Auto1 SIN

—

Électronique numérique

Brice Colombier, Florent Bernard

BUT GEII
Saint-Étienne

2025 – 2026

Inspiré de :

- *"SIN1 - Systèmes d'Information Numériques"* - Viktor Fischer, Florent Bernard
- *"Digital Design and Computer Architecture"* - David Money Harris, Sarah L. Harris

Chapitre 1

—

Introduction

Organisation du cours

Intervenants :

- Brice Colombier b.colombier@univ-st-etienne.fr
- Florent Bernard florent.bernard@univ-st-etienne.fr

Séances :

- 28h de cours/TD ($18 \times 1\text{h}30 + 1\text{h}$)
- 9h de TP ($3 \times 3\text{h}$)

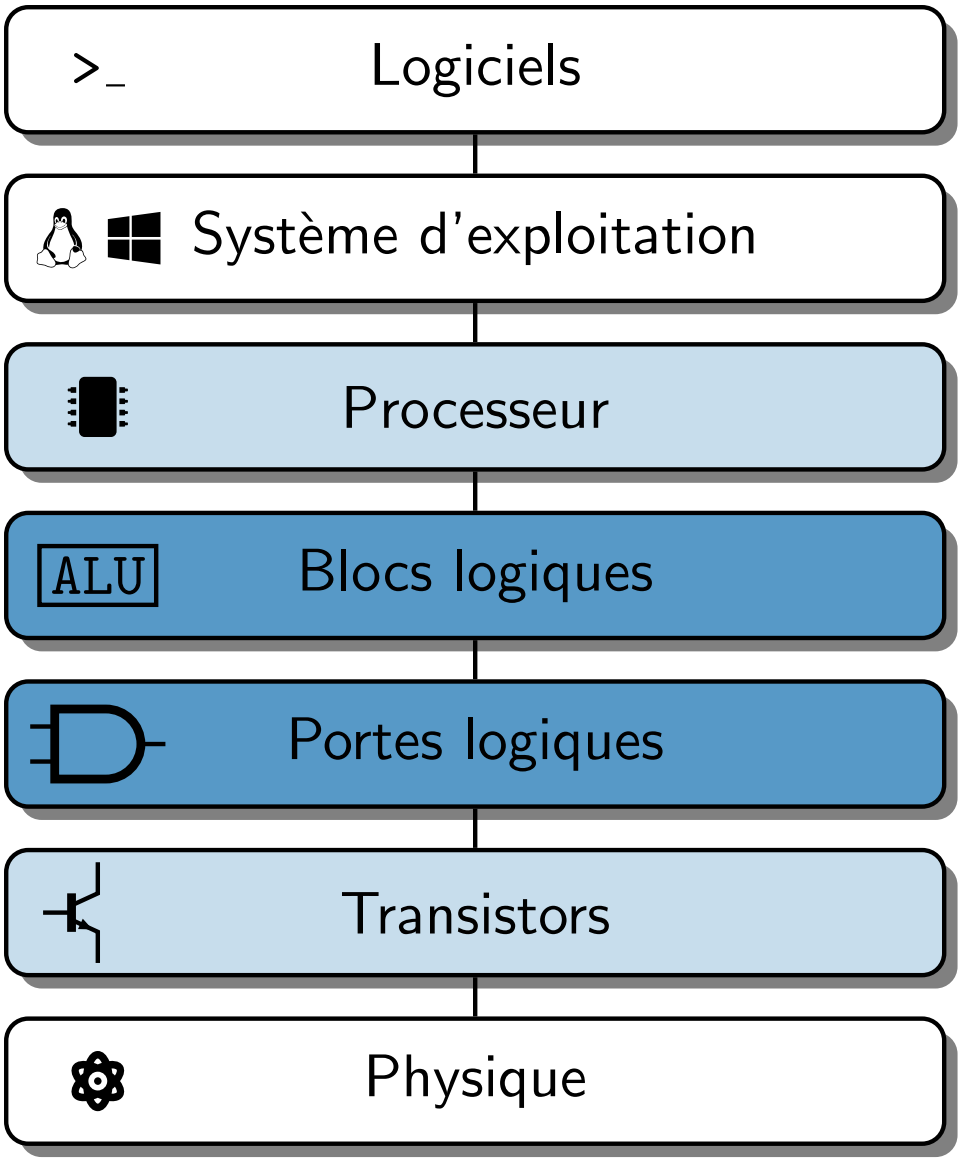
Évaluation :

- QCM
- contrôle à mi-parcours
- examen final

Électronique numérique : Qu'est-ce que c'est ?

Électronique numérique

Niveaux d'abstraction d'un système électronique numérique



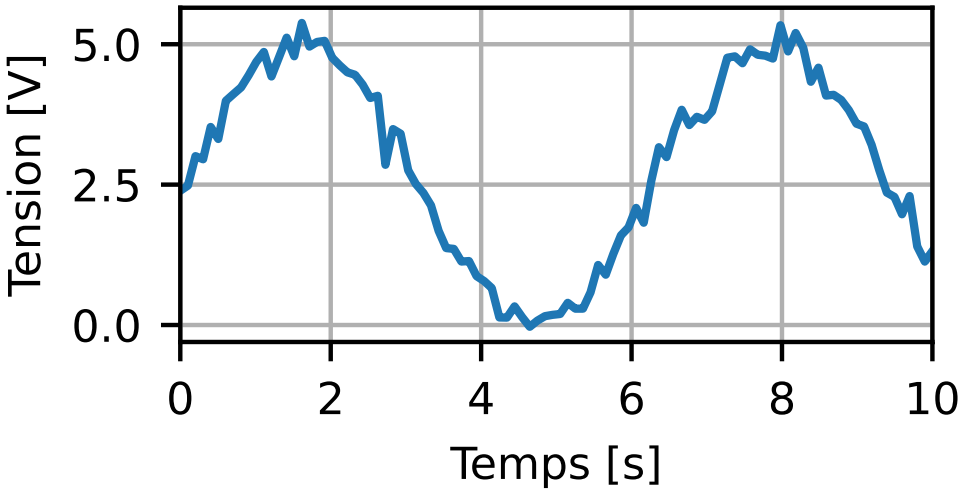
Informatique (Info)

Électronique analogique (Elen)

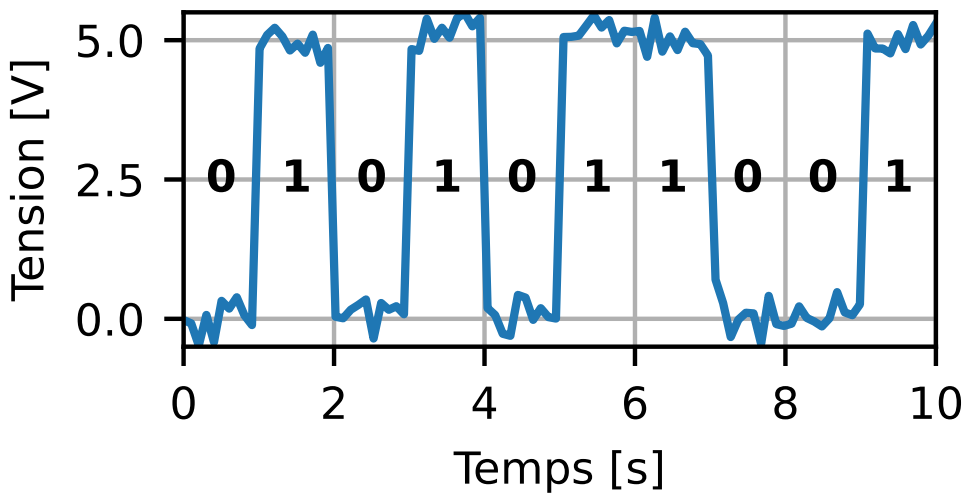
Électronique numérique ?

Définition :
Utilisation de **signaux électriques** pour **manipuler** de **l'information**.

Électronique **analogique**



Électronique **numérique**



- Signaux **continus**
  - capteurs
  - actionneurs
- Signaux (réputés) **discrets**
  - ordinateurs
  - systèmes embarqués

Approche pédagogique suivie dans ce cours

Approche **ascendante** :

Étudier et **comprendre** les blocs **élémentaires**, puis les **assembler** pour créer des blocs **plus complexes**.

- Progression par identification des **limites** de ce qu'on sait déjà,
- Points réguliers pour faire le point sur les **méthodes apprises**,
- Exemples **"fil rouge"** qui nous suivront pendant plusieurs séances.

Le fascicule est votre **outil de travail** en séance de cours/TD.

Attention

Toujours avoir son fascicule lors des séances !

Objectifs du cours

Compétences :

- À partir d'un **cahier des charges**, concevoir un système électronique manipulant de l'information numérique,
- Lire un **schéma électronique** comportant des éléments **logiques**,
- **Vérifier** qu'un système numérique fonctionne **selon les spécifications**,
- Programmer un **FPGA** à l'aide d'un logiciel de **synthèse** d'une **description matérielle**.

Chapitre 2

—

Logique booléenne

Information et logique

L'information la plus élémentaire à propos d'une proposition est sa **véracité**. Ces propositions peuvent prendre **deux, et seulement deux** valeurs :

- `VRAI`
- `FAUX`

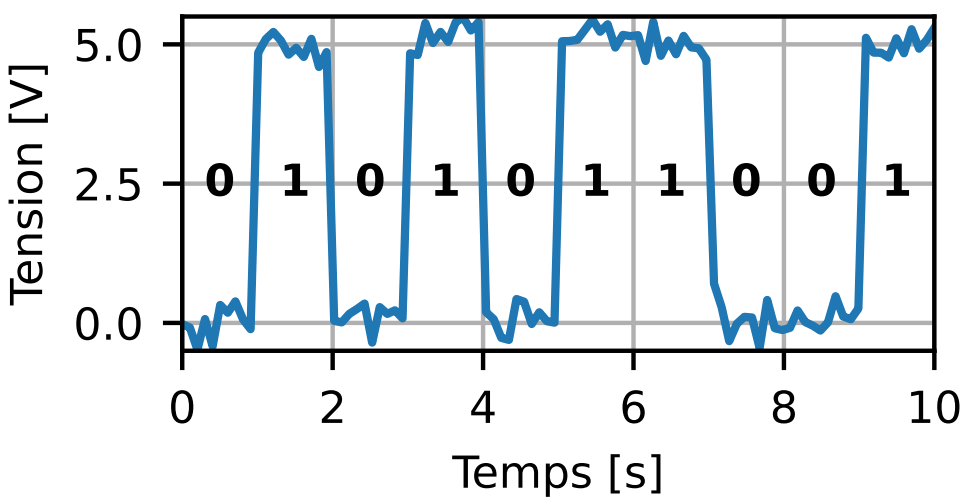
Exemples : la porte est fermée, la lampe est allumée, il fait jour, etc.

On encode cette information sur un **bit**, qui peut prendre deux valeurs :

- 0
- 1

En pratique, on compare la tension à un niveau de référence :

- Si $V < \frac{V_{dd}}{2}$: 0
- Si $V > \frac{V_{dd}}{2}$: 1



L'identité

Définition : l'information déduite est **identique** à l'information initiale.

