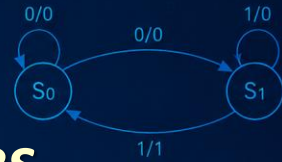


IV. Opérateurs arithmétiques

Principes et structures



ENSGTI
ÉCOLE D'INGÉNIEURS

Arnaud Martin

1

1



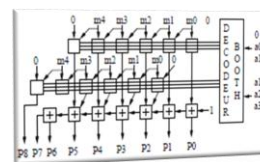
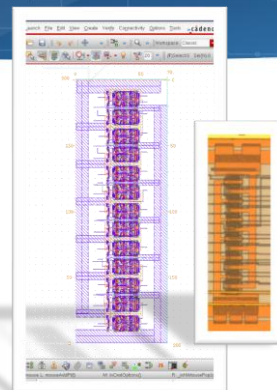
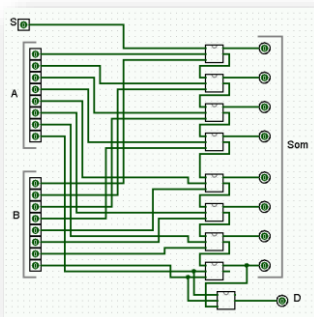
IV. Opérateurs
arithmétiques

Opérateurs arithmétiques

Additionneur

Soustracteur

Multiplieur



2

2

Cas d'un additionneur :

N nombre de bits des données

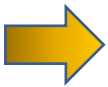
Table de vérité à $2 \times N$ entrées et $N+1$ sorties

$N+1$ fonctions logiques, table à 2^{2N} lignes

AN : N=8 $\Rightarrow$ 65536 de lignes

N=12 $\Rightarrow$ 16,78 millions de lignes

N=16 $\Rightarrow$ 4,3 milliards de lignes



On ne peut pas avoir une approche à partir des tables de vérité. Par ailleurs, une approche à partir des tables de vérité ne permettrait pas de définir les temps de propagation (temps de calcul) des opérateurs.

On crée et on assemble des briques de base.



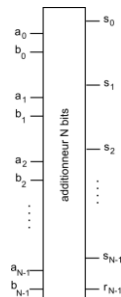
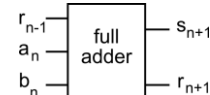
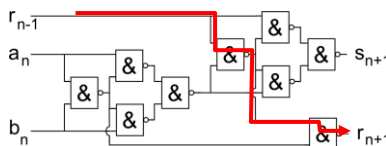
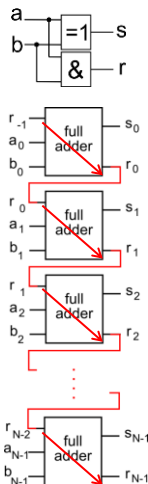
Structures ordonnées, cascadables à partir de sous-structures élémentaires, souvent calquées sur l'opération arithmétique élémentaire.

Remarque : on peut alors prévoir le temps de propagation (temps de calcul), la consommation de puissance et la surface (le coût)

3

3

Additionneur à propagation simple (cf TD-TP)



La propagation de la retenue limite la vitesse de l'opérateur.

Temps de propagation de la retenue :

(Tp délai d'une porte logique (ex: NAND))

4

4

Additionneur à retenue anticipée (carry look-ahead adder)

$$r_i = a_i \cdot b_i + r_{i-1} \cdot (a_i + b_i) \text{ expression de la retenue de rang } i$$

On a une retenue au rang i : si l'on propage une retenue de rang $i-1$ ou si le full adder i génère une retenue.

On introduit :

$P_i = a_i + b_i$ propagation de rang i

$G_i = a_i \cdot b_i$ génération de rang i

Alors :

$$r_i = G_i + r_{i-1} \cdot P_i \text{ expression de la retenue de rang } i$$

Les P_i et G_i peuvent également servir à calculer la somme s_i .

$$a_i \oplus b_i = G_i \oplus P_i = \overline{G_i} \oplus P_i ; \text{ alors } s_i = a_i \oplus b_i \oplus r_{i-1} = G_i \oplus P_i \oplus r_{i-1} = \overline{G_i} \cdot P_i \oplus r_{i-1}$$

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

5

5

Grâce à un **circuit anticipateur de retenue**, on peut calculer la retenue de sortie en 3Tp. Le mécanisme d'anticipation peut s'appliquer par bloc de bits.

