Chapitre 7.

Éléments d'arithmétique binaire

La plupart des calculateurs utilisent la base 2 pour représenter les nombres. Les opérations entre nombres sont donc basées sur l'arithmétique binaire. Dans la première partie de ce chapitre, nous rappellerons les notions élémentaires de codage et de représentation des nombres. Nous définirons ensuite les principes élémentaires de l'arithmétique binaire. Le principe des opérations de base (addition, soustraction, multiplication, division) sera présenté.

7.1. Représentation des nombres

Un code constitue une correspondance entre des symboles et des objets à désigner. Les codes utilisés pour représenter des nombres sont des codes pondérés : dans une base de travail donnée, la valeur d'un rang donné est un multiple par la base de celle du rang inférieur.

7.1.1. Représentation des entiers naturels

De manière générale, un nombre entier naturel N exprimé dans une base b est un ensemble ordonné de n chiffres chacun d'eux prenant une valeur comprise entre 0 et b-1.

$$N_b = a_{n-1}\ a_{n-2}\ \ldots.\ a_1\ a_0$$

Les nombres exprimés dans la base 10 sont appelés nombres décimaux. Les nombres exprimés dans la base 2 sont appelés nombres binaires. Pour les calculateurs électroniques ces nombres ont un intérêt particulier du fait qu'ils ne font intervenir que deux valeurs (0,1).

La valeur d'un nombre exprimé dans une base b est la somme pondérée de ces n chiffres, les poids de chacun d'eux étant des puissances de la base de numération. Cette valeur est comprise entre 0 est b^n-1.

$$N = a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \ldots.. + a_1b^1 + a_0b^0 \quad => \quad N = \sum_{i=0}^{n-1} a_i b^i$$

Exemple : Le nombre binaire $N_2 = 1011$ représente le nombre décimal $N_{10}=13$ $(2^3+2^2+2^0)$.

7.1.2. Représentation des entiers relatifs

Un entier relatif est un entier pouvant être négatif. Soit un nombre de n bits s'écrivant de la manière suivante :

$$N_b = a_{n-1}\ a_{n-2}\ \ldots..\ a_1\ a_0$$

Par convention, le bit de poids fort (a_{n-1}), appelé bit de signe est utilisé pour représenter le signe. Les autres bits (a_{n-2} ….. a_1 a_0) sont utilisés pour représenter la valeur du nombre. Ainsi, un nombre signé prend ses valeurs dans l'intervalle $[-(b^{n-1}-1) , b^{n-1}-1]$.

7.1.2.a. Codage signe et valeur absolue

Dans ce code, les chiffres (bits) a_{n-2} ….. a_1a_0 représentent la valeur absolue du nombre et le bit de signe a_{n-1} prend les valeurs suivantes :

$a_{n-1} = 0$ si $N \geq 0$
$a_{n-1} \neq 0$ si $N < 0$

Exemple : Interprétation de nombres binaires en représentation Signe et Valeur absolue (n=3)

2^2 2^1 2^0	N
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	-0
1 0 1	-1
1 1 0	-2
1 1 1	-3

Remarque : Un des problèmes induits par cette représentation réside dans le processus d'addition/soustraction de tels nombres. En effet, lorsque les 2 nombres à additionner sont de signes opposés, il est nécessaire de comparer la valeur des 2 nombres pour effectuer l'opération de soustraction et pour déterminer le signe du résultat. Un autre problème lié à cette représentation est la double représentation du 0.

7.1.2.b. Codage signe et complément

Le codage en complément a été introduit pour faciliter les opérations d'addition/soustraction de nombre rationnels. Dans ce code, le bit de signe a_{n-1} prend les valeurs suivantes :

$a_{n-1} = 0$ si $N \geq 0$
$a_{n-1} = b-1$ si $N < 0$

Les chiffres (bits) a_{n-2} ….. a_1a_0 représentent la valeur du nombre si le nombre est positif ($a_{n-1}=0$) ou la valeur codée en complément si le nombre est négatif ($a_{n-1}=b-1$). Le complément des nombres est un codage particulier. Les deux codes en complément les plus couramment utilisés sont le complément à b-1 (complément à la base moins 1) et le complément à b (complément à la base).

7.1.2.b.1. Complément à b-1

Le complément à b-1 d'un nombre N exprimé sur n chiffres (bits), est obtenu en soustrayant le nombre N du radical R diminué d'une unité. Le radical R d'un nombre N exprimé en base b sur n chiffres (ou bits) est la puissance de b immédiatement supérieure à la valeur maximum de N ($R = b^n$).

$$C_{b-1}(N) = b^n - 1 - N$$

Exemple :

$C_9(5230)_{10} = 10^4 - 1 - (5230)_{10} = (4769)_{10}$

$C_1(1100)_2 = 2^4 - 1 - (1100)_2 = (16 - 1 - 12)_{10} = (3)_{10} = (0011)_2$

Nous pouvons constater que le complément à b-1 de N peut s'obtenir en complémentant à b-1 tous les chiffres. Dans le cas de la base 2 cela se traduit par une inversion de tous les bits.

Exemple : Interprétation de nombres binaires en représentation Signe et Valeur absolue (n=3)

$2^2\ 2^1\ 2^0$	N
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	-3
1 0 1	-2
1 1 0	-1
1 1 1	-0

Remarque : Un des problèmes induits par cette représentation réside dans la double représentation du 0.

7.1.2.b.2. Complément à b

Le complément à b d'un nombre N exprimé sur n chiffres (bits), est obtenu en soustrayant le nombre N du radical R.

$$C_b(N) = b^n - N$$

Exemple :

$C_{10}(5230)_{10} = 10^4 - (5230)_{10} = (4770)_{10}$

$C_n(1100)_2 = 2^4 - (1100)_2 = (16 - 12)_{10} = (4)_{10} = (0100)_2$

Nous pouvons constater que le complément à b de N peut s'obtenir à partir du complément à b-1 :
$C_b(N) = C_{b-1}(N)+1$ mais également par la procédure de scrutation / complémentation suivante :

- Scruter le nombre à partir de la droite
- Tant que les bits rencontrés sont à 0, les conserver
- Complémenter à b le premier chiffre non nul
- Complémenter à b-1 tous les suivants

Dans le cas de la base 2 cela se traduit par :

- Scruter le nombre à partir de la droite
- Tant que les bits rencontrés sont à 0, les conserver
- Conserver le premier 1
- Inverser tous les bits suivants

Du fait de la plus grande simplification apportée au processus de soustraction, la représentation en complément la plus usuelle est la représentation en complément à b (complément à 2 en binaire). A partir de cette représentation en complément à b, la valeur décimale d'un nombre peut être obtenue par la relation suivante :

$$(N)_{10} = -[(a_{n-1})/(b-1)]b^{n-1} + \sum_{i=0}^{n-2} a_i b^i$$

Cette valeur est comprise entre $-b^{n-1}$ et $b^{n-1}-1$.

Pour des nombres binaires, cette relation devient :

$$(N)_{10} = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Exemple : Interprétation de nombres binaires signé exprimés en complément à 2 (n=3)

$2^2\ 2^1\ 2^0$	N
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	-4
1 0 1	-3
1 1 0	-2
1 1 1	-1

Remarque : Outre son intérêt pour les opérations d'addition/soustraction, le codage en complément à b permet d'éviter la double représentation du 0 ce qui n'est pas le cas en codage complément à b-1.

7.1.3. Représentation des nombres rationnels

7.1.3.a. Représentation en virgule fixe

Les représentations précédentes peuvent s'étendre aux nombres rationnels. Pour ces nombres, la partie fractionnaire est séparée de la partie entière, par une virgule ou un point. Ce type de codage des nombre rationnels est appelé représentation en virgule fixe.

Soit $(N)_b = a_{n-1} \ldots. a_0 , a_{-1} \ldots. a_{-m}$. La valeur décimale d'un tel nombre est :

$$(N)_{10} = a_{n-1}b^{n-1} + ... + a_0 b^0 + a_{-1}b^{-1} + \ldots. + a_{-m}b^{-m} \quad => \quad (N)_{10} = \sum_{i=-m}^{n-1} a_i b^i$$

Exemple : Le nombre binaire N = 1101,11 représente la valeur 13,75 ($2^3+2^2+2^0+2^{-1}+2^{-2}$).

7.1.3.b. Représentation en virgule flottante

Le codage en virgule fixe sur *n* bits ne permet de représenter qu'un intervalle de 2^n valeurs. Pour un grand nombre d'applications, cet intervalle de valeurs est trop restreint. La représentation à virgule flottante (*floating-point*) a été introduite pour répondre à ce besoin et améliorer la précision des calculs.

Cette représentation consiste à représenter un nombre binaire sous la forme $1,M * 2^E$ ou M et la mantisse et E l'exposant.

La représentation en virgule flottante a été normalisée (norme IEEE 754) afin que les programmes aient un comportement identique d'une machine à l'autre. La norme IEEE754 défini la façon de coder en virgule flottante un nombre binaire sur 32 bits ou 64 bits (double précision) en définissant 3 composantes :

- le signe S est représenté par un seul bit : le bit de poids fort (le plus à gauche)

- l'exposant E est codé sur les 8 bits consécutifs au signe en excès de 127 (simple précision) ou 1023 (double précision)
- la mantisse M (bits situés après la virgule) est représentée sur les 23 (simple précision) ou 55 (double précision) bits restants.

Remarque : les exposants peuvent aller de -126 à +127 (simple précision) ou de -1022 à +1023 (double précision).