Exemples :

- la lampe est allumée $\rightarrow$ la lampe est allumée
- l'interrupteur est fermé $\rightarrow$ l'interrupteur est fermé

Équation logique

$s =$

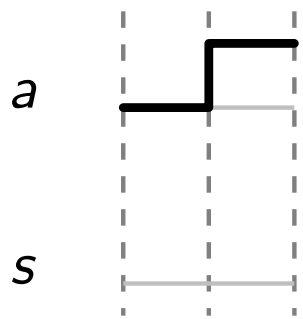
Table de vérité

a	$s =$
0	
1	

Symbole

a

Chronogramme



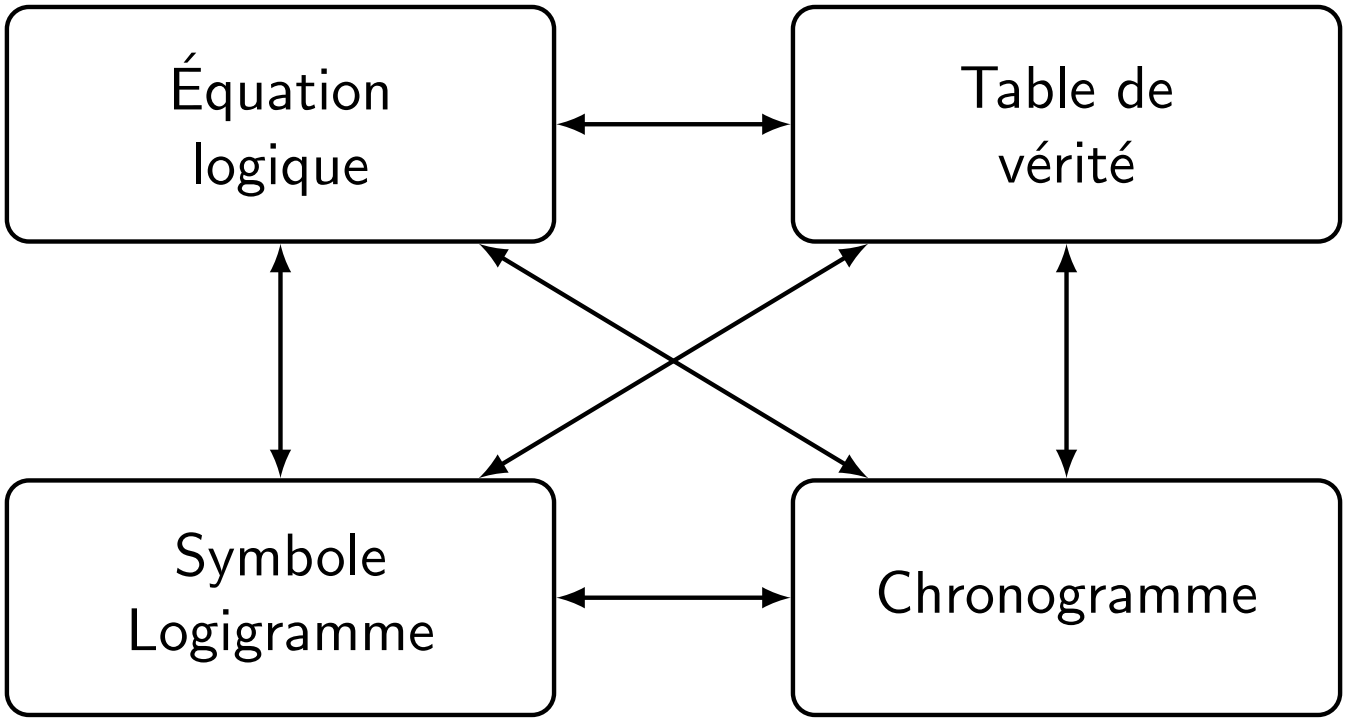
Méthode : Lecture d'équations logiques

$$s = a$$

s vaut 1 lorsque a vaut 1

Quatre représentations d'une fonction logique

Il existe **quatre** façons **équivalentes** de **décrire** une fonction logique :



L'usage de l'une ou l'autre dépendra du **contexte**.

Il faut **maîtriser** les 6 "conversions" possibles.

Négation (NOT)

Définition : l'information déduite est **le contraire** de l'information initiale.

Exemples :

- la lampe est éteinte → la lampe est allumée
- l'interrupteur est fermé → l'interrupteur est ouvert

Équation logique

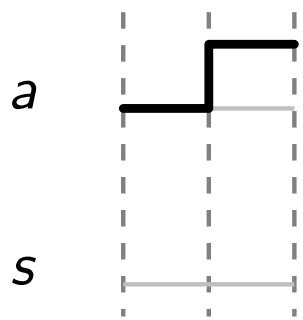
$s =$

Table de vérité

a	$s =$
0	
1	

Symbole

Chronogramme



Méthode : Lecture d'équations logiques

$$s = a$$

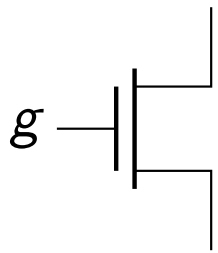
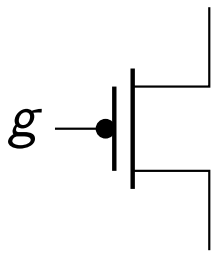
s vaut 1 lorsque a vaut 1

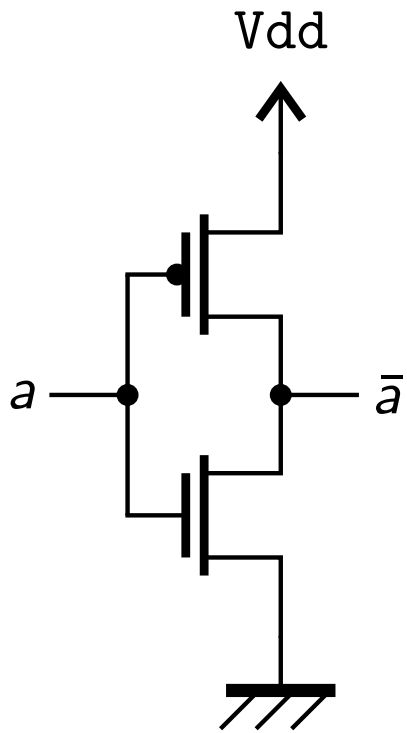
$$s = \bar{a}$$

s vaut 1 lorsque a vaut 0

Inverseur au niveau transistors

Voici “l’intérieur” d’un inverseur, composé de 2 transistors :

	nmos	pmos
		
$g = 0$		
$g = 1$		



Un transistor est constitué de matériaux **semi-conducteurs**.

Limite

Pour l'instant, on travaille avec **une seule** information.
Parfois on l'inverse, parfois non...

Peut-on "combiner" deux informations
pour en déduire une **nouvelle** information ?

Conjonction (AND)

Définition : l'information déduite est vraie **si et seulement si les deux** informations initiales sont vraies.

Exemples :

- Le match aura lieu si l'équipe A est présente **ET** l'équipe B est présente
- Je peux être alternant au S5 si j'ai de bonnes notes au S4 **ET** j'ai une entreprise
- Le moteur démarre si le carter est fermé **ET** l'interrupteur est fermé

Équation logique

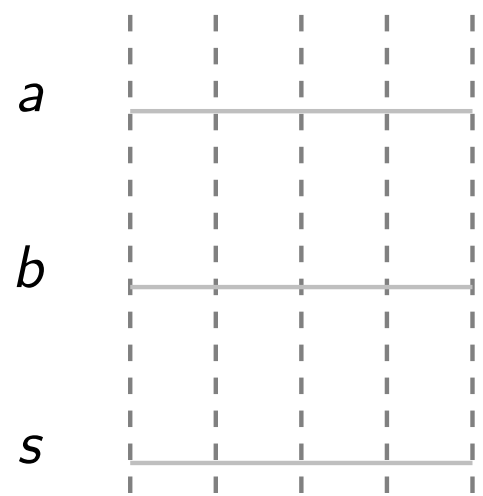
$s =$

Table de vérité

a	b	$s =$
0	0	
0	1	
1	0	
1	1	

Symbole

Chronogramme



Méthode : Lecture d'équations logiques

$$s = a \cdot b$$

s vaut 1 lorsque a vaut 1 ET b vaut 1

Disjonction (OU)

Définition : l'information déduite est vraie **si au moins une des deux** informations initiales est vraie.

Exemples :

- le match sera annulé si le vent dépasse 100 km/h **OU** il neige
- je suis inscrit en S5 si je suis alternant en S5 **OU** je suis étudiant en S5
- le moteur doit s'arrêter si le carter est ouvert **OU** l'interrupteur est ouvert

Équation logique

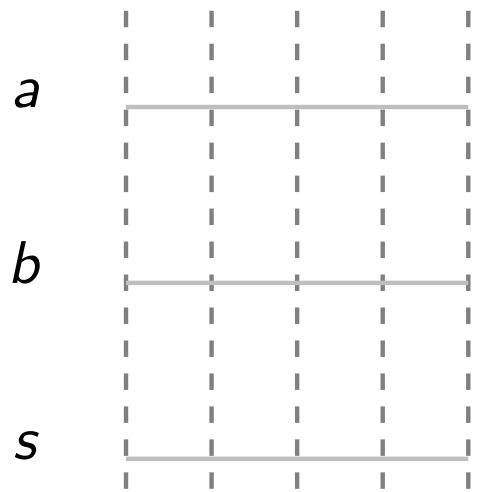
$s =$

Table de vérité

a	b	$s =$
0	0	
0	1	
1	0	
1	1	

Symbole

Chronogramme



Méthode : Lecture d'équations logiques

$$s = a \cdot b$$

s vaut 1 lorsque a vaut 1 ET b vaut 1

$$s = a \vee b$$

s vaut 1 lorsque a vaut 1 OU b vaut 1

Disjonction exclusive (OU exclusif)

Définition : l'information déduite est vraie si **une et seulement une** des deux informations est vraie.

Exemple :

- une équipe vient de marquer si l'équipe A vient de marquer **OU** l'équipe B vient de marquer (impossible de marquer simultanément)
- interrupteurs **va-et-vient** dans une pièce

Équation logique

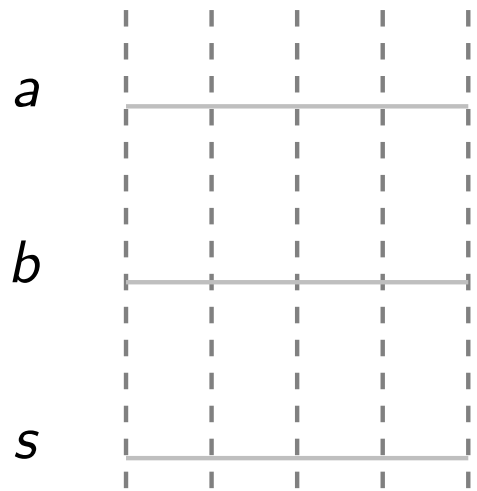
$s =$

Table de vérité

a	b	$s =$
0	0	
0	1	
1	0	
1	1	

Symbole

Chronogramme



Portes logiques à 3 entrées et plus

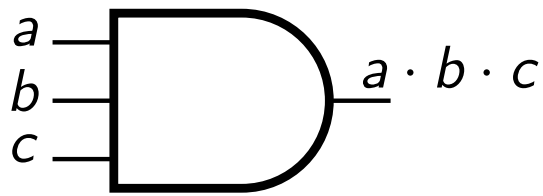
Équation logique

$$s = a \cdot b \cdot c$$

Table de vérité

a	b	c	$s = a \cdot b \cdot c$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Symbole



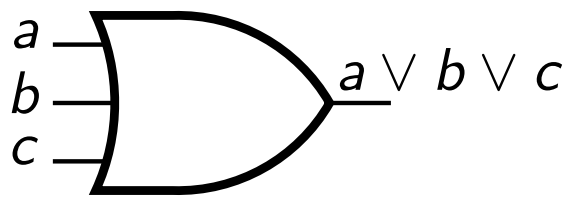
Équation logique

$$s = a \vee b \vee c$$

Table de vérité

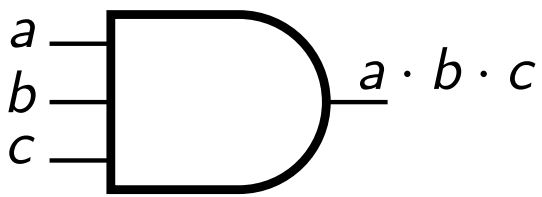
a	b	c	$s = a \vee b \vee c$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

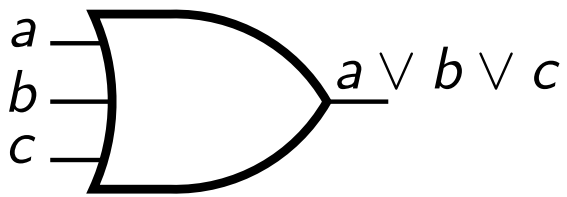
Symbole



Exercice : Portes logiques à 3 entrées et plus

Exercice : avec seulement des portes à 2 entrées, réaliser une porte ET et une porte OU à 3 entrées.





Fil rouge : Cahier des charges

Objectif : concevoir un boîtier de vote pour une association, utilisé lors de l'assemblée générale pour un vote à la majorité.

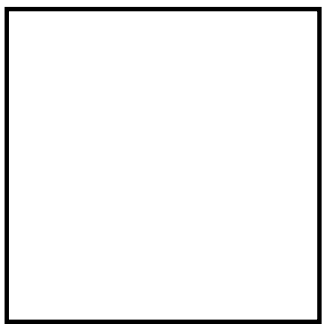
Cahier des charges :

Les membres du bureau votent :

- la présidente p,
- le secrétaire s,
- le trésorier t.

La sortie m commande une LED, qui s'allume si la majorité est atteinte.

