Addition entière

Alain MÉRIGOT

Université Paris Saclay

Plan

Additionneurs 1 bit

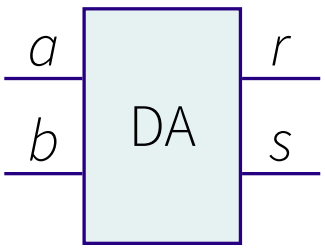
demi-additionneur

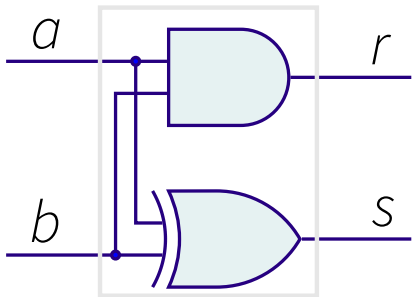
Addition de deux bits : **demi-additionneur**
entrées : opérandes a et b,
sorties : somme des deux opérandes sur 2 bits s et r
Le bit de poids fort r du résultat est appelé **retenue** (*carry*)
Le bit de poids faible s est appelé **somme**

a	b	$a+b$	r	s
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

$$a + b = 2^1 \times r + 2^0 \times s$$

$$s = a \oplus b$$
$$r = a \,\&\, b$$





Additionneurs 1 bit (cont.)

additionneur complet

Addition de mots de n bits : nécessité de prendre en compte la retenue :
additionneur complet
3 entrées : opérandes a b et c, 2 sorties : s et r

a	b	c	$a+b+c$	r	s
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

$$a+b+c = 2^1 \times r + 2^0 \times s$$

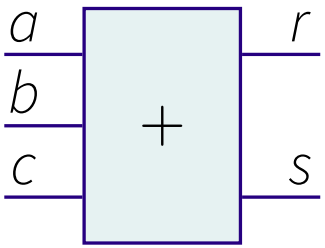
$$s = a \oplus b \oplus c$$
$$r = a \,\&\, b + a \,\&\, c + b \,\&\, c$$

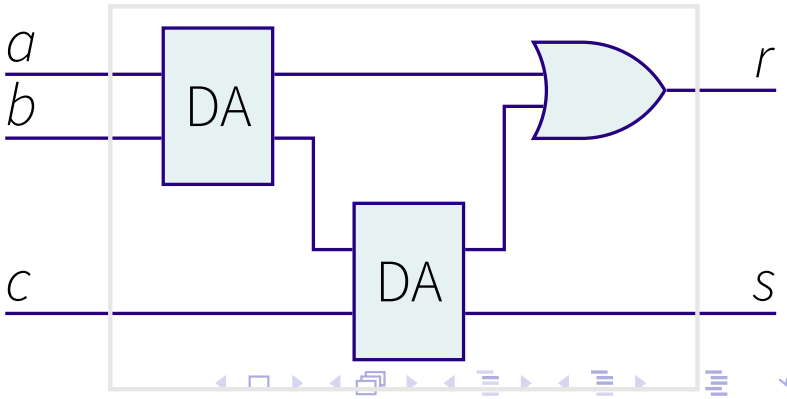
Additionneurs 1 bit (cont.)

additionneur complet

$$s = \underbrace{a \oplus b}_{s_{a+b}} \oplus c$$

$$\begin{aligned} r &= a \,\&\, b + a \,\&\, c + b \,\&\, c \\ &= a \,\&\, b + a \,\&\, (\overline{b} + b) \,\&\, c + (\overline{a} + a) \,\&\, b \,\&\, c \\ &= a \,\&\, b + a \,\&\, b \,\&\, c + a \,\&\, b \,\&\, c + a \,\&\, \overline{b} \,\&\, c + + \overline{a} \,\&\, b \,\&\, c \\ &= (a \,\&\, b + \cancel{a \,\&\, b \,\&\, c} + \cancel{a \,\&\, b \,\&\, c}) + (a \,\&\, \overline{b} + + \overline{a} \,\&\, b) \,\&\, c \\ &= \underbrace{a \,\&\, b}_{r_{a+b}} + (\underbrace{a \oplus b}_{s_{a+b}}) \,\&\, c \end{aligned}$$



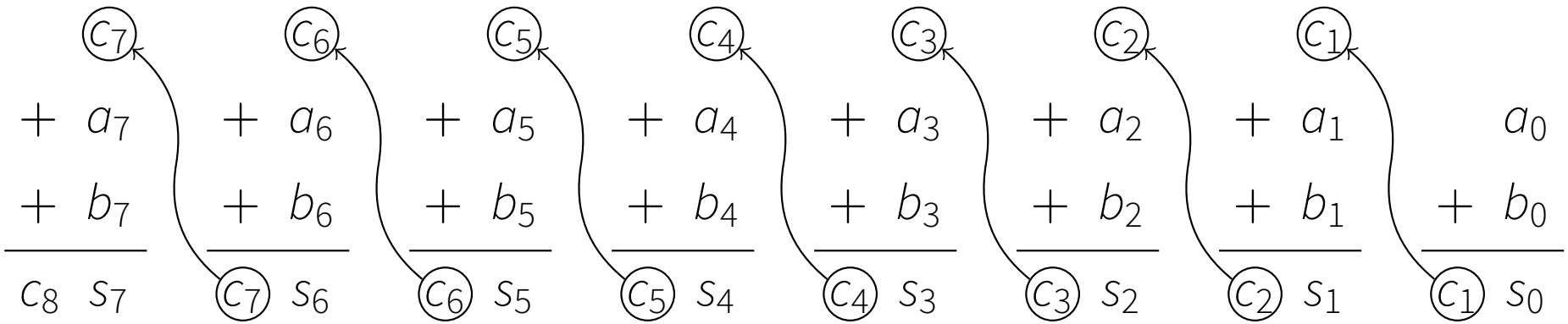


Plan

Plan

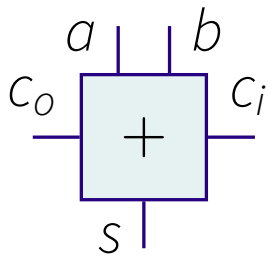
Additionneur à propagation de retenue

Addition de 2 nombres entiers non signés $A = (a_{n-1}, a_{n-2}, \ldots, a_0)$ et $B = (b_{n-1}, b_{n-2}, \ldots, b_0)$.
Addition de a_0 et $b_0 \rightarrow$ somme de poids 2^0 (s_0) et retenue de poids 2^1 (c_1).
c_1 sera pris en compte avec les bits de même poids a_1 et b_1.