Ainsi le codage sur un mot de 32 bits se fait sous la forme suivante :

seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm

- le **s** représente le bit relatif au signe
- les **e** représentent les bits relatifs à l'exposant avec un excès de 127
- les **m** représentent les bits relatifs à la mantisse.

Dans cette représentation la valeur d'un nombre sur 32 bits est donnée par l'expression :

$N = (-1)^s * 1,M * 2^{(E-127)}$

$N = (-1)^s * (1 + M*2^{-23}) * 2^{(E-127)}$

Exemple : Soit à coder la valeur 525,5.

525,5 s'écrit en base 2 de la façon suivante : 1000001101,1

On veut l'écrire sous la forme 1.0000011011×2^9. Par conséquent :

- le bit s vaut 1
- l'exposant vaut 9+127 = 136, soit 10010000
- la mantisse est 0000011011

La représentation du nombre 525.5 en binaire avec la norme IEEE754 est :

1 10010000 00000110110000000000000

Remarque : Le zéro est le seul nombre qui ne peut pas se représenter dans le système à virgule flottante à cause de la contrainte de non nullité du premier digit ($1,M * 2^E$). Le zéro est donc représenté par une convention de représentation. La norme IEEE754 précise cette convention de représentation (E=M=0) ainsi que d'autre conditions particulières :

E	M	Valeur représentée
Max	0	∞
Max	≠ 0	NaN (Not a number)
0	0	0
0	≠ 0	Nombre dénormalisé

7.1.4. Conversion de bases

Compte tenu de la facilité de conversion entre les bases 2, 8 et 16 (puissances de 2), les bases 8 (octal) et 16 (hexadécimal) sont également utilisées pour représenter les nombres binaires sous forme plus synthétique. Un nombre octal ou hexadécimal peut effectivement être convertit en nombre binaire par conversion de chacun de ses coefficients individuellement dans sa représentation binaire équivalente.

Exemple : $(3D)_{hex}$ => $(00111101)_{bin}$

Cette propriété est vraie pour les bases puissance de 2. Pour les bases non-puissance de deux, cette conversion est plus complexe.

Une solution permettant d'effectuer ces conversions est d'utiliser directement les expressions polynomiales. Cette technique d'évaluation directe d'expressions polynomiales de nombre est une méthode générale de conversion d'une base b1 en une base b2. Cette méthode est appelée méthode polynomiale.

7.1.4.a. Méthode polynomiale

Principe de la méthode :

- Exprimer le nombre $(N)_{b1}$ en polynôme avec dans le polynôme des nombres exprimés dans la base b2,
- Évaluer le polynôme en utilisant l'arithmétique de la base b2.

Exemple : A titre d'exemple considérons le nombre binaire (1011,101). L'expression polynomiale est la suivante :

$$A = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$$

$(A)_{10} = 1*8 + 0*4 + 1*2 + 1*1 + 1*1/2 + 0*1/4 + 1*1/8$
$= 11 + 5/8$
$= 11,625$

$(A)_8 = 1*10 + 0*4 + 1*2 + 1*1 + 1*1/2 + 0*1/4 + 1*1/10$
$= 13 + 5/10$
$= 13,5$

$(A)_3 = 1*22 + 0*11 + 1*2 + 1*1 + 1*1/2 + 0*1/11 + 1*1/22$
$= 102 + 12/22$
$= 102,121212 ...$

Ce type de conversion est en fait très utilisé pour passer d'une base quelconque à une expression décimale. Dans la pratique un changement de base entre deux bases quelconques se fait généralement par l'intermédiaire de l'expression décimale :

base b1 => expression décimale (base 10) => base b2

Nous avons vu que le passage d'une base quelconque à la base décimale par la méthode algorithmique est simple. L'inverse l'est moins. Une solution pour réaliser ce passage (base 10 à base quelconque) est d'appliquer une méthode itérative.

7.1.4.b. Méthode itérative

Soit A la représentation d'un nombre entier positif dans la base 10. Pour obtenir sa représentation dans la base b quelconque il suffit de diviser A par b, puis le quotient par b, jusqu'à ce que le quotient devienne nul. Les restes successifs lu de bas en haut sont la représentation de A dans la base b.

$A = a_3b^3 + a_2b^2 + a_1b^1 + a_0$
$= b(a_3b^2 + a_2b^1 + a_1) + a_0 = bQ1 + a_0$
$Q1 = b(a_3b^1 + a_2b^0) + a_1 = bQ2 + a_1$
$Q2 = ba_3 + a_2 = bQ3 + a_2$
$Q3 = a_3$

Exemple : Ecrire 88 en base 2 et 3

88 = (44*2) + 0 88 = (29*3) + 1
44 = (22*2) + 0 29 = (9*3) + 2
22 = (11*2) + 0 9 = (3*3) + 0
11 = (5*2) + 1 3 = (1*3) + 0
5 = (2*2) + 1 1= (0*3) + 1
2 = (1*2) + 0
1 = (0*2) + 1 $(88)_{10} = (10021)_3$

$(88)_{10} = (1011000)_2$

Exemple : Ecrire 23 en base 2 et 8
23 = (11*2) + 1 23 = (2*8) + 7
11 = (5*2) + 1 2 = (0*3) + 2
5 = (2*2) + 1
2 = (1*2) + 0 $(23)_{10} = (27)_8$
1 = (0*2) + 1

$(23)_{10} = (10111)_2$

Remarque : Cet exemple illustre les relations simples qui existent entre la base 2 et la base 8. Les digits binaires (appelés bits) sont pris 3 par 3 et exprimés en décimal pour obtenir le nombre octal. Ainsi l'équivalent du nombre binaire 10111 en base 8 est 27.

010 111
2 7

Il en est de même pour toutes les bases puissance de 2 et notamment pour la base 16 (hexadécimal). Dans le cas de la base 16, le nombre de bits à considérer est 4.

7.1.4.c. Cas particulier de la base 2

La conversion du système décimal au système binaire peut s'effectuer en remarquant que le reste de la division est 0 ou 1 selon que le dividende est pair ou impair.

La conversion décimale binaire peut donc être représentée plus simplement en écrivant les quotients de droite à gauche ; on écrit « 1 » sous chaque quotient impair et « 0 » sous chaque quotient pair.

Exemple **:** Conversion de $(53)_{10}$ en binaire

1 3 6 13 26 53
1 1 0 1 0 1 => $(53)_{10} = (110101)_2$

7.1.4.d. Cas des nombres rationnels

Considérons maintenant le cas d'un nombre $(N)_{10}$ ayant une partie fractionnaire et dont nous voulons l'expression dans la base b.

Pour le radical, la conversion s'effectue comme nous venons de le voir.

Pour la partie fractionnaire on opère de la façon suivante :

- On multiplie son expression en base 10 par b, on obtient un résultat $(n1)_{10}$.
- Le premier chiffre de la partie fractionnaire exprimé en base b est la partie entière de $(n1)_{10}$.
- On calcule $(n2)_{10} = (n1)_{10}$ - partie entière de $(n1)_{10}$

- La valeur du second chiffre de la partie fractionnaire exprimée en base b est obtenue en utilisant $(n2)_{10}$ comme nous avons utilisé $(n1)_{10}$ pour calculer le premier chiffre.

Exemple : Soit à convertir $(88,23)_{10}$ en base 2

Nous savons que $(88)_{10} = (1011000)_2$. Effectuons la conversion de la partie fractionnaire.

0,23 * 2 = 0,46 0,0
0,46 * 2 = 0,92 0,00
0,92 * 2 = 1,84 0,001
0,84 * 2 = 1,68 0,0011
0,68 * 2 = 1,36 0,00111
0,36 * 2 = 0,72 0,001110

d'où $(88,23)_{10} = (1011000,001110...)_2$

On voit bien que cette conversion peut ne pas se terminer et donc que l'on obtient lorsque l'on s'arrête à un nombre fini de positions, une approximation de la représentation du nombre.

Le cas particulier des bases multiples de 2 s'applique également dans le cas des nombres fractionnaires. Dans le cas de la base 8 par exemple, les bits sont pris 3 par 3 de chaque côté de la virgule.

Ainsi $(010111,011101)_2 = (27,35)_8$

7.2. Comparateurs binaires

7.2.1. Comparateur égalité

Pour savoir si deux nombres binaires A ($A= a_n...a_0$) et B ($B = b_n ... b_0$) sont égaux, il faut que trous les bits de ces deux nombres soient identiques. On a donc :

$$A = B \text{ si et seulement si } (a_0 = b_0) \text{ et } (a_1 = b_1) \text{ et } (a_2 = b_2) \text{ et ...}$$

Soit S la sortie du circuit comparateur égalité (S=1 si les 2 nombres sont identiques)

$$S = (a_0 \oplus b_0)' \,.\, (a_1 \oplus b_1)' \,.\, (a_2 \oplus b_2)' \,....$$

Un tel circuit peut également être déterminé en raisonnant à partir d'une structure itérative (en tranches) telle que celle présentée sur la figure 7.1. Ainsi, il suffit de déterminer la structure d'une cellule et de cascader cette cellule.

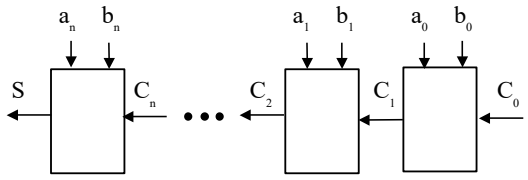


Figure 7.1. *Structure en tranche*

Pour déterminer la structure interne des cellules, considérons un étage i.

