Unité Arithmétique et Logique

Christophe Mouilleron

ensiie
école nationale supérieure d'informatique
pour l'industrie et l'entreprise

Plan

Half-adder

Addition sur 1 bit :

- 2 entrées *a* et *b*
- 2 sorties = somme s + retenue c

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

Bilan ?

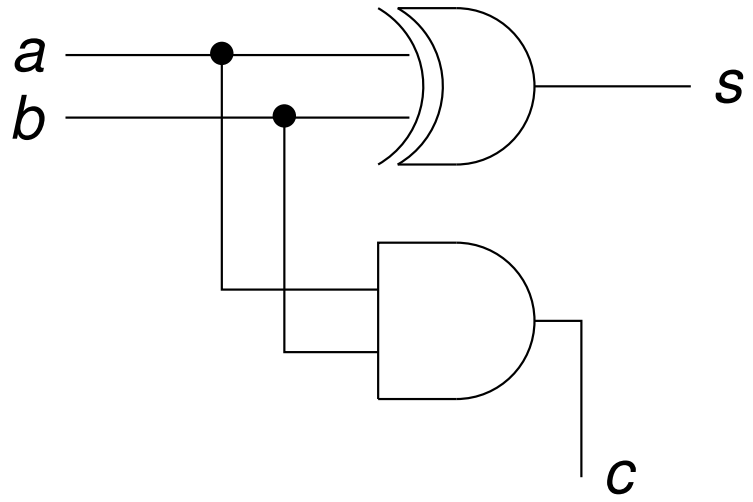
Half-adder

Addition sur 1 bit :

- 2 entrées a et b
- 2 sorties = somme s + retenue c

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Bilan : $s = a \oplus n$ $c = a \cdot b$



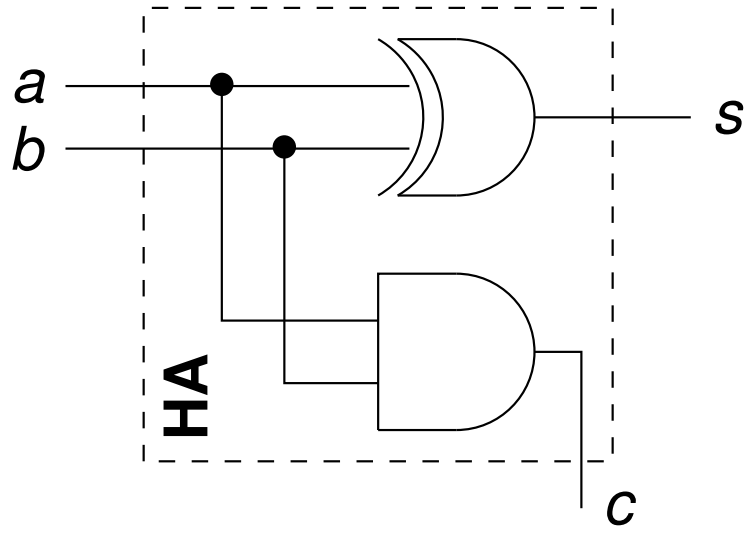
Half-adder

Addition sur 1 bit :

- 2 entrées a et b
- 2 sorties = somme s + retenue c

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Bilan : $s = a \oplus n$ $c = a \cdot b$



Full-adder

Addition sur n bits $\Rightarrow$ gestion des retenues

Full-adder sur 1 bit :

- $2 + 1$ entrées = a, b, et c_{in}
- $1 + 1$ sorties = s et c_{out}

a	b	c_{in}	s	c_{out}
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

Formules ?

Full-adder

Addition sur n bits $\Rightarrow$ gestion des retenues

Full-adder sur 1 bit :

- $2 + 1$ entrées = a, b, et c_{in}
- $1 + 1$ sorties = s et c_{out}

a	b	c_{in}	s	c_{out}
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

Formules :

- $s = a \oplus b \oplus c_{\text{in}}$

Full-adder

Addition sur n bits $\Rightarrow$ gestion des retenues

Full-adder sur 1 bit :

- $2 + 1$ entrées = a, b, et c_{in}
- $1 + 1$ sorties = s et c_{out}

a	b	c_{in}	s	c_{out}
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

Formules :

- $s = a \oplus b \oplus c_{\text{in}}$
- $c_{\text{out}} = a \cdot b + a \cdot c_{\text{in}} + b \cdot c_{\text{in}}$

Full-adder

Addition sur n bits $\Rightarrow$ gestion des retenues

Full-adder sur 1 bit :

- $2 + 1$ entrées = a, b, et c_{in}
- $1 + 1$ sorties = s et c_{out}

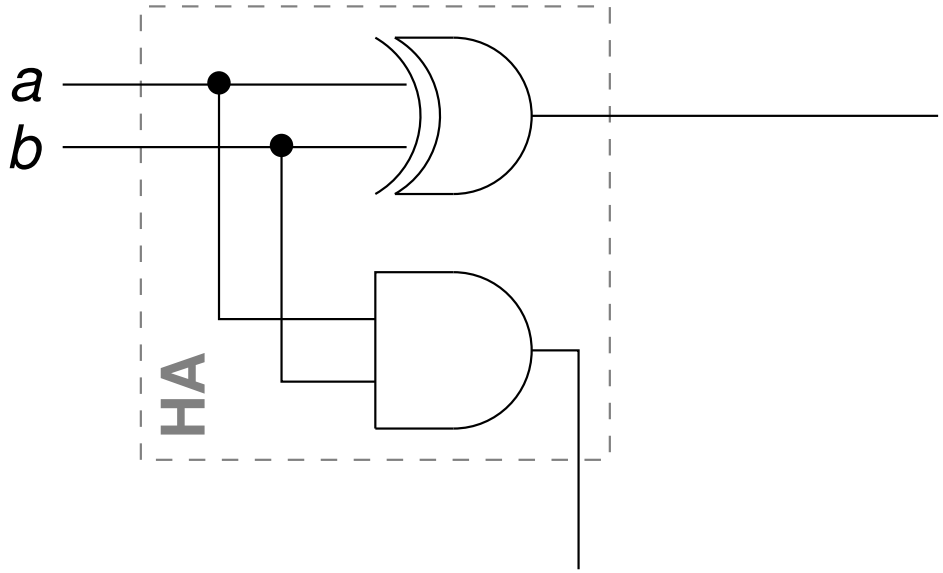
a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Formules :

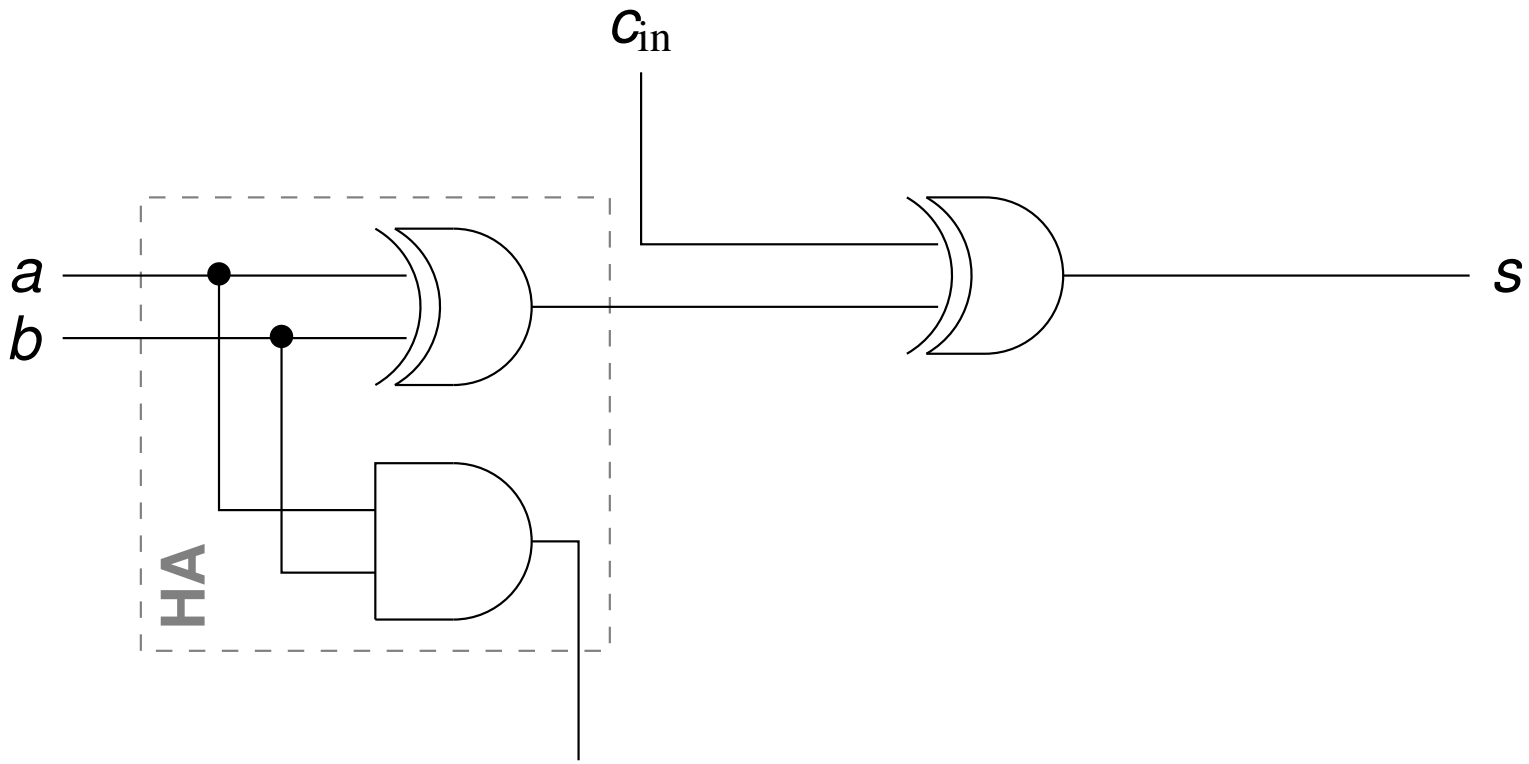
- $s = a \oplus b \oplus c_{\text{in}}$
- $c_{\text{out}} = a \cdot b + a \cdot c_{\text{in}} + b \cdot c_{\text{in}}$

$\rightsquigarrow$ $c_{\text{out}} = a \cdot b + (a \oplus b) \cdot c_{\text{in}}$

