Chapitre

5

Arithmétique binaire

Les codes sont manipulés au quotidien sans qu'on s'en rende compte, et leur compréhension est quasi instinctive. Le seul fait de lire fait appel au codage alphabétique, auquel la civilisation moderne doit son développement rapide. Les nombres aussi sont représentés par des codes, et ces codes sont formés par des chiffres. Il existe à ce titre de multiples codes pour représenter les nombres; ainsi les symboles graphiques 'XX' et '20' représentent tous deux la même quantité : vingt. Dans cet ouvrage, nous entendons le mot code dans sa définition représentative, associant un symbole à un objet, un signifiant à un signifié : '123' n'est pas la quantité cent vingt trois, de la même façon que 'oie' n'est pas l'animal.

Ce chapitre traite plus particulièrement des codes analytiques auxquels les ingénieurs ont recours pour effectuer des opérations arithmétiques. Le chapitre 7 couvrira quant à lui les codes dits représentatifs. Ces notions sont importante car les systèmes numériques que l'on rencontre dans la pratique recourent à ces codes et en respectent les standards.

5.1 Notions

Dans le cadre de notre étude, nous considérons les codes sous une représentation binaire. C'est à dire que nous représenterons tout code avec un ensemble de '0' et de '1'. Ces deux symboles représentent les formes possibles d'un *bit*.

5.1.1 Bit

Bit : Le mot fut utilisé pour la première fois par Claude Shannon dans un article publié en 1948. On attribue cependant son origine à John Wilder Tukey, mathématicien américain, qui inventa également le mot *software*. Bit est une contraction des mots **b**inary dig**it**, ou également **bi**nary uni**t**. Un bit peut prendre deux valeurs possibles, '0' ou '1'. Il est à la base des codes que nous allons présenter.

5.1.2 Mot

Un mot est un ensemble de bits agencés de sorte à représenter un objet dans un code. Le mot '0110000' représente le caractère '0' (zéro) en code Ascii (voir le chapitre 7), ou le nombre 48 dans la représentation dite décimale des entiers.

5.2 Codes analytiques

Les codes analytiques sont utilisés pour représenter les nombres (une quantité numérale). La matière théorique s'y rattachant sera traitée en plusieurs sous-sections. Nous verrons d'abord la théorie des systèmes de numération à base entière *b*, avant de traiter plus particulièrement la base 2. Une fois cette partie de la matière traitée, nous considérerons les représentations des nombres binaires signés, et finalement considérerons les opérations arithmétiques pouvant être utilisées sur ces représentations.

5.2.1 Systèmes de numération

On peut écrire les nombres réels sous différentes bases. La base dix est la plus utilisée. La forme générale s'écrit :

$$N = [\underbrace{a_{n-1}\ a_{n-2}\ \cdots\ a_1\ a_0}_{\text{partie entière, n chiffres}}\ ,\ \underbrace{a_{-1}\ a_{-2}\ \cdots\ a_{-m}}_{\text{partie fractionnaire, m chiffres}}]\underbrace{_{(b)}}_{\text{base}}$$

Les indices des a_i sont associés aux puissances de b et vérifient tous l'inégalité suivante : $0 \leq a_i < b$. Ainsi, la valeur de N est :

$$N = a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + \ldots + a_1 \times b^1 + a_0 \times b^0 + a_{-1} \times b^{-1} + a_{-2} \times b^{-2} + \ldots + a_{-m} \times b^{-m}$$

Considérons à titre d'exemple quelques nombres en base 10 (notre base usuelle) :

$123{,}561_{(10)} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 5 \times 10^{-1} + 6 \times 10^{-2} + 1 \times 10^{-3}$

$= 1 \times 100 + 2 \times 10 + 3 \times 1 + 5 \times 1/10 + 6 \times 1/100 + 1 \times 1/1000$

$= 100 + 20 + 3 + 0.5 + 0.06 + 0.001$

$3623{,}71_{(10)} = 3 \times 10^3 + 6 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 7 \times 10^{-1} + 1 \times 10^{-2}$

$= 3 \times 1000 + 6 \times 100 + 2 \times 10 + 3 \times 1 + 7 \times 1/10 + 1 \times 1/100$

$= 3000 + 600 + 20 + 3 + 0.7 + 0.01$

Utiliser des bases autre que la décimale n'est pas plus compliqué. Regardons à ce titre les exemples suivants :

$123{,}561_{(8)} = 1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 + 5 \times 8^{-1} + 6 \times 8^{-2} + 1 \times 8^{-3}$

$= 1 \times 64 + 2 \times 8 + 3 \times 1 + 5 \times 1/8 + 6 \times 1/64 + 1 \times 1/512$

$= 64 + 16 + 3 + 0{,}625 + 0{,}09375 + 0{,}001953125 = 83{,}720703125$

$3623{,}71_{(16)} = 3 \times 16^3 + 6 \times 16^2 + 2 \times 16^1 + 3 \times 16^0 + 7 \times 16^{-1} + 1 \times 16^{-2}$

$= 3 \times 4096 + 6 \times 256 + 2 \times 16 + 3 \times 1 + 7 \times 1/16 + 1 \times 1/256$

$= 12288 + 1536 + 32 + 3 + 0{,}4375 + 0{,}00390625 = 13859{,}44140625$

Les bases 2, 8 et 16 sont souvent utilisées dans le monde informatique et celui de l'électronique. Leur utilité réside dans le fait que ces bases sont des puissances de 2 et permettent de ce fait de manipuler des bits.

Pour les bases 2 et 8, l'ensemble des chiffres a_i utilisés sont respectivement {0, 1} et {0, 1, 2, 3, 4, 5, 6, 7} (ce qui respecte l'inégalité énoncée précédemment : $0 \leq a_i < b$). Cependant, la base 16 a recours à des chiffres représentant les quantités 10 à 15. L'usage veut que ces quantités soient représentées par les six premières lettres de l'alphabet. On utilise donc l'ensemble {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. Ainsi, le nombre CAF représentera la quantité 3 247 (en base 10) :

$$\begin{aligned} CAF_{(16)} &= C \times 162 + A \times 161 + F \times 160 \\ &= C \times 256 + A \times 16 + F \times 1 \\ &= 12 \times 256 + 10 \times 16 + 15 \times 1 \\ &= 3072 + 160 + 15 = 3247 \end{aligned}$$

Le tableau qui suit illustre l'écriture des nombres 0 à 16 dans les quatre bases usuelles de l'informatique :

Décimal (10)	Binaire (2)	Octal (8)	Hexadécimal (16)
0	00000	00	0
1	00001	01	1
2	00010	02	2
3	00011	03	3
4	00100	04	4
5	00101	05	5
6	00110	06	6
7	00111	07	7
8	01000	10	8
9	01001	11	9
10	01010	12	A
11	01011	13	B
12	01100	14	C
13	01101	15	D
14	01110	16	E
15	01111	17	F
16	10000	20	10

Il est important de noter certains points :

1. Les zéros à gauche de la partie entière n'ajoute et n'enlève rien au nombre : $00010_{(2)}$ et $10_{(2)}$ valent tous deux $2_{(10)}$. Il en est de même pour les zéros à la droite de la partie fractionnaire : $0{,}1_{(2)}$ et $0{,}1000_{(2)}$ valent tous deux $0{,}5_{(10)}$.
2. Quelle que soit la base *b* utilisée, les chiffres 10 donnent un nombre qui vaut *b* : $10_{(b)} = b$
3. Quelle que soit la base *b* utilisée, le nombre auquel on associe le symbole constitué du chiffre 1 suivi d'un nombre k de 0, vaut b à la puissance k : $\underbrace{10000\ldots000}_{\text{k zéros}}{}_{(b)} = b^k$

5.2.2 Conversion entre les bases

Il est utile de pouvoir passer de la représentation d'un nombre dans une base à une autre. Nous étudierons trois méthodes de conversion dans ce manuel.