Supposons que $C_i = 1$ si et seulement si $\forall j$ compris entre 0 et i-1, aj = bj

La même information (C_{i+1}) doit être fournie à l'étage suivant

$$\Rightarrow C_{i+1} = C_i \,.\, (a_i \oplus b_i)' \quad (\text{avec } C_0 = 1)$$

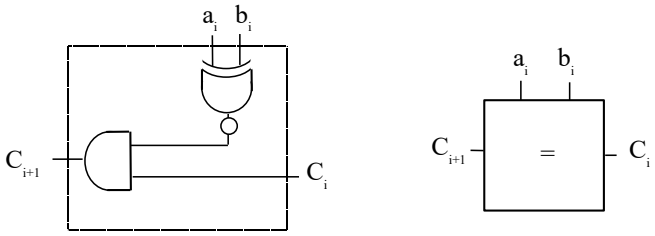


Figure 7.2. *Cellule du comparateur égalité*

7.2.2. Comparateur supériorité

Pour savoir si un nombre binaire A (A= $a_n...a_0$) est supérieur ou pas à un nombre binaire B (B = b_n ... b_0) il faut scruter ce nombre à partir des poids forts. On aura :

$$A > B \text{ si et seulement si } (a_n > b_n) \text{ ou } [(a_n = b_n) \text{ et } (a_{n-1} > b_{n-1})] \text{ ou } ...$$

Soit S la sortie du circuit comparateur égalité (S=1 si les 2 nombres sont identiques)

$$S = (a_n.b_n') + (a_n \oplus b_n)' \,.\, (a_{n-1}.b_{n-1}') + \ ....$$

Un tel circuit peut également être déterminé en raisonnant à partir d'une structure itérative (en tranches) telle que celle présentée sur la figure 7.1. Pour déterminer la structure interne des cellules, considérons un étage i

Supposons que $C_i = 1$ si et seulement si, en ne considérant que les bits de poids plus faible on a A > B.

La même information (C_{i+1}) doit être fournie à l'étage suivant

$$C_{i+1} = \ a_i.b_i' + (a_i \oplus b_i)'.\, C_i \qquad (\text{avec } C_0 = 0)$$

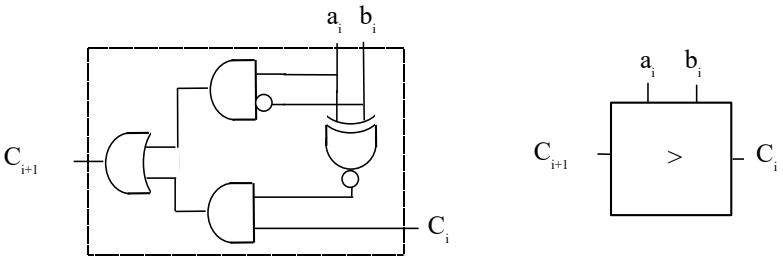


Figure 7.3. *Cellule du comparateur supériorité*

7.3. Addition binaire

7.3.1. Structure de l'additionneur binaire

Ce paragraphe présente la structure de base permettant de réaliser l'addition entre deux nombres binaires de n bits.

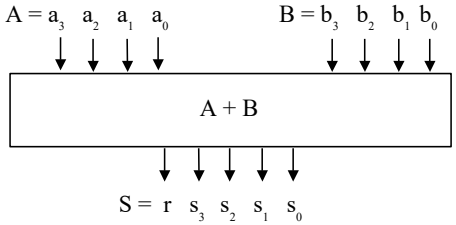


Figure 7.4. *Additionneur binaire*

La table d'addition binaire sur 2 bits est 1a suivante :

a + b	s	r
0 + 0	0	0
0 + 1	1	0
1 + 0	1	0
1 + 1	0	1

$s = a \oplus b$

$r = a.b$

L'opération d'addition de deux représentations binaires s'effectue de façon similaire à l'addition décimale c'est à dire en additionnant digit par digit les deux représentations alignées sur la virgule et en reportant une éventuelle retenue sur la colonne suivante.

Exemple :

```
Retenues -> 10011 11
              1001,011   = ( 9,375)10
            + 1101,101   = (13,625)10
              -----------  ----------
              10111,000  = (23,000)10
```

Pour réaliser un additionneur, le module de base doit donc réaliser l'addition sur 3 bits. L'addition étant une opération associative, on peut aisément déduire la table d'addition binaire sur 3 bits de la table d'addition binaire sur 2 bits.

Soit deux nombres A ($A= a_n ... a_0$) et B ($B= b_n ... b_0$).

a_i b_i r_i	S_i	r_{i+1}
0 0 0	0	0
0 0 1	1	0
0 1 0	1	0
0 1 1	0	1
1 0 0	1	0
1 0 1	0	1
1 1 0	0	1
1 1 1	1	1

$s_i = a_i \oplus b_i \oplus r_i$

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

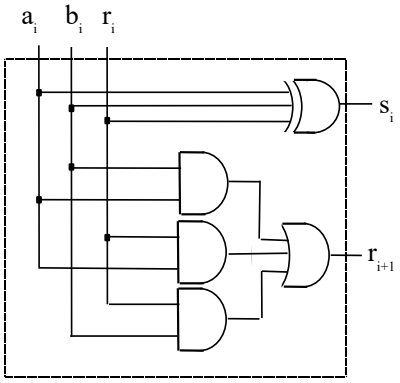


Figure 7.5. *Cellule additionneur 2 fois 1 bit plus retenue*

Les fonctions s_i et r_i peuvent également s'exprimer de la manière suivante :

$s_i = (a_i \oplus b_i) \oplus r_i$

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

Démonstration :

$$\begin{aligned} r_{i+1} &= r_i.(a_i \oplus b_i) + a_i.b_i \\ &= r_i.a_i'.b_i + r_i.a_i.b_i' + a_i.b_i + a_i.b_i \\ &= a_i.(b_i + r_i.b_i') + b_i.(a_i + r_i.a_i') \\ &= a_i.(b_i + r_i) + b_i.(a_i + r_i) \\ &= a_i.b_i + a_i.r_i + b_i.r_i \end{aligned}$$

Ceci conduit à la structure suivante :

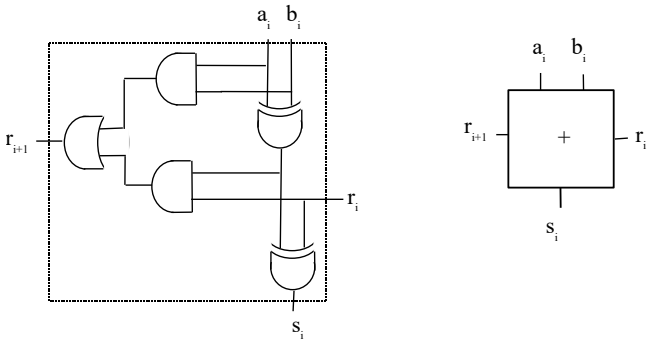


Figure 7.6. *Cellule additionneur 2 fois1 bit plus retenue*

La structure permettant d'additionner deux nombres de n bits est obtenue en cascadant cette structure de base.

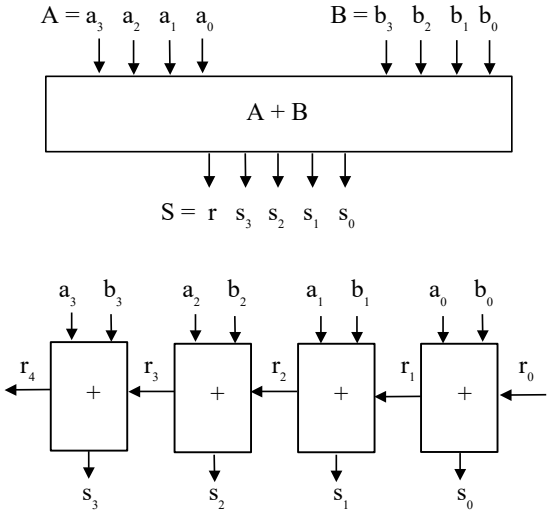


Figure 7.7. *Structure de l'additionneur binaire*

7.3.2. Incrémenteur

La structure d'un incrémenteur peut être déterminée à partir de celle de l'additionneur en considérant qu'un des 2 mots est à 0 et que la retenue entrante vaut 1.

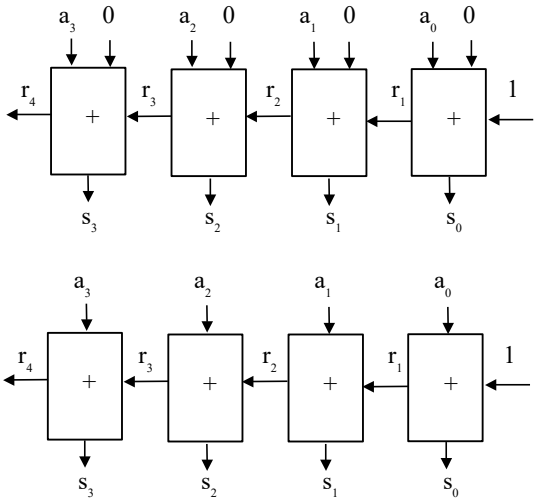


Figure 7.8. *Structure de l'incrémenteur*

$s_i = (a_i \oplus b_i) \oplus r_i$ $\quad b_i = 0 \quad \Rightarrow s_i = a_i \oplus r_i$

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

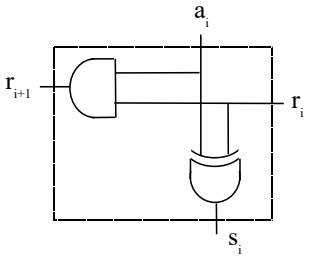


Figure 7.9. *Cellule élémentaire de l'incrémenteur*

7.3.3. Additionneur à propagation anticipée

L'inconvénient de la structure précédente est le temps nécessaire à la réalisation de l'addition. Ce temps est en effet conditionné par la propagation de la retenue à travers tous les additionneurs élémentaires. Dans un additionneur à carry anticipée on évalue en même temps la retenue de chaque étage. Pour cela on détermine pour chaque étage les quantités Pi et Gi suivantes :

$P_i = a_i \oplus b_i$ (propagation d'une retenue)

$G_i = a_i b_i$ (génération d'une retenue)

La retenue entrante à l'ordre i vaut 1 si :

- soit l'étage i-1 a généré la retenue ($G_{i-1} = 1$)
- soit l'étage i-1 a propagé la retenue générée à l'étage i-2 ($P_{i-1}=1$ et $G_{i-2}=1$)
- soit les étages i-1 et i-2 ont propagé la retenue générée à l'étage i-3 ($P_{i-1}=P_{i-2}=1$ et $G_{i-3}=1$)
- .....
- soit tous les étages inférieurs ont propagé la retenue entrante dans l'additionneur ($P_{i-1}=P_{i-2}=...=P_0=r_0=1$).