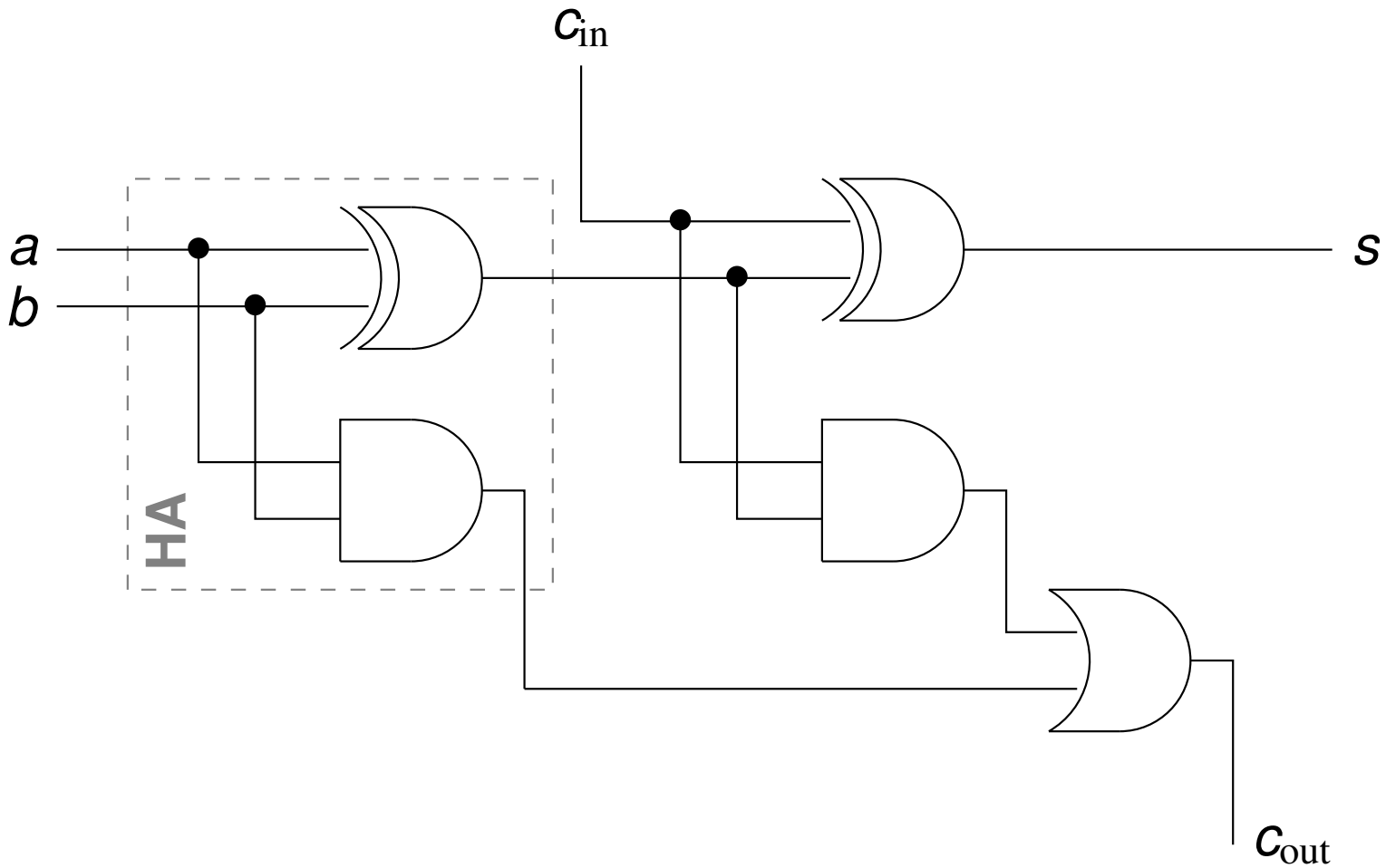
Full-adder, circuit



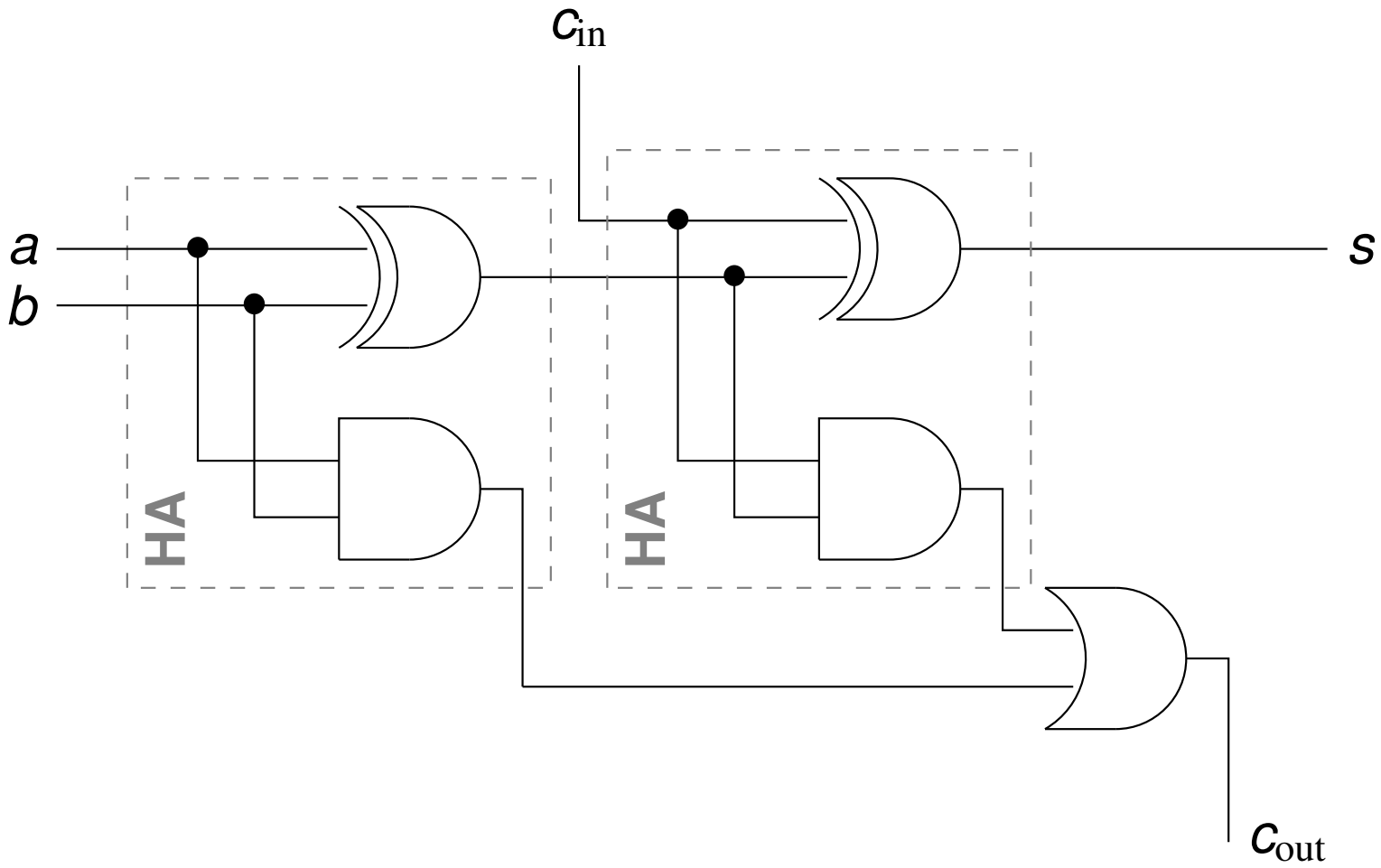
Full-adder, circuit



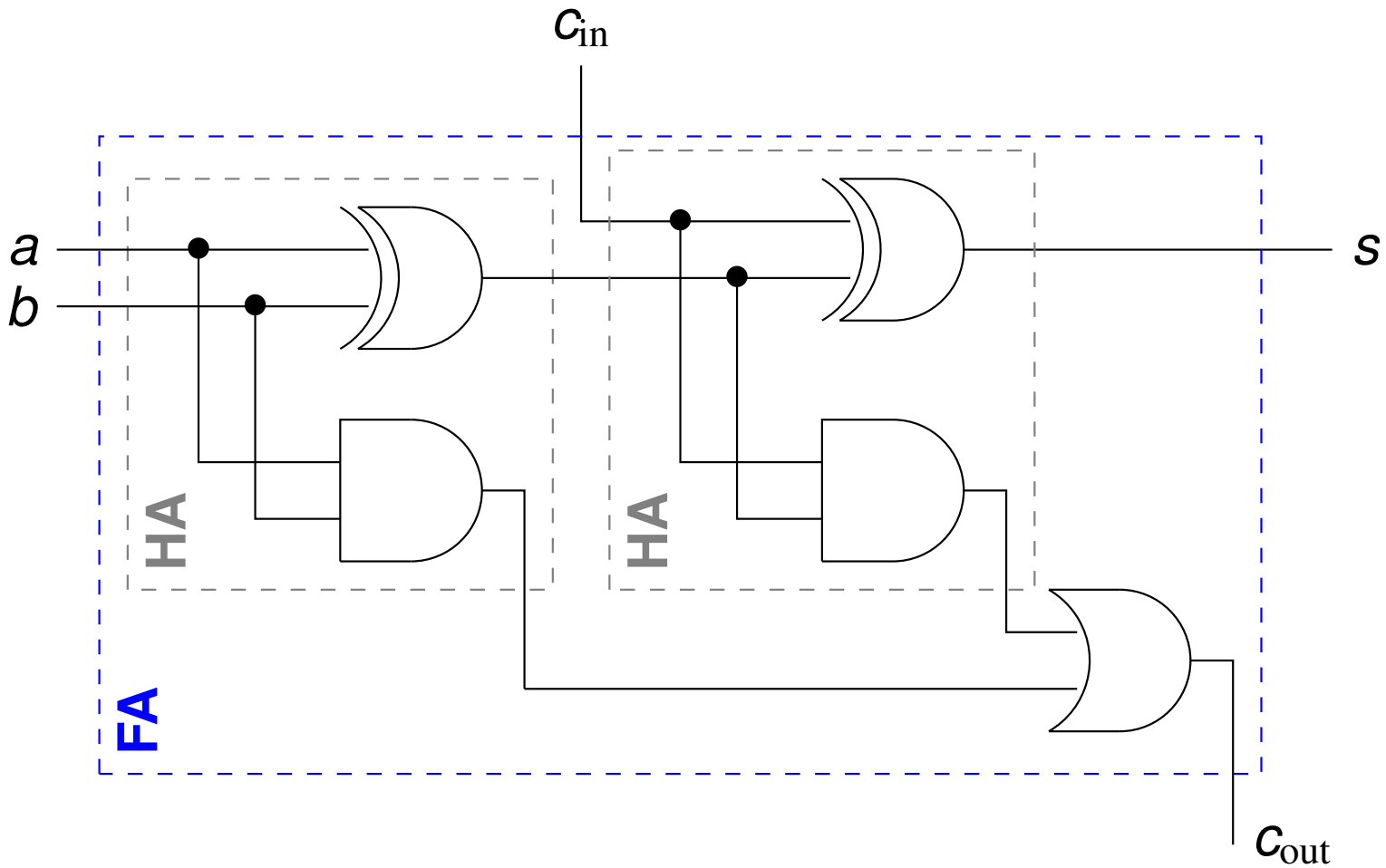
Full-adder, circuit



Full-adder, circuit



Full-adder, circuit



Bilan sur les briques de bases

Half-adder = addition 1 bit sans retenue

- 2 entrées + 2 sorties
- 2 portes logiques — surface
- profondeur 1 — latence

Bilan sur les briques de bases

Half-adder = addition 1 bit sans retenue

- 2 entrées + 2 sorties
- 2 portes logiques — surface
- profondeur 1 — latence

Full-adder = Addition sur 1 bit avec retenue

- 3 entrées + 2 sorties
- 2 half-adders + 1 `OR`
- 5 portes logique — surface
- profondeur 3 — latence

Bilan sur les briques de bases

Half-adder = addition 1 bit sans retenue

- 2 entrées + 2 sorties
- 2 portes logiques — surface
- profondeur 1 — latence

Full-adder = Addition sur 1 bit avec retenue

- 3 entrées + 2 sorties
- 2 half-adders + 1 `OR`
- 5 portes logique — surface
- profondeur 3 — latence

FA = compresseur $3 \rightarrow 2$

Plan

Additionneur n bits

Entrées :