1. La méthode polynomiale

Pour retrouver la valeur d'un nombre, cette méthode utilise l'expression :

$$N = a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + \ldots + a_1 \times b^1 + a_0 \times b^0 + a_{-1} \times b^{-1} + a_{-2} \times b^{-2} + \ldots + a_{-m} \times b^{-m}$$

Dans la pratique, elle nous sera utile pour exprimer la valeur numérale du nombre N lorsqu'on l'exprime dans tout autre base *b* autre que la décimale. Nous l'avons illustré par de nombreux exemples précédemment et ferons l'économie d'autre exemples (voir la section 5.2.1).

2. La méthode itérative

Cette méthode consiste à convertir un nombre vers une base *b*. Pour ce faire, considérons le nombre N:

$$N = [\underbrace{a_{n-1}\ a_{n-2}\ \cdots\ a_1\ a_0}_{\text{partie entière, n chiffres}}\ ,\ \underbrace{a_{-1}\ a_{-2}\ \cdots\ a_{-m}}_{\text{partie fractionnaire, m chiffres}}]\underbrace{_{(b)}}_{\text{base}}$$

Notons la partie entière E et la partie fractionnaire F.

$E = [a_{n-1}\ a_{n-2} \ldots a_1\ a_0]_{(b)}$

$F = [0\ ,\ a_{-1}\ a_{-2} \ldots a_{-m}]_{(b)}$

Partant du constat que

$E/b = [a_{n-1}\ a_{n-2} \ldots a_1\ a_0]_{(b)}/b = \{ a_{n-1} \times b^{n-1} + a_{n-2} \times b^{n-2} + \ldots + a_1 \times b^1 + a_0 \times b^0\}/b$

$E/b = a_{n-1} \times b^{n-2} + a_{n-2} \times b^{n-3} + \ldots + a_1 \times b^0 + a_0 / b = Q_1$ et reste a_0

$Q_1/b = a_{n-1} \times b^{n-3} + a_{n-2} \times b^{n-4} + \ldots + a_2 \times b^0 + a_1 / b = Q_2$ et reste a_1

$Q_2/b = a_{n-1} \times b^{n-4} + a_{n-2} \times b^{n-5} + \ldots + a_3 \times b^0 + a_2 / b = Q_3$ et reste a_2

Et que le processus peut se poursuivre jusqu'à obtenir a_{n-1}, nous verrons dans cette opération algorithmique une méthode permettant d'exprimer un nombre entier dans une base b.

Les exemples suivants illustrent le concept :

- $57_{(10)} = ?_{(2)}$

$57 \div 2 = 28$ reste **1**
$28 \div 2 = 14$ reste **0**
$14 \div 2 = 7$ reste **0**
$7 \div 2 = 3$ reste **1**
$3 \div 2 = 1$ reste **1**
$1 \div 2 = 0$ reste **1**
d'où $57_{(10)} = \mathbf{111001}_{(2)}$

- $637_{(10)} = ?_{(16)}$

$637 \div 16 = 39$ reste **13** (représenté par **D** en hexadécimal)
$39 \div 16 = 2$ reste **7**
$2 \div 16 = 0$ reste **2**
d'où $637_{(10)} = \mathbf{27D}_{(16)}$

La partie fractionnaire, elle, est traitée inversement :

$F \times b = [a_{-1} \times b^{-1} + a_{-2} \times b^{-2} + \ldots + a_{-m} \times b^{-m}] \times b$
$= a_{-1} + a_{-2} \times b^{-1} + \ldots + a_{-m} \times b^{-m+1} = a_{-1} + F_1$
$F_1 \times b = a_{-2} + a_{-3} \times b^{-1} + \ldots + a_{-m} \times b^{-m+2} = a_{-2} + F_2$
$F_2 \times b = a_{-2} + a_{-3} \times b^{-1} + \ldots + a_{-m} \times b^{-m+2} = a_{-2} + F_3$

Le processus peut se poursuivre jusqu'à obtenir a_{-m}.

Les exemples suivants illustrent le concept :

- $0{,}6875_{(10)} = ?_{(2)}$

$0{,}6875 \times 2 = 1{,}375$ partie entière = **1**
$0{,}375 \times 2 = 0{,}75$ partie entière = **0**
$0{,}75 \times 2 = 1{,}5$ partie entière = **1**
$0{,}5 \times 2 = 1{,}0$ partie entière = **1**
d'où $0{,}6875_{(10)} = \mathbf{0{,}1011}_{(2)}$

- $0{,}8125_{(10)} = ?_{(8)}$

$0{,}8125 \times 8 = 6{,}5$ partie entière = **6**
$0{,}5 \times 8 = 4{,}0$ partie entière = **4**
d'où $0{,}8125_{(10)} = \mathbf{0{,}64}_{(8)}$

3. *Cas des bases* b *et* b^k

Cette méthode est utilisée pour passer d'une base b à une base b^k, et vice versa. Le cas le plus courant est celui ou il s'agit de passer d'une représentation binaire à une représentation octale ou hexadécimale, ou de procéder à l'opération inverse.

Afin de mieux illustrer le concept, considérons les exemples suivants :

$1101001_{(2)} = 1\ 101\ 001_{(2)} = 001\ 101\ 001_{(2)} = 151_{(8)}$
$100100111_{(2)} = 1\ 0010\ 0111_{(2)} = 0001\ 0010\ 0111_{(2)} = 127_{(16)}$
$110111101111_{(2)} = 1101\ 1110\ 1111_{(2)} = DEF_{(16)}$

Le principe utilisé ici est de regrouper k chiffres pour passer de b à b^k. L'inverse est également applicable, comme l'illustrent les exemples suivants :

$1EC5_{(16)} = 0001\ 1110\ 1100\ 0101_{(2)} = 0001111011000101_{(2)} = 1111011000101_{(2)}$
$1672_{(8)} = 001\ 110\ 111\ 010_{(2)} = 001110111010_{(2)} = 1110111010_{(2)}$

Dans ce cas, on traduit chaque chiffre en k chiffres binaires. Notons finalement que l'opération est tout aussi faisable pour les nombres à partie fractionnaires, comme l'illustrent les exemples suivants, où la virgule est considérée comme un délimiteur pour rassembler les k chiffres.