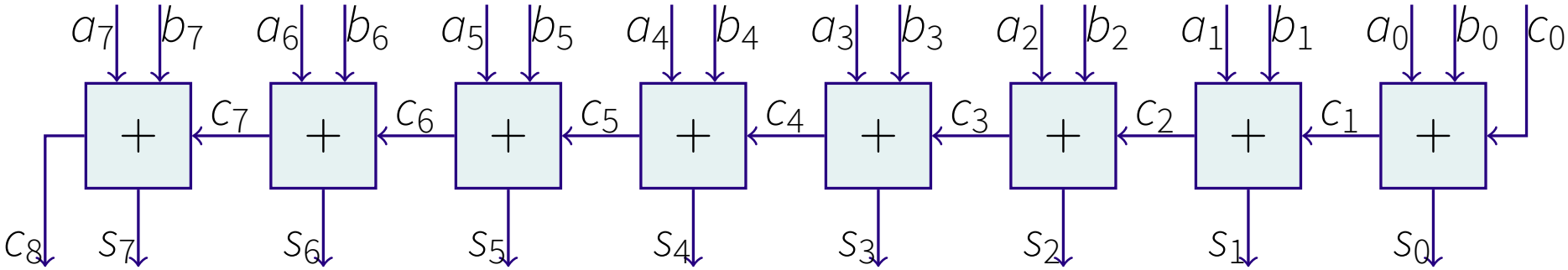


Addition à propagation de retenue (*carry propagate adder*) :
la retenue de c_{i+1} de l'étage i est réinjectée à l'entrée c de l'addition de l'étage $i + 1$.

Additionneur à propagation de retenue (cont.)



c_i et c_o sont respectivement la **retenue entrante** et la **retenue sortante** de l'additionneur complet.



La *retenue entrante* de l'additionneur n bits c_0 est généralement à 0, mais peut permettre de calculer $A + B + 1$ ou de cascader des additionneurs.
La *retenue sortante* c_n peut être interprétée de deux façons :

- Si A et B sont sur n bits, $0 \leq A, B \leq 2^n - 1$ et $0 \leq S = A + B \leq 2^{n+1} - 2$.
  S nécessite donc en général $n + 1$ bits de codage.
  c_n est alors le $n + 1^{\text{ième}}$ bit du résultat (poids 2^n).
- Si le résultat S *doit* être codé sur n bits, il sera valide ssi $S < 2^n$, donc si $c_n = 0$.
  c_n est alors un indicateur de **débordement de capacité**.

Additionneur à propagation de retenue (cont.)

temps de calcul

Prise en compte des **temps de traversée** des additionneurs.

Nous supposons que pour les additionneurs complets, la somme s_i et la retenue c_{i+1} seront stables τ après la stabilisation des entrées a_i, b_i et c_i

Si à l'instant $t = 0$, les entrées ai et b_i et la retenue c_0 sont stables $\forall i$

- à l'instant $t = \tau$, la somme s_0 et la retenue c_1 seront stables.
- comme les entrées de l'additionneur 1 sont toutes stables à l'instant τ, les sorties c_2 et s_1 seront stables en $t = 2 \times \tau$
- ...
- à l'instant $t = n \times \tau$, les sorties c_n et s_{n-1} seront stables.

Dans le pire cas, le temps de calcul de l'additionneur à propagation de retenue de n bits est de $t_n = n \times \tau$.
Évolution linéaire du temps.
Très problématique pour des valeurs de n au delà de quelques unités.
Nécessité de schémas d'additioneurs rapides

Plan

Plan

Addition de nombres signés

additionneur-soustracteur

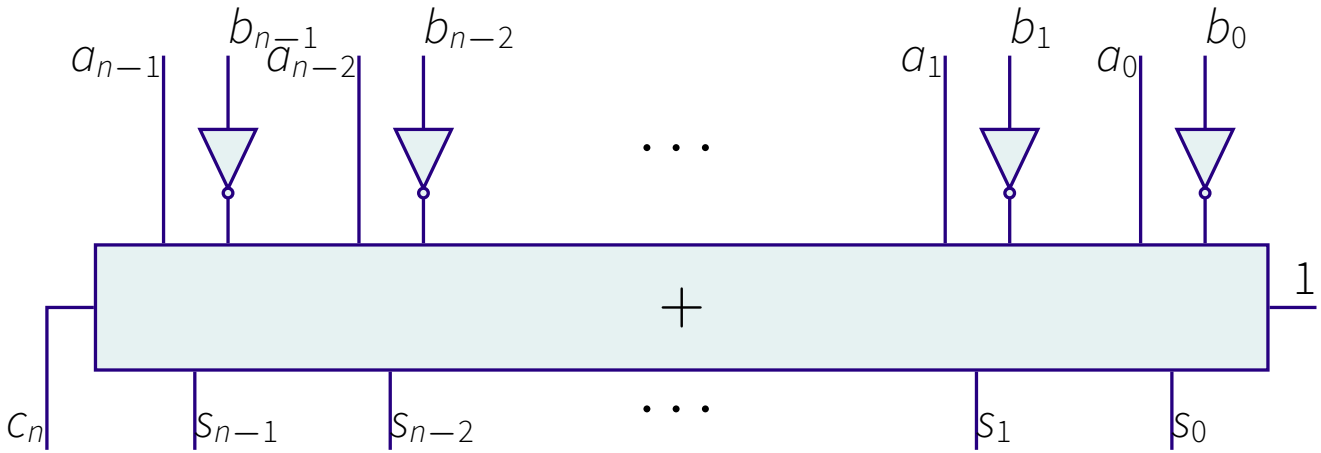
Comme $C_2(B) = 2^n - |B|$, à 2^n près, il est équivalent de calculer $A - B$ ou $A + C_2(B)$