- opérande $A = \sum_{i=0}^{n-1} a_i 2^i \in [\![0, 2^n - 1]\!]$ n bits
- opérande $B = \sum_{i=0}^{n-1} b_i 2^i \in [\![0, 2^n - 1]\!]$ n bits

Résultat $S = A + B \quad \in [\![0, 2^{n+1} - 2]\!]$:

- somme tronquée à n bits $s_0, \ldots, s_{n-1}$
- retenue sortante c_n

$$S = c_n \cdot 2^n + \sum_{i=0}^{n-1} s_i \cdot 2^i$$

Additionneur n bits

Entrées :

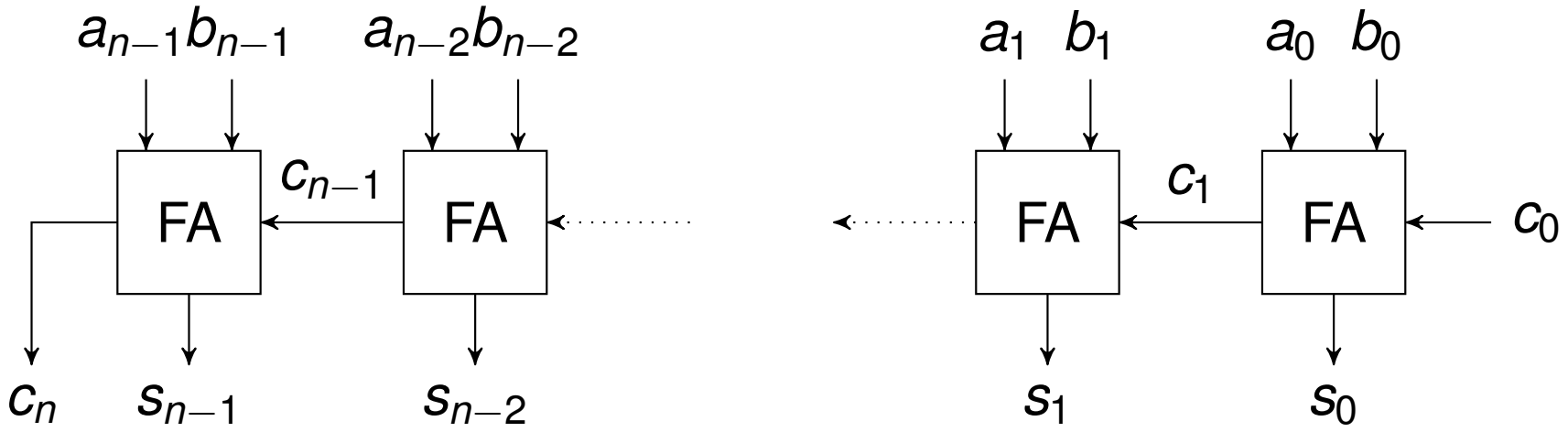
- opérande $A = \sum_{i=0}^{n-1} a_i 2^i \in [\![0, 2^n - 1]\!]$ — n bits
- opérande $B = \sum_{i=0}^{n-1} b_i 2^i \in [\![0, 2^n - 1]\!]$ — n bits
- retenue entrante c_0 — 1 bit

Résultat $S = A + B + c_0 \in [\![0, 2^{n+1} - 1]\!]$:

- somme tronquée à n bits — $s_0, \ldots, s_{n-1}$
- retenue sortante c_n

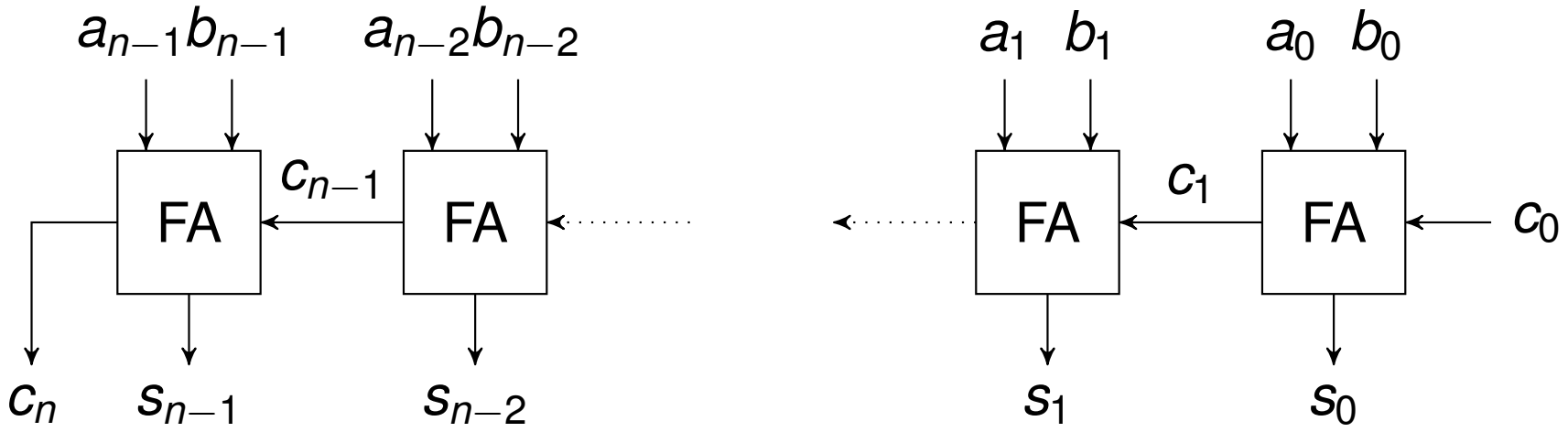
$$S = c_n \cdot 2^n + \sum_{i=0}^{n-1} s_i \cdot 2^i$$

Addition par propagation de retenue



Propriétés :

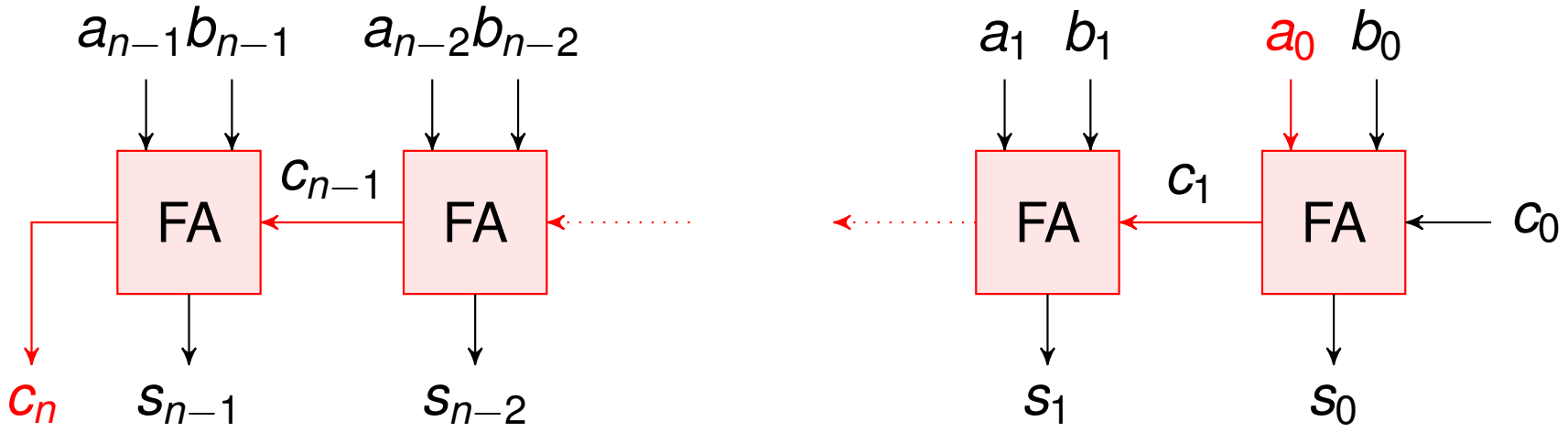
Addition par propagation de retenue



Propriétés :

- $5n$ portes logiques — linéaire en n

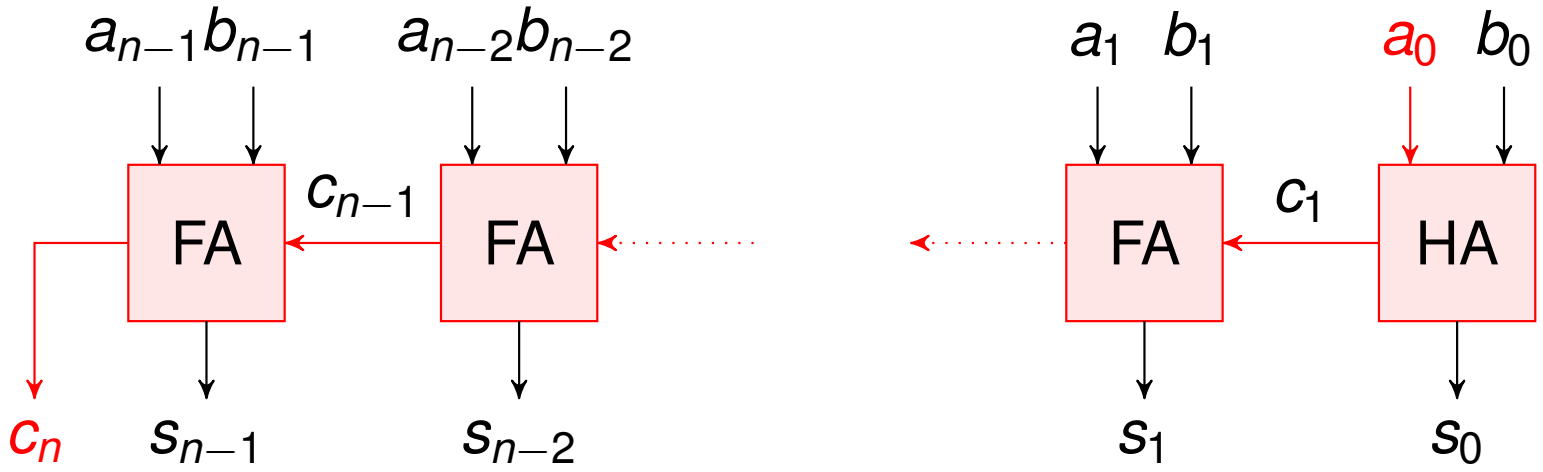
Addition par propagation de retenue



Propriétés :

- $5n$ portes logiques — linéaire en n
- profondeur $3n$ — linéaire en $n \Rightarrow$ LENT !!!

Addition par propagation de retenue



Propriétés :

- $5n - 3$ portes logiques
- profondeur $3n - 2$

linéaire en n

linéaire en $n \Rightarrow$ LENT !!!

Plan

Problème

Constat : calcul séquentiel à cause de la propagation de retenue

Solutions ?

Problème

Constat : calcul séquentiel à cause de la propagation de retenue

Solutions :

1. accélérer le calcul des retenues — Anticiper
2. choisir la valeur d'une retenue — Prédire ?
3. ignorer les retenues — Tricher ?