On considère 4 fulladder 1 bits, d'indice i à $i+3$.

On a une retenue dans $i+3$ si:

$$r_{i+3} = G_{i+3} + P_{i+3} \cdot G_{i+2} + P_{i+3} \cdot P_{i+2} \cdot G_{i+1} + P_{i+3} \cdot P_{i+2} \cdot P_{i+1} \cdot G_i + P_{i+3} \cdot P_{i+2} \cdot P_{i+1} \cdot P_i \cdot r_{i-1}$$

Fonctions de génération et propagation de retenue par bloc de 4:

$$G_{i+3,i} = G_{i+3} + P_{i+3} \cdot G_{i+2} + P_{i+3} \cdot P_{i+2} \cdot G_{i+1} + P_{i+3} \cdot P_{i+2} \cdot P_{i+1} \cdot G_i$$

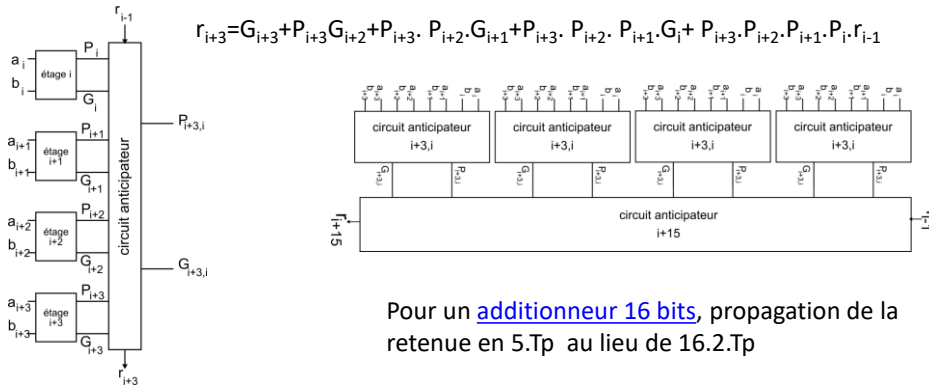
$$P_{i+3,i} = P_{i+3} \cdot P_{i+2} \cdot P_{i+1} \cdot P_i$$

$$r_{i+3} = G_{i+3,i} + P_{i+3,i} \cdot r_{i-1}$$

6

6

Addition – Additionneur à retenue anticipée (cont.)



Pour un additionneur 16 bits, propagation de la retenue en 5.Tp au lieu de 16.2.Tp

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

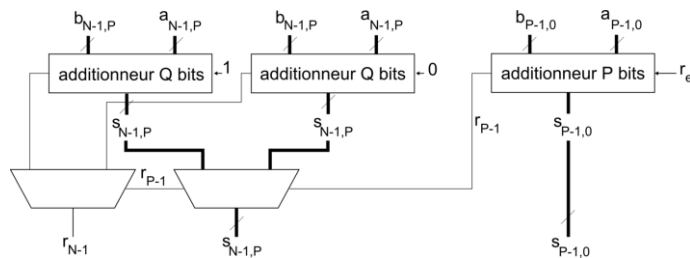
Idee : Le temps de propagation ne varie plus avec N (nb bits) mais avec $\log_2(N)$

7

7

Addition – Additionneur à sélection de retenue

Additionneur à sélection de retenue



$$N = P + Q$$

Idee : les calculs sont faits en parallèle pour les deux cas possibles de retenue. Le bon résultat est sélectionné quand la retenue est connue.

8

8

- 3 types principaux d'additionneurs

	vitesse	surface	consommation
Additionneur à propagation simple			
Additionneur à retenue anticipée			
Additionneur à sélection de retenue			

9

9

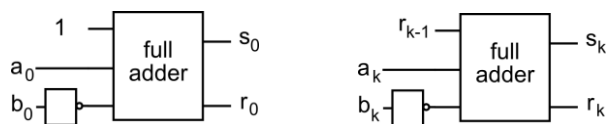
Principe

$$A - B = A + (-B)$$

Utilisation d'additionneur et d'inverseur.

Codage complément à deux, $-B$ s'obtient par :

De manière générale : complémentation des entrées et r_0 à 1.



Additionneur/soustracteur : utilisation d'inverseurs commandés (XOR).