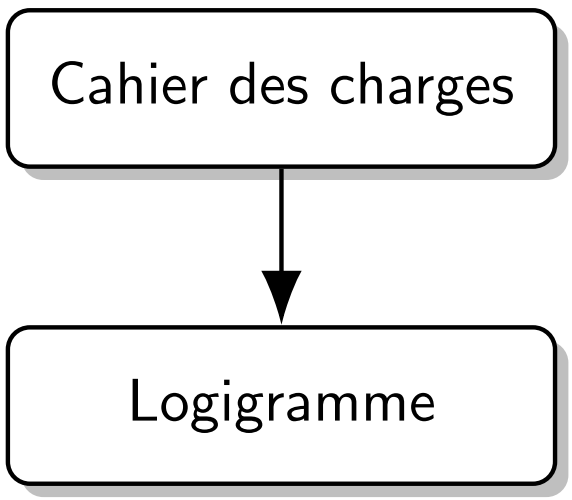
Diagramme :



Dessiner le logigramme générant la sortie `maj` à partir des entrées p, s et t. On considérera :

- Vote POUR : 1 (donc vote CONTRE : 0),
- Majorité atteinte : 1 (donc majorité non atteinte : 0).

Méthode : Conception numérique



Décrire en français de ce que fait le système.

Placer les portes logiques et les connecter.

Limite

Il est difficile de déduire **directement** le logigramme du cahier des charges.

Fil rouge : CdC → Table de vérité → Équation logique

La majorité est atteinte et la LED s'allume ($m = 1$) si :

Table de vérité

p	s	t	m
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

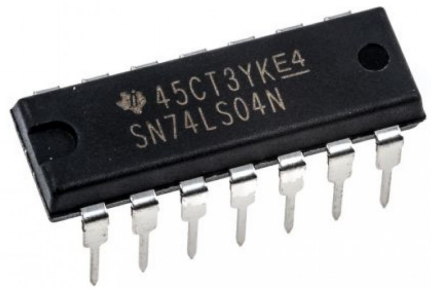
Équation logique finale :

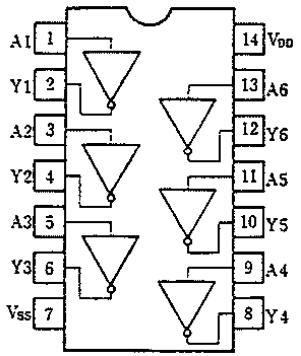
$m =$

Fil rouge : De l'équation logique au logigramme

$m =$

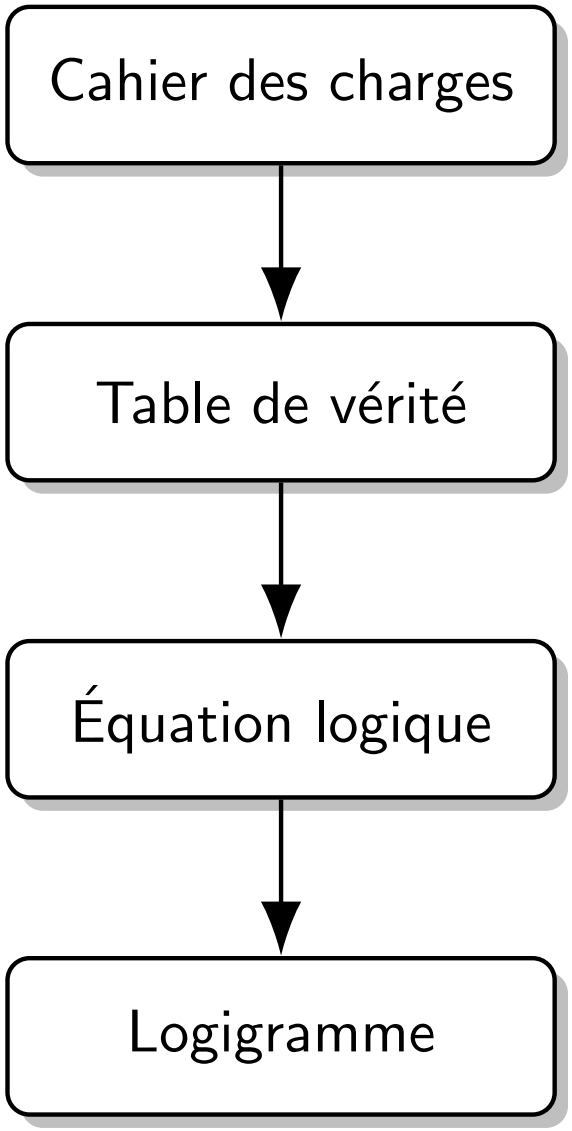
Coût en ressources logiques[1] :





[1] https://fr.wikipedia.org/wiki/Liste_des_circuits_int%C3%A9gr%C3%A9s_de_la_s%C3%A9rie_7400

Méthode : Conception numérique



Décrire en français de ce que fait le système.

Donner la valeur des sorties pour
toutes les valeurs possibles des entrées.

Écrire l'équation comme une somme de produits,
c'est à dire un OU de plusieurs ET.

Placer les portes logiques et les connecter.

Limite

Peut-on faire la **même chose** pour un **coût moindre** ?

Propriétés

AND

a	b	$s = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	$s = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Commutativité

- $a \cdot b = b \cdot a$
- $a \vee b = b \vee a$

a	b	$s = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$s = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Idempotence

- $a \cdot a = a$
- $a \vee a = a$

Propriétés – suite

Associativité

- $a \cdot b \cdot c = (a \cdot b) \cdot c$
- $a \vee b \vee c = (a \vee b) \vee c$

Distributivité

- $(a \cdot b) \vee c = (a \vee c) \cdot (b \vee c)$
- $(a \vee b) \cdot c = (a \cdot c) \vee (b \cdot c)$

Double-complémentation

- $\bar{\bar{a}} = a$

Fil rouge : Équation avec des fonctions à 2 entrées

$$
\begin{aligned}
m &= \bar{p} \cdot s \cdot t \vee p \cdot \bar{s} \cdot t \vee p \cdot s \cdot \bar{t} \vee p \cdot s \cdot t \\
&= \\
&=
\end{aligned}
$$

Fil rouge : Logigramme avec des fonctions à 2 entrées

$m =$

Coût :

Limite

Est-il possible de réduire encore le nombre de portes logiques nécessaires ?

En particulier, peut-on **simplifier** le circuit ?

Propriétés – suite

AND

a	b	$s = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	$s = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$s = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$s = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$s = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$s = a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Éléments neutres

- $a \cdot 1 = a$
- $a \vee 0 = a$

Éléments absorbants

- $a \cdot 0 = 0$
- $a \vee 1 = 1$

Complémentation

- $a \cdot \bar{a} = 0$
- $a \vee \bar{a} = 1$

Propriétés – suite

Absorption d'un terme

- $a \vee \bar{a} \cdot b = a \vee b$
  $= (a \vee \bar{a}) \cdot (a \vee b)$
  $= 1 \cdot (a \vee b)$
- $a \cdot (\bar{a} \vee b) = a \cdot b$
  $= a \cdot \bar{a} \vee a \cdot b$
  $= 0 \vee a \cdot b$

a	b	$\bar{a} \cdot b$	$a \vee \bar{a} \cdot b$	$(\bar{a} \vee b)$	$a \cdot (\bar{a} \vee b)$
0	0	0	0	1	0
0	1	1	1	1	0
1	0	0	1	0	0
1	1	0	1	1	1

Absorption d'un terme

- $a \vee a \cdot b = a$
  $= a \cdot 1 \vee a \cdot b$
  $= a \cdot (1 \vee b)$
- $a \cdot (a \vee b) = a$
  $= (a \vee 0) \cdot (a \vee b)$
  $= a \vee 0 \cdot b$

a	b	$a \cdot b$	$a \vee a \cdot b$	$(a \vee b)$	$a \cdot (a \vee b)$
0	0	0	0	0	0
0	1	0	0	1	0
1	0	0	1	1	1
1	1	1	1	1	1

Fil rouge : Simplification de l'équation logique

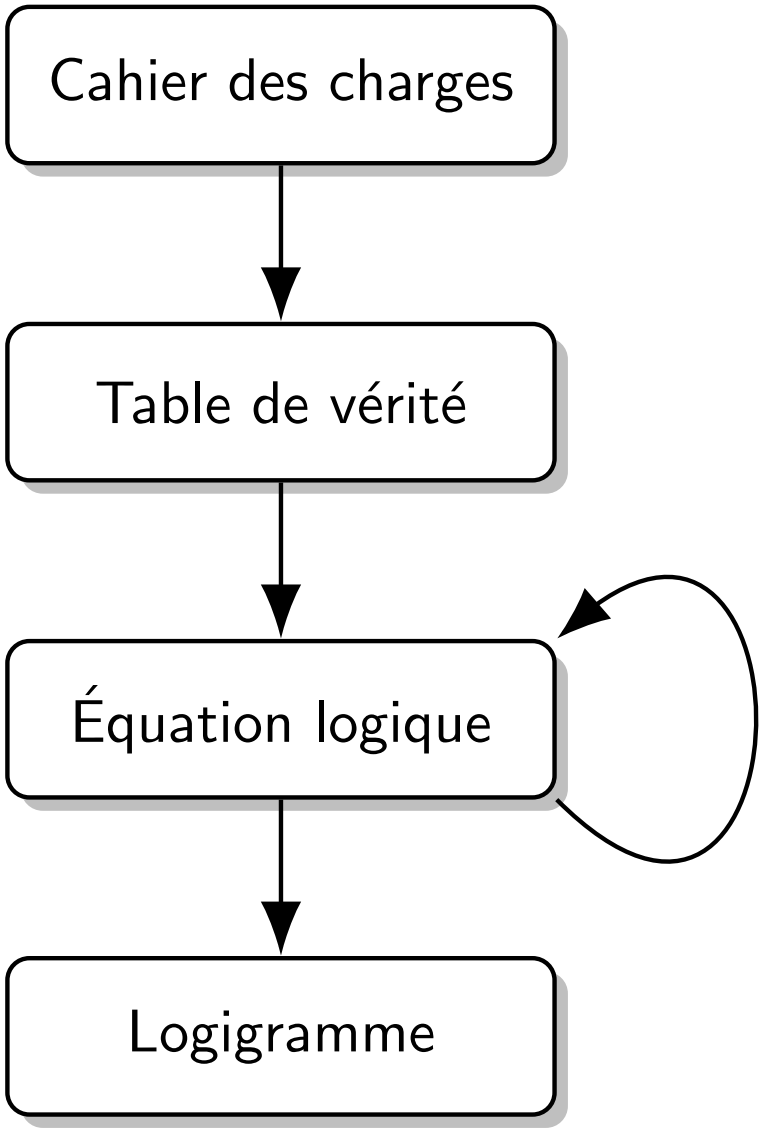
$$m = \bar{p} \cdot s \cdot t \vee p \cdot \bar{s} \cdot t \vee p \cdot s \cdot \bar{t} \vee p \cdot s \cdot t$$