Donc $r_i = G_{i-1} + P_{i-1}.G_{i-2} + P_{i-1}.P_{i-2}.G_{i-3} + ... + P_{i-1}.P_{i-2}.P_{i-3}....P_0.r_0$

- $r_1 = G_0 + P_0.r_0$

- $r_2 = G_1 + P_1.G_0 + P_1.P_0.r_0$
- $r_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.r_0$
- $r_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.r_0$

Dans un additionneur à retenue anticipée, on évalue en parallèle :

- les couples (G_i,P_i)
- les retenues r_i
- les bits de somme $s_i=a_i \oplus b_i \oplus r_i = P_i \oplus r_i$

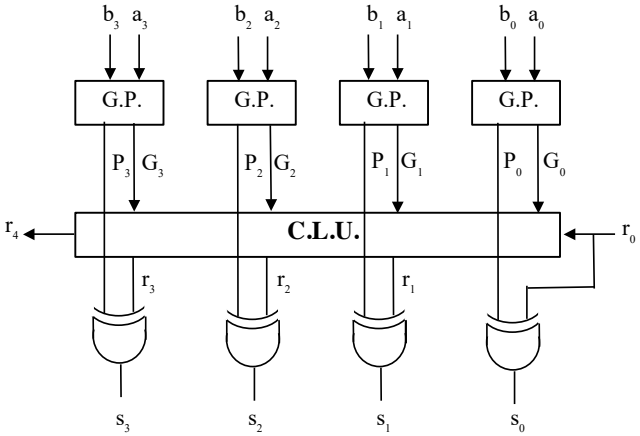


Figure 7.10. *Additionneur à retenue anticipée*

La structure du bloc CLU peut être déterminée à partir des équation donnant les retenues r_i.

7.4. Soustraction binaire

7.4.1. Structure du soustracteur binaire

Ce paragraphe présente la structure de base permettant de réaliser la soustraction entre deux nombre binaires de n bits.

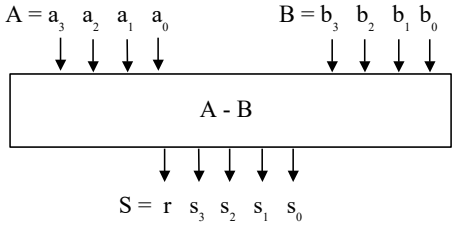


Figure 7.11. *Soustracteur binaire*

La table de soustraction sur 2 bits est la suivante :

a - b	s	r
0 - 0	0	0
0 - 1	1	1
1 - 0	1	0
1 - 1	0	0

$s = a \oplus b$

$r = a' . b$

La procédure de soustraction de nombre binaire est semblable à celle utilisée en décimal c'est à dire :

- le signe du résultat est le signe du nombre le plus grand en valeur absolue.
- le résultat est obtenu en soustrayant le nombre le plus petit en valeur absolue du nombre le plus grand.

Exemple :

```
Retenues ->    111                          011
                 101,0   ( 5  )              1100   ( 12)
               - 011,1   (-3,5)           - 0011   (- 3)
                 ------  -----              ----   ----
                 001,1   ( 1,5)             1001   (  9)
```

Pour réaliser un soustracteur, le module de base doit donc donc travailler sur 3 bits. Les relations donnant la somme et la retenue sur trois bits sont les suivantes.

Soit deux nombres A (A= a_n ... a_0) et B (B= b_n ... b_0) et soit à réaliser l'opération A-B.

a_i b_i r_i	S_i	r_{i+1}
0 0 0	0	0
0 0 1	1	1
0 1 0	1	1
0 1 1	0	1
1 0 0	1	0
1 0 1	0	0
1 1 0	0	0
1 1 1	1	1

$s_i = a_i \oplus b_i \oplus r_i$

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

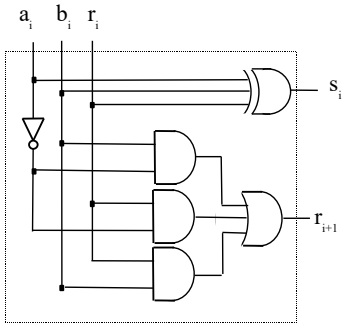


Figure 7.12. *Cellule soustracteur 2 fois 1 bit plus retenue*

Les fonctions S_i et R_i peuvent aussi s'exprimer de la manière suivante :

$s_i = (a_i \oplus b_i) \oplus r_i$

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

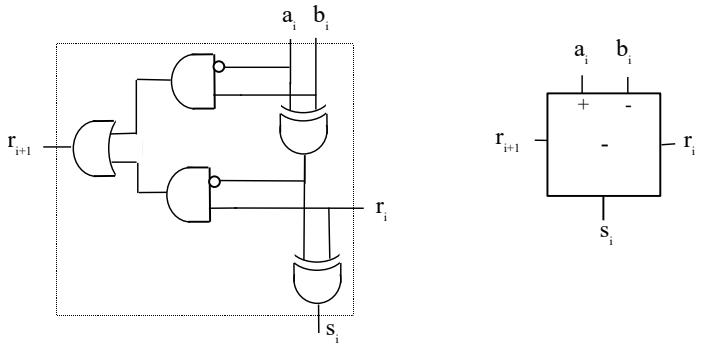


Figure 7.13. *Cellule soustracteur 2 fois 1 bit plus retenue*

$s_i' = (a_i' \oplus b_i) \oplus r_i$

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

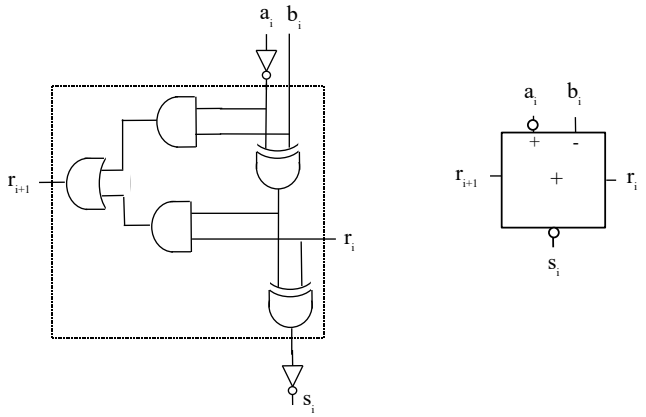
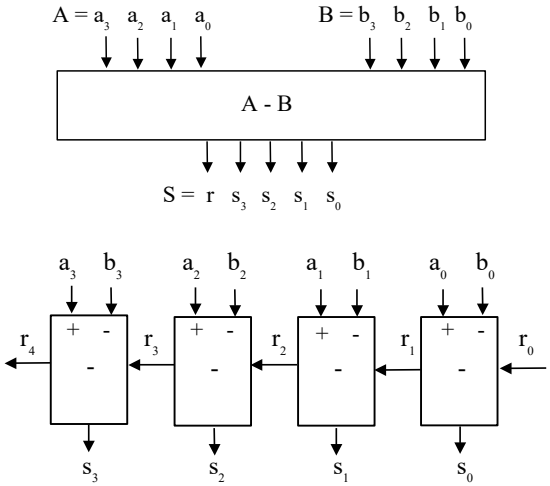


Figure 7.14. *Cellule soustracteur 2 fois 1 bit plus retenue*

Comme pour l'additionneur, un soustracteur peut être réalisé en cascadant les cellules de base.



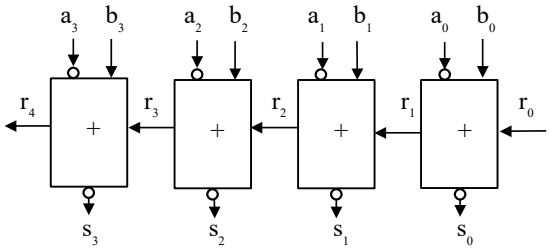


Figure 7.15. *Structure du soustracteur binaire*

Remarque : Relation entre la structure du soustracteur binaire et le complément à 1.

$C_1 (C_1(A) + B) = C_1 (2^n - 1 - A + B)$

$= 2^n - 1 - 2^n + 1 + A - B$

$= A - B$

Remarque : La soustraction n'est pas commutative. Cette propriété impose un dispositif amont de traitement et/ou d'orientation des entrées.

7.4.2. Décrémenteur

La structure d'un décrémenteur peut être déterminée à partir de celle du soustracteur en considérant que le mot à retrancher est nul ($b_i = 0$) et que la retenue entrante vaut 1.