10

10

High-Speed CMOS Logic 4-Bit Binary Full Adder with Fast Carry

1 Features

- Adds two binary numbers
- Full internal lookahead
- Fast ripple carry for economical expansion
- Operates with both positive and negative logic
- Fanout (over temperature range)
 - Standard outputs 10 LSTTL loads
 - Bus driver outputs 15 LSTTL loads
- Wide operating temperature range: -55°C to 125°C
- Balanced propagation delay and transition times
- Significant power reduction compared to LSTTL Logic ICs
- HC types



CD74HC283M96 4-bit logic adder & subtractor Hi-Speed CMOS
0,602 € Mouser 2026

2 Description

The CDx4HC283 and CDx4HCT283 contain 4-bit binary adders. The HCT device has TTL-voltage compatible inputs.

Device Information

PART NUMBER	PACKAGE ⁽¹⁾	BODY SIZE (NOM)
CD54HC283	J (CDIP, 16)	24.38 mm × 6.92 mm
CD74HC283CD74HC283	D (SOIC, 16)	9.90 mm × 3.90 mm
	N (PDIP, 16)	19.31 mm × 6.35 mm
CD74HCT283CD74HCT283	D (SOIC, 16)	9.90 mm × 3.90 mm
	N (PDIP, 16)	19.31 mm × 6.35 mm

11

11

Unité arithmétique et logique

Opérations arithmétiques : +, -

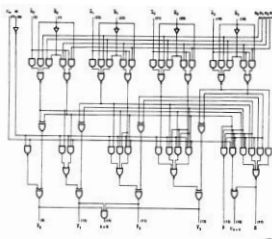
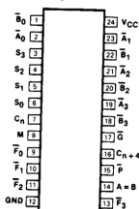
Opérations logiques : ET, OU, inversion, XOR

pour info
obsolète

- Exemple d'UAL : (74181TTL)

$$G_i = S_3 \cdot a_i \cdot b_i + S_2 \cdot a_i \cdot \bar{b}_i$$

$$P_i = a_i + S_0 \cdot b_i + S_1 \cdot \bar{b}_i$$



SELECTION				ACTIVE-HIGH DATA		
S3	S2	S1	S0	M = H LOGIC FUNCTIONS	M = L; ARITHMETIC OPERATIONS	
					C _n = H (no carry)	C _n = L (with carry)
L	L	L	L	F = A	F = A	F = A PLUS 1
L	L	L	H	F = A + B	F = A + B	F = (A + B) PLUS 1
L	L	H	L	F = A ⊕ B	F = A ⊕ B	F = (A ⊕ B) PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = ZERO
L	H	L	L	F = A ⊗ B	F = A PLUS A ⊗ B	F = A PLUS A ⊗ B PLUS 1
L	H	L	H	F = B	F = (A + B) PLUS A ⊗ B	F = (A + B) PLUS A ⊗ B PLUS 1
L	H	H	L	F = A ⊕ B	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	F = A ⊗ B	F = A ⊗ B MINUS 1	F = A ⊗ B
H	L	L	L	F = A + B	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	F = A ⊕ B	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = (A ⊕ B) PLUS AB	F = (A ⊕ B) PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS A ¹	F = A PLUS A PLUS 1
H	H	L	H	F = A + B	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	F = (A ⊕ B) PLUS A	F = (A ⊕ B) PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A

¹ Each bit is shifted to the next more significant position.

12

12

Multiplication combinatoire

Structure calquée sur la structure du calcul

[illegible]
$$\begin{array}{cccc} & x_3 & x_2 & x_1 & x_0 \\ \times & y_3 & y_2 & y_1 & y_0 \end{array}$$

- Fait intervenir 2 types d'opérations élémentaires :
 1. des produits de bits
 2. des additions de sommes partielles

					x_3	x_2	x_1	x_0
				$\times$	y_3	y_2	y_1	y_0
					$x_3 y_0$	$x_2 y_0$	$x_1 y_0$	$x_0 y_0$
			$x_3 y_1$		$x_2 y_1$	$x_1 y_1$	$x_0 y_1$	
		$x_3 y_2$	$x_2 y_2$		$x_1 y_2$	$x_0 y_2$	r_1	
	$x_3 y_3$	$x_2 y_3$	$x_1 y_3$		$x_0 y_3$			
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

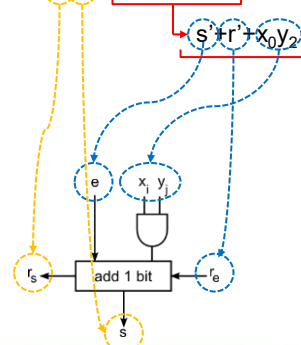
cellule élémentaire dédiée à ces opérations :