Fil rouge : Logigramme simplifié

$m =$

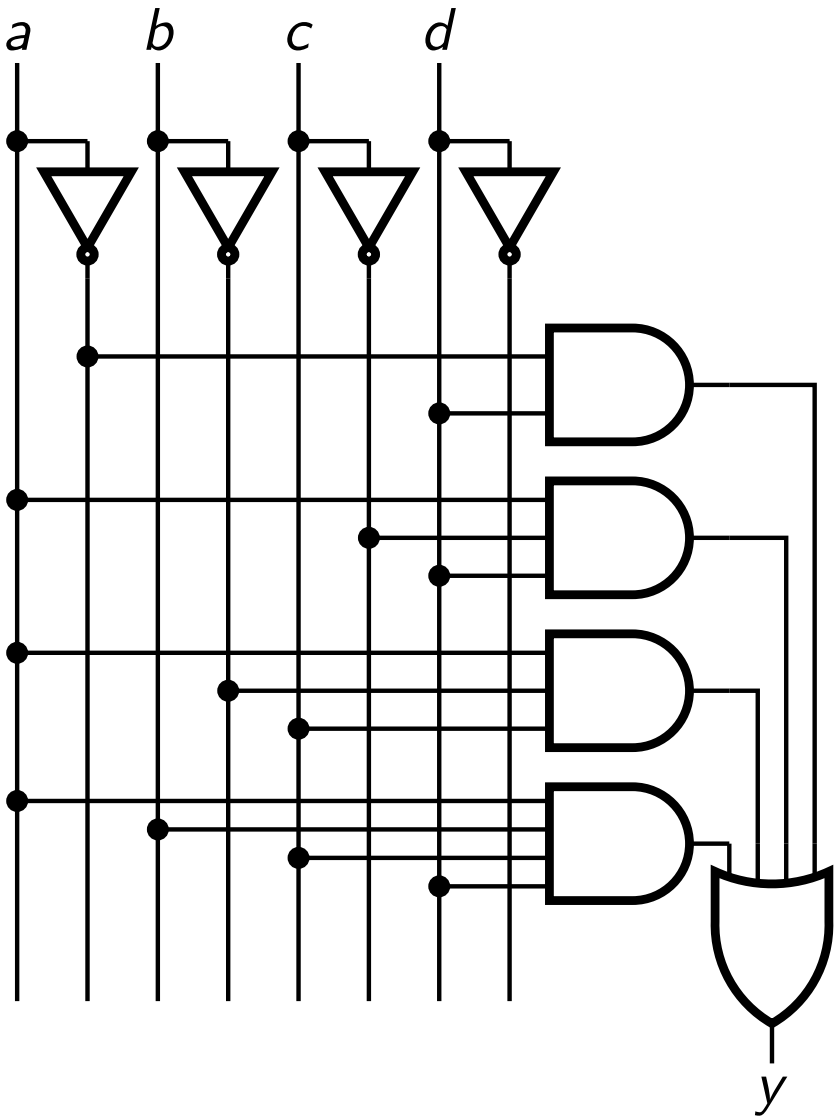
Coût en ressources logiques :

Méthode : Conception numérique



Simplification grâce aux règles de l'algèbre de Boole

Exercice : Simplification de logigramme



$y =$

Versions inversées des opérateurs binaires

NAND

Équation logique

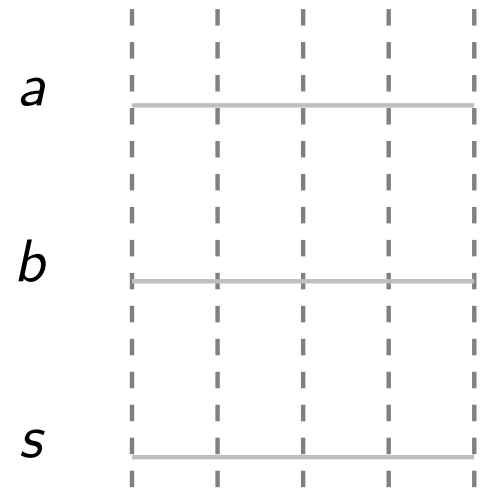
$s =$

Table de vérité

a	b	$s =$
0	0	
0	1	
1	0	
1	1	

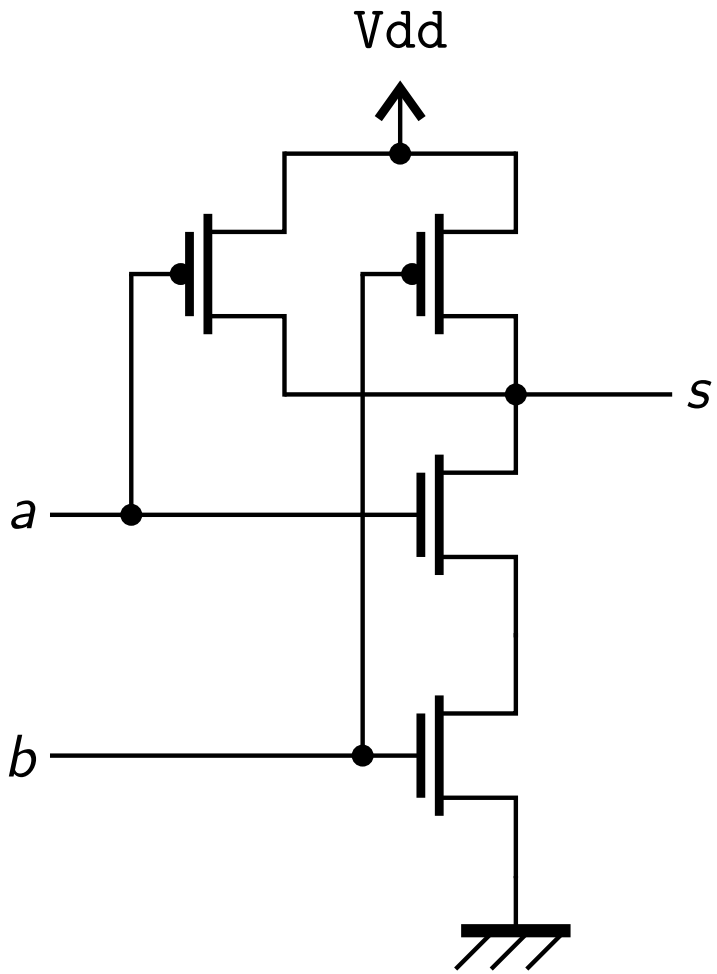
Symbole

Chronogramme



Porte NAND au niveau transistors

Voici “l’intérieur” d’une porte NAND, composée de 4 transistors :



Versions inversées des opérateurs binaires

NOR

Équation logique

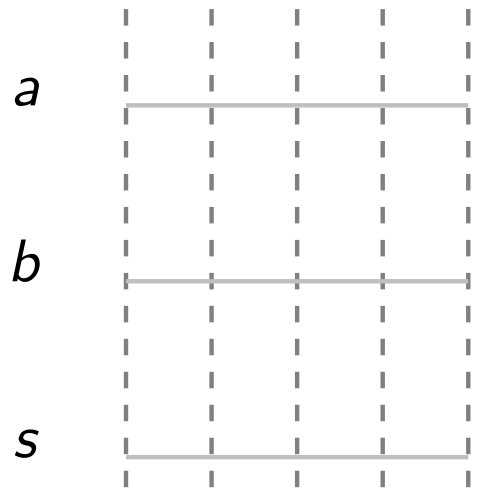
$s =$

Table de vérité

a	b	$s =$
0	0	
0	1	
1	0	
1	1	

Symbole

Chronogramme



Versions inversées des opérateurs binaires

XNOR

Équation logique

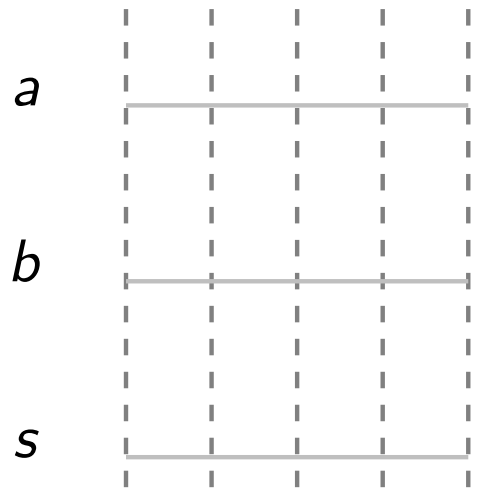
$s =$

Table de vérité

a	b	$s =$
0	0	
0	1	
1	0	
1	1	

Symbole

Chronogramme



Règles de De Morgan[2]

Les règles de De Morgan permettent :

- de "découper" une grande négation,
- de transformer un ET logique en OU logique, et inversement.

$\overline{a \cdot b} =$

a	b	$\overline{a \cdot b}$	$\bar{a}$	$\bar{b}$	
0	0	1			
0	1	1			
1	0	1			
1	1	0			

$\overline{a \vee b} =$

a	b	$\overline{a \vee b}$	$\bar{a}$	$\bar{b}$	
0	0	1			
0	1	0			
1	0	0			
1	1	0			

[2] Augustus De Morgan (1806 - 1871)

Universalité de la porte NAND

Toute fonction logique est réalisable avec **seulement** des portes NAND.

NOT

$$s = \bar{a} =$$

a	b	$s = \overline{a \cdot b}$
0	0	1
0	1	1
1	0	1
1	1	0

AND

$$s = a \cdot b =$$

Universalité de la porte NAND

Toute fonction logique est réalisable avec seulement des portes NAND.

OR

$s = a \vee b =$

NOR

$s =$

Universalité de la porte NAND

Fonction logique	Coût en portes NAND
NOT	1
AND2	2
OR2	3
NAND2	1
NOR2	4

On peut faire le même raisonnement avec la porte NOR !

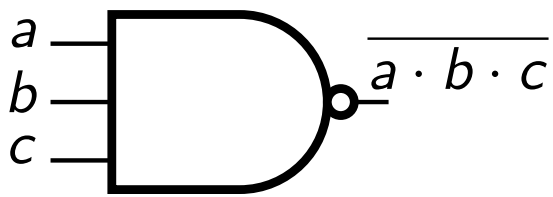
Attention

Le coût **unitaire** est très **élevé** mais, dans un **logigramme complet**, de nombreuses **simplifications par double-complémentation** sont possibles.

Exercice : Porte NAND à 3 entrées

Construire une porte NAND à 3 entrées avec des portes NAND à 2 entrées.

$$\overline{a \cdot b \cdot c} =$$

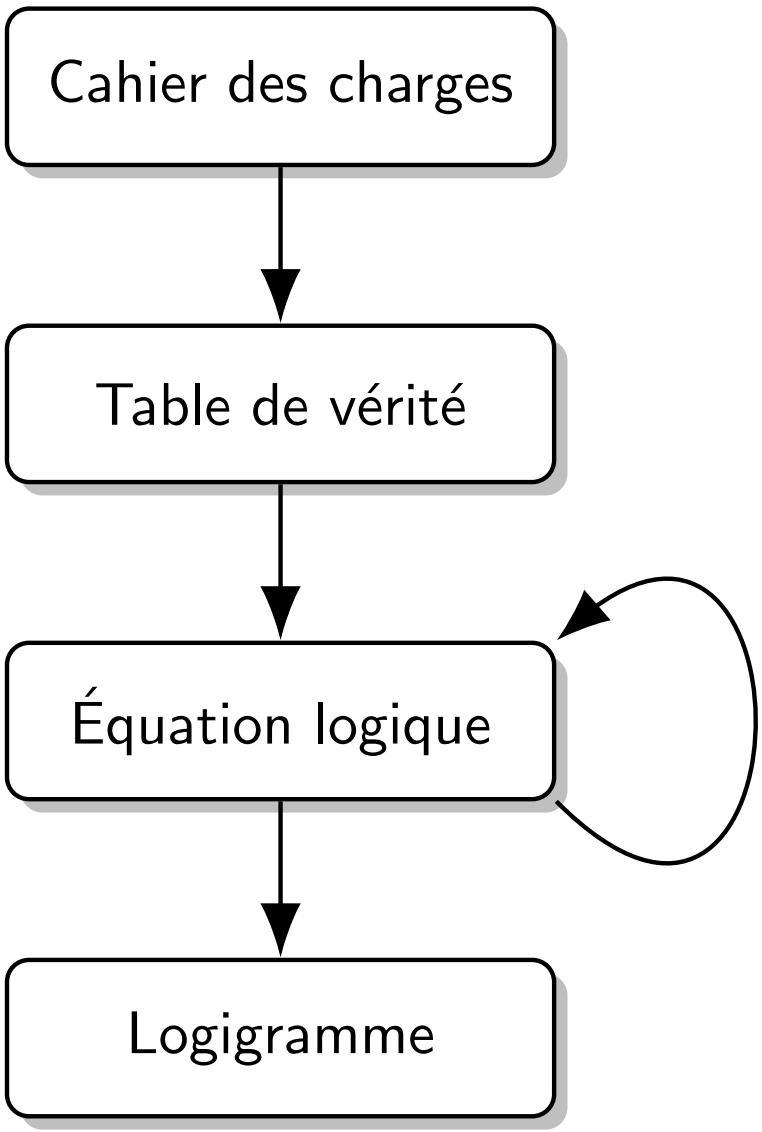


Fil rouge : Implémentation avec des portes NAND

$$m = t \cdot s \vee p \cdot t \vee p \cdot s =$$

Coût en ressources logiques :

Méthode : Conception numérique



- Simplification grâce aux règles de l'algèbre de Boole
- Utilisation de portes NAND/NOR grâce aux règles de De Morgan

Limite

Les règles de l'algèbre de Boole sont **complexes**.

Ces règles n'offrent **pas de garantie** de réussite.

Tableaux de Karnaugh[3] : remplissage

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

s

ab \ c	0	1
00	0	0
01	0	1
11	1	1
10	0	1

Attention

D'une valeur des entrées à l'autre, **un et un seul bit change d'état**.

