Multiplication entière

Alain MÉRIGOT

Université Paris Saclay

Plan

Multiplication non signée

Soient $A = (a_{n-1}, \ldots, a_1, a_0)$ et $B = (b_{n-1}, \ldots, b_1, b_0)$ deux entiers non signés de n bits.

$$\begin{aligned} A \times B &= A \times \sum_{j=0}^{n-1} b_j \times 2^j \\ &= \sum_{j=0}^{n-1} A \times b_j \times 2^j \end{aligned}$$

Comme

$$A \times b_j = \begin{cases} 0 & \text{si } b_j = 0 \\ A & \text{si } b_j = 1 \end{cases}$$

et

$$A \times 2^j = (a_{n-1}, \ldots, a_1, a_0, \overbrace{0, \ldots, 0, 0}^{\times j})$$

la multiplication se ramène à une suite d'additions et de décalages vers la gauche.

Le résultat sera sur $2n$ bits car $0 \leq A < 2^n$ et $0 \leq B < 2^n \implies 0 \leq A \times B < 2^{2n}$

Multiplication non signée par addition décalage

Déduction d'un algorithme de l'expression $A \times B = \sum_{j=0}^{n-1} A \times b_j \times 2^j$

1 résultat $\leftarrow 0$
2 **pour** $j \leftarrow 0$ **à** $n-1$
3 résultat $\leftarrow$ résultat + $b_j \times A \times 2^j$

Réécriture de l'algorithme sous une forme plus adaptée à une mise en oeuvre matérielle

1 résultat $\leftarrow 0$
2 **pour** $j \leftarrow 0$ **à** $n-1$
3 **si** $b_j = 1$
4 résultat $\leftarrow$ résultat + A ▷ *Addition*
5 $A \leftarrow A \times 2$ ▷ *Décalage*

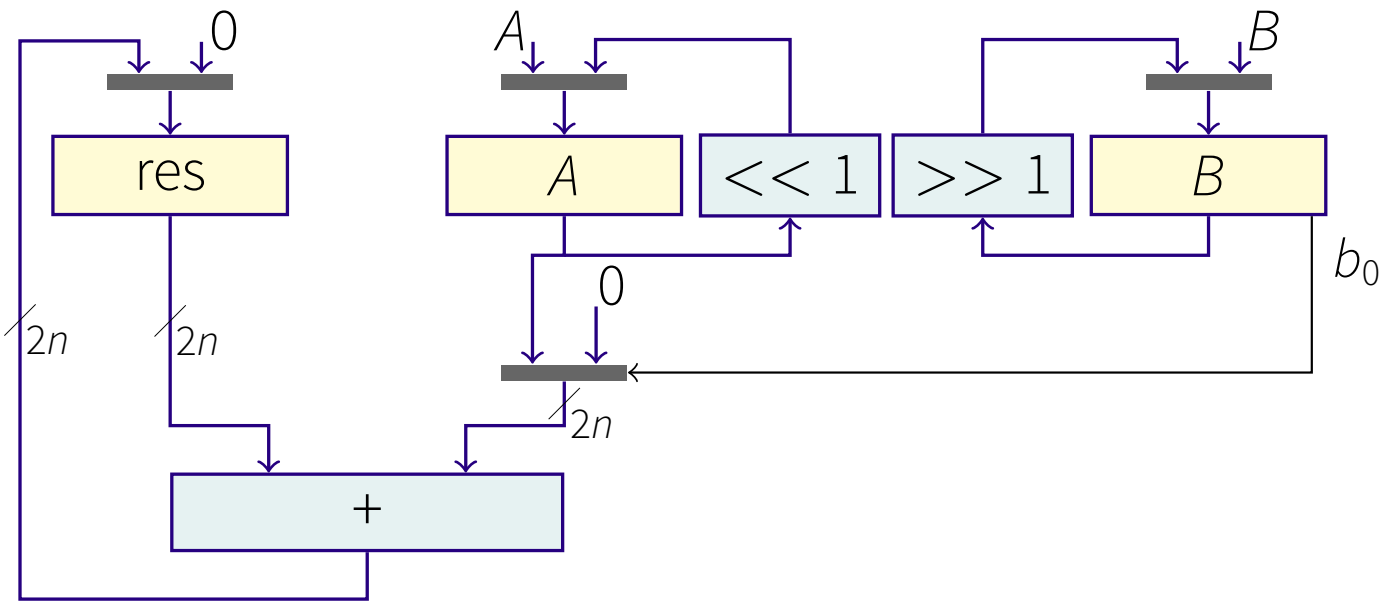
Algorithme de multiplication non signée par addition–décalage (ou série–parallèle)