→ $[r_1 \ p_1] = x_1 y_0 + x_0 y_1$

$$p_1 = x_1 y_0 \oplus x_0 y_1$$

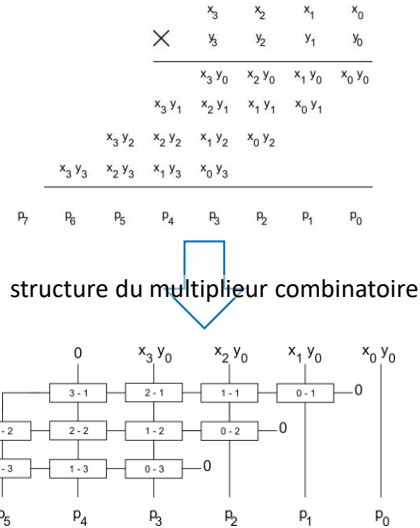
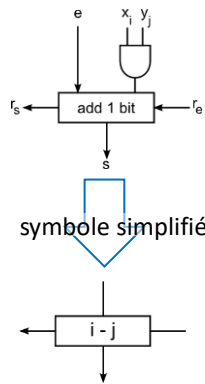
$$r_1 = (x_1 y_0) \cdot (x_0 y_1)$$

→ $[r_2, p_2] = x_2 y_0 + x_1 y_1 + r_1 + x_0 y_2$



Multiplication – Multiplication combinatoire (cont.)

Cellule élémentaire

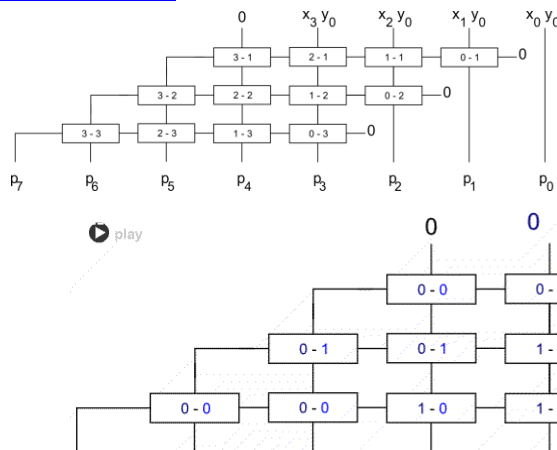


15

15

Multiplication – Multiplication combinatoire (cont.)

Exemple de calcul : X=3, Y=4



Fonctionne pour les
nombres non entiers en
virgule fixe ?
Fonctionne pour les
nombres signés ?

16

16

Multiplieur Booth

Repose sur l'algorithme de Booth, largement utilisé pour les multiplications, qui permet de **réduire de moitié le nombre de sommes partielles** tout en conservant une **structure régulière**.

Algorithme de Booth :

Soit A l'opérande de multiplication : $A = -a_{N-1} 2^{N-1} + a_{N-2} 2^{N-2} + \dots + a_2 2^2 + a_1 2^1 + a_0 2^0$

En utilisant la relation : $2^i = 2^{i+1} - 2 \cdot 2^{i-1}$

On peut écrire :

$$A = (-2 \cdot a_{N-1} + a_{N-2} + a_{N-3}) 2^{N-2} + (-2 \cdot a_{N-3} + a_{N-4} + a_{N-5}) 2^{N-4} + \dots \\ + (-2 \cdot a_3 + a_2 + a_1) 2^2 + (-2 \cdot a_1 + a_0 + 0) 2^0$$

Il s'agit d'une décomposition de Booth d'ordre 2.

17

17

$$A = (-2 \cdot a_{N-1} + a_{N-2} + a_{N-3}) 2^{N-2} + (-2 \cdot a_{N-3} + a_{N-4} + a_{N-5}) 2^{N-4} + \dots \\ + (-2 \cdot a_3 + a_2 + a_1) 2^2 + (-2 \cdot a_1 + a_0 + 0) 2^0$$

Quand un nombre M est multiplié par A, les produits partiels obtenus sont du type :

$$M \cdot 2^{-2}; M \cdot 2^{-1}; M \cdot 2^0; M \cdot 2^1; M \cdot 2^2$$

Le calcul du produit final résulte de la conjugaison de 3 types de commandes :

- complémentation à 2 ($\cdot -1$),
- décalage à gauche ($\cdot 2$),
- annulation ($\cdot 0$).