Anticiper (1)

Cycle de vie d'une retenue :

- naissance quand $a_i = b_i = 1$
- vie tant que $a_i \neq b_i$
- mort quand $a_i = b_i = 0$

$\{a_i,\ b_i\} = \{0,\ 1\}$

Anticiper (1)

Cycle de vie d'une retenue :

- naissance quand $a_i = b_i = 1$
- vie tant que $a_i \neq b_i$
- mort quand $a_i = b_i = 0$

$\{a_i,\ b_i\} = \{0,\ 1\}$

Notations :

- c_i = retenue à la position i

Anticiper (1)

Cycle de vie d'une retenue :

- naissance quand $a_i = b_i = 1$
- vie tant que $a_i \neq b_i$
- mort quand $a_i = b_i = 0$

$\{a_i,\ b_i\} = \{0,\ 1\}$

Notations :

- c_i = retenue à la position i
- g_i = génération de retenue à la pos. i

Anticiper (1)

Cycle de vie d'une retenue :

- naissance quand $a_i = b_i = 1$
- vie tant que $a_i \neq b_i$
- mort quand $a_i = b_i = 0$

$\{a_i,\ b_i\} = \{0,\ 1\}$

Notations :

- c_i = retenue à la position i
- g_i = génération de retenue à la pos. i
- p_i = propagation de retenue à la pos. i

Anticiper (1)

Cycle de vie d'une retenue :

- naissance quand $a_i = b_i = 1$
- vie tant que $a_i \neq b_i$ $\{a_i,\ b_i\} = \{0,\ 1\}$
- mort quand $a_i = b_i = 0$

Notations :

- c_i = retenue à la position i
- g_i = génération de retenue à la pos. i
- p_i = propagation de retenue à la pos. i indépendamment de c_i

Anticiper (2)

Calcul de g_i ? $g_i = a_i \cdot b_i$

Calcul de p_i ? $p_i = a_i \oplus b_i$ ou $p_i = a_i + b_i$

- calculs en parallèle
- valeurs dispo. après une latence de 1 en portes logiques

Calcul de c_{i+1} ?

Anticiper (2)

Calcul de g_i ? $g_i = a_i \cdot b_i$

Calcul de p_i ? $p_i = a_i \oplus b_i$ ou $p_i = a_i + b_i$

- calculs en parallèle
- valeurs dispo. après une latence de 1 en portes logiques

Calcul de c_{i+1} ? retenue si génération ou propagation

$$c_{i+1} = g_i + p_i \cdot c_i$$

Anticiper (3)

Latence ?

- pour $c_0 = 0$

entrée

Anticiper (3)

Latence ?

- pour $c_0 = 0$
- pour $c_1 = 3$

entrée

$$c_1 = g_0 + p_0 \cdot c_0$$

Anticiper (3)

Latence ?

- pour $c_0 = 0$ — entrée
- pour $c_1 = 3$ — $c_1 = g_0 + p_0 \cdot c_0$
- pour $c_{i+1} = 2$ + latence pour c_i — $c_{i+1} = g_i + p_i \cdot c_i$

Anticiper (3)

Latence ?

- pour $c_0 = 0$
- pour $c_1 = 3$
- pour $c_{i+1} = 2 +$ latence pour c_i

$\rightsquigarrow$ profondeur pour avoir $c_n = 2\,n + 1$ portes

entrée

$c_1 = g_0 + p_0 \cdot c_0$

$c_{i+1} = g_i + p_i \cdot c_i$

Est-ce vraiment mieux qu'avant ?

Anticiper (3)

Latence ?

- pour $c_0 = 0$ — entrée
- pour $c_1 = 3$ — $c_1 = g_0 + p_0 \cdot c_0$
- pour $c_{i+1} = 2 +$ latence pour c_i — $c_{i+1} = g_i + p_i \cdot c_i$

$\rightsquigarrow$ profondeur pour avoir $c_n = 2n + 1$ portes

Est-ce vraiment mieux qu'avant ? — pas vraiment

- gain de $\approx 33\%$ en latence . . . en apparence
- p_i et g_i calculés par le 1er HA de chaque FA
- dans un FA, seulement 2 portes entre c_{in} et c_{out}

Anticiper (4)

Vraie amélioration = additionneur à retenue anticipée

- choisir une petite taille k — taille typique = 4 ou 8
- concevoir un additionneur k bits
  - calcul séquentiel avec des FA
  - calcul optimisé de c_k — réduc. latence
- additionneur n bits = $\frac{n}{k}\times$ additionneur k bits

Anticiper (4)

Vraie amélioration = additionneur à retenue anticipée

- choisir une petite taille k — taille typique = 4 ou 8
- concevoir un additionneur k bits
  - calcul séquentiel avec des FA
  - calcul optimisé de c_k — réduc. latence
- additionneur n bits = $\frac{n}{k}\times$ additionneur k bits

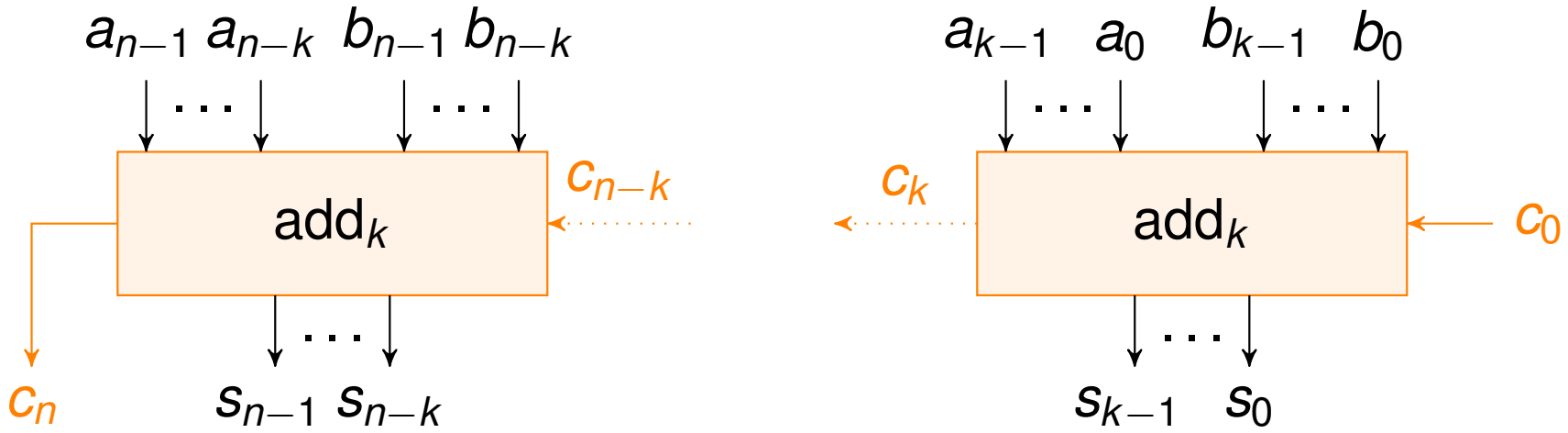
Exemple :

$$c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3 \cdot (g_2 + p_2 \cdot c_2) = \ldots$$
$$= g_3 + g_2 \cdot p_3 + g_1 \cdot p_2 \cdot p_3 + g_0 \cdot p_1 \cdot p_2 \cdot p_3 + c_0 \cdot p_0 \cdot p_1 \cdot p_2 \cdot p_3$$

à factoriser

$\rightsquigarrow$ profondeur 6 au lieu de 9 — plusieurs solutions

Anticiper (5)



Bilan sur l'additionneur à retenue anticipée :

- ✓ réduc. de latence
- ✗ latence toujours linéaire
- ✓ nb. de portes logiques toujours linéaire

chemin optimisé

Prédire ? (1)

Idée :

- utiliser deux additionneurs de $\frac{n}{2}$ bits — $2 \times \mathrm{add}_{n/2}$
- prédire la valeur de $c_{n/2}$ — retenue reliant les 2 $\mathrm{add}_{n/2}$
- calculer avec les deux additionneurs en parallèle
- valider a posteriori le choix pour $c_{n/2}$

Prédire ? (1)

Idée :

- utiliser deux additionneurs de $\frac{n}{2}$ bits — $2 \times \mathrm{add}_{n/2}$
- prédire la valeur de $c_{n/2}$ — retenue reliant les 2 $\mathrm{add}_{n/2}$
- calculer avec les deux additionneurs en parallèle
- valider a posteriori le choix pour $c_{n/2}$
- . . . et relancer le calcul en cas de mauvais choix !

Dupliquer (1)

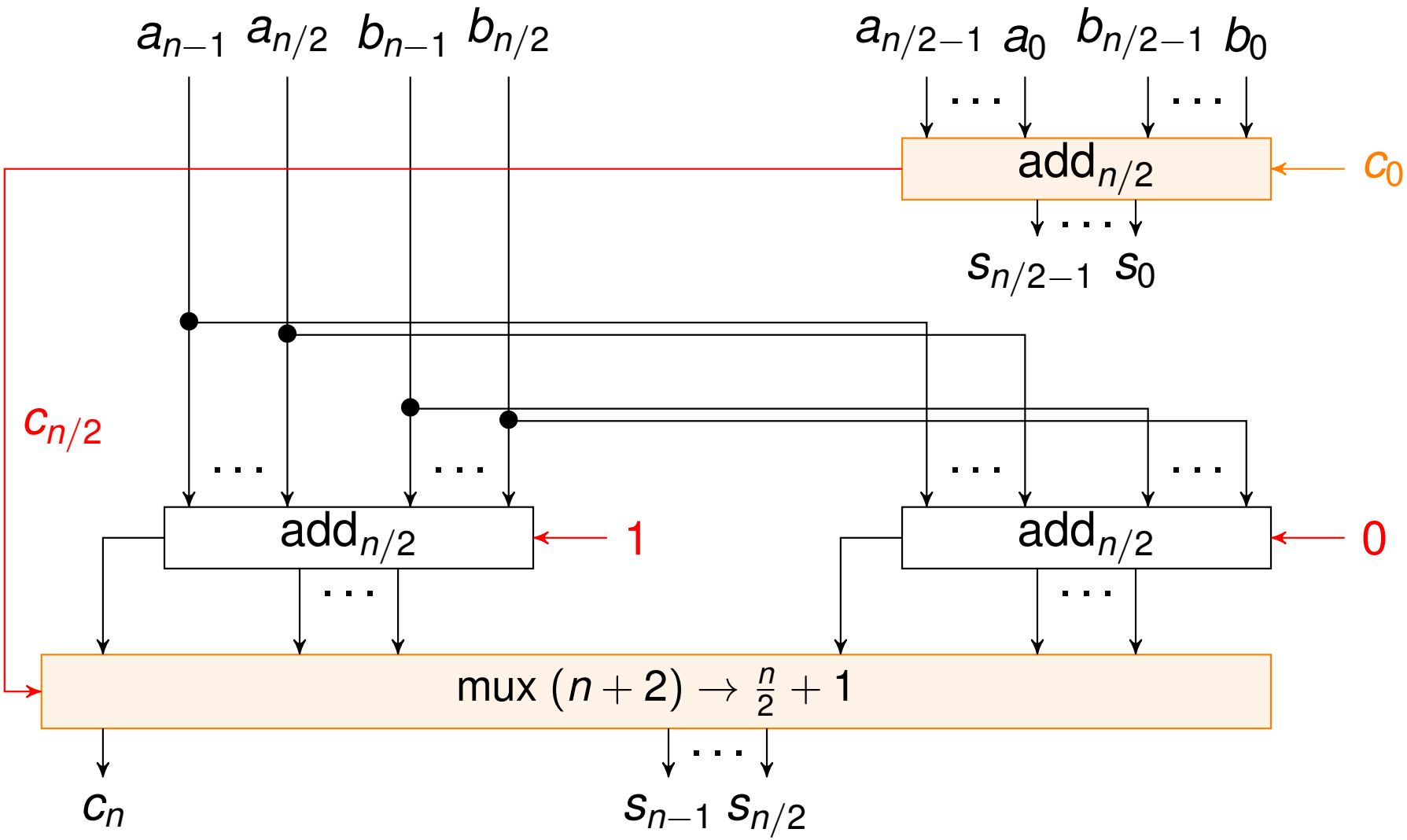
Idée :

- utiliser deux additionneurs de $\frac{n}{2}$ bits — $2 \times \text{add}_{n/2}$
- prédire la valeur de $c_{n/2}$ — retenue reliant les 2 $\text{add}_{n/2}$
- calculer avec les deux additionneurs en parallèle
- valider a posteriori le choix pour $c_{n/2}$
- . . . et relancer le calcul en cas de mauvais choix !