[3] Maurice Karnaugh (1924 - 2022)

Tableaux de Karnaugh : construction des groupes de 1

s

ab \ c	0	1
00	0	0
01	0	1
11	1	1
10	0	1

Les groupes de 1 :

- sont des **rectangles**
  - ou *a fortiori* des carrés
- contiennent $\mathbf{2^k}$ éléments
  - $1, 2, 4$ ou 8
- peuvent **“s’enrouler”** autour du tableau
  - sortir en haut pour revenir en bas,
  - sortir en bas pour revenir en haut,
  - sortir à gauche pour revenir à droite,
  - sortir à droite pour revenir à gauche,
- sont aussi **grands** que possible
- peuvent se **chevaucher**
- sont aussi **peu nombreux** que possible

Tableaux de Karnaugh : déduction de l'équation logique

s	c \ ab	0	1
	00	0	0
	01	0	1
	11	1	1
	10	0	1

L'équation logique d'un groupe se construit avec les valeurs des entrées qui restent **fixes**.

Groupe bleu

- a
- b
- c

$s_{bleue} =$

Groupe rouge

- a
- b
- c

$s_{rouge} =$

Groupe vert

- a
- b
- c

$s_{vert} =$

L'équation logique de la sortie est le **OU logique** des équations des groupes :

$$s = s_{bleue} \vee s_{rouge} \vee s_{vert}$$

$$=$$

Tableaux de Karnaugh : pourquoi cela fonctionne ?

s

ab \ c	0	1
00	0	0
01	0	1
11	1	1
10	0	1

En groupant et en identifiant les valeurs des entrées qui restent fixes, on **factorise**.

Groupe bleu

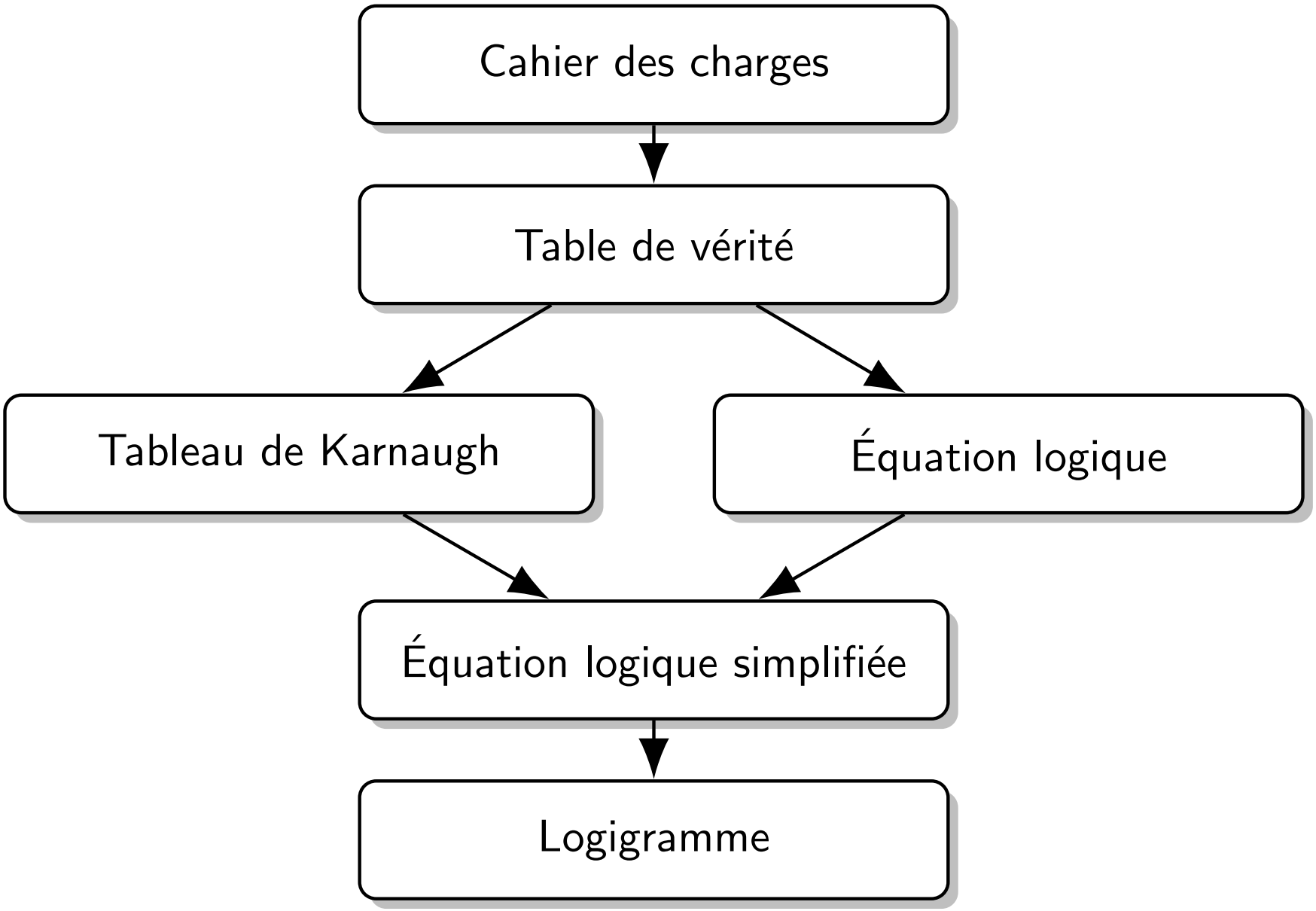
- $a = 1$
- $b = 1$
- c varie

$s_{bleue} = a \cdot b$

D'après la table de vérité et les règles de l'algèbre de Boole :

$$\begin{aligned} s_{bleue} &= a \cdot b \cdot c \vee a \cdot b \cdot \bar{c} \\ &= a \cdot b \cdot (c \vee \bar{c}) \\ &= a \cdot b \cdot 1 \\ &= a \cdot b \end{aligned}$$

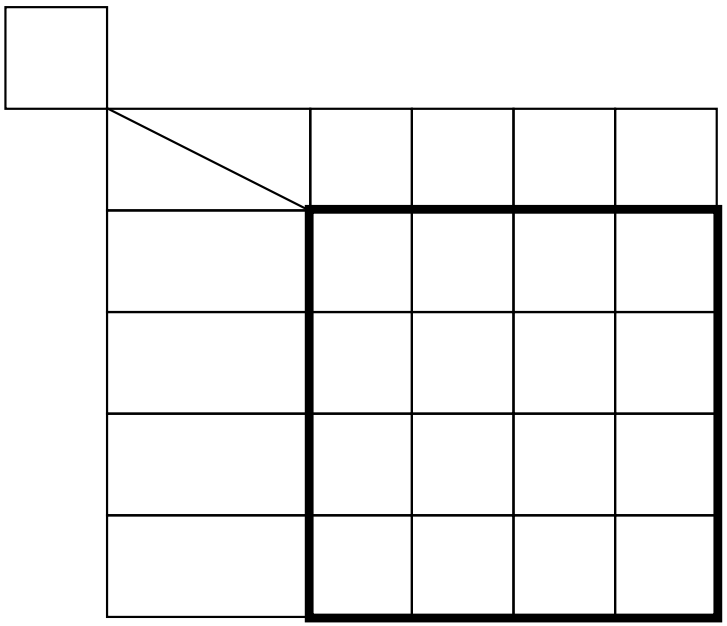
Méthode : Conception numérique



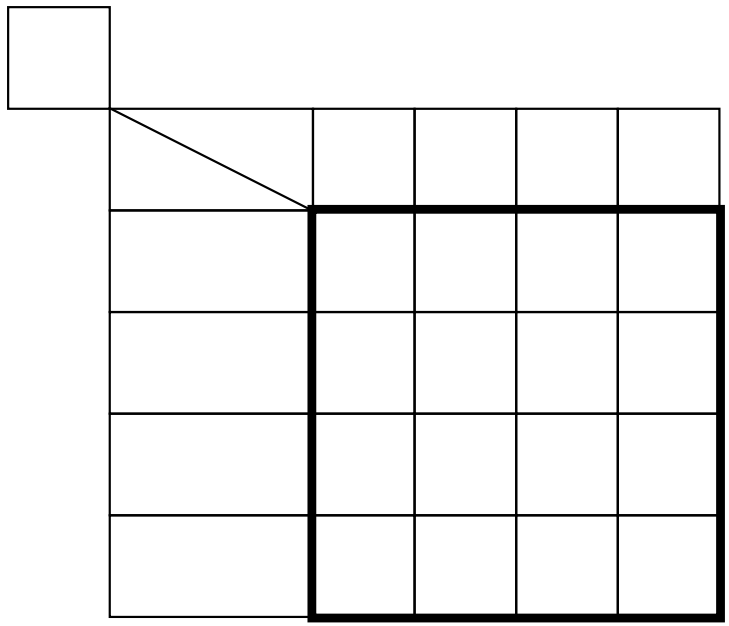
Exercice : Tableaux de Karnaugh

Exercice : trouver l'équation simplifiée des sorties s et y.

h	i	j	k	s	y
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	1	0	1
1	1	1	0	0	0
1	1	1	1	0	0



$s =$



$y =$

Pire cas : un damier

b

kl \ mn	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

En remontant à la table de vérité, on reconnaît celle du **XOR**.

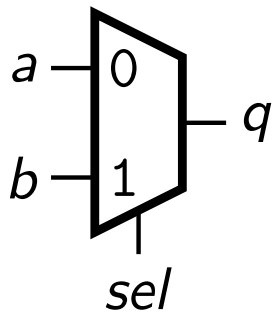
Un damier “inversé” correspond à la table de vérité du **XNOR**.

On traîte d'éventuels groupes de 1 en ajoutant un terme en OU dans la somme de produits.

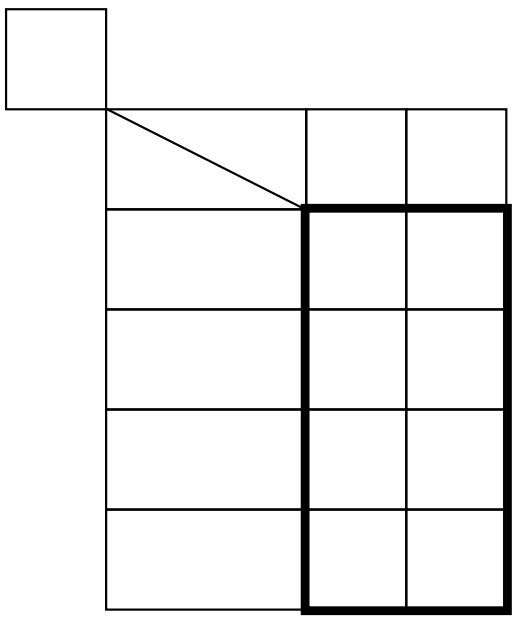
Exercice : Multiplexeur 2-vers-1

Donner le logigramme d'un multiplexeur 2-vers-1, un circuit à deux entrées de données a et b, une entrée de sélection sel et une sortie q. La sortie q vaut :

- a si $sel = 0$,
- b si $sel = 1$.



a	b	sel	q



Équation logique simplifiée :

$$s =$$

Logigramme :

Bilan du chapitre

Vous êtes à présent capables de :

- ✔ **traduire** un cahier des charges en une table de vérité,
- ✔ suivre une **méthodologie de conception** aboutissant à un logigramme correct,
- ✔ **transformer** une équation logique ou un logigramme pour tenir compte de la **disponibilité** d'une ou plusieurs portes logiques,
- ✔ **optimiser** un logigramme pour **réduire son coût**.

Limite

Peut-on manipuler des informations plus complexes que la qualité de **véracité** (VRAI/FAUX) ?

En particulier, comment représenter des **quantités** ?

Chapitre 3

—

Numération

Représentation des nombres

Un **nombre** est découpé en "petits morceaux" et est donc représenté par une séquence de **chiffres**. Chaque **chiffre** représente une valeur.

En base 10, les chiffres sont : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9.
En base b, il y a b chiffres, de 0 à $b-1$.