$1101001{,}1001_{(2)} = 1\ \ 101\ 001\ ,\ 100\ 1_{(2)} = 001\ 101\ 001\ ,\ 100\ 100_{(2)} = 151{,}44_{(8)}$
$100100111{,}011001_{(2)} = 1\ 0010\ 0111\ ,\ 0110\ 01_{(2)} = 0001\ 0010\ 0111\ ,\ 0110\ 0100_{(2)} = 127{,}64_{(16)}$
$11011110{,}11101_{(2)} = 1101\ 1110\ ,\ 1110\ 1000_{(2)} = DE{,}E8_{(16)}$

5.2.3 Nombres entiers négatifs

Le problème de la représentation des nombres négatifs se pose assez rapidement lorsque l'on veut manipuler des mots formés par des bits à cause de l'absence des symboles +/--. Il existe plusieurs approches pour résoudre le problème. La représentation des nombres entiers négatifs en complément à deux est la plus pratique et la plus utilisée de toutes. La représentation des nombres négatifs en complément à deux consiste à utiliser un bit supplémentaire (noté s) pour le signe pour un total de n+1 bits à gauche de la virgule et m bits à droite :

$$\overline{\overline{N}} = s\ a_{n-1}\ a_{n-2} \ldots a_1\ a_0\ ,\ a_{-1}\ a_{-2} \ldots a_{-m}$$

$\overline{\overline{N}}$ désigne ici la représentation d'un nombre négatif N. On dira de $\overline{\overline{N}}$ qu'il est l'opérateur de complément à deux de la valeur de $|N|$ représentée sur n +1 bits à gauche de la virgule et m bit à droite de la virgule. La valeur numérale associée au mot ainsi obtenu est donnée par la relation :

$$\overline{\overline{N}} = 10^{n}_{(2)} - |N|$$

$\overline{\overline{N}}$ est le complément à 2 du nombre $|N|$. On relèvera alors que :

$$\overline{\overline{N}} = 10^{n}_{(2)} - |N| = 10^{n}_{(2)} - |N| - 10^{-m}_{(2)} + 10^{-m}_{(2)}$$

Définissons l'opérateur dit de complément à un qui consiste à inverser les bits de $|N|$, l'opérateur de complément est donné par la relation :

$$\overline{N} = 10^{n}_{(2)} - |N| - 10^{-m}_{(2)}$$

Il en résulte l'écriture suivante de l'opérateur de complément à deux :

$$\overline{\overline{N}} = (10^{n}_{(2)} - |N| - 10^{-m}_{(2)}) + 10^{-m}_{(2)} = \overline{N} + 10^{-m}_{(2)}$$

Ce qui revient à dire que le complément à deux peut être effectué par un complément à un (inversion de tous les bits) suivi d'une addition d'une unité égale à la valeur du bit le moins significatif.

Le procédé sera clarifié à l'aide des quelques exemples qui suivent :

- Comment écrit-on N = -15 (5 bits + 1 bit de signe) ?
  $|N| = 15 = 001111_{(2)} \Rightarrow \overline{N} = 110000_{(2)} \Rightarrow -15 \equiv \overline{\overline{N}} = 110000_{(2)} + 000001_{(2)} = 110001_{(2)}$
- Comment écrit-on -35 (6 bits + 1 bit de signe) ?
  $|N| = 35 = 0100011_{(2)} \Rightarrow \overline{N} = 1011100_{(2)} \Rightarrow -35 \equiv \overline{\overline{N}} = 1011100_{(2)} + 0000001_{(2)} = 1011101_{(2)}$

Le bit de signe des nombres négatifs dans la représentation binaire signée prend la valeur 1. Les nombres positifs sont représentés en binaire signé en posant le bit de signe à 0 et en gardant la représentation non signée :

- Comment écrit-on +15 en binaire signé (5 bits + 1 bit de signe) ?
  $15 = 001111_{(2)}$
- Comment écrit-on +35 en binaire signé (6 bits + 1 bit de signe) ?
  $35 = 0100011_{(2)}$

De ce fait, pour un nombre de bits donné, certaines valeurs sont impossible à représenter en binaire signé. Par exemple :

- Il est impossible de représenter +15 en binaire signé avec 3 bits + 1 bit de signe.
- Il est impossible de représenter –15 en binaire signé avec 3 bits + 1 bit de signe.
- Il est impossible de représenter +35 en binaire signé avec 4 bits + 1 bit de signe.
- Il est impossible de représenter –35 en binaire signé avec 4 bits + 1 bit de signe.

En admettant une acception plus large de l'opérateur de complément à deux :

$$\overline{\overline{N}} = 10^{n}_{(2)} - N - 10^{-m}_{(2)} + 10^{-m}_{(2)}$$

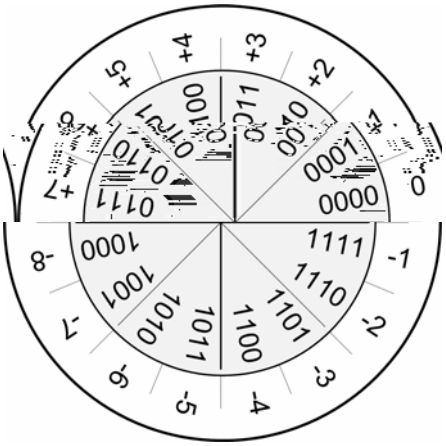
On trouve que le complément à deux est involué :

$$\overline{\overline{(N)}} = N$$

Le tableau qui suit montre, à titre d'illustration, l'ensemble des nombres représentables avec 4 bits (3 bits + 1 bit de signe) en binaire signé.

-8	1000	**-4**	1100	**0**	0000	**+4**	0100
-7	1001	**-3**	1101	**+1**	0001	**+5**	0101
-6	1010	**-2**	1110	**+2**	0010	**+6**	0110
-5	1011	**-1**	1111	**+3**	0011	**+7**	0111

Il est intéressant de relever que l'écriture en complément à 2 offre un dénombrement cyclique :



5.2.4 Addition et soustraction

Comme nous l'avions précisé précédemment, les codes analytiques sont utilisés pour leur signification numérale. Ainsi, ils sont utilisés dans des opérations arithmétiques, suivant des algorithmes que nous voudrons connaître.

1. Formulation naïve de l'addition

Considérons d'abord le principe d'addition dans une formulation un peu naïve. Supposons que nous voulions additionner deux mots binaires d'une même longueur (par exemple 16). Pour ce faire, on commencera par poser la table d'addition binaire :

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

Grâce à cette table, il devient possible d'effectuer certaines opérations d'addition de la même façon que cela se fait couramment en base dix:

Retenues								1	1	1	1	1	1	1	1		
A		1	0	1	0	1	0	1	0	1	0	0	1	0	0	1	1
B	+	0	1	0	0	0	1	0	0	0	1	1	0	1	1	0	1
Somme		1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0

En passant d'un colonne à la suivante, il peut arriver que l'on ait à additionner trois 1. Dans ce cas, la table d'addition binaire doit être légèrement modifiée. À la colonne *i*, nous aurons :

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

Grâce à cette nouvelle table, il devient possible d'effectuer toutes opérations d'addition binaire :

Retenues					1			1	1	1	1	1	1	1	1		
A		1	0	1	0	1	1	1	0	1	0	0	1	0	0	1	1
B	+	0	1	0	0	1	1	0	0	0	1	1	0	1	1	0	1
Somme		1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Comme les mots ont une limite de longueur, on dira que le résultat a provoqué une retenue si il nous faut un bit supplémentaire pour le conserver. Par exemple :

Retenues					1			1	1	1	1	1	1	1	1		
A		1	0	1	0	1	1	1	0	1	0	0	1	0	0	1	1
B	+	0	1	0	0	1	1	0	0	0	1	1	0	1	1	0	1
Somme		1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Le résultat de cette addition a nécessité 17 bits. Or si nous limitons la longueur des mots à 16 bits, cette addition ne serait pas possible.