Cet algorithme peut aisément se mettre en œuvre à l'aide d'un automate sur un chemin de données adapté.

Multiplication non signée par addition décalage (cont.)

Le calcul $A +$ res doit se faire sur $2n$ bits.
L'extraction du $i$ème bit b_i de B est un peu complexe.
$\Longrightarrow$ Utilisation de b_0 et d'un décalage à droite de B.

```
1   résultat ← 0
2   pour j ← 0 à n − 1
3       résultat ← résultat + si b_0 = 1 alors A sinon 0 fin si
4       A ← A × 2 ; B ← B ÷ 2
```



Un additionneur $2n$ bits

2 registres $2n$ bits(A, res), un registre n bits (B)

Multiplication non signée par addition décalage (cont.)

De nombreuses optimisations sont possibles.

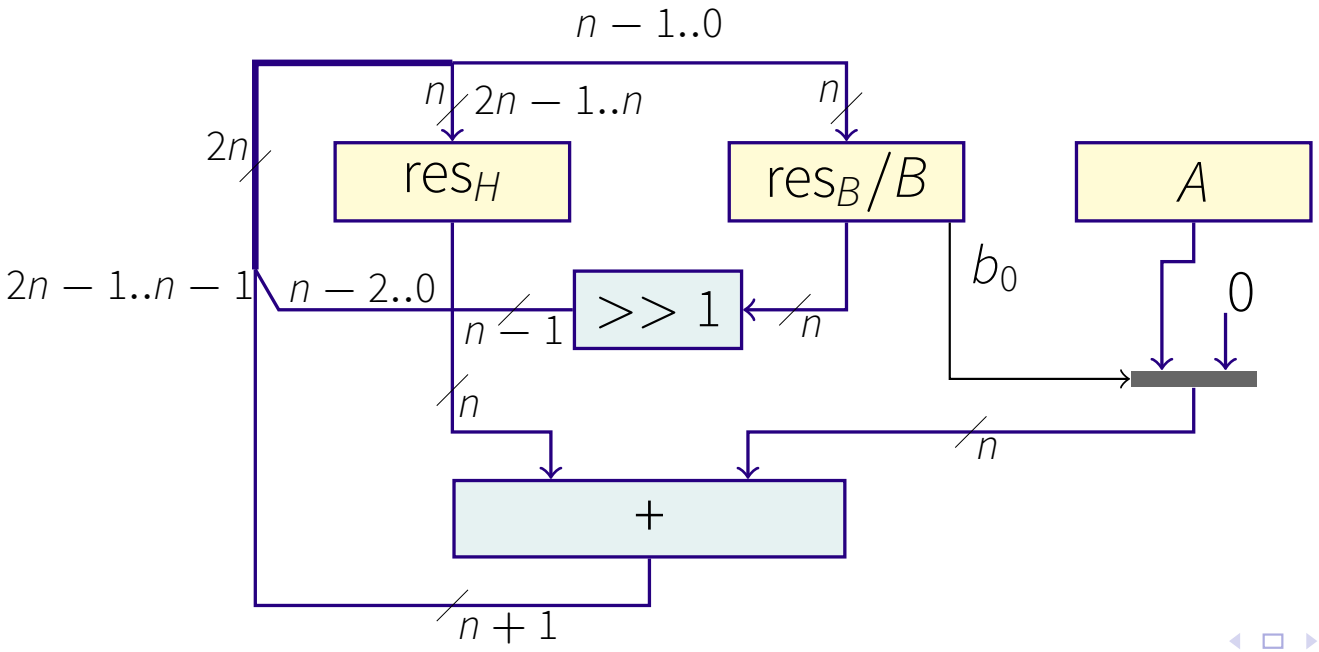
- A sur $2n$ bits et à l'étape i, les i LSB et les $n - i$ MSB de A sont à 0.
- pour $i = 0$, res $= 0$ et à l'étape i, ses $n - 1 - i$ bits de poids fort sont nulls.