En écrivant 27, **en base 10**, on écrit $2 \times 10 + 7 = \mathbf{2} \times \mathbf{10}^{\mathbf{1}} + \mathbf{7} \times \mathbf{10}^{\mathbf{0}}$

Un **nombre** s'écrit de manière **unique** comme une **somme** de **puissances** de la **base** multipliées par des **chiffres** de la **base**.

$$\begin{aligned} n &= \sum_{k=0}^{N} a_k \times b^k \\ &= a_0 \times b^0 + a_1 \times b + a_2 \times b^2 + \ldots + a_N \times b^N \\ &\text{avec } 0 \leq a_n \leq b-1 \text{ et } b \geq 2 \end{aligned}$$

Écrire 1024 en base 10 sous cette forme :
$1024 =$

D'autres bases que vous connaissez déjà

En français, nous utilisons parfois d'autres bases :

- base vingt
  - $96 = 4 \times 20 + 16$
    $= 4 \times 20^1 + 16 \times 20^0$
  - Les Français disent soixante, soixante-dix, quatre-vingt et quatre-vingt-dix,
  - Les Suisses disent soixante, septante, huitante et nonante.

- base cent
  - $1789 = 17 \times 100 + 89$
    $= 17 \times 100^1 + 89 \times 100^0$
  - appréciée des historiens.

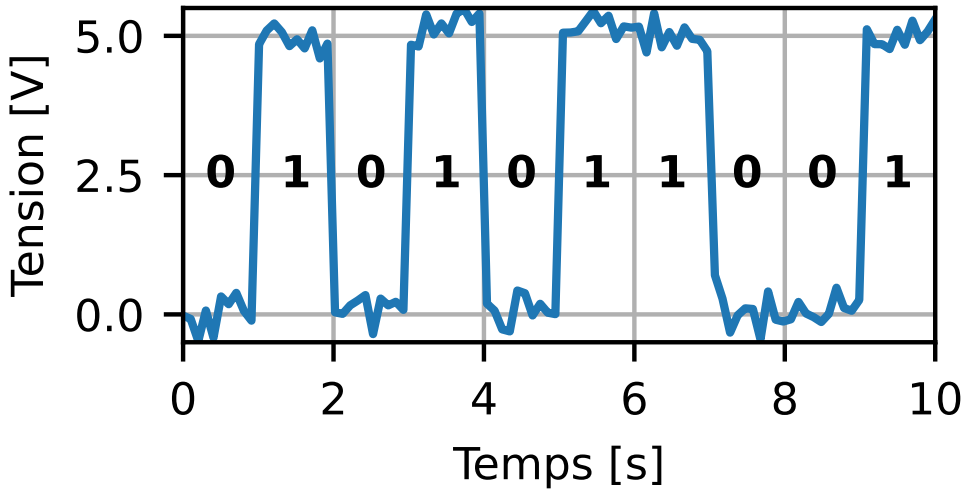
Remarque : on exprime facilement un nombre décimal en base 100 en **groupant les chiffres**, car 100 est une **puissance** de 10.
On fait ainsi des groupes de **2** chiffres car $100 = 10^{\mathbf{2}}$.

Base binaire

En base binaire, il n'y a que **deux chiffres** : 0 et 1.

Pourquoi la base binaire ?

C'est la base la plus **simple** **à implanter** dans le matériel.



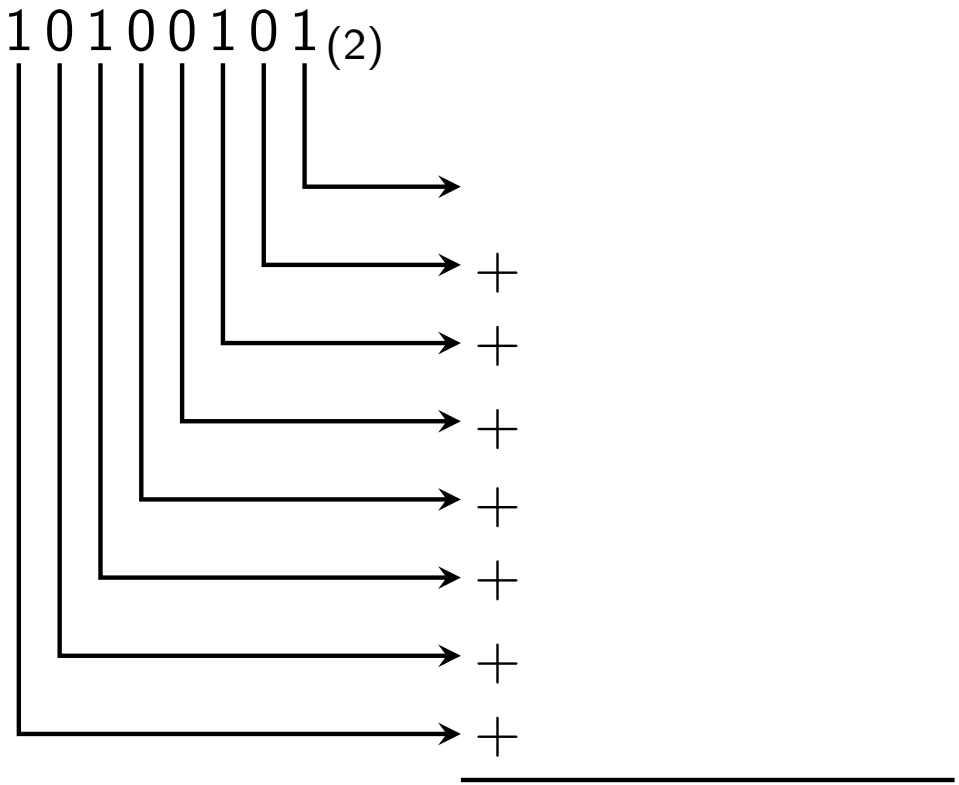
Le principe d'écriture des nombres est identique :
un nombre s'écrit en **binaire** comme une **somme de puissances de 2**.

Exemple : $6 = 4 + 2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 110$

La notation **indicielle** permet d'**expliciter la base** et évite les confusions.

$$6_{(10)} = 110_{(2)}$$

Conversion de binaire vers décimal



Il est utile de connaître :

- les puissances de 2 jusqu'à $2^{16} = 65536$
  - $2^3 = 8$
  - $2^4 = 16$
  - $2^5 = 32$
  - $2^6 = 64$
  - $2^7 = 128$
  - $2^8 = 256$
  - ...
- les ordres de grandeurs :
  - $2^{10} = 1024 \simeq 10^3$
  - $2^{20} \simeq (10^3)^2 \simeq 10^6$
  - $2^{30} \simeq 10^9$

Conversion de décimal vers binaire

Comment trouver l'écriture décimale (les chiffres) de la quantité 456 ?
Divisions entières successives par la base jusqu'à obtenir un quotient nul.

456/10

Le principe est **identique** en base 2.

Conversion de décimal vers binaire

Trouver l'écriture binaire de $456_{(10)}$

Divisions entières successives par 2 :

$456/2$

Conversion de décimal vers binaire

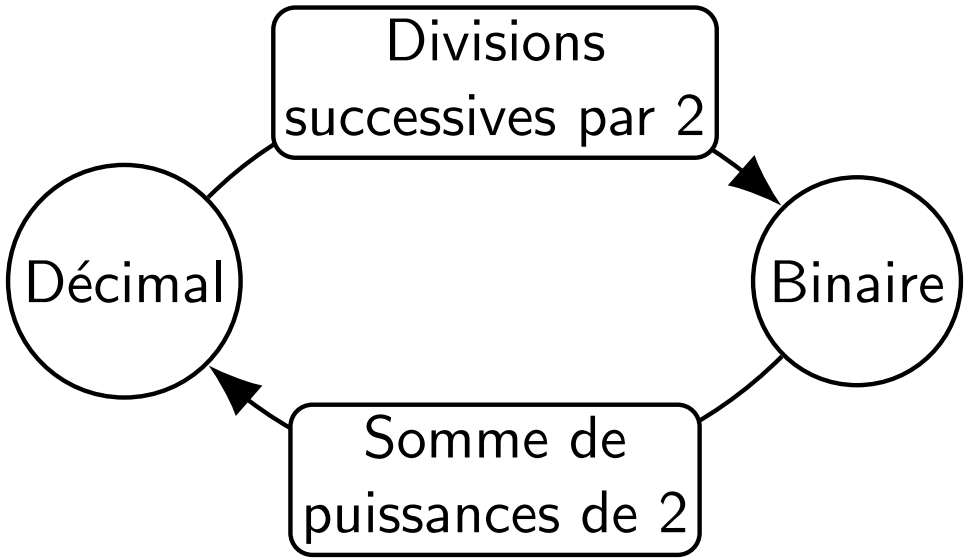
On peut aussi procéder par **additions successives**.

Cela fonctionne bien pour les **petits nombres** et est assez rapide avec un peu d'habitude, à condition de bien connaître les premières **puissances de 2**.

Exemple : $76_{(10)}$

Donc $76_{(10)} =$

Méthode : Conversions entre bases



Nombre de chiffres et capacité

En décimal, en utilisant 3 chiffres, on représente les nombres de à soit nombres différents.

En base b avec n chiffres, on peut donc représenter b^n nombres différents.

Donc en base binaire avec n bits on peut représenter nombres différents, allant de 0 à $2^n - 1$.

Exemple : avec 10 bits, on représente les nombres de à .

Exercice : jusqu'où peut-on compter avec 10 doigts ?

Inversement, pour représenter un nombre N en binaire, il faut bits.

11..1	$2^n - 1$
00..1	1
00..0	0

Exercice : Conversions binaire - décimal

Exercice 1 : Convertir en décimal les nombres binaires suivants :

$10_{(2)} =$ $1110_{(2)} =$ $1000111_{(2)} =$

$101101_{(2)} =$ $1101100_{(2)} =$ $11101101_{(2)} =$

Exercice 2 : Convertir en binaire les nombres décimaux suivants :

$9_{(10)} =$ $13_{(10)} =$ $58_{(10)} =$

$64_{(10)} =$ $175_{(10)} =$ $255_{(10)} =$

Exercice 3 : Déterminer le nombre de bits nécessaires pour coder les nombres décimaux suivants :

$7_{(10)}$: bits $45_{(10)}$: bits $230_{(10)}$: bits

Limite

Exercice : coder en binaire votre année de naissance.

L'écriture binaire de grands nombres est parfois **difficile à lire**.

On aimerait pouvoir "grouper" les 0 et les 1.

Base hexadécimale

En base 16, les chiffres sont : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F.

Chiffre	Quantité	Codage binaire
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001

Chiffre	Quantité	Codage binaire
A	**10**	1010
B	**11**	1011
C	**12**	1100
D	**13**	1101
E	**14**	1110
F	**15**	1111

Les chiffres de 0 à 9 codés en binaire forment le code BCD (*binary-coded decimal*).

Méthode : Valeur d'un nombre dans une base donnée

Chiffres dans la base (symboles)	s_3	s_2	s_1	s_0
Indice (position)	3	2	1	0
Poids = base$^{\text{indice}}$	b^3	b^2	b^1	b^0
Valeur = symbole $\times$ poids	$s_3 \times b^3$	$s_2 \times b^2$	$s_1 \times b^1$	$s_0 \times b^0$

$a = 325_{(10)}$

Chiffres	3	2	5
Indice	2	1	0
Poids			
Valeur			

$a =$

$b = 1001_{(2)}$

Chiffres	1	0	0	1
Indice				
Poids				
Valeur				

$b =$

$c = 5E_{(16)}$

Chiffres		
Indice		
Poids		
Valeur		

$c =$

Conversion de binaire vers hexadécimal

Grouper par 4 les chiffres en binaire en commençant à droite, par les **chiffres de poids faible**, et remplacer chaque bloc par un chiffre hexadécimal.

$$80000_{(10)} = \quad 1001110001000000_{(2)}$$

Donc $80000_{(10)} = 1001110001000000_{(2)} =$ $_{(16)}$

Cela fonctionne car 16 (hexadécimal) est une puissance de 2 (binaire).
On fait des groupes de **4** chiffres car $16 = 2^{\mathbf{4}}$.