Solution plus réaliste ici :

- utiliser trois additionneurs de $\frac{n}{2}$ bits — $3 \times \text{add}_{n/2}$
- faire les calculs avec $c_{n/2} = 0$ ET $c_{n/2} = 1$
- choisir a posteriori le bon résultat selon la valeur de $c_{n/2}$

Dupliquer (2)


a_{n-1} $a_{n/2}$ b_{n-1} $b_{n/2}$
$a_{n/2-1}$ a_0 $b_{n/2-1}$ b_0
$\text{add}_{n/2}$
c_0
$s_{n/2-1}$ s_0
$c_{n/2}$
$\text{add}_{n/2}$
1
$\text{add}_{n/2}$
0
$\text{mux}\ (n+2) \rightarrow \frac{n}{2}+1$
c_n
s_{n-1} $s_{n/2}$

Dupliquer (3)

Bilan sur l'additionneur à sélection de retenue :

- ✓ latence divisée par ≈ 2
- ✗ $>$ 50% de portes logique en plus

Découper mieux pour gagner plus :

- (sous-)additionneurs de tailles $\neq$ latence un peu meilleure

Dupliquer (3)

Bilan sur l'additionneur à sélection de retenue :

- ✓ latence divisée par ≈ 2
- ✗ $>$ 50% de portes logique en plus

Découper mieux pour gagner plus :

- (sous-)additionneurs de tailles $\neq$ — latence un peu meilleure
- découper en $>$ 2 additionneurs parallèles — latence $\Theta(\sqrt{n})$

Dupliquer (3)

Bilan sur l'additionneur à sélection de retenue :

- ✓ latence divisée par ≈ 2
- ✗ $>$ 50% de portes logique en plus

Découper mieux pour gagner plus :

- (sous-)additionneurs de tailles $\neq$ — latence un peu meilleure
- découper en >2 additionneurs parallèles — latence $\Theta(\sqrt{n})$
- approche récursive possible — latence $\Theta(\log n)$

Dupliquer (3)

Bilan sur l'additionneur à sélection de retenue :

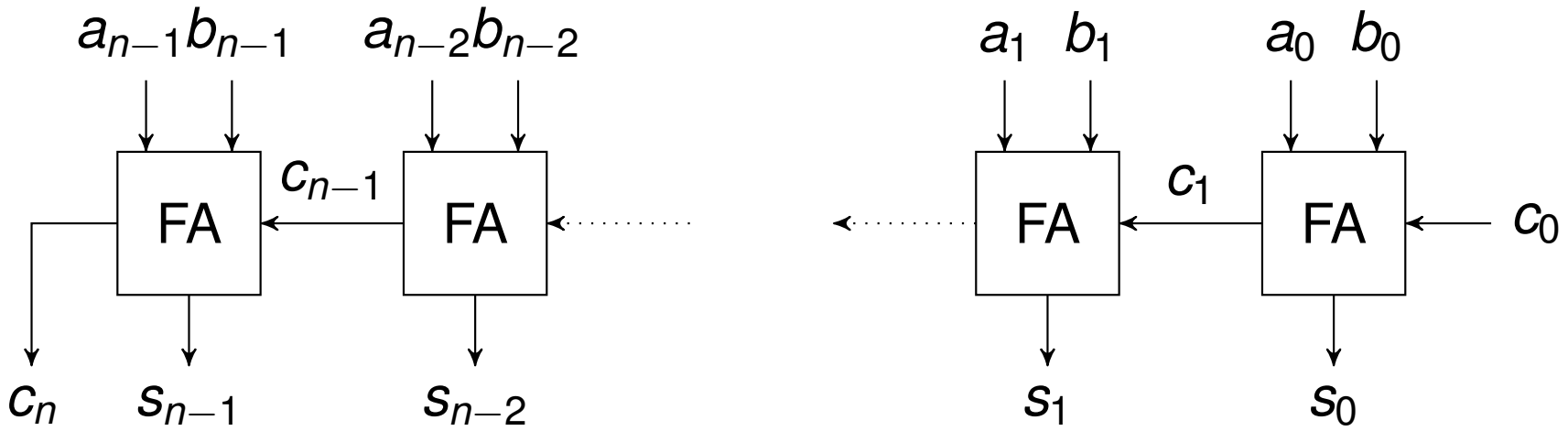
- ✓ latence divisée par ≈ 2
- ✗ $>$ 50% de portes logique en plus

Découper mieux pour gagner plus :

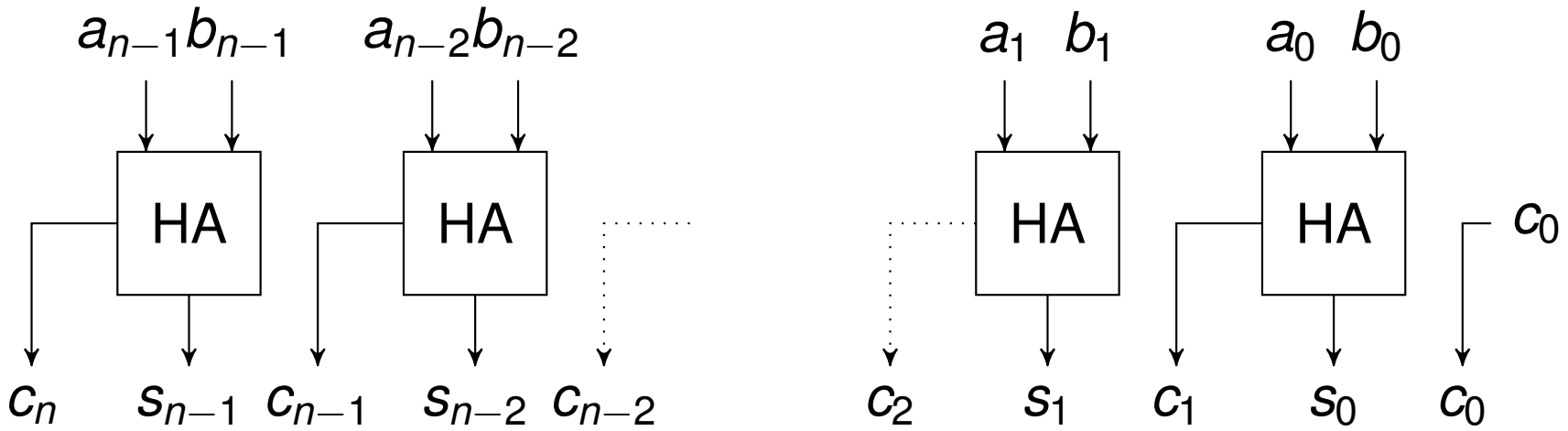
- (sous-)additionneurs de tailles $\neq$ — latence un peu meilleure
- découper en >2 additionneurs parallèles — latence $\Theta(\sqrt{n})$
- approche récursive possible — latence $\Theta(\log n)$

$\rightsquigarrow$ **Compromis** entre latence et surface/simplicité

Tricher ? (1)



Tricher ? (1)



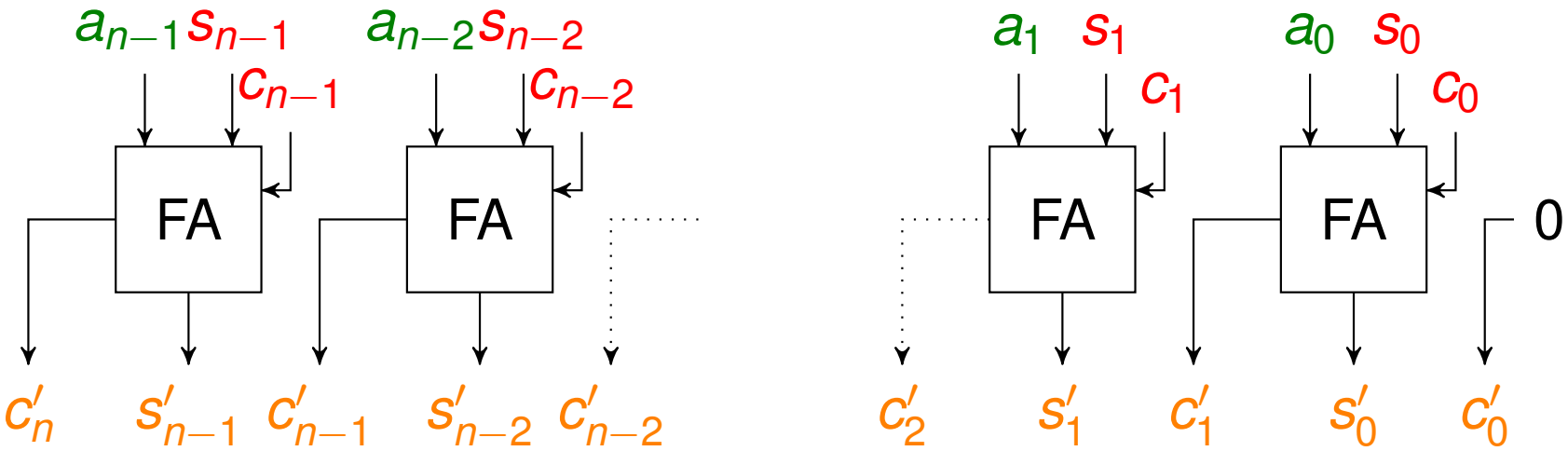
- entrée = représentation binaire classique
- sortie = nouvelle représentation
- redondance $\Rightarrow$ pas de propagation de retenue cf bouliers

On parle de représentation *carry-save*.

$$S = c_n + \sum_{i=0}^{n-1} (s_i + c_i) \cdot 2^i$$

Tricher ? (2)

Bonus : *binaire* + *carry-save* = *carry-save*

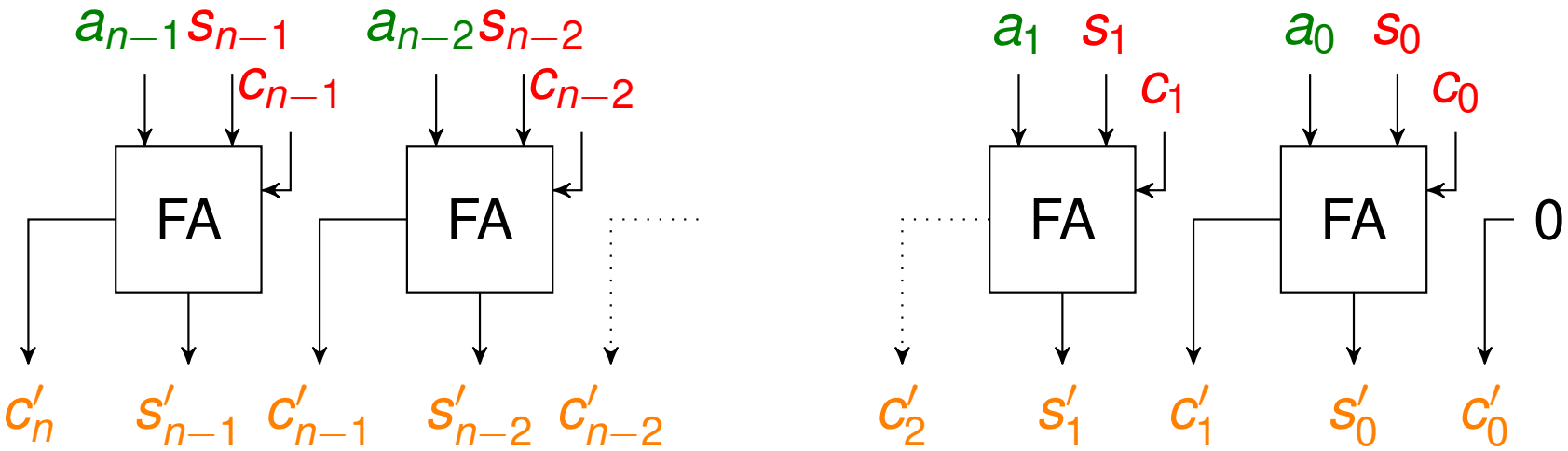


Bilan sur l'additionneur *carry-save* :