Soustracteur = calcul du complément à deux du 2e opérande + addition avec le 1er opérande.

Comme $C_2(B) = \overline{B} + 1, A - B = A + \overline{B} + 1$.

Pour transformer un additionneur en soustracteur, il faut :

- inverser tous les bits de l'opérande B
- calculer $A + B$ en mettant la retenue entrante c_0 à 1



Construction d'un soustracteur n bits à partir d'un additionneur

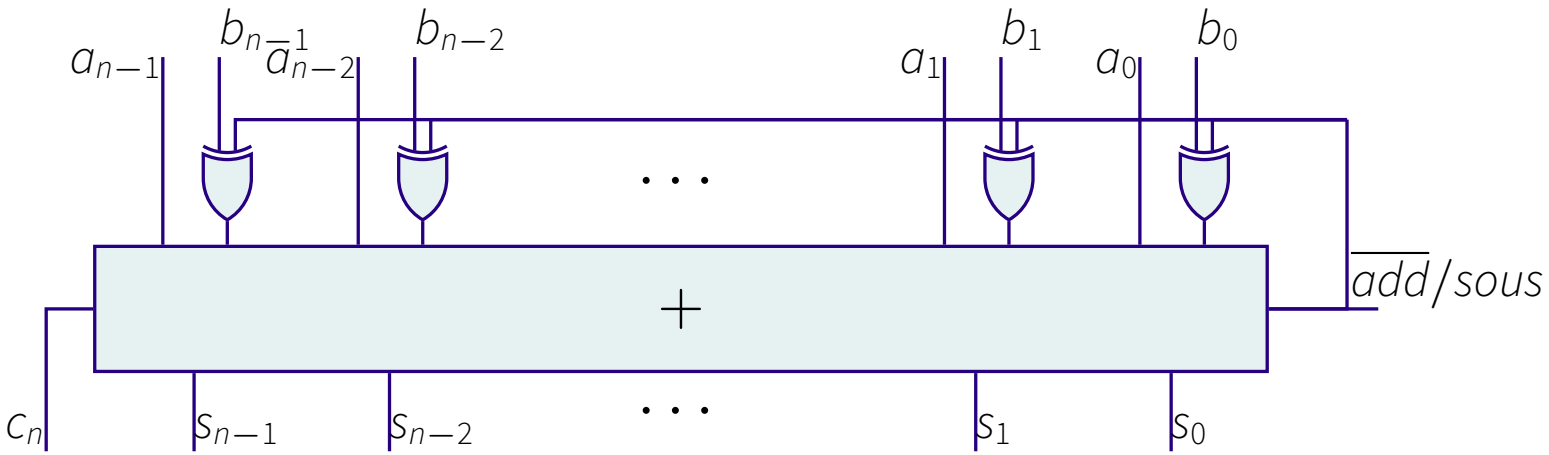
Addition de nombres signés (cont.)

additionneur-soustracteur

Il est possible de faire simplement un **additionneur/soustracteur**

Différence entre addition et soustraction :

- complémentation ou non du deuxième opérande
- mise à zéro ou à un de la retenue entrante c_0.



Si le signal $\overline{add}/sous = 0$ alors $S = A + (00\ldots0 \oplus B) + 0 = A + B$

Si le signal $\overline{add}/sous = 1$ alors $S = A + (11\ldots1 \oplus B) + 1 = A + \overline{B} + 1 = A - B$

Plan

Addition de nombres signés (cont.)

validité du résultat

La retenue sortante n'est **pas** un indicateur de validité du résultat pour l'addition de nombres signés.

3 cas à considérer :

A et B positifs ou nuls

Nous aurons $A + B \geq 0$ (toujours)

$$\begin{array}{rcccccccc} & & c_n = 0 & c_{n-1} & c_{n-2} & & c_1 & \\ A & & & 0 & a_{n-2} & \dots & a_1 & a_0 \\ +B & & + & 0 & b_{n-2} & \dots & b_1 & b_0 \\ \hline S & & & c_{n-1} & s_{n-2} & \dots & s_1 & s_0 \end{array}$$

Le résultat est valide si $0 \leq A + B < 2^{n-1}$.
Il faut donc $s_{n-1} = 0$.
Pour cela, il faut $c_{n-1} = 0 (= c_n)$.

A et B négatifs

$A + B$ est toujours < 0.

$$
\begin{array}{rccccccc}
 & & c_n = 1 & c_{n-1} & c_{n-2} & & c_1 & \\
A & & & 1 & a_{n-2} & \dots & a_1 & a_0 \\
+B & & + & 1 & b_{n-2} & \dots & b_1 & b_0 \\
\hline
S & & & c_{n-1} & s_{n-2} & \dots & s_1 & s_0
\end{array}
$$

Résultat valide si $-2^{n-1} \leq A + B < 0$
le bit de poids fort s_{n-1} doit être à 1.
Il faut donc $c_{n-1} = 1 (= c_n)$ pour que le résultat soit valide.