Idée : Multiplication transformée en étapes logiques élémentaires (complémentation, décalage, annulation)

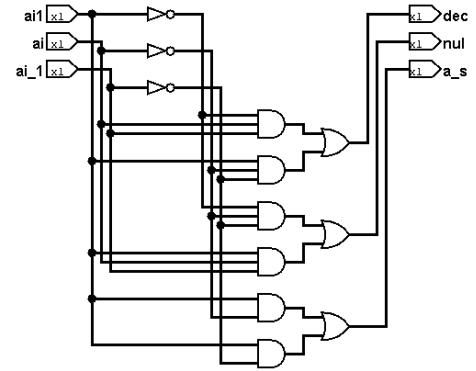
Multiplieur Booth constitué d'un **décodeur Booth** et de **cellules élémentaires** capables d'exécuter les 3 types de commande.

18

18

- Table de vérité du décodeur Booth

a_{i+1}	a_i	a_{i-1}	produit partiel	déc. (dec)	ann. (nul)	compl. (a/s)
0	0	0	$M*0$	0	1	0
0	0	1	$M*1$	0	0	0
0	1	0	$M*1$	0	0	0
0	1	1	$M*2$	1	0	0
1	0	0	$M*-2$	1	0	1
1	0	1	$M*-1$	0	0	1
1	1	0	$M*-1$	0	0	1
1	1	1	$M*0$	0	1	0

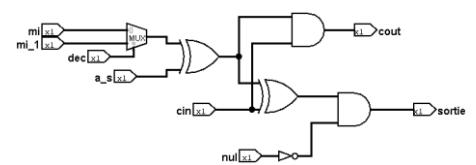


19

19

- Table de vérité d'une cellule élémentaire

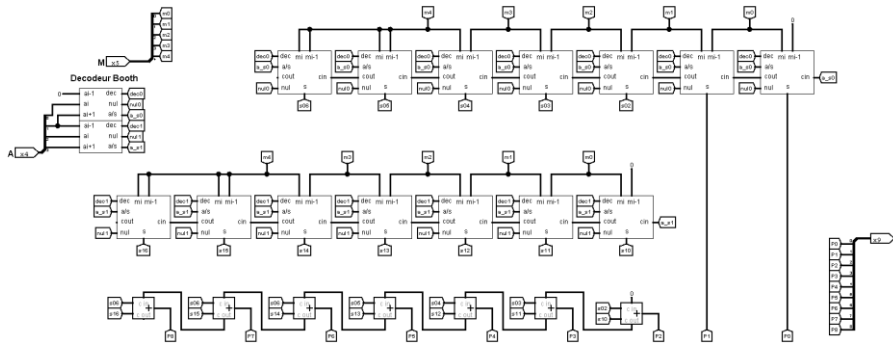
déc. (dec)	ann. (nul)	compl. (a/s)	sortie	retenue (carry out)
0	0	0	$m_i \oplus carry_{in}$	$m_i \cdot carry_{in}$
0	0	1	$\overline{m_i} \oplus carry_{in}$	$\overline{m_i} \cdot carry_{in}$
0	1	0	0	0
1	0	0	$m_{i-1} \oplus carry_{in}$	$m_{i-1} \cdot carry_{in}$
1	0	1	$\overline{m_{i-1}} \oplus carry_{in}$	$\overline{m_{i-1}} \cdot carry_{in}$



20

20

- Structure du multiplieur Booth



Réduit les propagations de retenues.

Le nombre d'opérations "utiles" dépend de la configuration des bits du multiplieur.

21

21

Multiplieur Wallace

Le multiplieur de Wallace n'a pas une structure régulière mais une structure arborescente.

Il repose sur une **décomposition** de la multiplication en **sous-multiplications** sur un nombre réduit de bits, et sur la combinaison rapide des produits partiels.

Soit A et B sur N bits les nombres à multiplier. On peut écrire :

$$A * B = 2^N \cdot A_{MSB} \cdot B_{MSB} + 2^{N/2} (A_{MSB} \cdot B_{LSB} + A_{LSB} \cdot B_{MSB}) + A_{LSB} \cdot B_{LSB}$$

Soit 4 multiplications partielles sur 2 fois moins de bits.