- ✓ addition toujours en parallèle — latence $\Theta(1)$
- ✓ processus itérable pour sommer > 2 entiers

Ruser/Paresser (2)

Bonus : *binaire* + *carry-save* = *carry-save*



Bilan sur l'additionneur *carry-save* :

- ✓ addition toujours en parallèle — latence $\Theta(1)$
- ✓ processus itérable pour sommer > 2 entiers
- ✗ conversion *carry-save* $\rightarrow$ binaire = vraie addition $C' + S'$ — coûteux

Plan

Retour sur l'arithmétique non-signée sur n bits

Valeur associée à $\overline{a_{n-1}a_{n-2}\cdots a_1 a_0}^{(2)}$: $\sum_{i=0}^{n-1} a_i\, 2^i$

Calculs effectués en conservant uniquement les n bits de poids faibles

$\Rightarrow$ calcul modulo 2^n + résultat $\in [\![0,\ 2^n - 1]\!]$

Exemple : $n = 4$

$$\overline{1101}^{(2)} + \overline{0110}^{(2)} = \overline{0011}^{(2)} = 3 \equiv 13 + 6 \mod 16$$

Représentation des entiers signés

Bit de signe + entier non-signé :

- valeurs $\in [\![-2^{n-1}+1,\ 2^{n-1}-1]\!]$
- signes à gérer à part
- $+0$ et -0

$$(-1)^{a_{n-1}} \times \sum_{i=0}^{n-2} a_i\, 2^i$$

Représentation des entiers signés

Bit de signe + entier non-signé :

- valeurs $\in [\![-2^{n-1}+1,\ 2^{n-1}-1]\!]$
- signes à gérer à part
- $+0$ et -0

$$(-1)^{a_{n-1}} \times \sum_{i=0}^{n-2} a_i\, 2^i$$

Complément à 1 **:**

- valeurs $\in [\![-2^{n-1}+1,\ 2^{n-1}-1]\!]$
- deux représentations pour 0

$$\begin{cases} \sum_{i=0}^{n-2} a_i\, 2^i & \text{si } a_{n-1}=0 \\ -\sum_{i=0}^{n-2} \bar{a}_i\, 2^i & \text{si } a_{n-1}=1 \end{cases}$$

Représentation des entiers signés

Bit de signe + entier non-signé :

$$(-1)^{a_{n-1}} \times \sum_{i=0}^{n-2} a_i\, 2^i$$

- valeurs $\in [\![-2^{n-1}+1,\ 2^{n-1}-1]\!]$
- signes à gérer à part
- $+0$ et -0

Complément à 1 **:**

$$\begin{cases} \sum_{i=0}^{n-2} a_i\, 2^i & \text{si } a_{n-1}=0 \\ -\sum_{i=0}^{n-2} \bar{a}_i\, 2^i & \text{si } a_{n-1}=1 \end{cases}$$

- valeurs $\in [\![-2^{n-1}+1,\ 2^{n-1}-1]\!]$
- deux représentations pour 0

Complément à 2 **:**

$$-a_{n-1}\, 2^{n-1} + \sum_{i=0}^{n-2} a_i\, 2^i$$

- valeurs $\in [\![-2^{n-1},\ 2^{n-1}-1]\!]$
- représentation unique

Complément à 2

Valeur associée à $\overline{a_{n-1}a_{n-2}\cdots a_1 a_0}^{(2)}$: $\left(\sum_{i=0}^{n-2} a_i\, 2^i\right) - a_{n-1}\, 2^{n-1}$

- si $a_{n-1} = 0$: même valeur qu'en non-signée
- si $a_{n-1} = 1$: valeur négative = valeur non-signée -2^n

Même règles de calcul qu'en arithmétique non-signée

$\Rightarrow$ calcul modulo 2^n + résultat $\in [\![-2^{n-1},\ 2^{n-1} - 1]\!]$

Exemple : $n = 4$

$$\overline{1101}^{(2)} + \overline{0110}^{(2)} = \overline{0011}^{(2)} = 3 \equiv -3 + 6 \mod 16$$

Complément à 2

Valeur associée à $\overline{a_{n-1}\,a_{n-2}\cdots a_1\,a_0}^{(2)}$: $\left(\sum_{i=0}^{n-2} a_i\,2^i\right) - a_{n-1}\,2^{n-1}$

- si $a_{n-1} = 0$: même valeur qu'en non-signée
- si $a_{n-1} = 1$: valeur négative = valeur non-signée -2^n

Même règles de calcul qu'en arithmétique non-signée

$\Rightarrow$ calcul modulo 2^n + résultat $\in [\![-2^{n-1},\ 2^{n-1}-1]\!]$

$\Rightarrow$ solution privilégiée pour l'arithmétique signée

Arithmétique non-signée et complément à 2

Caractéristiques communes :

- ✓ circuit de taille connue
- ✓ résultat garanti modulo 2^n
- ✗ risque de dépassement de capacité

Complément à 2 :

- ✓ permet de parler d'entiers négatifs
- ✓ mêmes règles / circuits qu'en non-signée
- ✗ parfois peu intuitif

colle : résoudre $x = -x$

Arithmétique non-signée et complément à 2

Caractéristiques communes :

- ✓ circuit de taille connue
- ✓ résultat garanti modulo 2^n
- ✗ risque de dépassement de capacité

Complément à 2 :

- ✓ permet de parler d'entiers négatifs
- ✓ mêmes règles / circuits qu'en non-signée
- ✗ parfois peu intuitif

colle : résoudre $x = -x$

2 arithmétiques = 2 lectures différentes :

- ✗ choix de lecture critique lors des test

$\overline{1101}^{(2)} < \overline{0110}^{(2)}$?

Réponse à la colle

Version info.

Représentation de $-x$?

- passer au complément à 1
- ajouter 1

$$x \rightsquigarrow x_{n-1} \dots x_1 x_0$$

$$\overline{x_{n-1}} \dots \bar{x_1} \bar{x_0}$$

$$-x \rightsquigarrow \overline{x_{n-1}} \dots \bar{x_1} \bar{x_0} + 1$$

Réponse à la colle

Version info.

Représentation de $-x$? $\quad x \rightsquigarrow x_{n-1} \dots x_1 x_0$

- passer au complément à 1 $\quad \overline{x_{n-1}} \dots \bar{x_1} \bar{x_0}$
- ajouter 1 $\quad -x \rightsquigarrow \overline{x_{n-1}} \dots \bar{x_1} \bar{x_0} + 1$

Déduction :

- $x_i \neq \overline{x_i}$, donc $x = -x$ ssi propagation retenue jusqu'à x_{n-1} $\quad \overline{x_i} = 1$
- $x_{n-1} = \overline{x_{n-1}} + 1$ toujours vrai

Réponse à la colle

Version info.

Représentation de $-x$? $\quad x \rightsquigarrow x_{n-1} \dots x_1 x_0$

- passer au complément à 1 $\quad \overline{x_{n-1}} \dots \bar{x}_1 \bar{x}_0$
- ajouter 1 $\quad -x \rightsquigarrow \overline{x_{n-1}} \dots \bar{x}_1 \bar{x}_0 + 1$

Déduction :

- $x_i \neq \overline{x_i}$, donc $x = -x$ ssi propagation retenue jusqu'à x_{n-1} $\quad \overline{x_i} = 1$
- $x_{n-1} = \overline{x_{n-1}} + 1$ toujours vrai
- soit $x_{n-1} = 0$ et $x = 0$, soit $x_{n-1} = 1$ et $x = -2^{n-1}$ sic

Réponse à la colle

Version info.

Représentation de $-x$? $\quad x \rightsquigarrow x_{n-1} \dots x_1 x_0$

- passer au complément à 1 $\quad \overline{x_{n-1}} \dots \bar{x_1} \bar{x_0}$
- ajouter 1 $\quad -x \rightsquigarrow \overline{x_{n-1}} \dots \bar{x_1} \bar{x_0} + 1$

Déduction :

- $x_i \neq \overline{x_i}$, donc $x = -x$ ssi propagation retenue jusqu'à x_{n-1} $\quad \overline{x_i} = 1$
- $x_{n-1} = \overline{x_{n-1}} + 1$ toujours vrai
- soit $x_{n-1} = 0$ et $x = 0$, soit $x_{n-1} = 1$ et $x = -2^{n-1}$ sic

Version math.

$$\begin{aligned} x = -x \quad &\Leftrightarrow 2x = 0 &&\Leftrightarrow \quad 2x \equiv 0 \bmod 2^n \\ &\Leftrightarrow x \equiv 0 \bmod 2^{n-1} &&\Leftrightarrow \quad x = 0 \text{ ou } -2^{n-1} \end{aligned}$$

Plan

Soustracteur n bits

Idée :

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

- remplacer B par son complément à 2
- utiliser un additionneur

Problème :

Soustracteur *n* bits

Idée : $A - B = A + (-B)$

- remplacer B par son complément à 2
- utiliser un additionneur

Problème :

passage au complément à 2 $\Rightarrow$ ajouter 1 coûteux

Soustracteur n bits

Idée : $A - B = A + (-B)$

- remplacer B par son complément à 2
- utiliser un additionneur

Problème :