2. Addition avec retranchement de la retenue

Posons une addition qui provoque un dépassement de retenue (retenue supplémentaire sur une longueur de bits prédéfinie :

Retenues			1				1			1	1	1	1	1	1	1	1	
A			1	1	1	0	1	1	1	0	1	0	0	1	0	0	1	1
B	+		0	1	0	0	1	1	0	0	0	1	1	0	1	1	0	1
Somme		1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

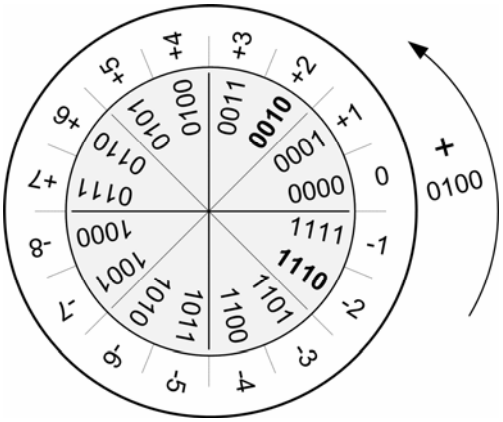
Le résultat de l'addition nécessite 17 bits. En admettant un retranchement de la retenue supplémentaire, on obtient l'algorithme de l'addition avec retranchement de retenue :

Retenues			1				1			1	1	1	1	1	1	1	1	
A			1	1	1	0	1	1	1	0	1	0	0	1	0	0	1	1
B	+		0	1	0	0	1	1	0	0	0	1	1	0	1	1	0	1
Somme		✗	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

La cyclicité du complément à deux dans la représentation des nombres signés permet d'utiliser correctement l'addition avec retranchement de retenue pour effectuer toutes les opérations d'addition. Par exemple :

Retenues			1			
A			1	1	1	0
B	+		0	1	0	0
Somme		✗	0	0	1	0

Ce qui s'illustre bien graphiquement :



Considérons un exemple un peu plus élaboré que le lecteur est invité à détailler :

Termes	Écriture binaire	Complément à 2
A	1110101010010011	-5485
B	0100010001101101	17517
Résultat	0010111100000000	12032
Résultat réel	---	12032

Aussi étonnant que cela puisse sembler, le simple retranchement du 17e bit a pour effet de rendre le résultat correct dans le cas du complément à 2.

3. Addition et soustraction dans le cas du complément à 2

La soustraction $N_1 - N_2$ peut être regardée comme une addition en inversant l'opérande de droite. Posons $\overline{\overline{N_2}}$ complément à 2 (inverse) de N_2. Selon la définition du complément à 2, nous avons :

$$N_1 + \overline{\overline{N_2}} = N_1 + (10^n_{(2)} - N_2)$$
$$N_1 + \overline{\overline{N_2}} = 10^n_{(2)} - (N_2 - N_1)$$

Si N_1 - N_2 est positif, en retirant la retenue supplémentaire, on élimine la valeur $(10^n_{(2)})$ et on obtient la bonne réponse :

$$N_1 + \overline{\overline{N_2}} = N_1 - N_2$$

Si N_1 - N_2 est négatif, $10^n_{(2)} - (N_2 - N_1)$ est positif et il n'y a pas de retenue supplémentaire. Sachant que :

$$N_1 + \overline{\overline{N_2}} = 10^n_{(2)} - (N_2 - N_1)$$
$$N_1 + \overline{\overline{N_2}} = 10^n_{(2)} - | N_2 - N_1 |$$
$$N_1 + \overline{\overline{N_2}} = \overline{\overline{N_2 - N_1}}$$

et le résultat obtenu est correct.

Encore une fois, nous allons nous appuyer sur des exemples pour illustrer la chose :

01101010	01101010	10101010
- 01100001	+ 10011111	+ 10011111
=----------	=100001001	= 00001001
Soustraction à effectuer	**Addition $N_1 + \overline{\overline{N_2}}$**	**Retranche la retenue donne le résultat**

L'équivalent décimal est 106 – 97 = 9

Inversons maintenant les termes de l'addition :

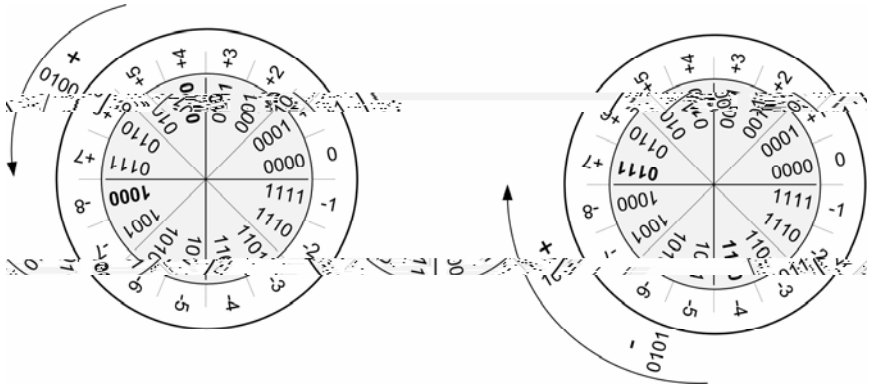
01100001	01100001
- 01101010	+ 10010110
=----------	= 11110111
Soustraction à effectuer	**Addition $N_1 + \overline{\overline{N_2}}$: pas de retenue? On laisse cela tel quel**

Et à nouveau, le résultat ici (11110111) est -9.

5.2.5 Dépassement

Une limitation intrinsèque des systèmes logiques est, comme nous avons pu le mentionner, la longueur des mots dans un code. Dans le cas des codes analytiques, cette limitation n'est pas à écarter et doit être considérée avec prudence Un problème important survient dans le cas de l'addition et la soustraction lorsque le résultat ne peut être représenté selon la longueur des mots binaires.

Supposons que nous disposions d'un code de 4 bits pour représenter les nombres en binaire signé. Cela signifie que nous disposons des nombres allant de -8 à +7. Si le résultat de l'opération dépasse ces limites, nous faisons face à un cas de débordement. Par exemple, si nous additionnons +4 et +4, le résultat est +8 et n'est pas représentable en complément à 2 sur 4 bits. Il en est de même lorsque nous additionnons -4 et -5.



0100 + 0100 = 1000 (résultat invalide) 1100 + 1011 = 0111 (résultat invalide)

Les dépassements surviennent uniquement lorsque l'on additionne deux valeurs positives ou deux valeurs négatives. On peut les reconnaître par l'incohérence du résultat final qui se trouve être négatif dans le cas de l'addition de deux nombres positifs, et négatif dans le cas de l'addition de deux nombres positifs.

5.3 Circuits usuels arithmétiques

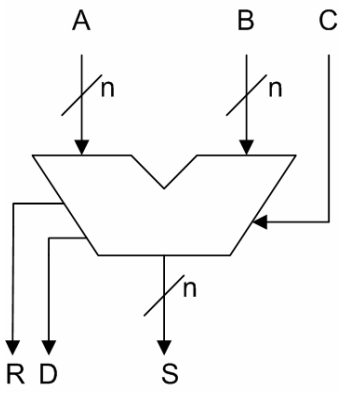
5.3.1 L'additionneur

L'additionneur est un circuit arithmétique permettant, comme l'indique son nom, d'additionner deux nombres binaires A et B. Les signaux A et B doivent avoir la même longueur (nombre de bits) et donnent un résultat de même longueur plus un bit de retenue supplémentaire.