$\Longrightarrow$ On peut effectuer les calculs sur uniquement n bits (et non $2n$).

De plus, à l'étape i, les i bits de poids faibles de res ne seront plus modifiés et les i bits de poids forts de B seront à zéro.

$\Longrightarrow$ utilisation d'un registre unique pour B et les poids faibles de res.

Décalage à gauche de A remplacé par décalage à droite de res/B.



un additionneur n bits

3 registres n bits.

Contrôlé par un automate

Nécessite n cycles de calcul

Plan

Cas des nombres signés

Soit $B = (b_{n-1}, \ldots, b_1, b_0)$ un nombre signé codé en complément à deux.

$$B = -2^{n-1} \times b_{n-1} + \sum_{i=n-2}^{0} b_i \times 2^i$$

L'irrégularité entre le bit $n-1$ et les autres complexifie la mise en oeuvre.

En remarquant que $2^i = 2^{i+1} - 2^i$, B peut se réécrire

$$\begin{array}{rcl}
B & = & -2^{n-1} \times b_{n-1} + 2^{n-2} \times b_{n-2} + 2^{n-3} \times b_{n-3} + \cdots + 2b_1 + b_0 \\
& = & -2^{n-1} \times b_{n-1} + 2^{n-1} \times b_{n-2} \\
& & - 2^{n-2} \times b_{n-2} + 2^{n-2} \times b_{n-3} \\
& & - 2^{n-3} \times b_{n-3} + \cdots \\
& & - 2b_1 + 2b_0 \\
& & - b_0 \\
& = & \sum_{i=n-1}^{0} (-b_i + b_{i-1}) \times 2^i
\end{array}$$

en posant $b_{-1} = 0$.

Algorithme (ou réécriture) de **Booth**

Cas des nombres signés (cont.)

En posant $b'_i = -b_i + b_{i-1}$

$$A \times B = \sum_{i=n-1}^{0} b'_i \times 2^i \times A$$

Suivant les cas, b'_i vaut 0, -1 ou $+1$ et il faut ajouter ou soustraire $A \times 2^i$

b_i	b_{i-1}	$b'_i = b_i - b_{i-1}$		
1	1	0	$\rightarrow$	$+0$
1	0	-1	$\rightarrow$	$-2^i \times A$
0	1	1	$\rightarrow$	$+2^i \times A$
0	0	0	$\rightarrow$	$+0$

Cas des nombres signés (cont.)

D'où l'algorithme de multiplication de nombres signés de Booth

```
1   résultat ← 0
2   pour j ← 0 à n − 1
3       selon b_j b_{j−1}
4           cas 10 :        résultat ← résultat - A
5           cas 01 :        résultat ← résultat + A
6           cas 00 ou 11 :  résultat ← résultat
7       A ← A × 2                                  ▷ Décalage
```

Adaptation du schéma du multiplieur non signé.