$s_i = (a_i \oplus b_i) \oplus r_i$ $\quad b_i = 0 \quad \Rightarrow s_i = a_i \oplus r_i$

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

Ainsi, un décrémenteur peut être réalisé à partir d'un incrémenteur en inversant les entrées a_i et les sorties s_i tel que représenté sur la figure 7.16.

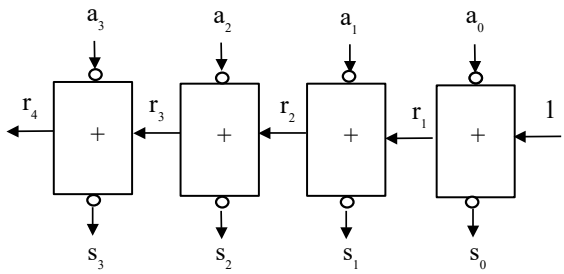


Figure 7.16. *Structure de l'incrémenteur*

7.4.3. Additionneur / Soustracteur

La structure d'un opérateur assurant soit l'addition soit la soustraction de deux nombres A et B en fonction d'une commande Op peut être conçu à partir des structures d'additionneurs et soustracteurs précédemment présentées. La structure d'un tel opérateur est représentée sur la figure 7.17.

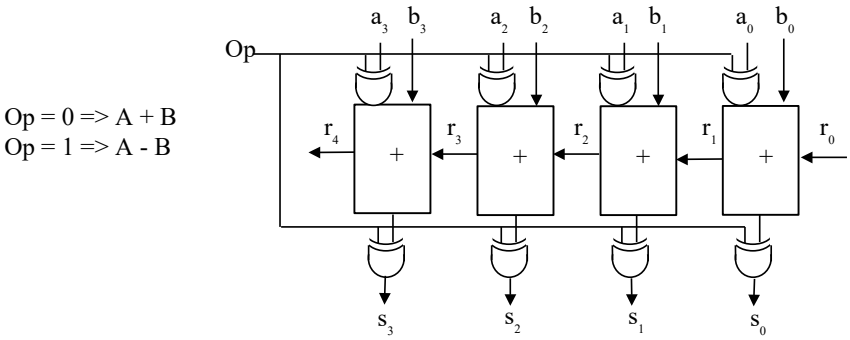


Figure 7.17. *Structure d'un additionneur / soustracteur binaire*

Remarque : Tout comme pour le soustracteur, la non-symétrie de cette cellule en mode soustraction impose un dispositif amont de comparaison et d'orientation des entrées.

Notons qu'une cellule assurant les opérations incrémentation / décrémentation en fonction d'une commande C peut être réalisée le même principe.

7.5. Addition algébrique

Un nombre entier relatif peut être représenté par un nombre binaire dont le bit de poids fort (a_{n-1}) indique le signe (0 correspond à un nombre positif, 1 à un nombre négatif) et les autres bits (a_{n-2} ….. a_1a_0) représentent la partie significative du nombre. Si la partie significative du nombre est représentée en valeur absolue, l'opération de soustraction est relativement complexe puisque non commutative. Elle nécessite effectivement la connaissance du plus grand des deux nombres, l'aiguillage du plus grand des deux nombre ssur l'entrée + du soustracteur (est celle du plus petit sur l'entrée -), un traitement particulier pour le signe (signe du résultat = signe du plus grand), etc. La structure réalisant l'opération d'addition sur des nombres signés exprimés en valeur absolue est représentée sur la figure 7.18.

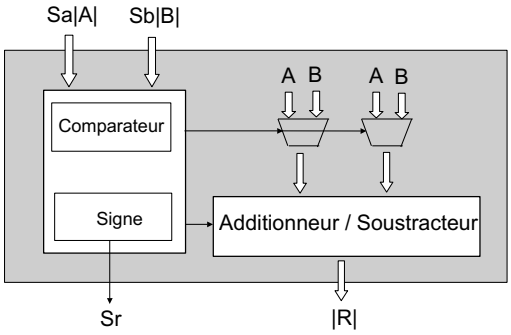


Figure 7.18. *Additionneur de nombre signés*

Un moyen d'éviter ce processus pour effectuer une soustraction est d'utiliser un codage en complément pour représenter les nombres négatifs. Il existe deux formes de compléments ; le complément à 2 (complément à b) et le complément à 1 (complément à b-1). Avec de tels codages, toute opération de soustraction se transforme en une opération d'addition entre nombre codés. La structure générale de l'additionneur algébrique utilisant un code complément est la suivante :

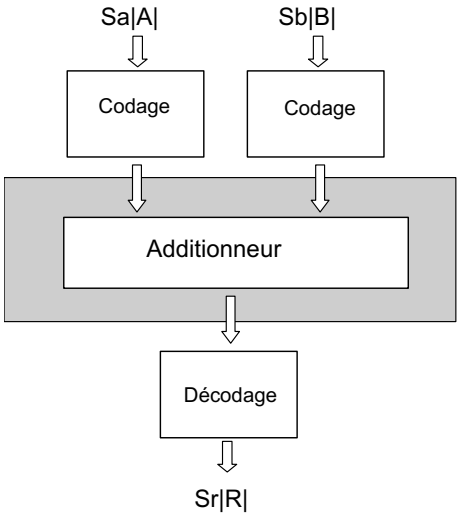


Figure 7.19. *Structure d'un additionneur algébrique*

7.5.1. Processus d'addition / soustraction en complément à 2

En utilisant un codage en complément à 2 pour les nombres négatifs, l'opération de soustraction se transforme en une simple opération d'addition binaire [A-B = A+(-B)].

7.5.1.a. Principe

1. Le bit de retenue de l'addition doit être ignoré dans tous les cas. Le résultat de l'addition est la valeur attendue. Ce résultat peut être positif ou négatif
2. Si le bits de signe du résultat vaut 0, le nombre obtenu est positif et codé en binaire naturel sur les bits significatifs. Si le bits de signe du résultat vaut 1, le nombre obtenu est négatif et codé en complément à 2.

Exemple :

```
 14 = 01110        14  =  01110
-13 =-01101    +(-13) =  10011
                         -----
                         00001     Signe=0 =>R=(00001)2=(1)10

 13 =  01101       13  =  01101
-14 = -01110    +(-14) =  10010
                         -----
                         11111     Signe=1 => R=-C2(11111)2
                                              =-(00001)2
                                              =-(1)10
```

Remarque : Le résultat de l'opération doit être inférieure à 2^n, c'est à dire qu'en valeur absolue sa représentation binaire comporte n bits au plus. Les cas qui peuvent poser problèmes sont ceux ou le signe des deux opérandes de l'addition est identique. Dans ce cas, il peut y avoir dépassement de capacité et l'addition de deux nombres positifs peut donner un résultat négatif.

```
   15 =   01111        15 = 01111
- (-2)= + 00010      + 2 = 00010
                          -----
                          10001     Signe=1 => R=-C2(10001)2
                                               =-(01111)2
                                               =-(15)10
```

Un bit de signalisation de dépassement de capacité peut être généré (Overflow)

$$Overflow = S(M).S(N).S'(R) + S'(M).S'(N).S(R)$$

7.5.1.b. Démonstration

Afin de démontrer le processus d'addition/soustraction en complément à 2, trois cas sont à envisager.

Cas 1 : M>0 et N>0 => signe(M)=signe(N)=0
=> retenue = 0
=> Résultat = M + N

Cas 2 : M>0 et N<0

N est représenté par son complément à 2 soit $(2^n - N)$.

On effectue donc l'addition $M + (2^n - N)$.

$$\begin{aligned} S &= (M - N) + 2^n \\ &= M - N \quad \text{(sur n bits)} \\ C_2(S) &= 2^n - M + N \\ &= N - M \quad \text{(sur n bits)} \end{aligned}$$

=> Résultat : $R = S$ si bit de signe = 0
$= -C_2(S)$ si bit de signe = 1

Cas 3 : M<0 et N<0

M et N sont représentés par leur complément à 2.

On effectue donc l'addition $(2^n - M) + (2^n - N)$.

$$\begin{aligned} S &= (- M - N) + 2^n + 2^n \\ &= - M - N \quad \text{(sur n bits)} \\ C_2(S) &= 2^n – S \\ &= M + N + 2^n \\ &= M + N \text{ (sur n bits)} \end{aligned}$$

=> Résultat : $R = -C_2(S)$

7.5.1.c. Additionneur / Soustracteur en complément à 2

Le changement de signe d'une opérande est obtenu en prenant son complément à 2. L'opération A-B se transforme donc en une opération $A+C_2(B)$. La structure d'un opérateur assurant soit l'addition soit la soustraction de deux nombres A et B en fonction d'une commande C peut donc être déterminée très simplement à partir d'un additionneur et d'un bloc multiplexé calculant le complément à 2.