L'additionneur est un circuit complexe qui a mérité une attention particulière dans le domaine des circuits numérique. Cette complexité est certes due aux normes régissant l'addition (représentation des nombres, format choisi pour représenter les nombres négatifs), et nous tâcherons d'aborder la conception du circuit par étapes de sorte à éclairer au mieux les connaissances théoriques connues jusqu'à présent.

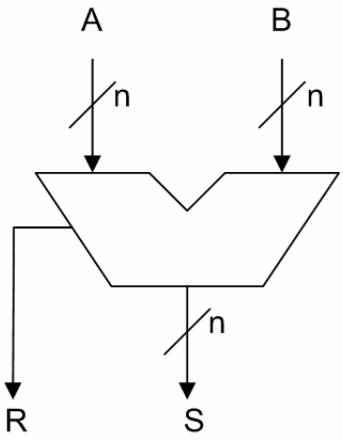
L'additionneur comprend deux entrées (A et B) de longueur n. Il comporte également une sortie correspondant au résultat de l'addition (S) de longueur n, ainsi que deux signaux de sortie, le signal (R) pour la retenue supplémentaire et le signal (D) pour le débordement. On va également vouloir utiliser le même circuit pour effectuer la soustraction. On ajoute donc en entrée un signal de contrôle (C) permettant d'effectuer l'opération A - B plutôt que A + B.

Nous présentons ici le schéma global d'un additionneur :

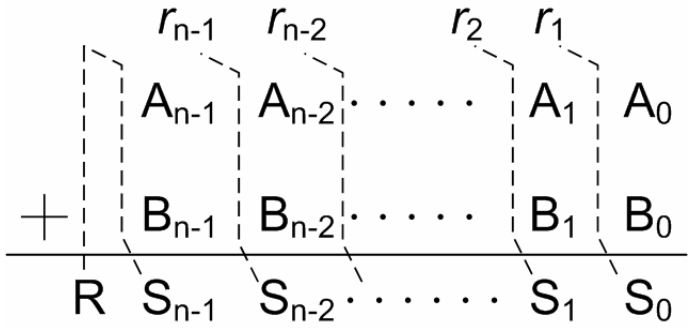


Nous allons faire appel à la conception itérative pour réaliser ce circuit. Mais nous commencerons par simplifier le problème en retirant les signaux (C) et (D), et les ajouterons au fur à mesure que nous pourrons enrichir notre implémentation.

Nous connaissons déjà un algorithme simple pour effectuer l'addition, celui de ce que nous avions appelé la formulation naïve de l'addition :



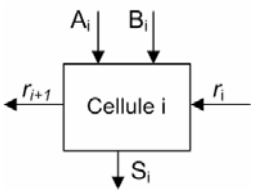
Additionneur simple



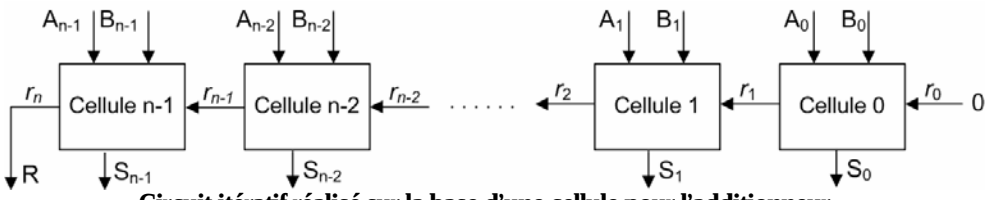
Algorithme itératif pour l'additionneur simple

Comme on peut s'en rendre compte, l'addition est réalisée au niveau de chaque bit et le résultat intermédiaire est propagé itérativement d'un niveau à l'autre jusqu'à obtenir le résultat total. À l'exception du premier bit, pour chaque niveau *i* nous considérons trois termes : A_i, B_i et r_i qui donnent le résultat S_i et une retenue pour le niveau suivant r_{i+1}. Le résultat R est récupéré du signal r_n mais n'est pas considéré pour appliquer l'algorithme d'addition avec retranchement de la retenue supplémentaire.

On peut concevoir une cellule à trois entrées et deux sorties pour réaliser l'ensemble du circuit :



Cellule i



Circuit itératif réalisé sur la base d'une cellule pour l'additionneur

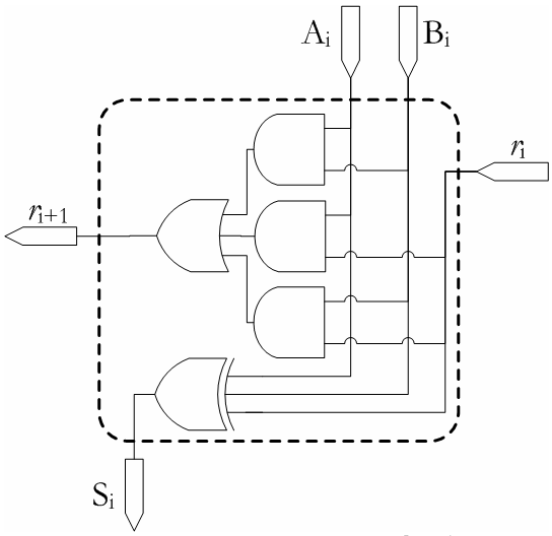
La table de vérité associée à cette cellule est la suivante :

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

On trouve alors que les signaux de sorties sont donnés par les équations :

- $S_i = A_i \oplus B_i \oplus r_i$
- $r_{i+1} = A_iB_i + A_ir_i + B_ir_i$

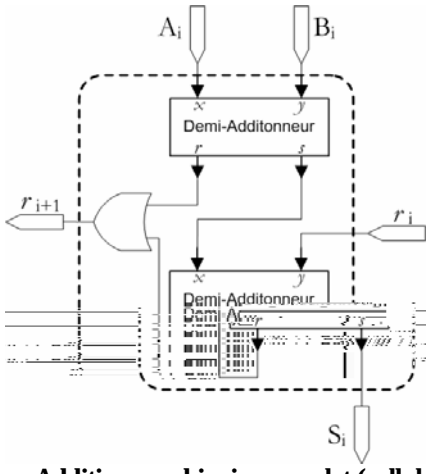
Ce qui nous donne le circuit :



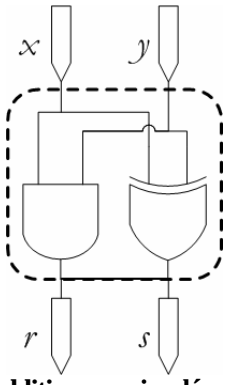
Cellule i de l'additionneur itératif

Mais cette implémentation n'est pas optimisée. La littérature en propose une autre qui la divise en deux demi-cellules (appelées demi-additionneurs binaires) au prix d'une réalisation multi-niveaux.