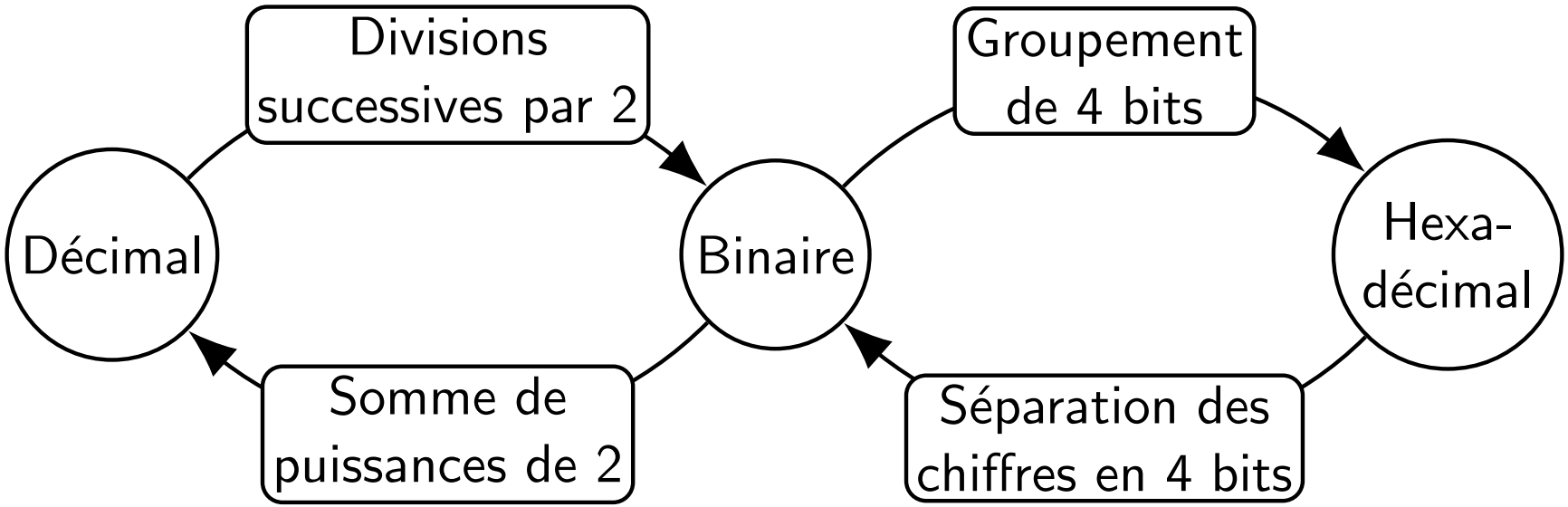
C'est le même procédé lorsqu'on convertit $1789_{(10)}$ en base cent : $17\ 89_{(100)}$ en disant "dix-sept cent quatre-vingt neuf", car 100 est une **puissance** de 10.

Conversion d'hexadécimal vers binaire

Remplacer chaque chiffre hexadécimal par sa représentation binaire.

Exemple : $5A71_{(16)} =$ $_{(2)}$

Méthode : Conversions entre bases



Exercice : Conversions binaire - hexadécimal

Exercice 1 : Convertir en binaire les nombres hexadécimaux suivants :

$B_{(16)} =$ $5C_{(16)} =$ $CAFE_{(16)} =$

Exercice 2 : Convertir en hexadécimal les nombres binaires suivants :

$110_{(2)} =$ $110010_{(2)} =$ $10010001101_{(2)} =$

Limite

Comment convertir de décimal en hexadécimal et vice versa **directement** ?

Conversion d'hexadécimal vers décimal

Un nombre s'écrit en **hexadécimal** comme une somme de puissances de **16**.

$$\mathrm{D\ 5\ A}_{(16)}$$

Conversion de décimal vers hexadécimal

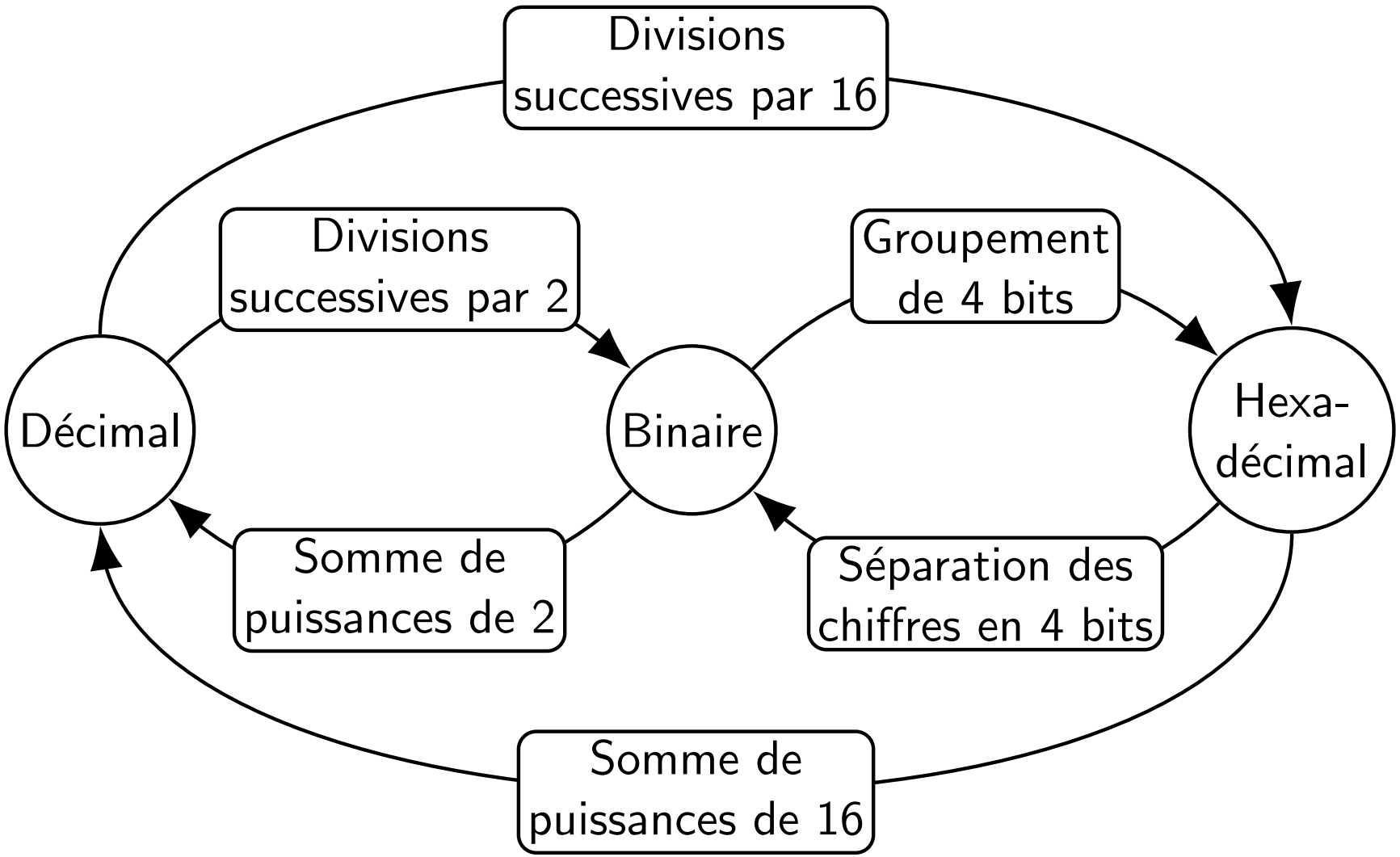
De la même manière que la conversion de décimal à binaire, on effectue des **divisions entières successives par la base** jusqu'à obtenir un quotient nul.

Trouver l'écriture hexadécimale de $8000_{(10)}$

Divisions entières successives par 16 :

Donc $8000_{(10)} =$ $_{(16)}$

Méthode : Conversions entre bases



Exercice : Conversions décimal - hexadécimal

Exercice 1 : Convertir en décimal les nombres hexadécimaux suivants :

$9_{(16)} =$ $85_{(16)} =$ $1000_{(16)} =$

Exercice 2 : Convertir en hexadécimal les nombres décimaux suivants :

$14_{(10)} =$ $192_{(10)} =$ $8228_{(10)} =$

Bilan du chapitre

Vous êtes à présent capables de :

- ✔ Représenter un entier naturel en base **binaire**,
- ✔ Représenter un entier naturel en base **hexadécimale**,
- ✔ **Convertir** directement entre les bases binaire, décimale et hexadécimale.

Limite

Peut-on concevoir des circuits électroniques numériques qui manipulent des nombres ?

Peut-on concevoir des circuits numériques pour réaliser des **opérations arithmétiques** en binaire ?

Comment fabriquer une **calculatrice** ?

(interdite à l'examen tant qu'on ne sait pas faire)

Chapitre 4

—

Arithmétique

Addition binaire

L'addition binaire s'effectue de la **même manière** que l'addition décimale.

Rappel :

$$\begin{array}{r} 1\ 2\ 3\ 4 \\ +\ 6\ 0\ 8\ 5 \\ \hline \end{array}$$

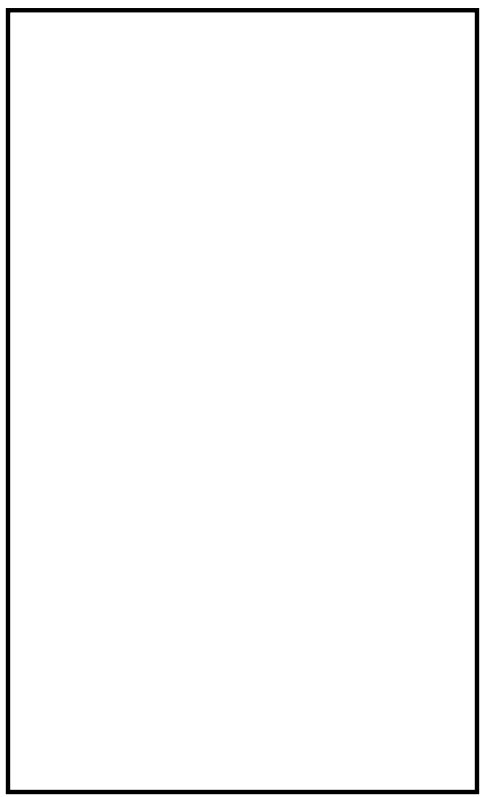
En binaire :

$$\begin{array}{r} 0\ 1\ 0 \\ +\ 0\ 1\ 1 \\ \hline \end{array} \qquad \begin{array}{r} \\ +\ \ \ \\ \hline \end{array}$$

Fil rouge : Additionneur binaire sur 4 bits

Objectif : concevoir un additionneur binaire sur 4 bits.

- 2 entrées : a et b sur 4 bits,
- 1 sortie : $s = a + b$ sur 4 bits.



Limite

Il faut écrire 4 tables de vérité, une pour chaque sortie de s_3 à s_0.

Chaque table de vérité a : $2^4 \times 2^4 = 2^{4+4} = 2^8 = 256$ lignes.

Conception structurelle

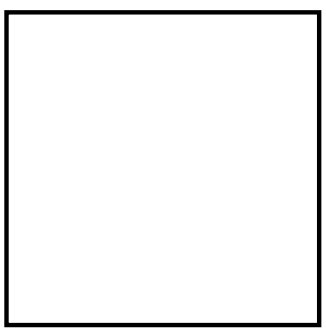
Le problème à résoudre peut être **découpé** en problèmes plus simples.

Observation :

Pour une addition binaire sur n bits, on effectue n additions sur 1 bit.

$$\begin{array}{rcccc} & {\scriptstyle 1} & & & \\ & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 \end{array}$$

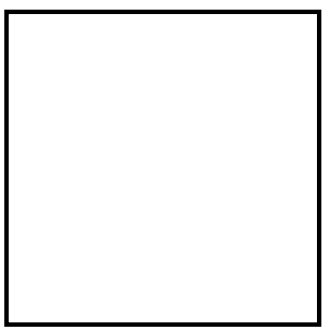
On peut donc concevoir un **additionneur sur 1 bit** puis les “chaîner” :



a	b	$s = a + b$
0	0	
0	1	
1	0	
1	1	

Additionneur avec retenue sortante (demi-additionneur)

On ajoute une sortie pour la retenue **sortante**.