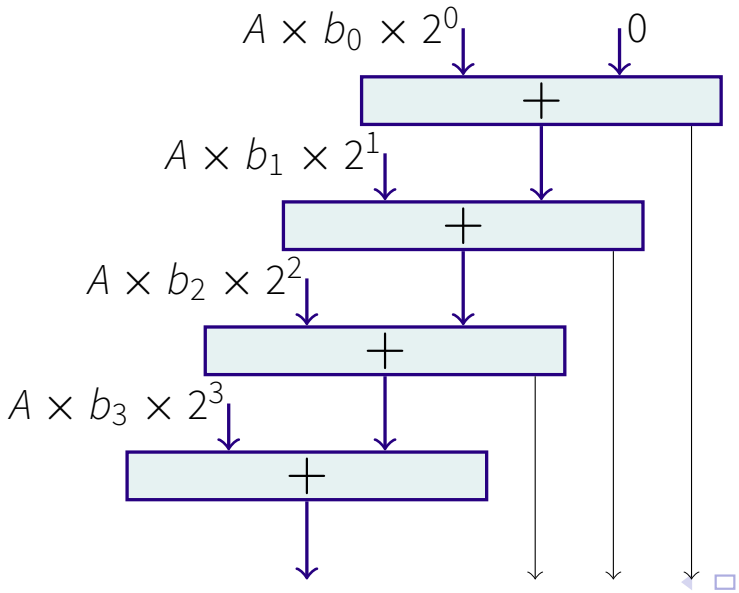
On utilise fréquemment une décomposition en base 4 (*algorithme de Booth modifié*).
A chaque étape, il faut alors ajouter au résultat, soit 0, soit $\pm A$, soit $\pm 2 \times A$
Le calcul en base 4 permet de réduire le nombre de couches d'additionneurs par 2. Intéressant y compris pour des multiplications non signées (cf TD).

Plan

Multiplieur en tableau

Pour accélérer, déroulage dans l'espace de la suite d'additions au moyen de plusieurs additionneurs.

$$
\begin{array}{cccccccc}
 & & & & a_{n-1} & a_{n-2} & \cdots & a_1 \quad a_0 \\
 & & & \times & b_{n-1} & b_{n-2} & \cdots & b_1 \quad b_0 \\
\hline
 & & & & b_0a_{n-1} & b_0a_{n-2} & \cdots & b_0a_1 \quad b_0a_0 \\
+ & & & b_1a_{n-1} & b_1a_{n-2} & \cdots & b_1a_1 & b_1a_0 \\
+ & & b_2a_{n-1} & b_2a_{n-2} & \cdots & b_2a_1 & b_2a_0 & \\
\vdots & & & \vdots & \vdots & & & \\
+ & b_{n-1}a_{n-1} & b_{n-1}a_{n-2} & \cdots & b_{n-1}a_1 & b_{n-1}a_0 & &
\end{array}
$$

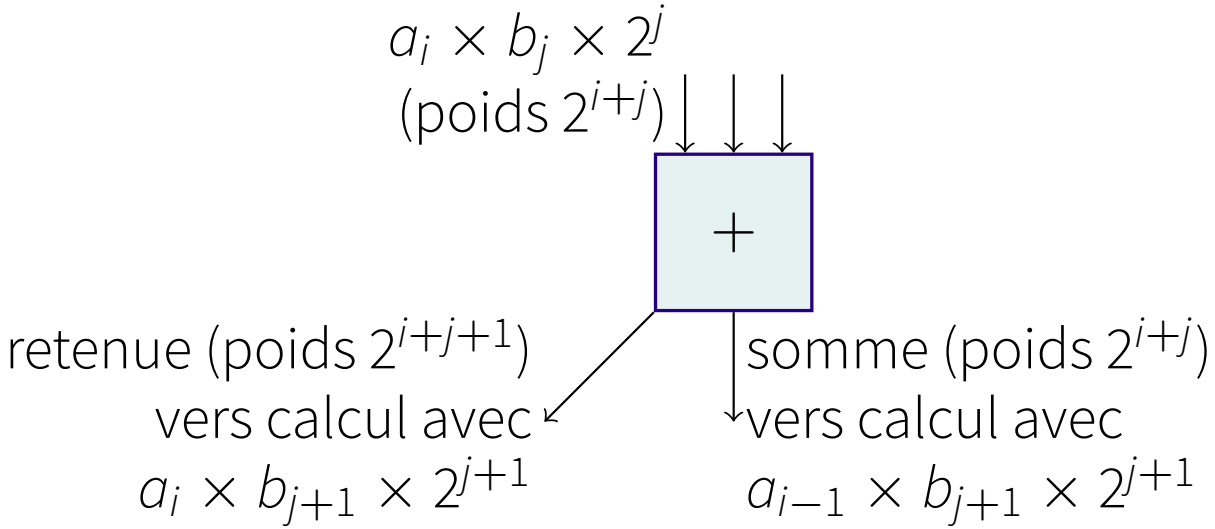


Multiplieur en tableau (cont.)