Remarque : la retenue sortante est *toujours* à 1, car $A = 2^n - |A|$, $B = 2^n - |B|, A + B = 2^{n+1} - (|A| + |B|) > 2^n$ si $|A|, |B| < 2^{n-1}$.

A et B de signes contraires

Nous supposerons $A \geq 0$ et $B < 0$.

Le résultat $A + B$ est de signe quelconque.

$$\begin{aligned} 0 \leq & \quad A \quad < 2^{n-1} \\ -2^{n-1} \leq & \quad B \quad < 0 \\ -2^{n-1} \leq & \; A + B \; < 2^{n-1} \end{aligned}$$

Le résultat est donc *toujours* valide.

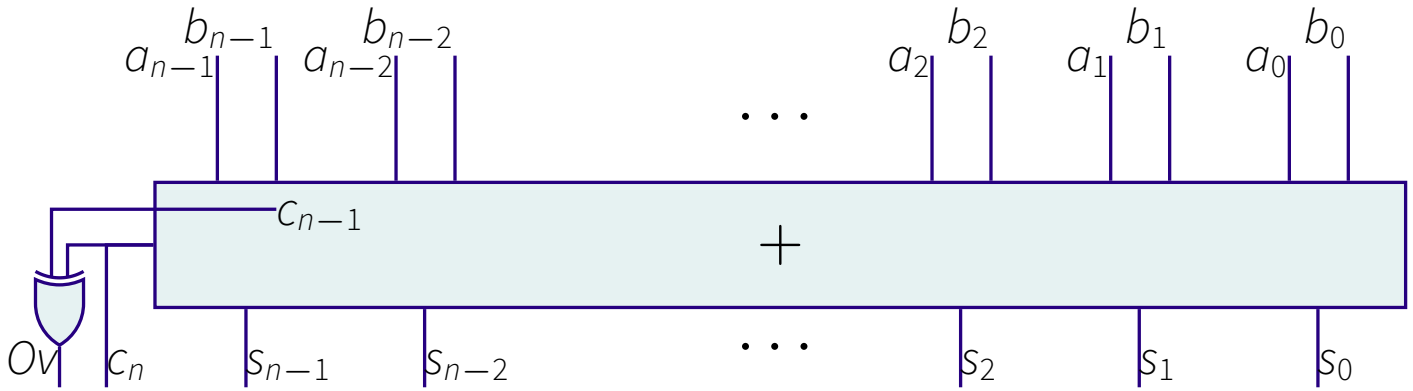
$$\begin{array}{rcccccccc} & c_n = c_{n-1} & & c_{n-1} & c_{n-2} & & c_1 & \\ A & & & 0 & a_{n-2} & \dots & a_1 & a_0 \\ +B & & + & 1 & b_{n-2} & \dots & b_1 & b_0 \\ \hline S & & & \overline{c_{n-1}} & s_{n-2} & \dots & s_1 & s_0 \end{array}$$

Dans ce cas, nous avons toujours $c_{n-1} = c_n$ et un résultat toujours valide.

Récapitulatif		
A et $B \geq 0$	S valide ssi $c_{n-1} = 0$	$c_n = 0$ toujours
A et $B < 0$	S valide ssi $c_{n-1} = 1$	$c_n = 1$ toujours
A et B de signes contraires	S toujours valide	$c_n = c_{n-1}$ toujours

Dans tous les cas, le résultats sera valide si et seulement si $c_n = c_{n-1}$

Le **ou** exclusif des deux dernières retenues est un indicateur de débordement de capacité (*overflow*)



Implications logicielles

Indicateurs de débordement de capacité en signé et non-signé existent dans les processeurs et sont exploitables en assembleur.
Mais *les langages de haut niveau ne font* ***aucune*** *vérification de la validité d'un résultat*.

```
unsigned char x=17*17; // 289 > 255 nécessiterait plus de 8 bits
                       // Interprété comme 33 (289-256)
char y=15*15;          // 225 codable sur 8 bits, mais représente
                       // alors un nombre négatif sur un 'char'.
                       // Interprété comme -256+225=-31
```

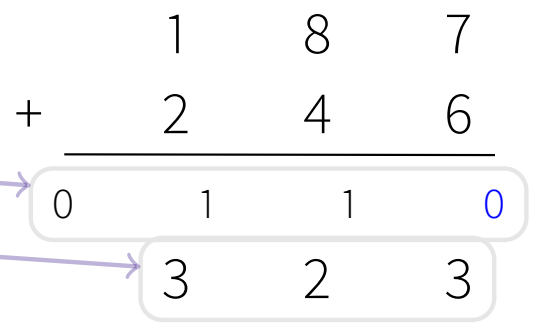
Problème similaire pour les calculs intermédiaires.
Pour limiter les problèmes de débordement de capacité, le langage C spécifie que les calculs intermédiaires entiers sont effectués en `long` (en `double` pour des données flottantes).
Toujours laisser le compilateur décomposer un calcul.

Plan

Additionneurs à sauvegarde de retenue

Une manière de réduire le temps de calcul d'un APR est de ne *pas* propager la retenue, mais de la mémoriser :
additionneur à sauvegarde de retenue (ASR) (*carry save adder*).

Le « résultat » du calcul est alors formé de deux parties : la retenue R et la somme S.



« Résultat sous forme RS »
Il faut les additionner pour avoir le « vrai » résultat.

Une addition à sauvegarde de retenue assure que le bit de poids faible de R c_0 est à zéro.
$n-1$ additions supplémentaires pour assurer que $(c_{n-1}c_{n-2}\ldots c_0)$ sont tous nuls.