Sachant que C2(N)=C1(N)+1, un tel opérateur peut être optimisé en jouant sur la retenue entrante pour réaliser l'opération +1 et sur le fait que l'opérateur complément à 1 est une simple inversion. Une telle structure est présentée sur la figure 7.20 :

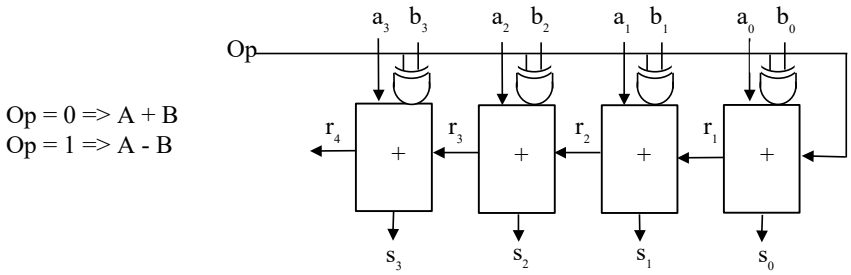


Figure 7.20. *Structure d'un additionneur / soustracteur en complément à 2*

7.5.2. Processus d'addition / soustraction en complément à 1

En utilisant un codage en complément à pour les nombres négatifs, l'opération de soustraction se transforme en une simple opération d'addition binaire [A-B = A+(-B)], mais dans ce cas, le résultat final n'est obtenu qu'après addition de la retenue au résultat partiel. Cette addition supplémentaire fait que le codage en complément à 1 ne présente que peu d'intérêt par rapport au codage en complément à 2. Le processus d'addition/soustraction en complément à 1 ne sera donc présenté ici qu'à titre pédagogique.

7.5.2.a. Principe

1. Le bit de retenue de l'addition doit être ignoré. Si ce bit vaut 0, le résultat de l'addition est la valeur attendue. Si ce bit vaut 1, le résultat attendu est le résultat de l'addition +1. Le résultat final peut être positif ou négatif.
2. Si le bit de signe du résultat final vaut 0, le nombre obtenu est positif et codé en binaire naturel sur les bits significatifs. Si le bit de signe du résultat final vaut 1, le nombre obtenu est négatif et codé en complément à 1.

Exemple :

```
 14 = 01110          14 = 01110
-13 =-01101        +(-13)= 10010
                           -----
                          100000
                        +      1
                           -----
                           00001   Signe=0 => R=(00001)2=(1)10

 13 =  01101          13 = 01101
-14 = -01110        +(-14)= 10001
                           -----
                          011110
                        +      0
                           -----
                           11110     Signe=1 => R=-C1(11110)2
                                                =-(00001)2
                                                =-(1)10
```

Remarque : Le résultat de l'opération doit être inférieure à 2^n, c'est à dire qu'en valeur absolue sa représentation binaire comporte n bits au plus. Les cas qui peuvent poser problèmes sont ceux ou le signe des deux opérandes de l'addition est identique.

```
   15 =   01111      15 =  01111
- (-2)= + 00010    + 2 =  00010
                          -----
                          10001
                        +     0
                          -----
                          10001    Signe=1 => R=-C1(10001)2
                                              =-(01111)2
                                              =-(15)10
```

Un bit de signalisation de dépassement de capacité peut être généré (Overflow)

Overflow = S(M).S(N).S'(R) + S'(M).S'(N).S(R)

7.5.2.b. Démonstration

Afin de démontrer le processus d'addition/soustraction en complément à 1, trois cas sont à envisager.

Cas 1 : M>0 et N>0 => signe(M)=signe(N)=0
=> retenue = 0
=> Résultat = M + N

Cas 2 : M>0 et N<0
N est représenté par son complément à 1 (2^n-1-N).
On effectue donc l'addition M + (2^n-1-N).
Si le résultat S est inférieur à 2^n (bit de retenue à 0) on obtient :

S1 = M + (2^n-1 - N)
S2 = S1 + 0
C1(S2) = 2^n-1 - (M - N + 2^n-1)
= N – M

=> Résultat : R = -C_1(S2)

Si le résultat est supérieur à 2^n (bit de retenue à 1) et si l'on ignore le bit de retenue (résultat - 2^n) on obtient :

S1 = M + (2n-1 - N) - 2n
= M - N – 1
S2 = S1 + 1
= M – N

=> Résultat : R = S2

Pour avoir le résultat exact, il faut donc rajouter le bit de retenue au résultat de l'addition.

Cas 3 : M<0 et N<0 => signe(M)=signe(N)=1
=> retenue = 1

N et M sont représentés par leur complément à 1.
On effectue donc l'addition (2^n-1-M) + (2^n-1-N).
Le résultat est supérieur à 2^n (bit de retenue à 1). Si l'on ignore le bit de retenue (résultat - 2^n) on obtient :

S1 = (2^n-1 - M) + (2^n-1 - N) - 2^n
= - M - N + 2^n - 1 –1
S2 = S1 + 1

$$= - M - N + 2^n - 1$$

$$C1(S2) = 2^n-1 - (- M - N + 2^n-1)$$

$$= M + N$$

=> Résultat : $R = -C_1(S2)$

7.6. Mutiplication binaire

7.6.1. Multiplication par une puissance de 2

Décaler un nombre écrit en base b de k positions vers la droite reviens à multiplier ce nombre par b^{-k}, et le décaler de k positions vers la gauche reviens à le multiplier par b^k.

Soit un nombre N écrit en base b :

$$(N)_b = \sum_{i=-m}^{n} p_i b^i = (p_n\ p_{n-1} \cdots p_1\ p_0\ ,\ p_{-1}\ p_{-2} \cdots p_{-m})_b$$

Ce nombre décalé de k position vers la gauche s'écrit :

$$(p_n\ p_{n-1} \cdots p_1\ p_0\ p_{-1} \cdots p_{-k}\ ,\ p_{-k-1} \cdots p_{-m})_b = \sum_{i=-m}^{n} p_i b^{i+k}$$

$$\sum_{i=-m}^{n} p_i b^{i+k} = b^k * \sum_{i=-m}^{n} p_i b^i = b^k * (N)_b$$

Décaler un nombre binaire de k positions (k positif pour un décalage à gauche et négatif pour un décalage à droite) reviens donc à multiplier ce nombre par 2^k.

Exemple :

$(110,101)_2 = (6,625)_{10}$

$(1,10101)_2 = 2^{-2}*(6,625)_{10} = (6,625 / 4)_{10} = (1,65625)_{10}$

$(11010,1)_2 = 2^{+2}*(6,625)_{10} = (6,625 * 4)_{10} = (26,5)_{10}$

7.6.2. Multiplieur cablé

Le processus de multiplication binaire est en fait beaucoup plus simple que le processus de multiplication décimale dans la mesure les tables de multiplications des chiffres se limitent au ET logique. La table de multiplication binaire est la suivante :

a * b	s
0 * 0	0
0 * 1	0
1 * 0	0
1 * 1	1

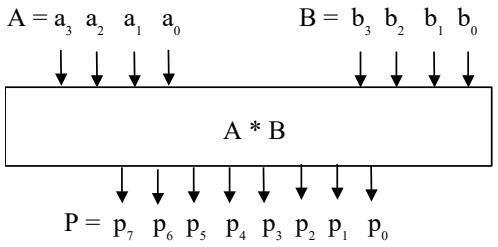
Pour chaque digit du multiplieur égal à 1, un produit partiel non nul est formé. Ce produit est constitué du multiplicande décalé d'un certain nombre de positions de manière à aligner son digit de poids le plus faible avec le 1 correspondant du multiplieur. Le produit final est obtenu par addition de tous les produits partiels.

Le processus de multiplication binaire est illustré par l'exemple suivant :

```
  11010    Multiplicande
    101    Multiplieur
 ------
  11010    Produit partiel
 00000     Produit partiel
11010      Produit partiel
-------
10000010
```

Cette opération de multiplication binaire se résume donc à une somme de produits partiels. Elle peut être réalisée par une structure cablée cascadant des additionneurs (Figure 7.21.).

				A_3	A_2	A_1	A_0	Multiplicande
				B_3	B_2	B_1	B_0	Multiplieur
				A_3B_0	A_2B_0	A_1B_0	A_0B_0	Produit partiel
			A_3B_1	A_2B_1	A_1B_1	A_0B_1		Produit partiel
		A_3B_2	A_2B_2	A_1B_2	A_0B_2			Produit partiel
	A_3B_3	A_2B_3	A_1B_3	A_0B_3				Produit partiel
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	



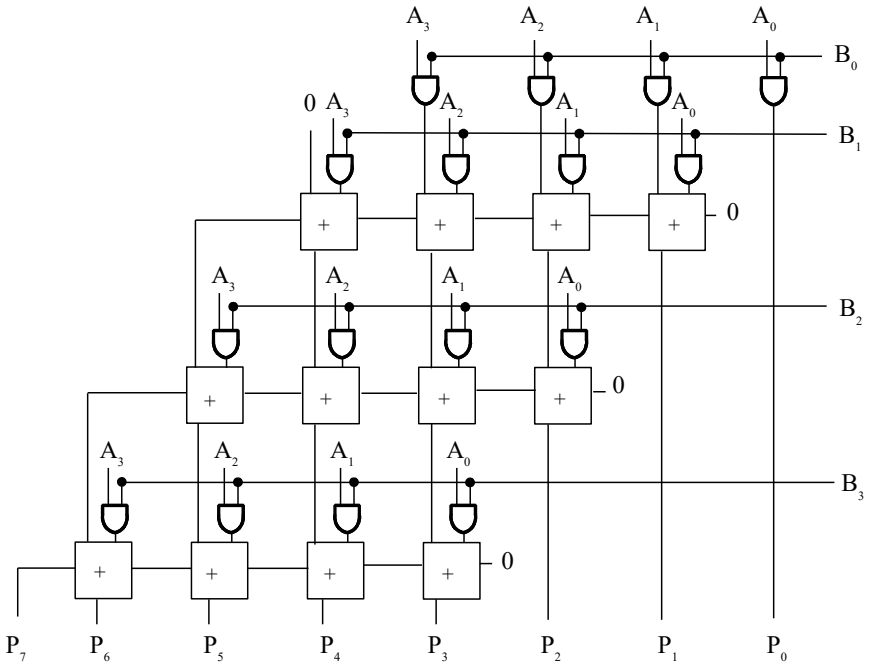


Figure 7.21. *Structure d'un multiplieur cablé*

Remarque : Si les 2 opérandes A et B s'écrivent sur n bits le ($A<2^n$ et $B<2^n$), le produit P s'écrit sur 2n bits ($P<2^{2n}$)

La multiplication des nombres radicaux peut s'effectuer sur le même principe. La virgule est placée en utilisant les mêmes règles que pour la multiplication décimale. Le nombre de digits à droite de la virgule est égal à la somme du nombre de digits à droite de la virgule du multiplicande et du multiplieur.

```
  110.10   Multiplicande
    10.1   Multiplieur
  ------
   11010   Produit partiel
  00000    Produit partiel
 11010     Produit partiel
 -------
10000.010
```

7.6.3. Multiplieur algébrique

Pour effectuer la multiplication de nombres signés, une solution est d'effectuer la multiplication des valeurs absolues et d'affecter au résultat un signe positif si les deux opérandes sont de même signe et un signe négatif si les deux opérandes sont de signe inverse (Signe(R) = Signe(A) OUEX Signe(B)).