Problème : n additionneurs à traverser

- avec des additionneurs à propagation, chaque addition prend un temps n et la multiplication est en n^2
- avec des additionneurs rapides (type à génération anticipée), le temps reste de l'ordre de $n \times \log n$

Mais une multiplication est une *accumulation*, et on peut utiliser des **additionneurs à sauvegarde de retenue**.
Chaque additionneur génère somme et retenue et la retenue est prise en compte à l'étage suivant.

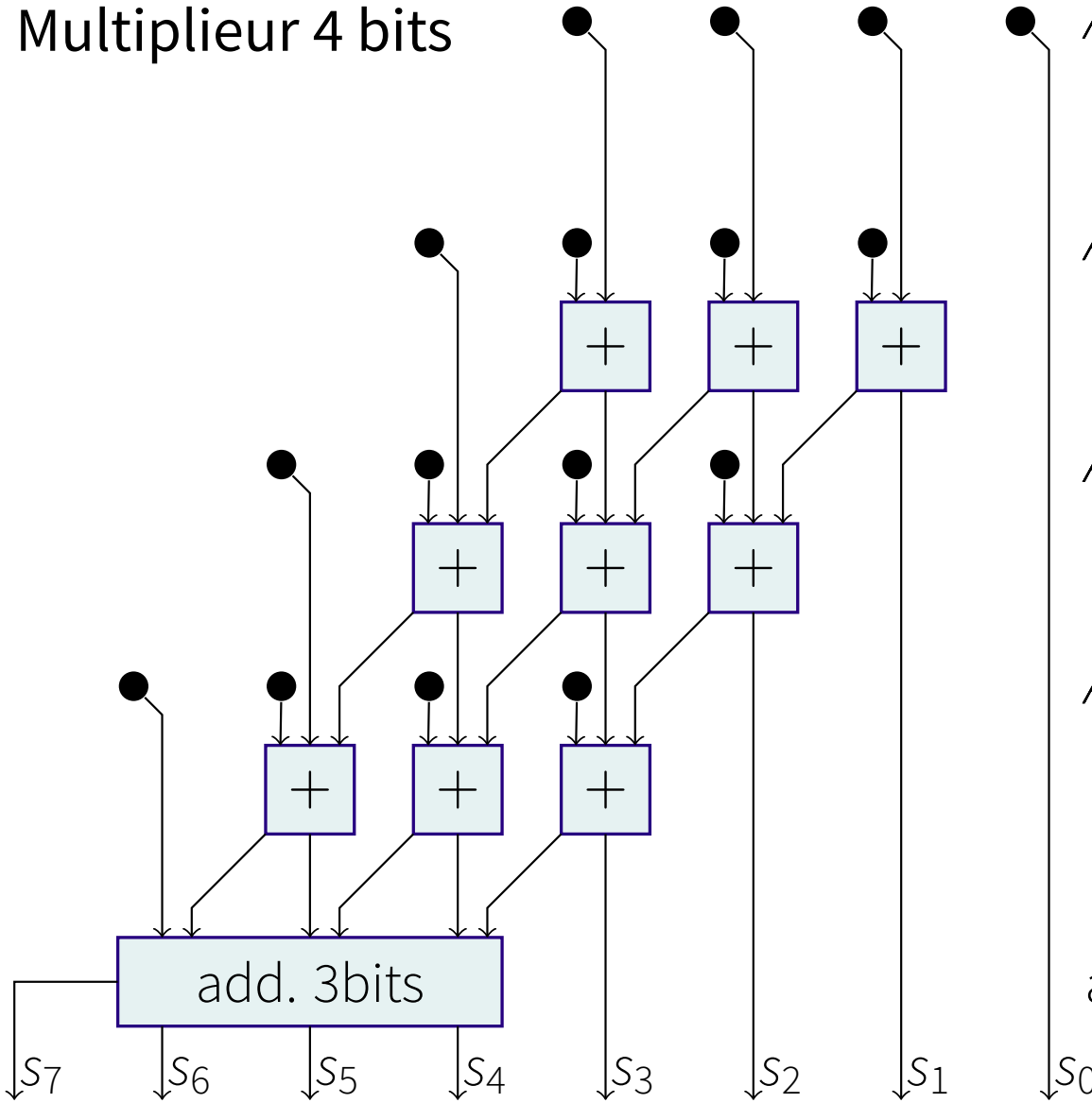


Chaque couche est traversée en temps 1.