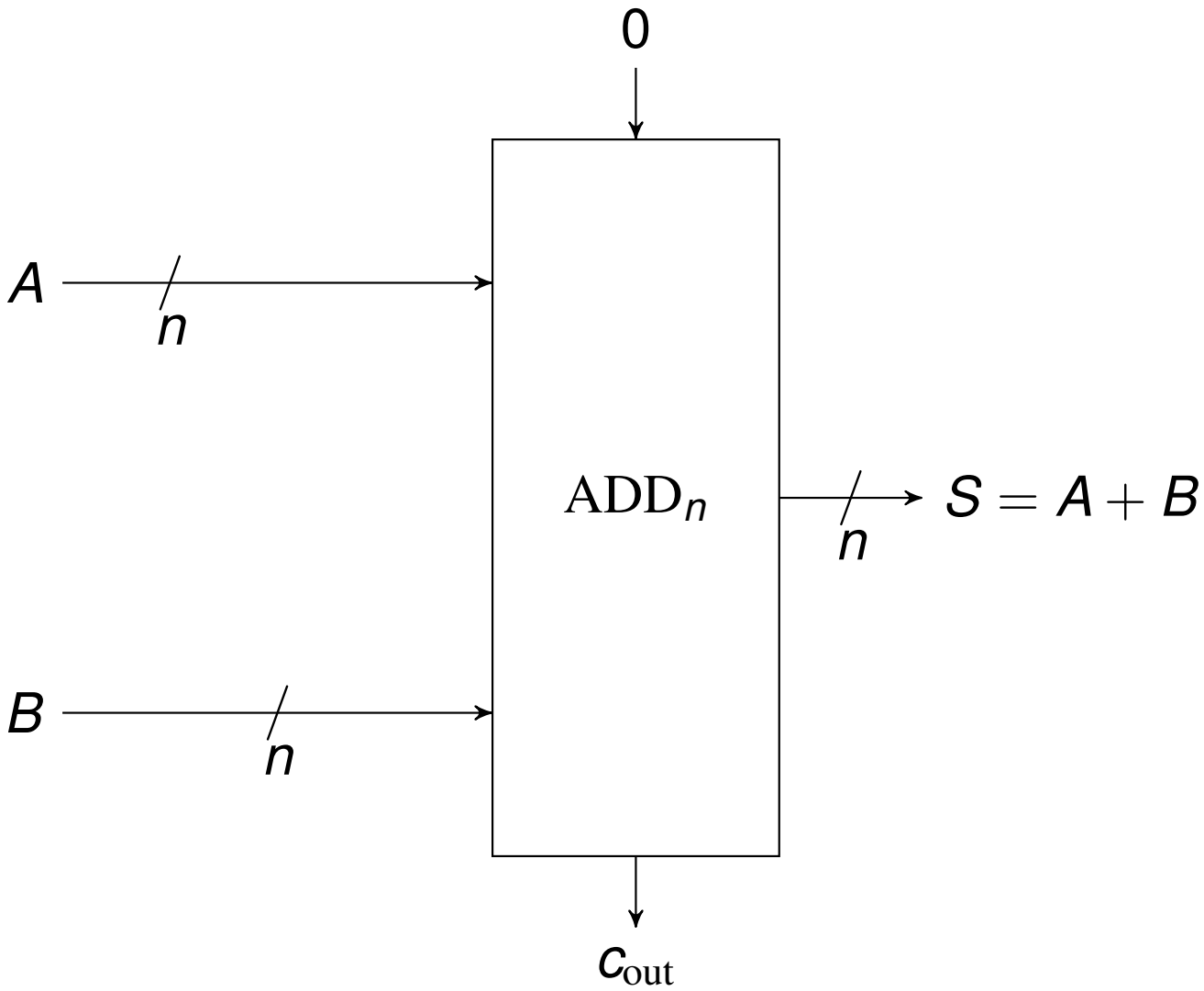
passage au complément à 2 $\Rightarrow$ ajouter 1 coûteux

Solution : $A - B = A + \overline{B} + 1$

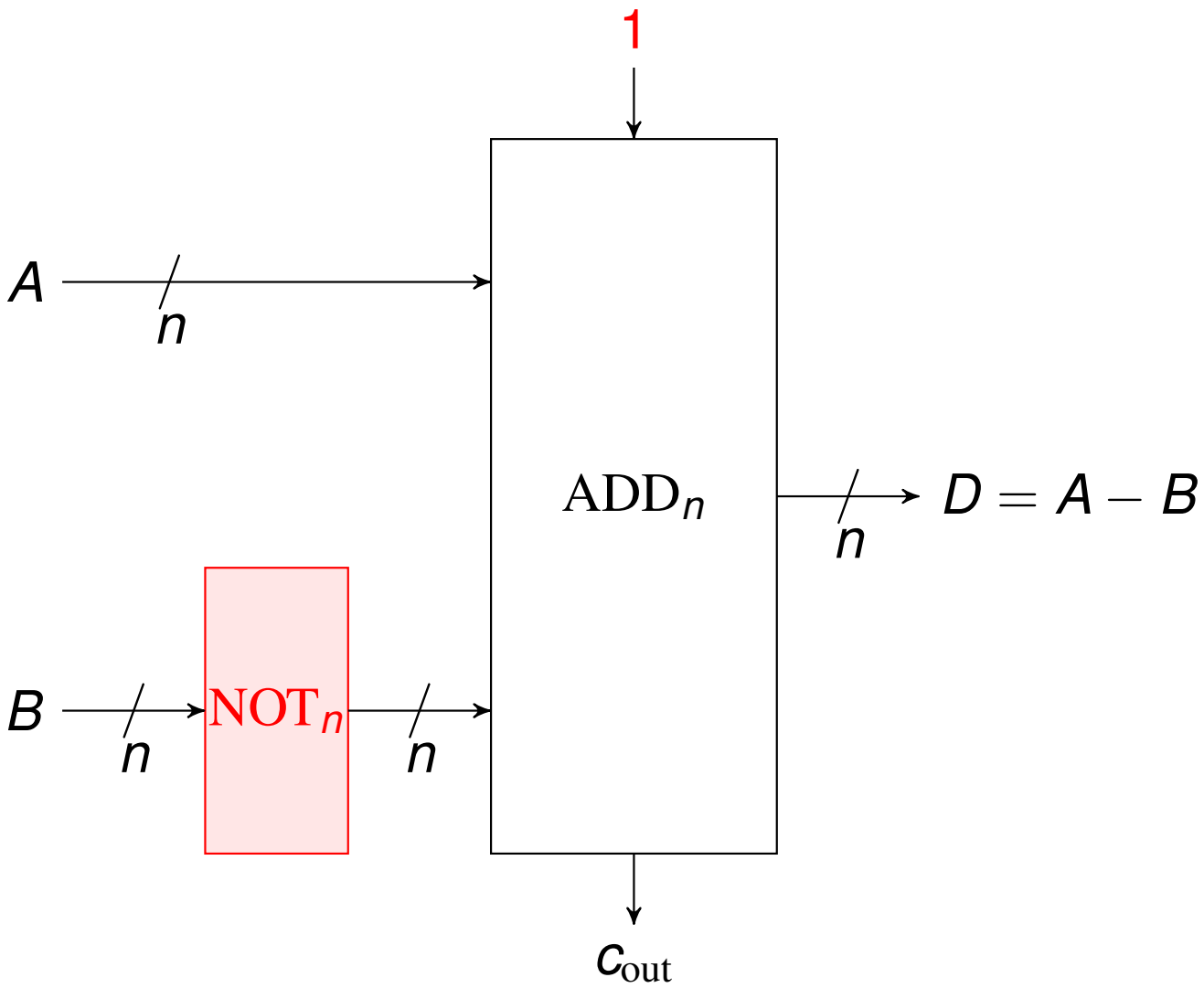
$\overline{B} = (2^n - 1) - B$

- se limiter au complément à 1
- lancer l'addition $A + \overline{B}$ avec $c_{\text{in}} = 1$

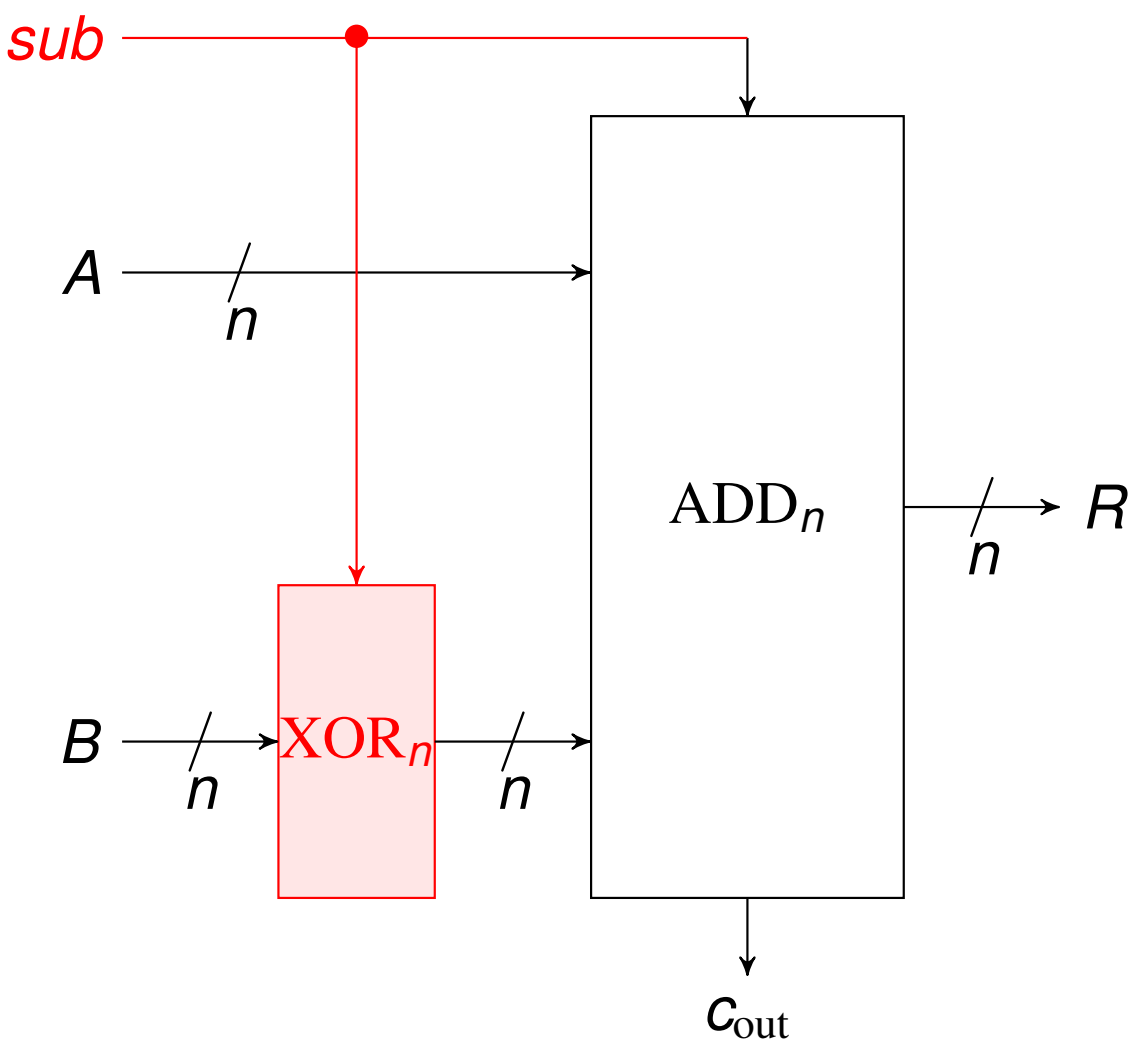
Additionneur n bits



Soustracteur n bits



Additionneur/Soustracteur n bits



Deux en un :

- bit de contrôle `sub`
- `sub` $= 0$
  $\rightsquigarrow R = A + B + 0$
- `sub` $= 1$
  $\rightsquigarrow R = A + \overline{B} + 1$

Bilan sur la soustraction

Bilan :

- soustraction = addition de l'opposé $A - B = A + (-B)$
- complément à 2 = $\mathbb{Z}/2^n\mathbb{Z}$
- réutilisation futée de l'additionneur n bits $A - B = A + \overline{B} + 1$

Améliorations ?

Bilan sur la soustraction

Bilan :

- soustraction = addition de l'opposé $A - B = A + (-B)$
- complément à 2 = $\mathbb{Z}/2^n\mathbb{Z}$
- réutilisation futée de l'additionneur n bits $A - B = A + \overline{B} + 1$

Améliorations ?

