour obtenir les fonctions $G_{i+3,i}$ et $P_{i+3,i}$ et 2 t_p sont nécessaires pour obtenir r_{i+15} en fonction des $G_{i+3,i}$ et $P_{i+3,i}$.

En généralisant, on constate qu'il faut 2 (p+1) t_p pour obtenir la retenue d'une addition de 4^p bits avec des circuits anticipateurs sur 4 bits. Avec cette approche, le temps nécessaire pour obtenir la retenue de sortie varie de manière logarithmique avec le nombre de bits.

Les sorties S_i sont obtenues en utilisant la formule $S_i = \overline{G_i}.P_i \oplus r_{i-1}$

ce qui implique de générer l'ensemble des retenues r_{i-1}, en utilisant des circuits d'anticipation ou de propagation simple de retenue en fonction des contraintes temporelles.

1.3.3.3 La sélection de retenue

Pour une addition de deux nombres de n bits, l'additionneur à sélection de retenue (Figure 16) utilise un additionneur de p bits pour les poids faibles, et deux additionneurs de q bits pour les poids forts avec n = p + q. L'un des additionneurs de q bits a une retenue d'entrée à 0 et l'autre à 1. Les sorties et la retenue de poids fort sont obtenues par multiplexage des sorties et de la retenue des deux additionneurs de q bits dès que la retenue sortante de l'additionneur des poids faibles est connue. Chacun des additionneurs sur p ou q bits peut utiliser des mécanismes de propagation rapide de retenue.

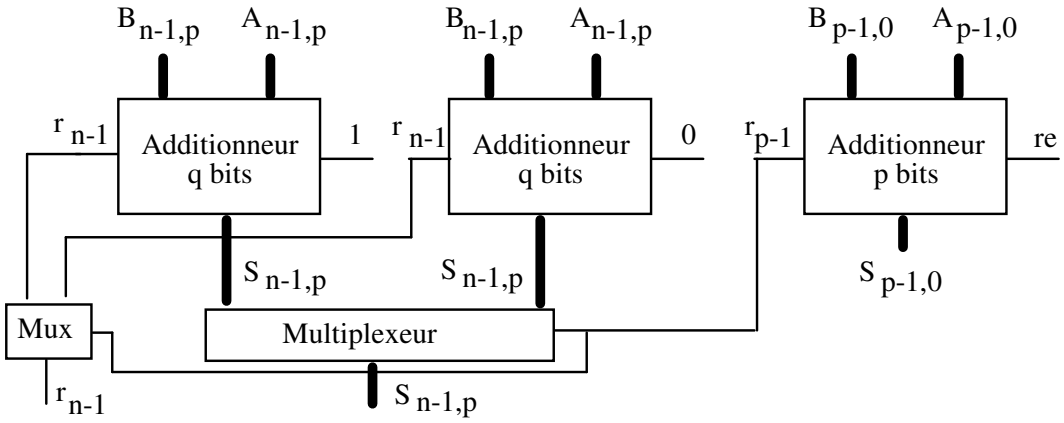

Figure 16 : Principe de l'additionneur à sélection de retenue

1.3.4 Les comparateurs

La logique en tranche utilisée pour l'additionneur peut également être utilisée pour d'autres opérateurs comme les comparateurs (exemple en TD).