Temps total $\approx n$
+ addition finale des sommes et retenue

Multiplieur en tableau (cont.)

Multiplieur 4 bits



$A \times b_0 \times 2^0$

$A \times b_1 \times 2^1$

$A \times b_2 \times 2^2$

$A \times b_3 \times 2^3$

addition finale

Pour une multiplication $n \times n$, il faut :
$(n-1) \times (n-1)$ add. 1 bit
$+ n$ add. 1 bit (add. finale à propagation).

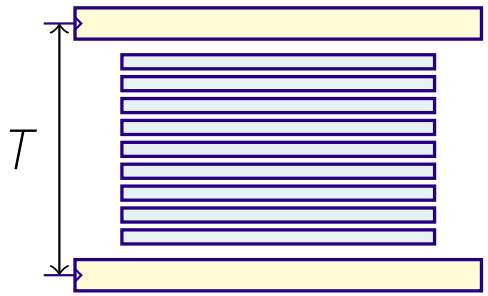
Le temps est $(n-1) \times \tau + (n-1) \times \tau$

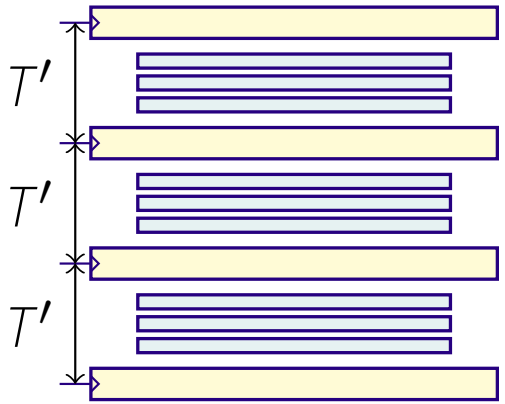
Plan

Mise en oeuvre pipeline

Le temps de calcul $2 \times (n-1)$ est supérieur au chemin critique d'un additionneur rapide.
La présence d'un multiplieur va imposer une période d'horloge T plus *longue*.

Pour éviter de dégrader T et donc l'ensemble des performances
$\Longrightarrow$ découpage du multiplieur en plusieurs *étages*, séparé par des registres.





Si p étages, la période d'horloge sera approximativement divisée par p;

Technique du « **pipeline** »

Mise en oeuvre pipeline (cont.)

L'introduction de registres permet d'effectuer plusieurs multiplications en même temps.
Soit à calculer
`for(int i=0;i<N;i++) R[i]=A[i]*B[i];`

A $t = 0$, début du calcul de `R[0]=A[0]*B[0]`
A $t = 1$, suite du calcul de `R[0]` et début du calcul de `R[1]=A[1]*B[1]`
A $t = 2$, suite des calculs de `R[0]` et `R[1]` et début du calcul de `R[2]`
etc.

t_0	t_1	t_2	t_3	t_4	t_5	t_6
$R^0 = A^0 \times b_0^0$	$R^0 += A^0 \times b_1^0 \times 2^1$	$R^0 += A^0 \times b_2^0 \times 2^2$	$R^0 += A^0 \times b_3^0 \times 2^3$			
	$R^1 = A^1 \times b_0^1$	$R^1 += A^1 \times b_1^1 \times 2^1$	$R^1 += A^1 \times b_2^1 \times 2^2$	$R^1 += A^1 \times b_3^1 \times 2^3$		
		$R^2 = A^2 \times b_0^2$	$R^2 += A^2 \times b_1^2 \times 2^1$	$R^2 += A^2 \times b_2^2 \times 2^2$	$R^2 += A^2 \times b_3^2 \times 2^3$	
			$R^3 = A^3 \times b_0^3$	$R^3 += A^3 \times b_1^3 \times 2^1$	$R^3 += A^3 \times b_2^3 \times 2^2$	$R^3 += A^3 \times b_3^3 \times 2^3$
				$\cdots$	$\cdots$	$\cdots$

Le pipeline permet de :
- réduire le temps de cycle
- lancer un calcul par cycle

$\Longrightarrow$ forte amélioration des performances

Mise en oeuvre pipeline (cont.)