N'est pas intéressant en général :
- n étapes supplémentaires pour le calcul de $R+S$
- chaque étape nécessite de mémoriser de S et C et le temps de calcul d'une étape est $\tau_{\text{add}}+\tau_{\text{reg}}$

sauf pour faire des **accumulations**.

Additionneurs à sauvegarde de retenue (cont.)

accumulation : suite d'opérations (généralement additions), telle que le résultat de l'opération k soit un des opérandes de l'opération $k+1$.

```
for(i=0;i<N;i++)
  S = S + A[i] ;
```

Accumulation des éléments du vecteur `A[]` dans `S`.

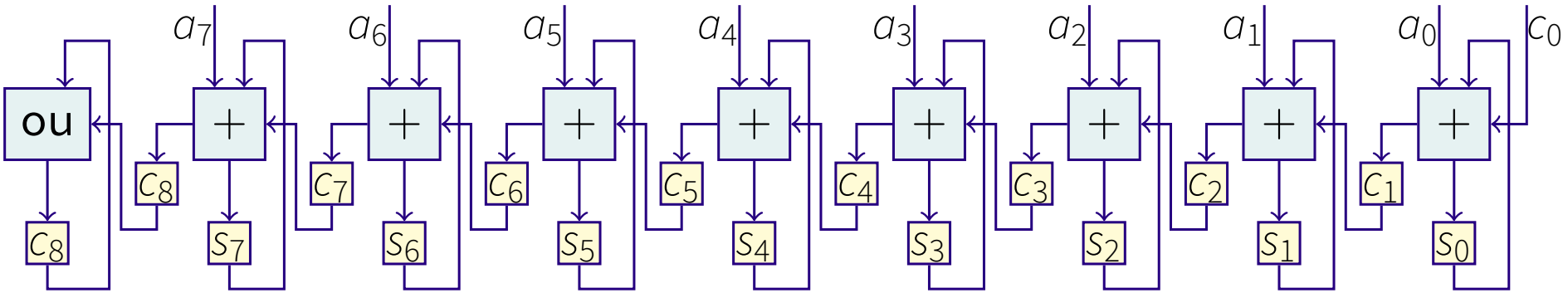
Seul compte le résultat final, et avoir les résultats intermédiaires sous forme RS n'est pas un problème si cela permet d'améliorer les performances.

Il sera nécessaire à la fin :

- soit d'accumuler $n-1$ fois fois zéro pour assurer que la retenue est identiquement nulle
- soit d'utiliser un additionneur normal pour calculer la somme de S et R

Les accumulations se rencontrent dans de nombreux problèmes (par exemple en traitement de signal) et sont à la base d'opérations comme la **multiplication**.

Additionneurs à sauvegarde de retenue (cont.)



Noter le **ou** pour préserver la retenue c_8 comme indicateur de débordement

Le temps d'une étape est $\tau + t_r$

- τ maximum entre temps (entrées $\rightarrow s$) et (entrées $\rightarrow co$) de l'additionneur
- t_r temps de traversée des registres ($\sqcap \rightarrow$ sortie)

Pour accumuler N nombres sur n bits :

	période horloge	nombre étapes	temps total
APR	$n \times \tau + t_r$	N	$N \times (n \times \tau + t_r)$
ASR	$\tau + t_r$	$N + (n-1)$	$(N + n - 1) \times (\tau + t_r)$

Si $\tau = t_r$ et $N \gg n \gg 1$, $t_{\text{APR}} \approx N \times n \times \tau$ et $t_{\text{ASR}} \approx N \times 2 \times \tau$

Plan

Additionneurs à génération de retenue anticipée

Principe des **additionneurs à génération de retenue anticipée** (ARA) (*carry lookahed adder*)

- on calcule préalablement *toutes* les retenues entrantes des différents bits
- on calcule ensuite l'addition en temps unitaire

Pour le $i$ème bit, calcul de la retenue sortante $c_i + 1$ en fonction de a_i, b_i et la retenue entrante c_i

$$\begin{aligned} c_{i+1} &= a_i b_i + a_i c_i + b_i c_i \\ &= g_i + p_i c_i \end{aligned}$$

en posant

$$\begin{aligned} g_i &= a_i b_i && \text{fonction de } \textit{génération} \text{ de retenue} \\ p_i &= a_i + b_i && \text{fonction de } \textit{propagation} \text{ de retenue} \end{aligned}$$

En itérant ce calcul, on peut calculer tous les c_i en fonction de c_0

Si l'on arrive à calculer rapidement les c_i, en fonction des a_k, b_k, $\forall k < i$ et de c_0, on accélère l'addition.

Additionneurs à génération de retenue anticipée (cont.)

Calcul des retenues successives

$$
\begin{aligned}
c_1 &= g_0 + p_0 c_0 \\
c_2 &= g_1 + p_1 c_1 \\
&= g_1 + p_1 (g_0 + p_0 c_0) \\
&= \underbrace{g_1 + p_1 g_0}_{G_{10}} + \underbrace{p_1 p_0}_{P_{10}}\, c_0 \\
&= G_{10} + P_{10} c_0
\end{aligned}
$$

avec

$$
\begin{array}{lcll}
G_{10} & = & g_1 + p_1 g_0 & \text{fonction de } \textit{génération} \text{ sur les bits 1–0} \\
P_{10} & = & p_1 p_0 & \text{fonction de } \textit{propagation} \text{ sur les bits 1–0}
\end{array}
$$

A l'ordre $i+1$

$$
\begin{aligned}
c_{i+1} &= g_i + p_i c_i \\
&= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \\
&= \cdots \\
&= \underbrace{g_i + p_i g_{i-1} + \cdots + p_i p_{i-1} \ldots p_1 g_0}_{G_{i0}} + \underbrace{p_i p_{i-1} \ldots p_0}_{P_{i0}}\, c_0 \\
&= G_{i0} + P_{i0}\, c_0
\end{aligned}
$$