Le demi-additionneur prend deux entrées au lieu de trois, et le circuit complet est le suivant :



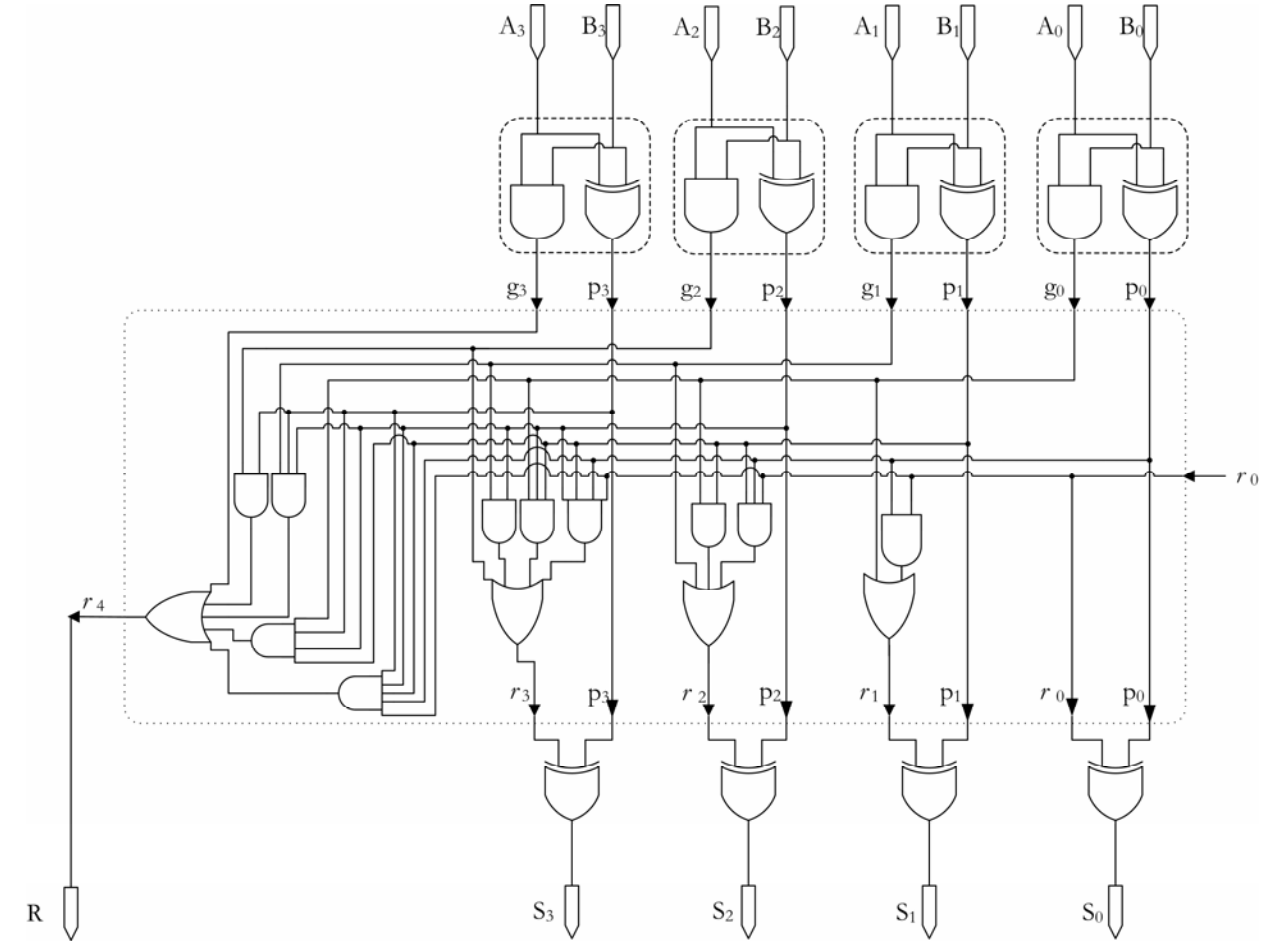
Additionneur binaire complet (cellule i) avec demi additionneurs



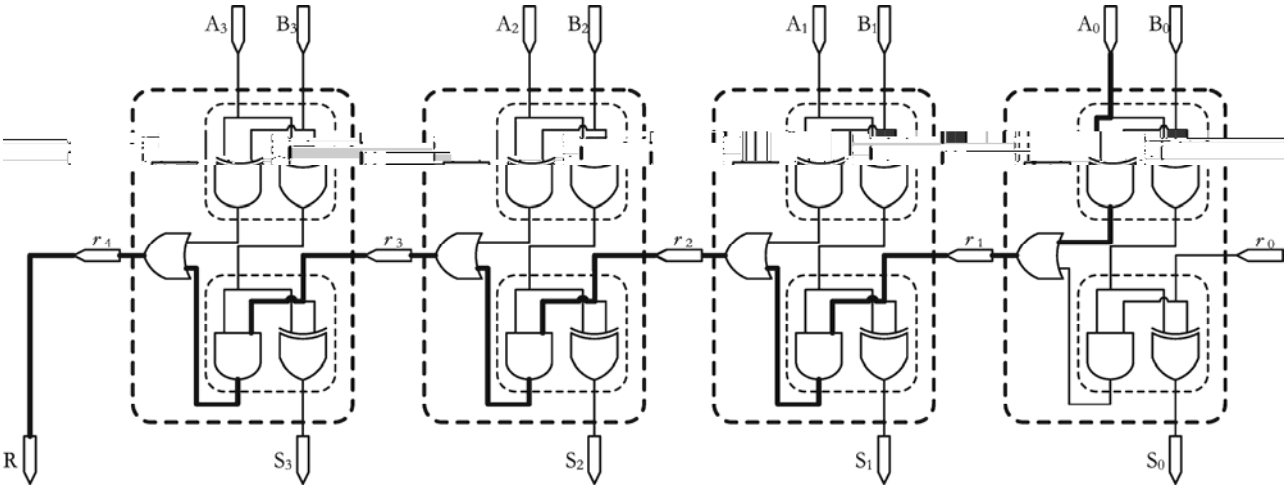
Demi Additionneur, implémentation

On se rappellera la table d'addition utilisée à la section 5.2.4 qui est aussi la table de

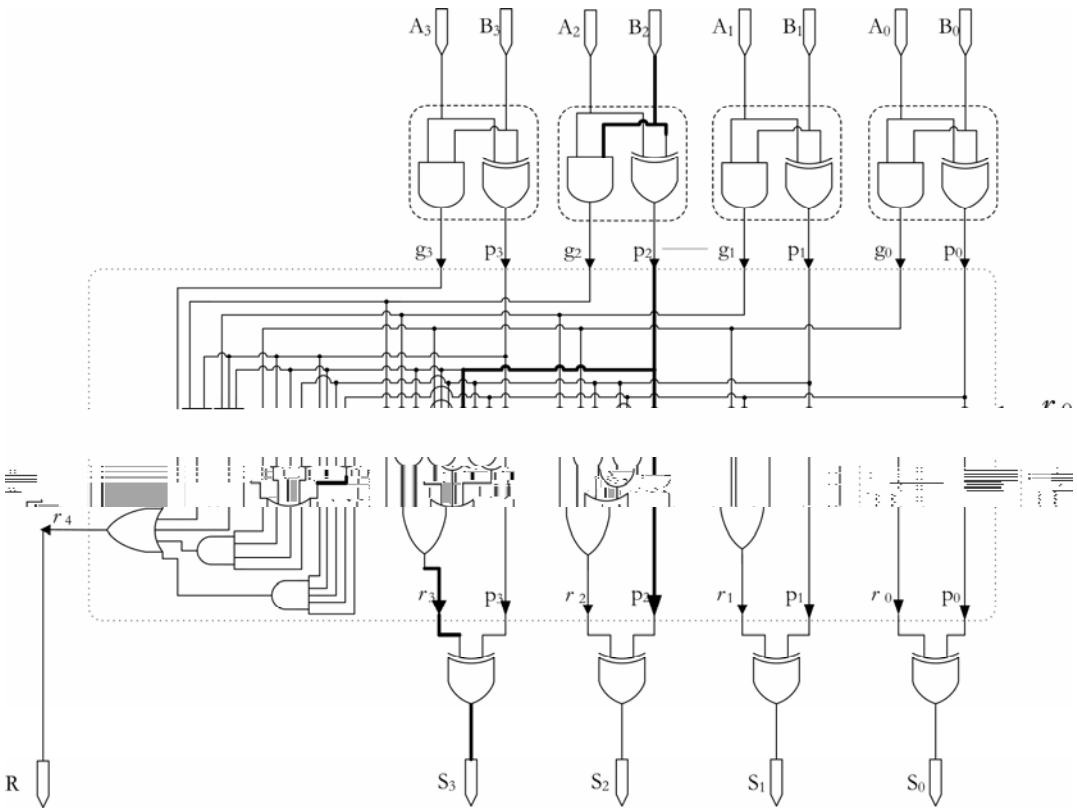
pour générer les s_i). On appellera ce circuit un anticipateur de retenue car il va permettre d'accélérer le calcul) du signal de retenue (R). Cet anticipateur de retenue va nous permettre de réaliser un additionneur rapide :