Il est également possible de réaliser directement la multiplication de nombres représentés en code complément à 2. De telles structures peuvent être établies à partir d'une analyse de l'opération de multiplication sur l'expression des opérandes codées en complément à 2.

$$A = -a_3 2^3 + \sum_{i=0}^{2} a_i 2^i$$

$$B = -b_3 2^3 + \sum_{i=0}^{2} b_i 2^i$$

Exemple de présentation des opérations et de structure en découlant :

$$A * B = a_3 b_3 2^6 + \sum_{i=0}^{2}\sum_{j=0}^{2} a_i b_i 2^{(i+j)} - a_3 2^3 \sum_{i=0}^{2} b_i 2^i - b_3 2^3 \sum_{i=0}^{2} a_i 2^i$$

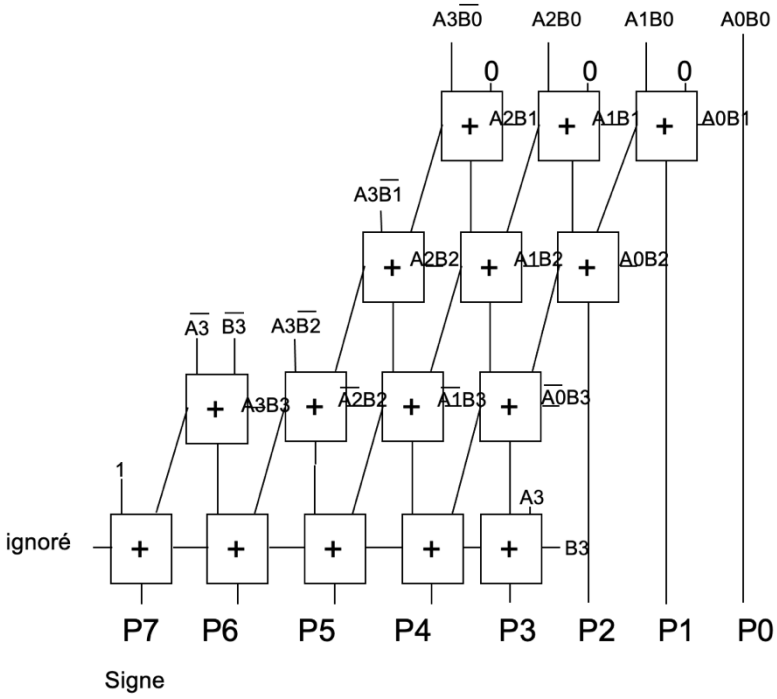
$$A * B = a_3 b_3 2^6 + \sum_{i=0}^{2}. \sum_{j=0}^{2} a_i b_i 2^{(i+j)} + 2^3(-\sum_{i=0}^{2} a_3 b_i 2^{(i)}) + 2^3(-\sum_{i=0}^{2} a_i b_3 2^{(i)})$$

$$C_2(X) = 2^n - X \Rightarrow -X = -2^n + C_2(X)$$
$$\Rightarrow -2^n + C_1(X) + 1$$

$$A * B = a_3 b_3 2^6 + \sum_{i=0}^{2}. \sum_{j=0}^{2} a_i b_i 2^{(i+j)} + 2^3[-2^3 + \sum_{i=0}^{2} \overline{a_3 b_i} 2^{(i)}) + 1] + 2^3[-2^3 + \sum_{i=0}^{2} \overline{a_i b_3} 2^{(i)} + 1]$$

$$A * B = -2^7 + a_3 b_3 2^6 + \sum_{i=0}^{2}\sum_{j=0}^{2} a_i b_i 2^{(i+j)} + \sum_{i=0}^{2} \overline{a_3} b_i 2^{(i+3)} + \sum_{i=0}^{2} \overline{a_i} b_3 2^{(i+3)}$$

$$
\begin{aligned}
A*B = \; & a_0b_0 2^0 + a_1b_0 2^1 + a_2b_0 2^2 \\
& + a_0b_1 2^1 + a_1b_1 2^2 + a_2b_1 2^3 \\
& + a_0b_2 2^2 + a_1b_2 2^3 + a_2b_2 2^4 \\
& + \overline{a_0b_3} 2^3 + \overline{a_1b_3} 2^4 + \overline{a_2b_3} 2^5 \\
& + \overline{a_3b_0} 2^3 + \overline{a_3b_1} 2^4 + \overline{a_3b_2} 2^5 \\
& + 2^4 \qquad + a_3b_3 2^6 - 2^7
\end{aligned}
$$



7.6.4. Multiplieur séquentiel

Supposons que l'on dispose de deux registres de n bits A et B pour stocker les opérandes et de deux registres R1 et R2 pour stocker le résultat final. La multiplication de deux nombres A et B peut être réalisée selon l'algorithme suivant :

```
            (A)  1110
            (B)  1011
             ----------
R2R1        0000 0000          Initialisation de R1 et R2
+A*2^0           1110          b0=1 => on ajoute A*2^0
             ----------
R2R1        0000 1110          Décalage de B => B=(0101)
+A*2^1         1 110            b0=1 => on ajoute A*2^1
             ----------
R2R1        0010 1010          Décalage de B => B=(0010)
+0*2^2                         b0=0 => on ajoute 0*2^2
             ----------
R2R1         0010 1010         Décalage de B => B=(0001)
+A*2^3        111 0            b0=1 => on ajoute A*2^3
             ----------
R2R1          1001 1010        Fin
```

D'après cet algorithme, il faut faire des additions de 2n bits entre R1R2 et A, A étant décalé au maximum de (n-1) positions vers la gauche. En réalité, les additions ne portent que sur des opérandes de n bits (résultat sur n+1 bits), le reste du résultat étant la recopie d'une partie de R1R2. Il est donc possible de n'utiliser qu'un additionneur de n bits.

A devant être décalé de R1 à R2, il faudrait donc le stocker dans un registre de 2n bits. En fait il revient au même de fixer A et de décaler R1R2 vers la droite, à condition de stocker le résultat de l'addition dans R2. On doit donc introduire une bascule R3 (registre 1 bit) pour la retenue éventuelle de l'addition (R2 + A -> R3R2). Cette approche permet de n'utiliser pour A qu'un registre n bits.

```
          (A) 1110
          (B) 1011
            ------------
R3R2R1      0 0000 0000     Initialisation de R3 R2 R1
+A            1110          b0=1 => (R2 + A -> R3R2)
            ------------
R3R2R1      0 1110 0000     Décalage de R3R2R1
R3R2R1      0 0111 0000     Décalage de B => B=(0101)
+A            1110          b0=1 => (R2 + A -> R3R2)
            ------------
R3R2R1      1 0101 0000     Décalage de R3R2R1
R3R2R1      0 1010 1000     Décalage de B => B=(0010)
+0            0000          b0=0 => (R2 + 0 -> R3R2)
            ------------
R3R2R1      0 1010 1000     Décalage de R3R2R1
R3R2R1      0 0101 0100     Décalage de B => B=(0001)
+A            1110          b0=1 => (R2 + A -> R3R2)
            ------------
R3R2R1      1 0011 0100     Décalage de R3R2R1
R3R2R1      0 1001 1010     Fin
```

Si on observe l'évolution de R1 et de B au cours des décalages successifs, on constate que l'on a toujours n zéros qui ne sont d'aucune utilité. On peut encore économiser un registre en plaçant B dans R1, d'où le schéma minimal de la multiplication.

```
        (A) 1110
          ------------
R3R2R1    0 0000 1011      Initialisation de R3R2R1 (B->R1)
+A          1110           b0(R1)=1 => (R2 + A -> R3R2)
          ------------
R3R2R1    0 1110 1011      Décalage de R3R2R1
R3R2R1    0 0111 0101      b0(R1)=1 => (R2 + A -> R3R2)
+A          1110
          ------------
R3R2R1    1 0101 0101      Décalage de R3R2R1
R3R2R1    0 1010 1010      b0(R1)=0 => (R2 + 0 -> R3R2)
+0          0000
          ------------
R3R2R1    0 1010 1010      Décalage de R3R2R1
R3R2R1    0 0101 0101      b0(R1)=1 => (R2 + A -> R3R2)
+A          1110
          ------------
R3R2R1    1 0011 0101      Décalage de R3R2R1
R3R2R1    0 1001 1010       Fin
```

7.7. Division binaire

La division est la plus complexe des 4 opérations arithmétiques. Le principe de la division décimale classique que l'on apprend à l'école est basée sur le principe d'essais successifs. Par exemple, pour diviser 11 par 4, on doit d'abord s'apercevoir que 11 est supérieur à 4 et se demander combien de fois 4 va dans 11. Si l'on fait un essai initial avec 3, la multiplication 3*4=12 nous indique que le choix est mauvais (12>11). Il faut recommencer avec 2 etc...