- complément à 1 = n `XOR` en parallèle — déjà optimal
- additionneur — cf section précédente

Plan

Support des opérations logiques

Objectif = créer un circuit réalisant le calcul logique

Contraintes :

- calcul bit à bit — calculs « bas niveau », masques
- expressivité — universalité
- profondeur réduite
- surface réduite
- intégration à l'additionneur/soustracteur

Support des opérations logiques (2)

Solution 1 :

- faire un circuit par opération
- utiliser ces circuits en parallèle
- choisir le résultat à l'aide de bits de contrôle

Solution 2 :

- faire un circuit pour `NAND`
- émuler les autres opérations

```
NOT x = x NAND x
x AND y = NOT (x NAND y)
...
```

Support des opérations logiques (2)

Solution 1 :

- faire un circuit par opération
- utiliser ces circuits en parallèle
- choisir le résultat à l'aide de bits de contrôle

✗ surface importante

Solution 2 :

- faire un circuit pour `NAND`
- émuler les autres opérations

```
NOT x = x NAND x
x AND y = NOT (x NAND y)
...
```

✓ surface réduite

Support des opérations logiques (2)

Solution 1 :

- faire un circuit par opération
- utiliser ces circuits en parallèle
- choisir le résultat à l'aide de bits de contrôle

✗ surface importante

✓ profondeur $\Theta(1)$

Solution 2 :

- faire un circuit pour `NAND`
- émuler les autres opérations

```
NOT x = x NAND x
x AND y = NOT (x NAND y)
...
```

✓ surface réduite

✓ profondeur 1

Support des opérations logiques (2)

Solution 1 :

- faire un circuit par opération
- utiliser ces circuits en parallèle
- choisir le résultat à l'aide de bits de contrôle

✗ surface importante
✓ profondeur $\Theta(1)$
✓ bon support matériel

Solution 2 :

- faire un circuit pour `NAND`
- émuler les autres opérations

```
NOT x = x NAND x
x AND y = NOT (x NAND y)
...
```

✓ surface réduite
✓ profondeur 1
✗ émulation coûteuse

Intégration à l'additionneur/soustracteur

Complément à 1 = `NOT` bit à bit

- `NOT` déjà disponible pour B

Calcul de $A \wedge B$ =

Intégration à l'additionneur/soustracteur

Complément à 1 = `NOT` bit à bit

- `NOT` déjà disponible pour B

Calcul de $A \wedge B$ = `AND` bit à bit en // du ADD_n + multiplexeur

Idée = Ajout d'un complément à 1 pour A et en sortie :

- ✓ autres calculs arith. $B - A$, $-A - B$
- ✓ limite les circuits logiques $A \vee B \equiv \overline{\overline{A} \wedge \overline{B}}$
- ✓ coût négligeable
- ✗ gestion des bits de contrôle un peu plus complexe

cf TP

Plan

Drapeaux de l'ALU

Drapeau = bit d'information complémentaire

2 drapeaux indiquant la nature du résultat :

- *Z* (Zero) 1 ssi résultat nul
- *S* (Sign) 1 ssi résultat négatif en arith. signée

2 drapeaux pour signaler les dépassements de capacité :

- *C* (Carry) 1 ssi dépassement en arith. non-signée
- *O* (Overflow) 1 ssi dépassement en arith. signée

Valeur pour C : cas de l'addition

Propriété

Lors du calcul de $a + b$ en arithmétique non-signée, on a dépassement de capacité ssi la retenue sortante de l'additionneur est 1.

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a + b \in ?$

1. $[\![0,\ 2^n - 1]\!] \rightsquigarrow$ ni retenue, ni dépassement
2. $[\![2^n,\ 2^{n+1} - 2]\!] \rightsquigarrow$ retenue et dépassement

Valeur pour C : cas de la soustraction

Propriété

Lors du calcul de $a - b$ en arithmétique non-signée, on a dépassement de capacité ssi ?

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a - b \in$?

Valeur pour C : cas de la soustraction

Propriété

Lors du calcul de $a - b$ en arithmétique non-signée, on a dépassement de capacité ssi ?

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a - b \in$?

1. $[\![0,\ 2^n - 1]\!]$ pas de dépassement

Valeur pour C : cas de la soustraction

Propriété

Lors du calcul de $a - b$ en arithmétique non-signée, on a dépassement de capacité ssi ?

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a - b \in$?

1. $[\![0,\ 2^n - 1]\!]$ pas de dépassement
   $\Rightarrow a \geq b \quad \Rightarrow \overline{b} = 2^n - 1 - b \geq 2^n - 1 - a$

Valeur pour C : cas de la soustraction

Propriété

Lors du calcul de $a - b$ en arithmétique non-signée, on a dépassement de capacité ssi ?

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a - b \in$?

1. $[\![0,\ 2^n - 1]\!]$ pas de dépassement
   $\Rightarrow a \geq b \quad \Rightarrow \overline{b} = 2^n - 1 - b \geq 2^n - 1 - a$
   $\Rightarrow$ l'additionneur calcule $a + \overline{b} + 1 \geq 2^n$
   $\Rightarrow$ retenue, pas de dépassement de capacité

Valeur pour C : cas de la soustraction

Propriété

Lors du calcul de $a - b$ en arithmétique non-signée, on a dépassement de capacité ssi ?

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a - b \in$?

1. $[\![0,\ 2^n - 1]\!]$ pas de dépassement
   $\Rightarrow a \geq b \quad \Rightarrow \overline{b} = 2^n - 1 - b \geq 2^n - 1 - a$
   $\Rightarrow$ l'additionneur calcule $a + \overline{b} + 1 \geq 2^n$
   $\Rightarrow$ retenue, pas de dépassement de capacité

2. $[\![1 - 2^n,\ -1]\!]$ dépassement

Valeur pour C : cas de la soustraction

Propriété

Lors du calcul de $a - b$ en arithmétique non-signée, on a dépassement de capacité ssi ?

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a - b \in$?

1. $[\![0,\ 2^n - 1]\!]$ pas de dépassement
   $\Rightarrow a \geq b \quad \Rightarrow \overline{b} = 2^n - 1 - b \geq 2^n - 1 - a$
   $\Rightarrow$ l'additionneur calcule $a + \overline{b} + 1 \geq 2^n$
   $\Rightarrow$ retenue, pas de dépassement de capacité

2. $[\![1 - 2^n,\ -1]\!]$ dépassement
   $\Rightarrow \overline{b} = 2^n - 1 - b < 2^n - 1 - a$

Valeur pour C : cas de la soustraction

Propriété

Lors du calcul de $a - b$ en arithmétique non-signée, on a dépassement de capacité ssi la retenue sortante de l'additionneur est 0.

Preuve : Soient $a, b \in [\![0,\ 2^n - 1]\!]$.

$a - b \in ?$

1. $[\![0,\ 2^n - 1]\!]$ pas de dépassement
   $\Rightarrow a \geq b \quad \Rightarrow \overline{b} = 2^n - 1 - b \geq 2^n - 1 - a$
   $\Rightarrow$ l'additionneur calcule $a + \overline{b} + 1 \geq 2^n$
   $\Rightarrow$ retenue, pas de dépassement de capacité

2. $[\![1 - 2^n,\ -1]\!]$ dépassement
   $\Rightarrow \overline{b} = 2^n - 1 - b < 2^n - 1 - a$
   $\Rightarrow$ l'additionneur calcule $a + \overline{b} + 1 \in [\![1,\ 2^n - 1]\!]$
   $\Rightarrow$ pas de retenue, dépassement de capacité

Valeur pour O : cas de l'addition

Soient $a, b \in [\![-2^{n-1},\ 2^{n-1}-1]\!]$.

4 cas :

1. $a \geq 0,\ b < 0$
   $\Rightarrow -2^{n-1} \leq b \leq a+b < a \leq 2^{n-1}-1$
   $\Rightarrow$ dépassement de capacité impossible
   $\Rightarrow O = 0$

2. $a < 0,\ b \geq 0$

Valeur pour O : cas de l'addition

Soient $a, b \in [\![-2^{n-1},\ 2^{n-1}-1]\!]$.

4 cas :

1. $a \geq 0,\ b < 0$
   $\Rightarrow -2^{n-1} \leq b \leq a+b < a \leq 2^{n-1}-1$
   $\Rightarrow$ dépassement de capacité impossible
   $\Rightarrow O = 0$

2. $a < 0,\ b \geq 0$
   $\Rightarrow -2^{n-1} \leq a \leq a+b < b \leq 2^{n-1}-1$
   $\Rightarrow$ dépassement de capacité impossible
   $\Rightarrow O = 0$

Valeur pour O : cas de l'addition (suite)

3. $a \geq 0$, $b \geq 0$
   $\Rightarrow 0 \leq a + b \leq 2^n - 2$
   $\Rightarrow$ dépassement ssi $a + b \in [\![2^{n-1},\ 2^n - 2]\!]$

Valeur pour O : cas de l'addition (suite)

3. $a \geq 0$, $b \geq 0$
   $\Rightarrow 0 \leq a + b \leq 2^n - 2$
   $\Rightarrow$ dépassement ssi $a + b \in [\![2^{n-1},\ 2^n - 2]\!]$
   $\Rightarrow$ dépassement ssi résultat de l'ALU = $a + b - 2^n < 0$
   $\Rightarrow O = 1$ ssi $S = 1$

Valeur pour O : cas de l'addition (suite)

3. $a \geq 0$, $b \geq 0$
   $\Rightarrow 0 \leq a + b \leq 2^n - 2$
   $\Rightarrow$ dépassement ssi $a + b \in [\![2^{n-1},\ 2^n - 2]\!]$
   $\Rightarrow$ dépassement ssi résultat de l'ALU = $a + b - 2^n < 0$
   $\Rightarrow O = 1$ ssi $S = 1$

4. $a < 0$, $b < 0$

Valeur pour O : cas de l'addition (suite)

3. $a \geq 0, b \geq 0$
   $\Rightarrow 0 \leq a + b \leq 2^n - 2$
   $\Rightarrow$ dépassement ssi $a + b \in [\![2^{n-1},\ 2^n - 2]\!]$
   $\Rightarrow$ dépassement ssi résultat de l'ALU = $a + b - 2^n < 0$
   $\Rightarrow O = 1$ ssi $S = 1$

4. $a < 0, b < 0$
   $\Rightarrow -2^n \leq a + b \leq -2$
   $\Rightarrow$ dépassement ssi $a + b \in [\![-2^n,\ -2^{n-1} - 1]\!]$
   $\Rightarrow$ dépassement ssi résultat de l'ALU = $a + b + 2^n \geq 0$
   $\Rightarrow O = 1$ ssi $S = 0$

Valeur pour O : cas général

Propriété

Lors du calcul de $a + b$ en arithmétique signée (complément à 2), on a dépassement de capacité si et seulement si :

- a et b sont de même signe,
- le résultat de l'ALU a le signe opposé de celui de a et b.

Propriété

Lors du calcul de $a - b$ en arithmétique signée (complément à 2), on a dépassement de capacité si et seulement si :

- a et b sont de signes opposés,
- le résultat de l'ALU a le même signe que b.

Drapeaux – Choses à retenir

Intérêt des drapeaux de l'ALU :

- signaler un résultat remarquable
- signaler un problème dépassement capacité

$\rightsquigarrow$ information nécessaire et suffisante pour comparer des entiers

Drapeaux – Choses à retenir

Intérêt des drapeaux de l'ALU :

- signaler un résultat remarquable
- signaler un problème — dépassement capacité

$\rightsquigarrow$ information nécessaire et suffisante pour comparer des entiers

Complément à 2 = structure algébrique simple

- calculs des drapeaux simples $\Rightarrow$ circuits petits/rapides
- correction prouvée

évite les erreurs