Si nous reprenons le circuit classique de l'additionneur :

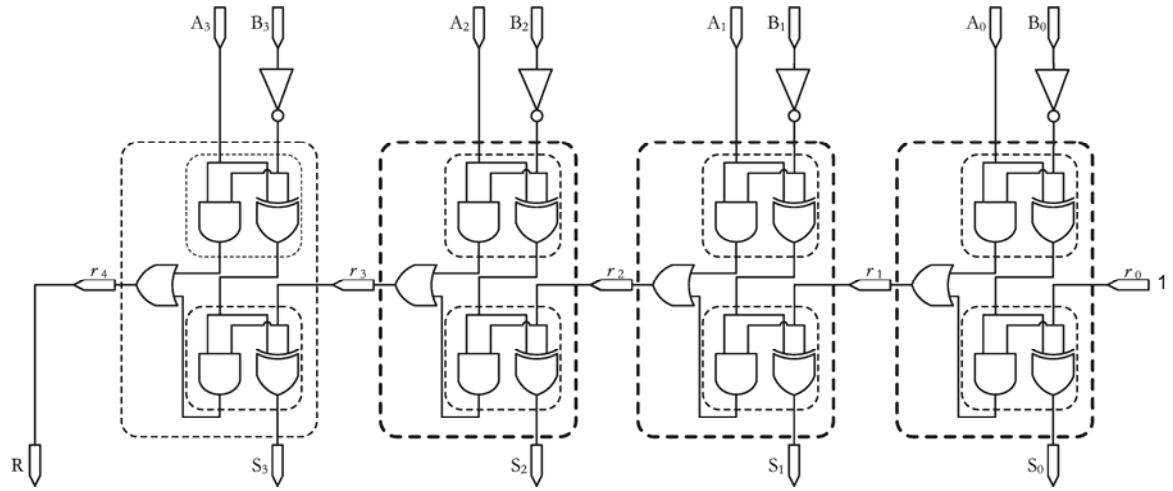


et celui de l'additionneur rapide utilisant un anticipateur de retenue :



Le coût du circuit rapide est plus élevé mais le chemin critique traverse quatre portes au plus. De plus, le chemin de r_0 à R est très court.

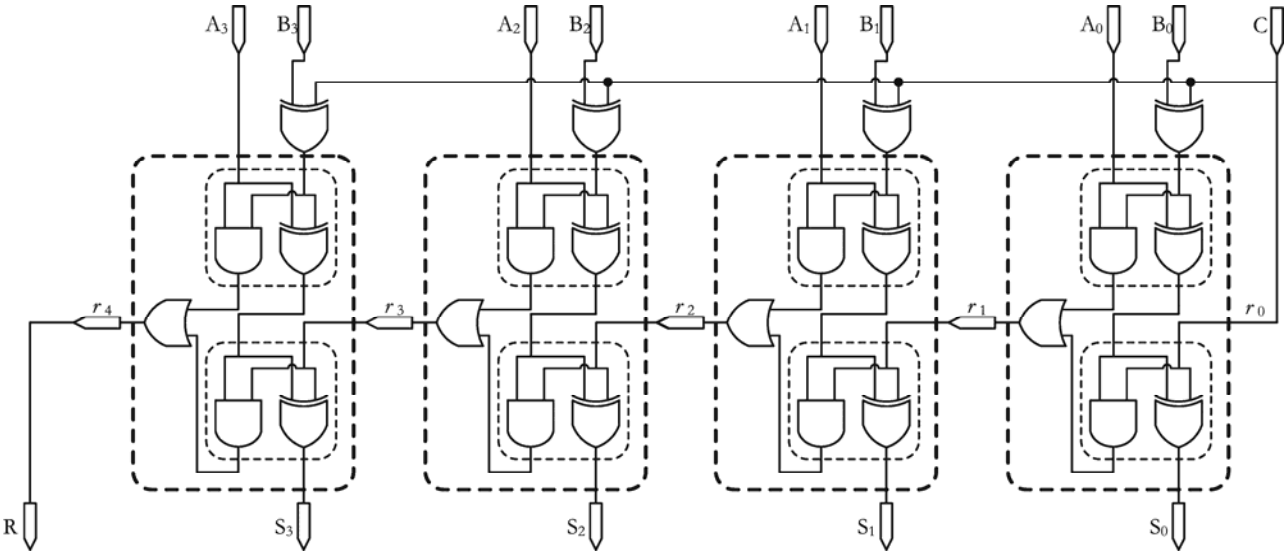
Il reste encore à discuter de l'option d'effectuer la soustraction. Afin de représenter les nombres négatifs, nous choisissons la représentation en complément à 2. On se rappelle que pour inverser un nombre en complément à 2 il suffit d'inverser tous les bits et d'ajouter 1 au bit le moins significatif. Aussi, pour obtenir un soustracteur, il suffit d'inverser les bits de B et de mettre 1 dans r_0. Que l'on choisisse l'additionneur itératif ou l'additionneur rapide avec anticipateur de retenue, la solution demeure identique :



Cependant, nous voulons commander l'opération de soustraction par le signa (C). Or se rappellera que :

- $X \oplus 1 = \overline{X}$
- $X \oplus 0 = X$

On aura donc une addition si C = 0 et une soustraction pour C=1 avec le circuit suivant:

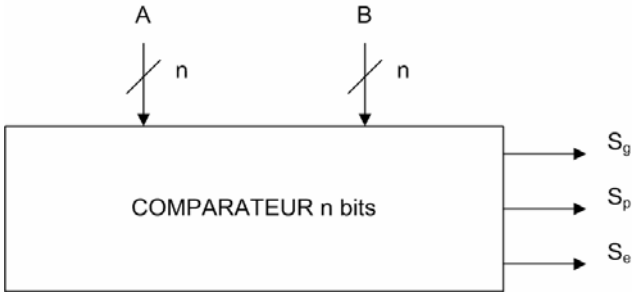


Finalement, le signal (D) s'obtient en effectuer un XOR entre les signaux r_n et r_{n-1}.

5.3.2 Le comparateur

Le comparateur est un circuit arithmétique permettant de comparer deux nombres binaires A et B. A et B doivent avoir la même longueur (nombre de bits). On cherche à savoir si A > B, A<B ou A=B. On comprend donc que le circuit répond à une question à trois choix.