Compte tenu du moins grand nombre de possibilités d'essais, ce principe d'essais successifs et en fait beaucoup plus simple dans la division binaire.

Exemple :

```
11  | 4
30  | 2,75
20  |
 0  |
```

```
1011     | 100
-100     | 10,1100
 ----    |
 0011    |
 -000    |
  ----   |
  0110   |
  -100   |
   ----  |
   0100  |
   -100  |
    ----
    0000
```

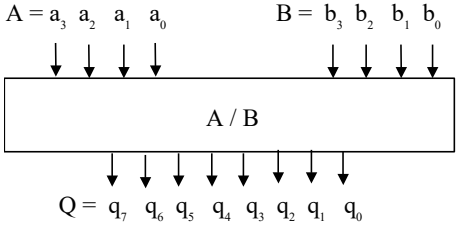
7.7.1. Diviseur cablé

Soit deux nombres binaires A et B. Q représente le quotient de la division de A par B et R les restes partiels. Le principe général de la division binaire peut se mettre sous la forme suivante :

```
Si R>B    Alors Q=1 et R-B
          Sinon Q=0 et R
```

```
                1 0 1 1
0 ⇦  - 0 1 0 0 ↓ | | |
     -------     | | |
       1 0       | | |
0 ⇦  - 0 1 0 0 ↓ | | |
     -------     | | |
       1 0 1     | | |
1 ⇦  - 0 1 0 0 ↓ | |
     -------     | |
       0 0 1 1   | |
0 ⇦  - 0 1 0 0 ↓ | |
     -------     | |
       0 0 1 1 0 | |
1 ⇦  - 0 1 0 0 ↓ |
     -------     |
       0 0 1 0 0 |
1 ⇦  - 0 1 0 0 ↓
     -------
       0 0 0 0 0
0 ⇦  - 0 1 0 0 ↓
     -------
       0 0 0 0 0
0 ⇦  - 0 1 0 0
```

La structure du diviseur binaire peut être directement déduite du principe exposé précédemment.



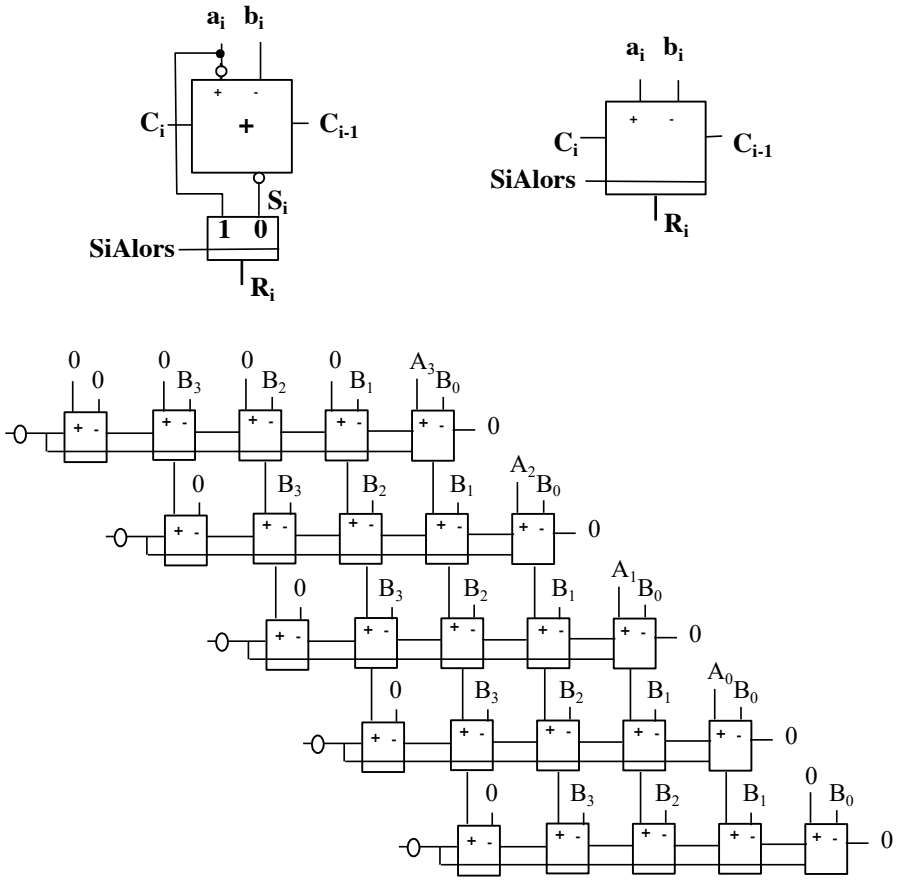


Figure 7.23. *Structure d'un diviseur combinatoire*

7.7.2. Diviseur séquentiel

Tout comme l'opération de multiplication, l'opération de division peut être réalisée de manière séquentielle selon les algorithmes suivants :

7.7.2.a. Algorithme de division avec restauration du dividende

1: Soustraire le diviseur du dividende avec le bits de poids fort du diviseur aligné sur le bit de poids fort du dividende.

	(12/2)	(2/12)
Dividende	1100	10
Diviseur	- 10	- 1100
	-------	--------
	0100	- 0100

2.a : Si le résultat est positif (ou nul) un 1 apparaît dans le quotient. Le poids de ce digit est identique à celui du premier digit droit du dividende sur lequel un digit du diviseur a été soustrait.

Dividende 1**1**00
Diviseur - 1**0**

0100 => Quotient 1..,

Le diviseur est décalé d'un bit vers la droite et est soustrait au dividende partiel (résultat de la soustraction)

```
Dividende partiel     0100
Diviseur            -   10
                    --------
                      0000
```

Le processus est réitéré à partir de là.

2.b : Si le résultat est négatif, un 0 apparaît dans le quotient. Le poids de ce digit est identique à celui du premier digit droit du dividende sur lequel un digit du diviseur a été soustrait.

```
Dividende       10
Diviseur      - 1100
              --------
              - 0100    => Quotient .,.0
```

Le diviseur est alors rajouté au résultat de manière à restaurer le dividende initial.

```
Diviseur        1100
Résultat      - 0100
              --------
                1000
```

Le diviseur est décalé d'un bit vers la droite et est soustrait au dividende restauré.

```
Dividende       1000
Diviseur      - 1100
              --------
              00100
```

Le processus est réitéré à partir de là.

Exemple 1 : (12/5)

```
Dividende               1100
Diviseur              - 101
                      ------
Dividende partiel       0010     => Quotient 1.,
Diviseur décalé       -  101
                      -------
Dividende partiel     - 0011     => Quotient 10,
Ajout du diviseur     +  101
                      -------
Dividende restauré      0010
Diviseur décalé       -   101
                      -------
Dividende partiel     - 00001    => Quotient 10,0
Ajout du diviseur     +   101
                      -------
Dividende restauré      00100
Diviseur décalé       -    101
                       --------
Dividende partiel       000011   => Quotient 10,01
Diviseur décalé       -     101
                      ---------
                        0000001   => Quotient 10,011
```

etc... => Résultat = (1*2 + 0*1 + 0*0,5 + 1*0,25 + 1*0,125 + ...

Exemple 2 : (5/12)

```
Dividende                101
Diviseur               - 1100
                       ------
Dividende partiel      - 0010     => Quotient 0,0
Ajout du diviseur      + 1100
                       -------
Dividende restauré       101
Diviseur décalé        -  1100
                       -------
Dividende partiel      + 01000    => Quotient 0,01
Diviseur décalé        -   1100
                        -------
Dividende partiel      + 000100   => Quotient 0,011
Diviseur décalé        -    1100
                        --------
Dividende partiel      - 0000100    => Quotient 0,0110
Ajout du diviseur        + 1100
                         --------
Dividende  restauré      0001000
Diviseur décalé        -     1100
                         --------
                       - 00000100    => Quotient 0,01100
```

etc... => Résultat = (0*1 + 0*0,5 + 1*0,25 + 1*0,125 + 0*0,0625 +...

7.7.3.b. Algorithme de division sans restauration du dividende

Dans l'algorithme de division sans restauration du dividende, l'étape d'addition du diviseur à un résultat négatif qui permettait de restaurer le dividende initial est supprimée et le diviseur décalé d'une position vers la droite est rajouté au résultat négatif. Cette étape d'addition du diviseur remplace donc les deux étapes de rajout du diviseur et de soustraction du diviseur décalé.

Ceci peut être justifié de la manière suivante : Si R représente le dividende partiel négatif et Y le diviseur alors 1/2Y représente le diviseur décalé d'une position vers la droite.

Algo. avec restauration		Algo. sans restauration
R + Y - 1/2Y	=	R + 1/2Y

Exemple : (12/5)

```
Dividende              1100
Diviseur             - 101
                     ------
Dividende partiel      0010     => Quotient 1.,
Diviseur décalé      -  101
                     -------
Dividende partiel    - 0011     => Quotient 10,
Diviseur décalé      +   101
                     -------
Dividende partiel    - 00001    => Quotient ..10,0
Diviseur décalé      +    101
                     --------
Dividende partiel      000011   => Quotient ..10,01
Diviseur décalé      -     101
                     ---------
                       0000001  => Quotient ..10,011
```

etc... => Résultat = (1*2 + 0*1 + 0*0,5 + 1*0,25 + 1*0,125 + ...