On distingue :

latence temps entre le début d'un calcul et la disponibilité des résultats
inchangée (voire légèrement augmentée) par le pipeline

débit nombre d'opérations possibles par unité de temps
lié au temps de cycle et fortement amélioré par le pipeline

Supposons un multiplieur avec n couches d'additionneurs.
τ_a et τ_r temps de traversée des additionneurs et des registres
mise en pipeline avec k étages de p couches d'additions ($n = k \times p$)

	temps cycle	latence	débit
sans pipeline	$\tau_r + n \times \tau_a$	$\tau_r + n \times \tau_a$	$\frac{1}{\tau_r + n \times \tau_a}$
avec pipeline	$\tau_r + p \times \tau_a$	$k \times (\tau_r + p \times \tau_a)$ $= k \times \tau_r + n \times \tau_a$	$\frac{1}{\tau_r + p \times \tau_a}$

Si $n = 64, p = k = 8, \tau_a = \tau_r$,

- débit $1/65 \rightarrow 1/9$ ($\times 7$)
- latence $65 \rightarrow 72$ ($+10\%$)

Mise en oeuvre pipeline (cont.)

Le pipeline permet

- d'améliorer fortement le débit
- au prix d'une légère augmentation de la latence
- et d'un surcoût matériel limité

Omniprésent dans les architectures actuelles

- opérateurs de calculs (multiplieurs, diviseurs, calcul flottant)
- processeurs (CPU, GPU) (exécution pipeline des instructions)
- opérateurs de calcul vectoriel, traitement de signal, IA, etc

En général le débit est ce qui détermine les performances
mais l'augmentation de latence peut être pénalisante dans certains cas (*récurrences*)

```
for(int i=0;i<N;i++) R *= A[i];
```

Il faut attendre la fin d'un calcul pour démarrer le calcul suivant.
Pas de possibilité d'exécution simple en pipeline.

Mise en oeuvre pipeline (cont.)

Version pipeline de l'additionneur (avec 1 couche d'additionneur par étage)