Idée : décomposition en sous-multiplications moins lourdes

22

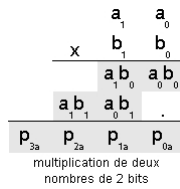
22

Multiplication – Multiplieur Wallace (cont.)

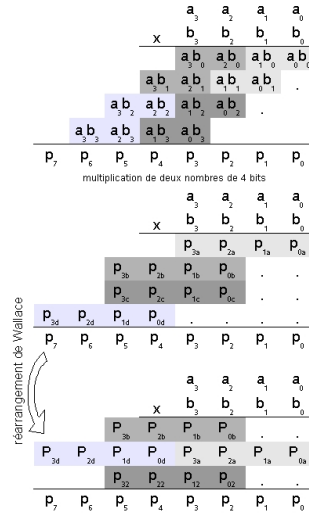
- Exemple pour N=4 bits

$$A \cdot B = 2^4 \cdot A_{\text{MSB}} \cdot B_{\text{MSB}} + 2^2 (A_{\text{MSB}} \cdot B_{\text{LSB}} + A_{\text{LSB}} \cdot B_{\text{MSB}}) + A_{\text{LSB}} \cdot B_{\text{LSB}}$$

$$A_{\text{MSB}} = a_3 a_2 ; A_{\text{LSB}} = a_1 a_0 ; B_{\text{MSB}} = b_3 b_2 ; B_{\text{LSB}} = b_1 b_0$$



multiplication de deux
nombres de 2 bits

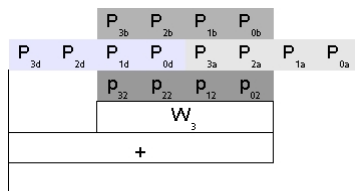


23

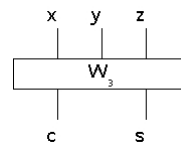
23

Multiplication – Multiplieur Wallace (cont.)

- Exemple pour N=4 bits



structure du multiplieur de Wallace pour la
multiplication de deux nombres de 4 bits

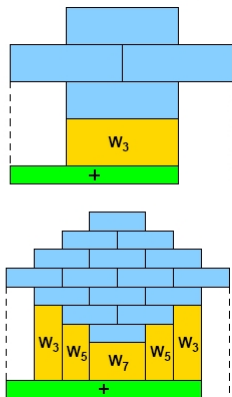
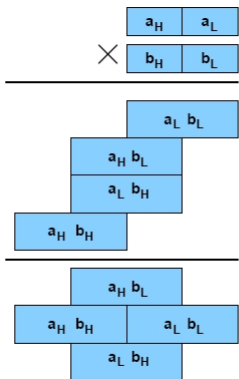
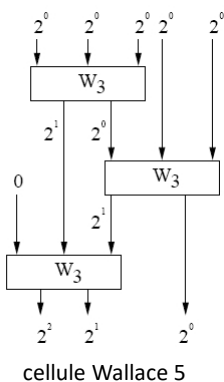


cellule Wallace 3 : full adder

24

24

- $N > 4$ bits
cellules de Wallace plus grosses.



Comparaison des multiplieurs

	temps de traversée	surface	consommation
multiplieur classique	54.70 ns	1810 (portes)	4270 $\mu\text{W}/\text{MHz}$
multiplieur Booth	42.18 ns	1020 (portes)	2195 $\mu\text{W}/\text{MHz}$
multiplieur 16 bits en technologie 0.7 μm , 3V			
multiplieur Booth	5.64 ns	21 242 (μm^2)	132.4 $\mu\text{W}/\text{MHz}$
multiplieur Wallace	3.28 ns	21 717 (μm^2)	116.9 $\mu\text{W}/\text{MHz}$

multiplieur 16 bits en technologie 0.18 μm , 1.95V



Remarque : il existe d'autre type de structures de multiplieur (Dadda par exemple), optimisées pour une caractéristique particulière.