Notre circuit final doit produire trois signaux S_g (actif si A>B), S_p (actif si A<B) et S_e (actif si A=B) en prenant en entrée les signaux A et B :



Ce circuit compare les deux entrées A et B et donne :

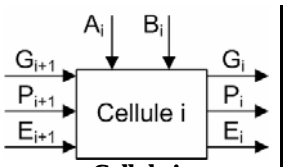
- $S_g\ S_p\ S_e = 100$ si A>B
- $S_g\ S_p\ S_e = 010$ si A<B
- $S_g\ S_p\ S_e = 001$ si A=B

La conception d'un tel circuit n'est pas évidente, d'autant plus que la longueur des mots A et B n'est pas connue à priori. On veut donc utiliser une conception modulable, permettant de générer le circuit selon les contraintes imposées. Une méthode de conception flexible et souvent utilisée dans la conception de circuits arithmétiques est l'utilisation de circuits itératifs.

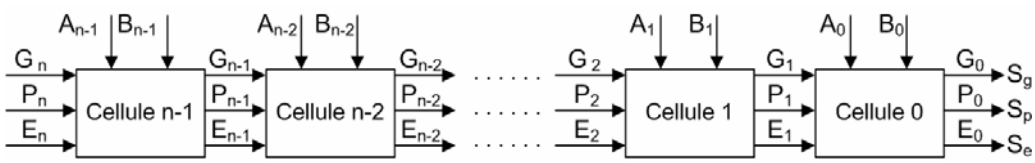
Les circuits itératifs sont l'application en circuits logiques d'algorithmes itératifs. Le principe général est de trouver une méthode ou un algorithme agissant sur un seul bit et de le répéter de façon itérative pour obtenir au final le résultat voulu pour une longueur de mot (en bits) donnée.

Si nous considérons le problème de la comparaison de deux nombres binaires, on constate que cette comparaison peut se réaliser en effectuant la comparaison depuis le bit le plus significatif et decrescendo jusqu'au bit le moins significatif.

Considérons pour cela la cellule du circuit itératif suivante :



Cellule i



Circuit itératif réalisé sur la base d'une cellule pour le comparateur

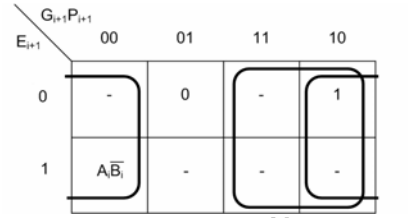
A_i et B_i correspondent aux bits unitaires des mots A et B. Les signaux intermédiaires G_i, P_i et E_i permettant aux cellules de communiquer entre elles. On associe les signaux G_i, P_i et E_i aux réponses intermédiaires du système, respectivement « Plus **G**rand que… », « Plus **P**etit que… » et « **É**gal… ». Dans la réalisation du circuit complet, on pose $G_n=P_n=0$, et $E_n=1$. Les signaux terminaux G_0, P_0 et E_0 correspondent aux sorties du comparateur, respectivement S_g, S_p et S_e. Au niveau de chaque cellule, on compare les signaux A_i et B_i et décide de la sortie en fonction de la réponse intermédiaire à donner (G_i, P_i et E_i) en fonction de la réponse intermédiaire de la cellule précédent (G_{i+1}, P_{i+1} et E_{i+1}).

- Si la réponse précédente est $G_{i+1}P_{i+1}E_{i+1}$ = 100, la sortie sera $G_iP_iE_i$ = 100 (car la valeur des bits moins significatifs n'a pas d'importance dans ce cas).
- Si la réponse précédente est $G_{i+1}P_{i+1}E_{i+1}$ = 010, la sortie sera $G_iP_iE_i$ = 010 (car la valeur des bits moins significatifs n'a pas d'importance dans ce cas).
- Si la réponse précédente est $G_{i+1}P_{i+1}E_{i+1}$ = 001, la sortie sera $G_iP_iE_i$ dépend de A_i et B_i. :
  - Si $A_i=B_i$ ⇨ $G_iP_iE_i$ = 001
  - Si A_iB_i=10 ⇨ $G_iP_iE_i$ = 100
  - Si A_iB_i=01 ⇨ $G_iP_iE_i$ = 010

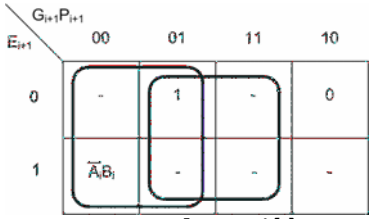
Cela se traduit par la table de vérité que voici :

G_{i+1}	P_{i+1}	E_{i+1}	A_i	B_i	G_i	P_i	E_i
1	0	0	-	-	1	0	0
0	1	0	-	-	0	1	0
0	0	1	0	0	0	0	1
0	0	1	0	1	0	1	0
0	0	1	1	0	1	0	0
0	0	1	1	1	0	0	1

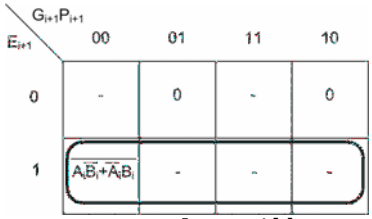
Tous les autres cas sont facultatifs. On peut associer à cette table de vérité les trois tables de Karnaugh à variables inscrites suivantes :



Karnaugh associé à G_i



Karnaugh associé à P_i

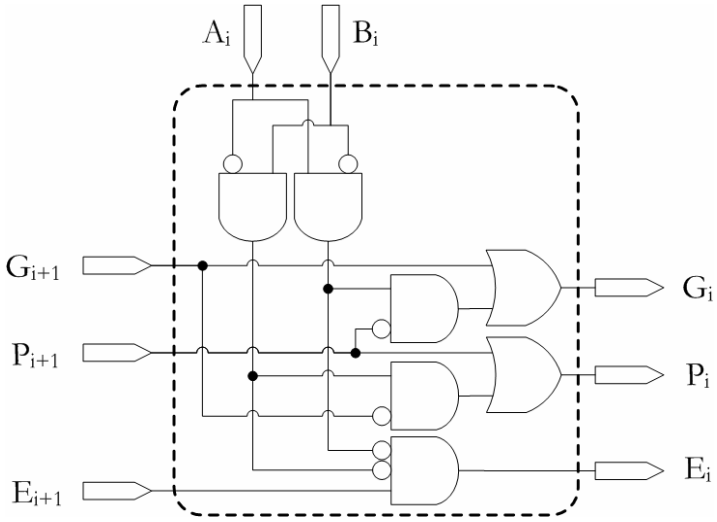


Karnaugh associé à E_i

On trouve donc :

$G_i = A_i \overline{B_i}\ \overline{P_{i+1}} + G_{i+1}$; $P_i = \overline{A_i}\ B_i \overline{G_{i+1}} + P_{i+1}$; $E_i = \overline{A_i \overline{B_i} + \overline{A_i} B_i}$ $E_{i+1} = \overline{A_i \overline{B_i}}\ \overline{\overline{A_i} B_i}\ E_{i+1}$

Ce qui nous donne le circuit suivant :



Notons que le circuit itératif aurait pu être pensé de façon inverse, en effectuant la comparaison depuis le bit le plus moins significatif et crescendo jusqu'au bit le plus significatif.