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Pour simplifier la notation, on considère :

- le doublet génération-propagation $(g_i, p_i) = (a_i b_i, a_i + b_i)$ associé à chaque bit
- l'opérateur $\blacksquare$ qui calcule c_{i+1} en fonction de (g_i, p_i) et de c_i
  $c_{i+1} = (g_i, p_i) \blacksquare c_i = g_i + p_i c_i$
- l'opérateur $\circ$ qui permet de construire des fonctions de génération-propagation sur plusieurs bits (G_{ij}, P_{ij}) en combinant des doublets (g_i, p_i)

$$\begin{aligned}(G_{i,i-1}, P_{i,i-1}) &= (g_i + p_i g_{i-1}, p_i p_{i-1}) \\ &= (g_i, p_i) \circ (g_{i-1}, p_{i-1})\end{aligned}$$

Le calcul de la retenue à l'ordre $i+1$ en fonction de c_0 s'écrit alors simplement:

$$\begin{aligned}c_{i+1} &= (G_{i,0}, P_{i,0}) \blacksquare c_0 \\ &= \big((g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ \cdots \circ (g_0, p_0) \big) \blacksquare c_0\end{aligned} \tag{1}$$

Le problème se ramène à combiner des applications de l'opérateur $\circ$.

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Propriétés de l'opérateur $\circ$

- il n'est ***pas*** commutatif
  $(g', p') \circ (g, p) = (g' + p'g, p'p) \neq (g, p) \circ (g', p') = (g + pg', pp')$
  On ne peut pas permuter des opérandes
- il est **idempotent** : $(g, p) \circ (g, p) = (g + pg, pp) = (g, p)$
- il est **associatif** : $((g'', p'') \circ (g', p')) \circ (g, p) = (g'', p'') \circ ((g', p') \circ (g, p))$

$$((g'', p'') \circ (g', p')) \circ (g, p) = (g'' + p''g', p''p') \circ (g, p) = (g'' + p''g' + p''p'g, p''p'p)$$

$$(g'', p'') \circ ((g', p') \circ (g, p)) = (g'', p'') \circ (g' + p'g, p'p) = (g'' + p''g' + p''p'g, p''p'p)$$

$$= (g'', p'') \circ (g', p') \circ (g, p)$$

Un opérateur associatif donne un résultat identique, quel que soit le parenthésage.
Mais des parenthésages différents peuvent donner des temps de calcul différents.

Nombreux opérateurs associatifs en arithmétique, traitement de signal, calcul numérique, etc.

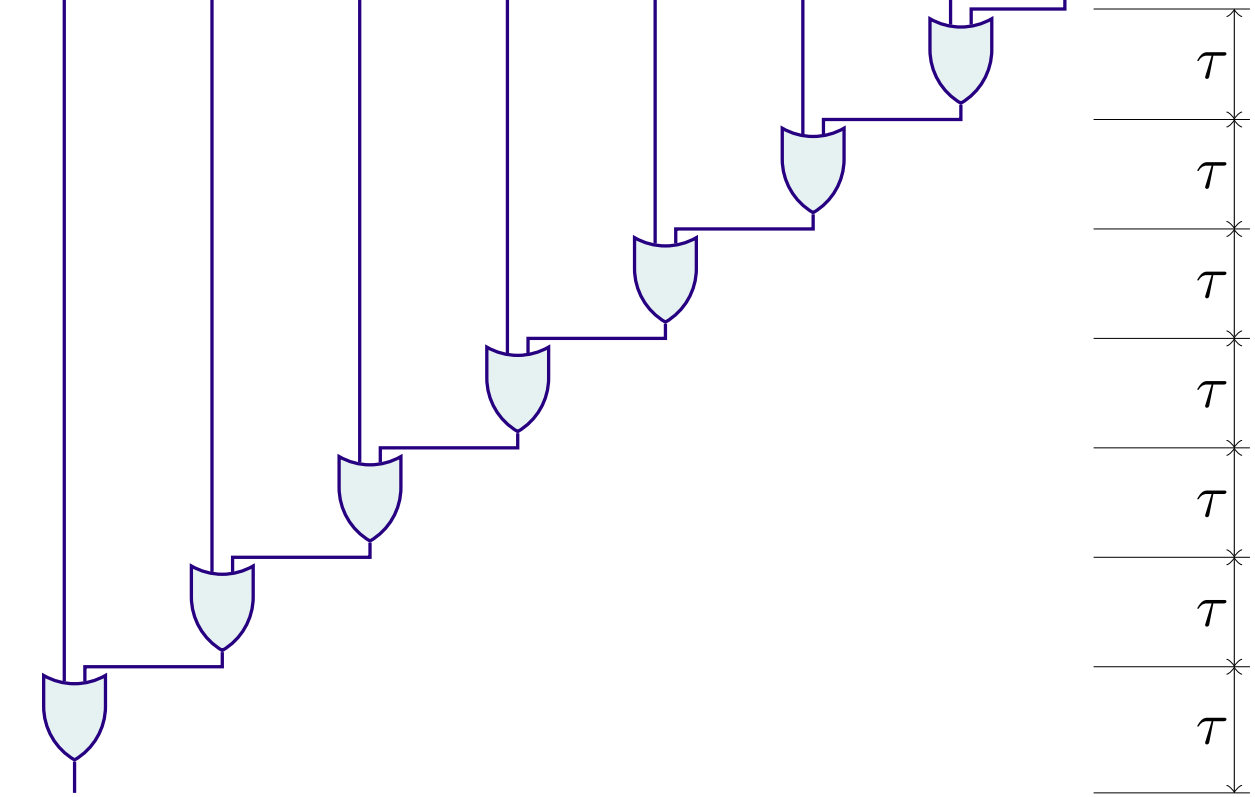
Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Exemple : **ou** (associatif) à 8 entrées à partir de portes **ou** à deux entrées

$$x_7+x_6+x_5+x_4+x_3+x_2+x_1+x_0 = (x_7+(x_6+(x_5+(x_4+(x_3+(x_2+(x_1+x_0)))))))$$

$$(x_7+(x_6+(x_5+(x_4+(x_3+(x_2+(x_1+x_0)))))))$$



Temps total : 7τ

Dans le cas général, pour n bits, temps : $(n-1)\times\tau$

Le temps varie linéairement avec le nombre de bits.