Division - pas d'opérateur dédié

- Algorithme de division s'appuyant sur d'autres opérateurs arithmétiques
 - Requière de la logique séquentielle

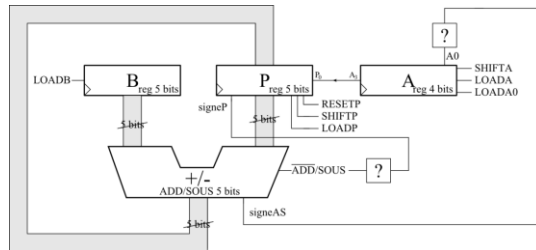
Algorithme de division sans restauration :

- Décalage des registres P et A de 1 bit vers la gauche.
 - Si le contenu du registre P est négatif :
 - on additionne le contenu du registre B à P
 - Si le contenu du registre P est positif :
 - on soustrait le contenu du registre B à P
 - Si le résultat chargé dans P est négatif on met le bit de poids faible de A (A0) à « 0 » sinon à « 1 ».
- Ces étapes sont répétées N fois, N étant le format des données, ici N=4.

Exemple de
l'algorithme de
division sans
restauration

Rmq

Remarque :
Comme on la
fait à la main



27

27

Opérateurs arithmétiques – Remarques générales - Erreurs de calculs

Sources d'erreur de calcul possibles

- Résolution liée au format des données (troncature, arrondi, erreur de représentation)
- Addition/soustraction :
 - débordement (overflow)
- Multiplication :
 - arrondi/quantification

$$\begin{array}{r}
 \begin{array}{cccc} x_3 & x_2 & x_1 & x_0 \end{array} \\
 + \begin{array}{cccc} y_3 & y_2 & y_1 & y_0 \end{array} \\
 \hline
 \begin{array}{cccc} s_3 & s_2 & s_1 & s_0 \end{array}
 \end{array}$$

$$\begin{array}{r}
 \begin{array}{cccc} x_3 & x_2 & x_1 & x_0 \end{array} \\
 \times \begin{array}{cccc} y_3 & y_2 & y_1 & y_0 \end{array} \\
 \hline
 \begin{array}{cccc} x_3 y_0 & x_2 y_0 & x_1 y_0 & x_0 y_0 \\
 x_3 y_1 & x_2 y_1 & x_1 y_1 & x_0 y_1 \\
 x_3 y_2 & x_2 y_2 & x_1 y_2 & x_0 y_2 \\
 x_3 y_3 & x_2 y_3 & x_1 y_3 & x_0 y_3 \end{array}
 \end{array}$$

$$\begin{array}{cccccccc} p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \end{array}$$

28

28

Opérations sur des nombres non-entiers :

Cas du codage en virgule fixe :



Rappel : la virgule n'a pas d'existence matérielle ; la représentation en virgule fixe est une interprétation du nombre binaire.
Donc pas d'impact sur les opérateurs. Ce sont les mêmes qui traitent les nombres entiers ou non-entiers à virgule fixe.

Cas du codage en virgule flottante :

Dans ce cas également, les architectures d'opérateurs sont les mêmes puisque les calculs en virgule flottante se traduisent par des additions/soustractions d'entiers pour les calculs de l'exposant, et par des opérations arithmétiques en virgule fixe pour la mantisse.



Opérations sur des nombres signés :

add/sous : les architectures d'opérateurs vues peuvent être utilisées directement
mult : dépend de la structure

En complément à deux

29

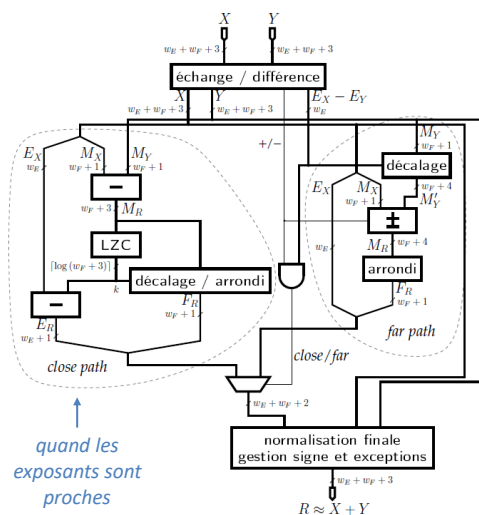
29

Multiplication plus simple

addition soustraction plus complexes.

Les nombres à additionner doivent avoir le même exposant donc il faut aligner les mantisses. C'est le nombre le plus grand qui impose la précision de calcul.

Les opérateurs arithmétiques sont les mêmes que l'on soit en virgule fixe ou virgule flottante. C'est l'utilisation qu'on en fait qui est différente



30

30



- Quels intérêts y a-t-il à avoir des opérateurs cascadables et à structure régulière ?
- Quel codage des nombres est compatible avec les architectures courantes d'opérateurs arithmétiques ?