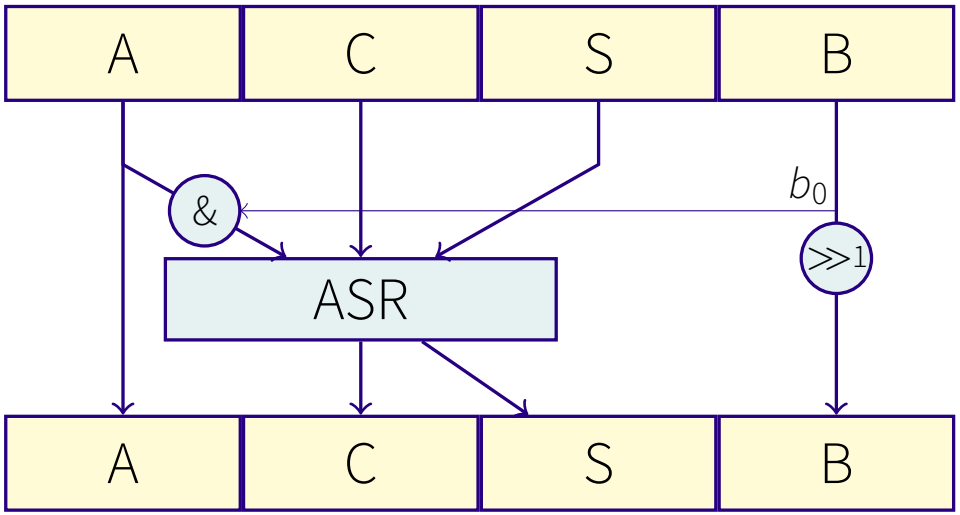
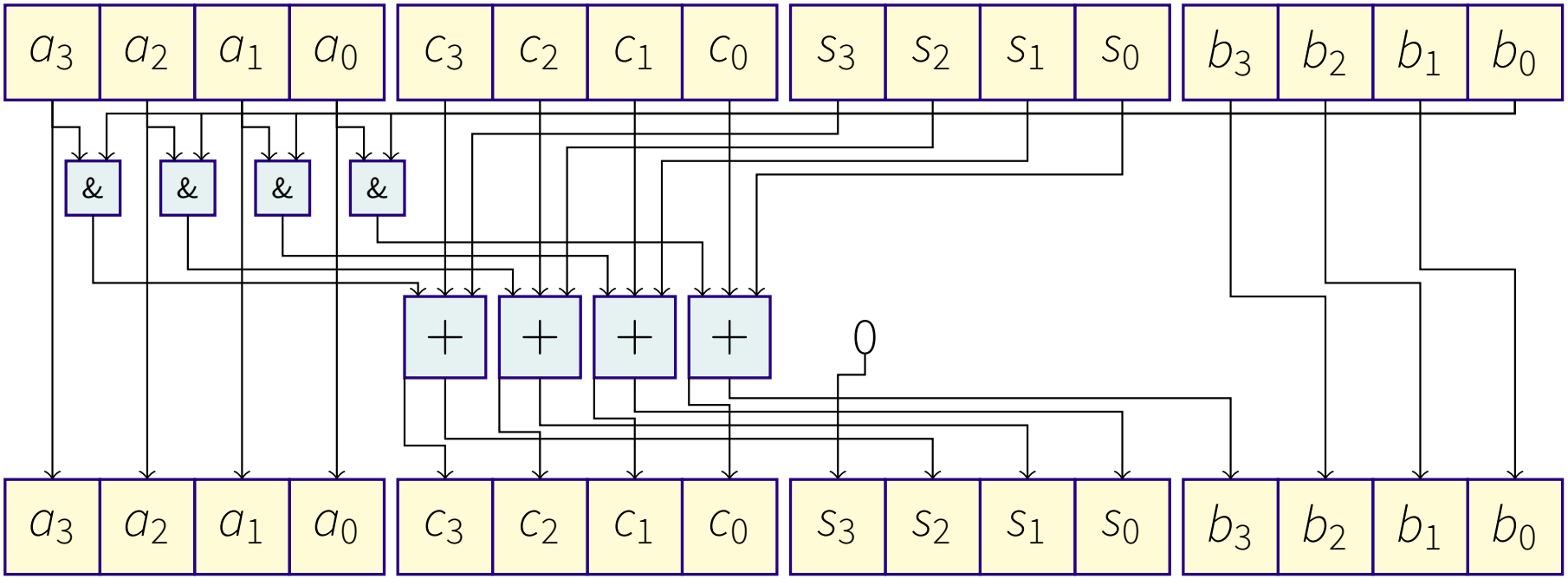


Schéma de principe d'un étage d'additionneur.

Le schéma détaillé assure que *tous* les étages sont identiques. (diverses optimisations sont possibles)



Mise en oeuvre pipeline (cont.)

Il existe des schémas de multiplieurs plus rapides (en un temps $\log n$) (Wallace-Dadda, Luk-Vuillemin).
Mais leur structure est plus irrégulière, et ils sont plus délicats à pipeliner.
Rarement utilisés.

En pratique, sur les processeurs actuels, multiplieurs ($64 \times 64 \rightarrow 64$ ou $32 \times 32 \rightarrow 64$) en tableau, pipelinés sur 3 à 5 étages.

Les premiers étages réalisent les additions à sauvegarde de retenue.
Le dernier étage permet d'effectuer l'addition finale avec un additionneur rapide.

Utilisation fréquente d'un algorithme de Booth en base 4 (algorithme de Booth modifié).