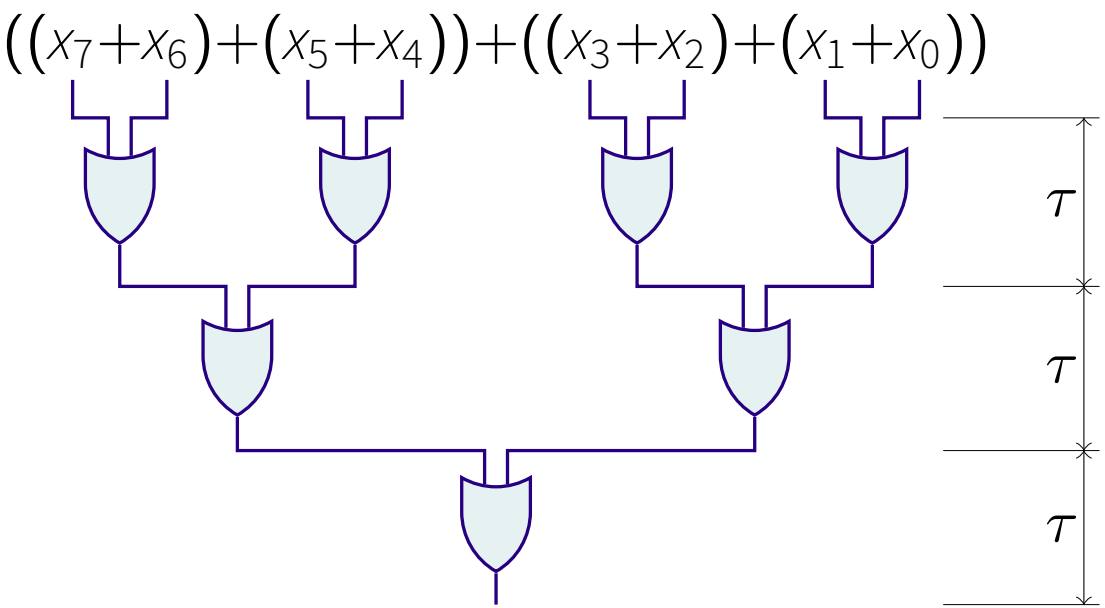
« **ou** à propagation »

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Utilisation d'un parenthésage différent

$$x_7+x_6+x_5+x_4+x_3+x_2+x_1+x_0 = ((x_6+x_7)+(x_4+x_5))+((x_2+x_3)+(x_0+x_1))$$



Trois couches à traverser.

Le temps de calcul n'est plus que 3τ

Doubler la taille ne rajoute qu'une couche

Dans le cas général de 2^k bits, temps de calcul
$t = k \times \tau$
ou pour n bits
$t = \log_2 n \times \tau$

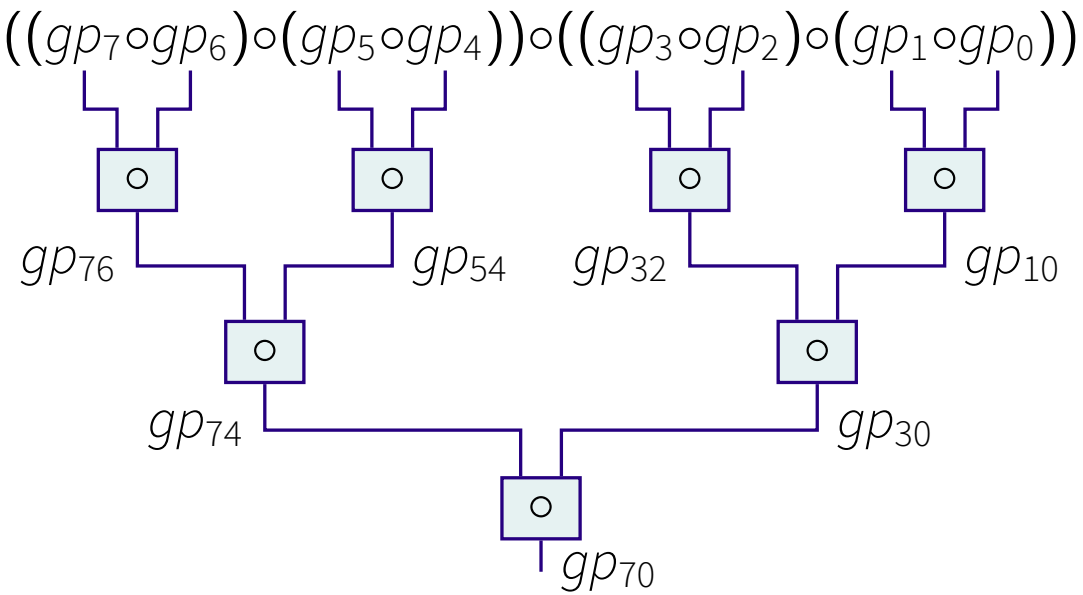
On passe d'un temps de calcul linéaire à un temps logarithmique.

Réduction du temps grâce au calcul simultané de plusieurs **ou**.
Exploitation du **parallélisme**

Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Même principe pour l'opérateur $\circ$



Pour n bits,
à l'étage $l \in \{1, \ldots, \log n\}$,
on calcule (G_{ij}, P_{ij}) tel que

- $j = k \times 2^l : j < n$
- $i = j + 2^l - 1$

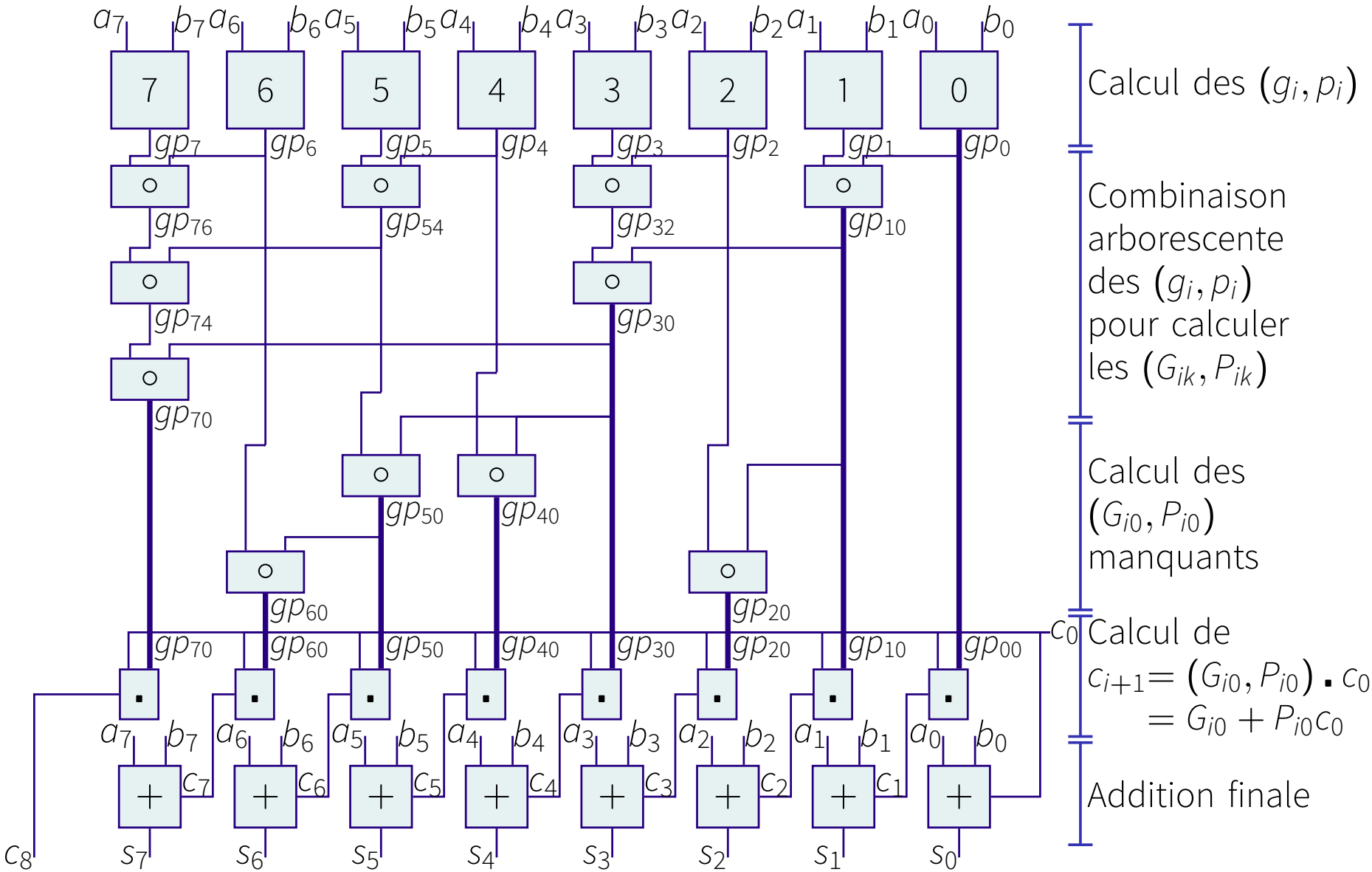
On obtient les (G_{i0}, P_{i0}) pour $i = 2^l - 1$

Problème à résoudre un peu différent : trouver *toutes* les retenues entrantes.
Pour cela on a besoin de *tous* les $(G_{i0}, P_{i0}), \forall i \in \{1, \ldots, 2^n - 1\}$

Revient à *distribuer* l'application de l'opérateur $\circ$ sur les données.
« calcul préfixe »

Peut se faire en combinant les différents (G_{ij}, P_{ij}) partiels calculés.

Additionneurs à génération de retenue anticipée (cont.)



Additionneurs à génération de retenue anticipée (cont.)

mise en oeuvre arborescente

Le chemin critique est pour le calcul de (G_{60}, P_{60}) (calcul de c_7).
Nécessite de traverser 4 couches de $\circ$

Dans le cas général, on traverse $2 \times (\log n - 1)$ couches.

	APR	ARA
	n	$2 \times (\log n - 1)$
8	8	4
16	16	6
32	32	8
64	64	10

Temps logarithmique

Doubler le nombre de bits ne rajoute que 2 couches à traverser.

Intéressant pour $n \geq 16$

Schéma présenté : Brent-Kung.
Il existe d'autres schémas d'ARA un peu plus réguliers et de temps de calcul identique. Ils utilisent le caractère idempotent de $\circ$ pour faire des calculs redondants et nécessitent donc plus de portes.

Il existe d'autre structures d'additionneurs rapides, par exemple en $\sqrt{n}$.