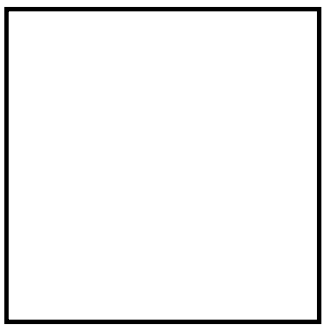


a	b		
0	0		
0	1		
1	0		
1	1		

Additionneur complet

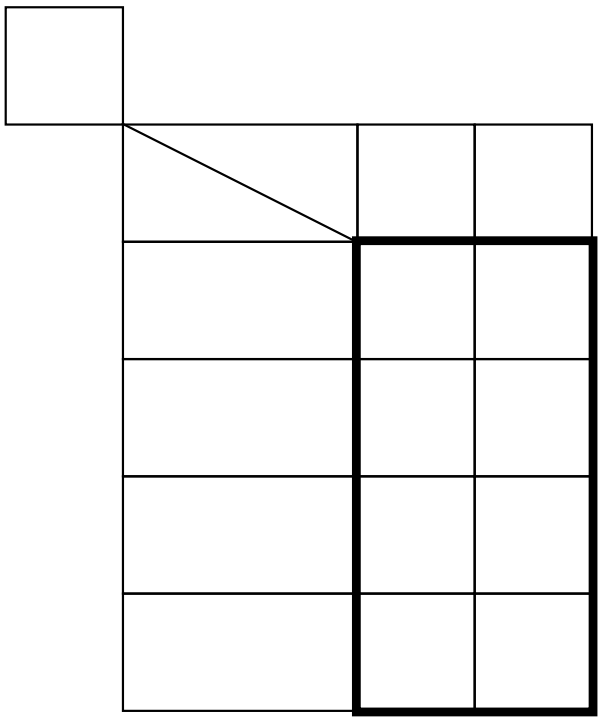
On ajoute une entrée pour la retenue **entrante**.

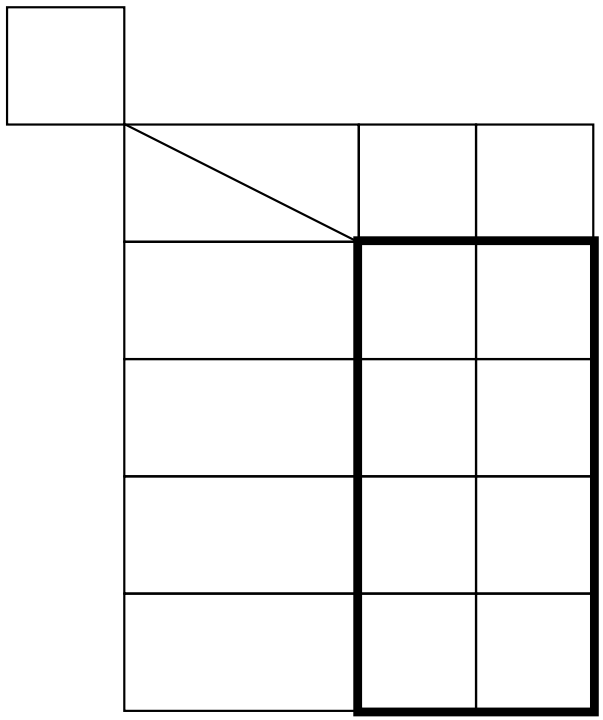
`add_complet`



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

Additionneur complet



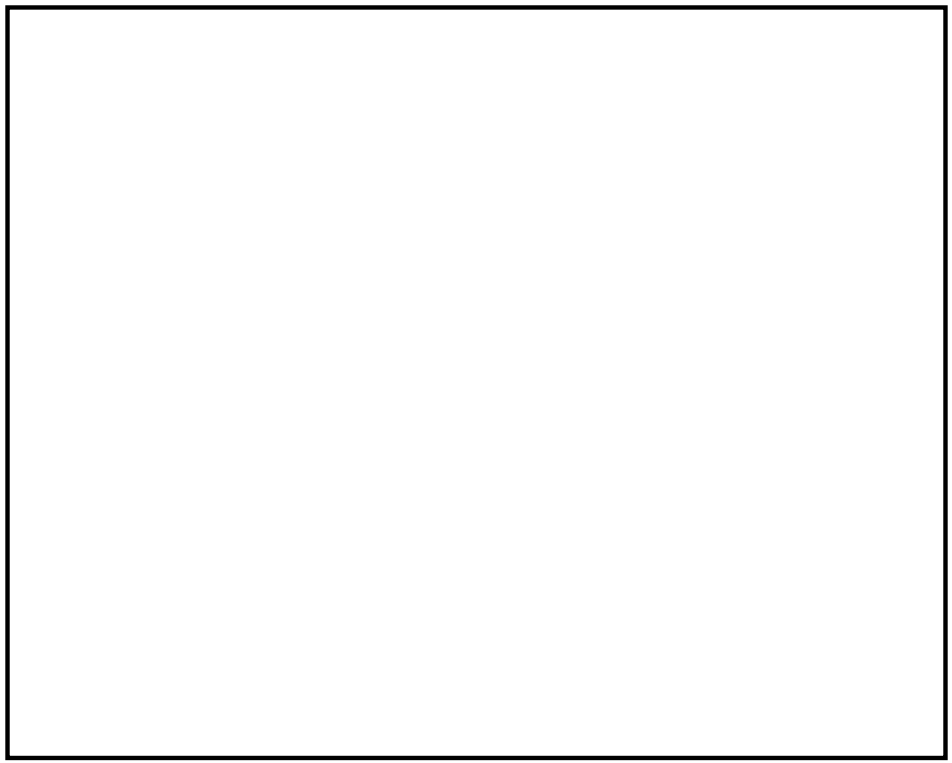


$s =$

On reconnaît

$r_{out} =$

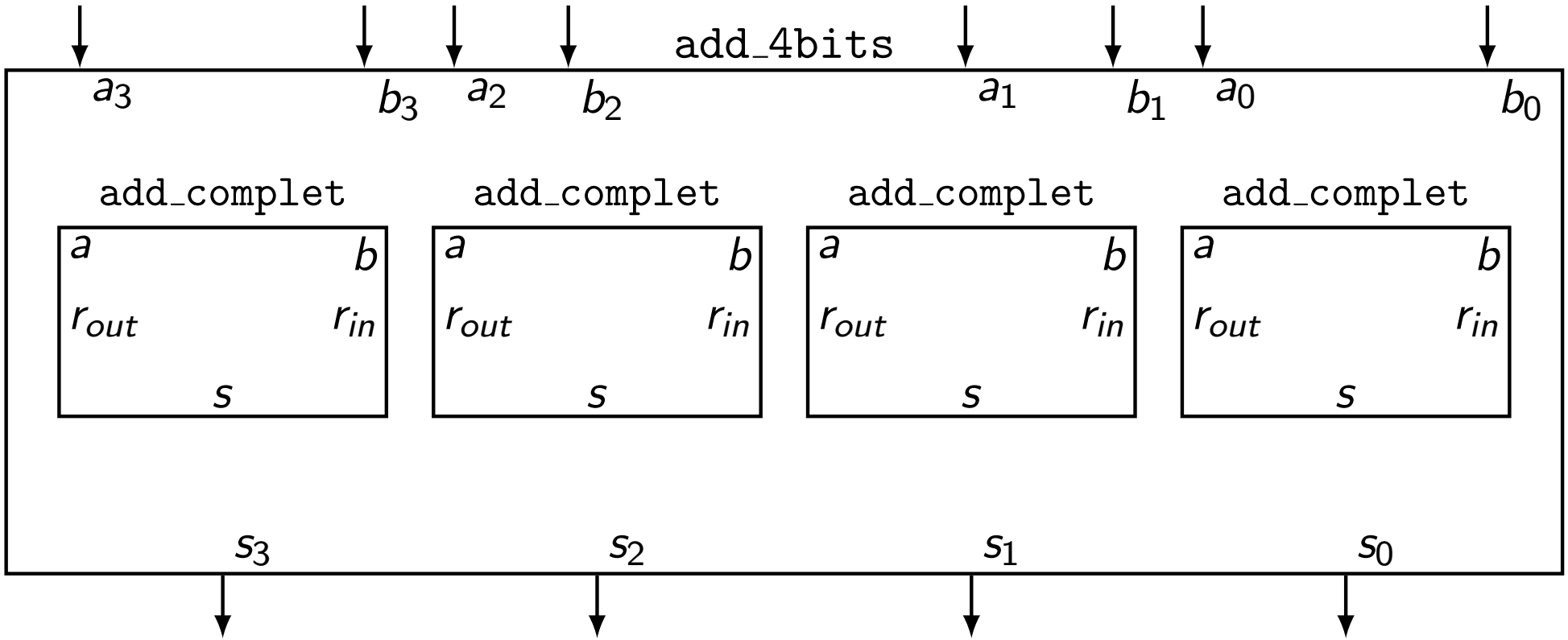
Additionneur complet



Conception structurelle : chaque “boîte” possède :

- un **nom**,
- des **entrées nommées**,
- des **sorties nommées**.

Fil rouge : Additionneur binaire sur 4 bits



Limite

On ne sait représenter pour l'instant que des nombres positifs ou nuls.

Comment représenter les nombres **positifs, nuls ou négatifs** ?
Comment représenter des nombres **signés** ?

Comment obtenir **l'opposé** d'un nombre ?

Représentation intuitive : signe & amplitude

Pour une **largeur de représentation donnée** sur n bits, on dédie le **bit le plus à gauche** à la représentation du **signe** : c'est le **bit de signe**.
Le bit de signe vaut :

- 0 si le nombre est positif,
- 1 si le nombre est négatif.

C'est (presque) ce qu'on fait en décimal, avec les symboles $+$ et $-$.

Les **autres** $n-1$ bits sont utilisés pour représenter **l'amplitude**.

Exemples **sur 4 bits** :

$$-5_{(10)} = \qquad\qquad 6_{(10)} =$$

Avantage/inconvénients de la représentation signe & amp.

- **Avantage :**
  - Intuitive / facile à comprendre

- **Inconvénients :**
  - Le nombre zéro admet **deux** représentations : 10...0 et 00...0.
    - Mais après tout, en décimal aussi.
  - L'addition binaire **ne fonctionne pas**

$$\begin{array}{rr} & 5 \\ + & -2 \\ \hline \end{array} \qquad \begin{array}{rrrrr} & 0 & 1 & 0 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

Méthode du complément à deux

Toujours pour une **largeur de représentation donnée** sur n bits, on obtient **l'opposé** d'un nombre par la méthode du complément à deux en :

1. **inversant chaque bit** de la représentation binaire puis,
2. **ajoutant 1**.

Exemple sur 4 bits :

$3_{(10)} = \quad _{(2)}$

↓ inversion des bits

↓ $+1$

$= -3$

Décimal	Repr. signée sur 4 bits	Décimal	Repr. signée sur 4 bits
7		-1	
6		-2	
5		-3	
4		-4	
3		-5	
2		-6	
1		-7	
0		-8	

Attention

Le **bit le plus à gauche** code toujours le **signe**.

Nombre signés : capacité

Le **bit le plus à gauche** code le **signe**.

Sur n bits, il reste $n-1$ bits pour coder l'amplitude.

Le **zéro** est codé comme un nombre **positif**.

Les nombres **positifs** représentables sur n bits **en représentation signée** vont donc de 0 à $2^{n-1}-1$.

Les nombres **négatifs** représentables sur n bits **en représentation signée** vont donc de -2^{n-1} à -1.

Sur n bits, on peut donc coder les nombres signés allant de -2^{n-1} à $2^{n-1}-1$.

11..1	-1
10..0	-2^{n-1}
01..1	$2^{n-1}-1$
00..0	0

Autre interprétation du bit de signe : puissance négative

Le bit de signe peut également être vu comme une **puissance de 2 négative**. Interprété ainsi, on obtient directement la **valeur** du nombre **sans avoir à appliquer la méthode du complément à deux**.

Exemple :

$$110_{(2s)} = -2^2 + 2^1 = -4 + 2 = -2$$

Le bit de signe vaut 1 : le nombre est négatif, on obtient sa valeur absolue par la méthode du complément à deux :

1. inversion des bits de la représentation binaire : 001
2. $+1$ pour obtenir la valeur absolue : $1 + 1 = 2$

Le nombre négatif dont la valeur absolue vaut 2 est bien -2.

Complément à deux : pourquoi est-ce que ça fonctionne ?

Soit un nombre binaire x écrit sur n bits comme une séquence :

$$x = x_{n-1}x_{n-2}...x_1x_0$$

Le complément à deux de ce nombre s'écrit, par définition :

$$cpta2(x) = \bar{x}_{n-1}\bar{x}_{n-2}...\bar{x}_1\bar{x}_0 + 1$$

Leur somme vaut :

$$\begin{array}{rccccc} & x_{n-1} & x_{n-2} & \dots & x_1 & x_0 \\ + & \bar{x}_{n-1} & \bar{x}_{n-2} & \dots & \bar{x}_1 & \bar{x}_0 \\ + & & & & & 1 \\ \hline & 1 & 1 & \dots & 1 & 1 \\ + & & & & & 1 \\ \hline & & & & & 0 \end{array}$$

On obtient donc bien l'**opposé**, car $x + cpta2(x) = 0$ donc $cpta2(x) = -x$.

Addition de nombres binaires signés

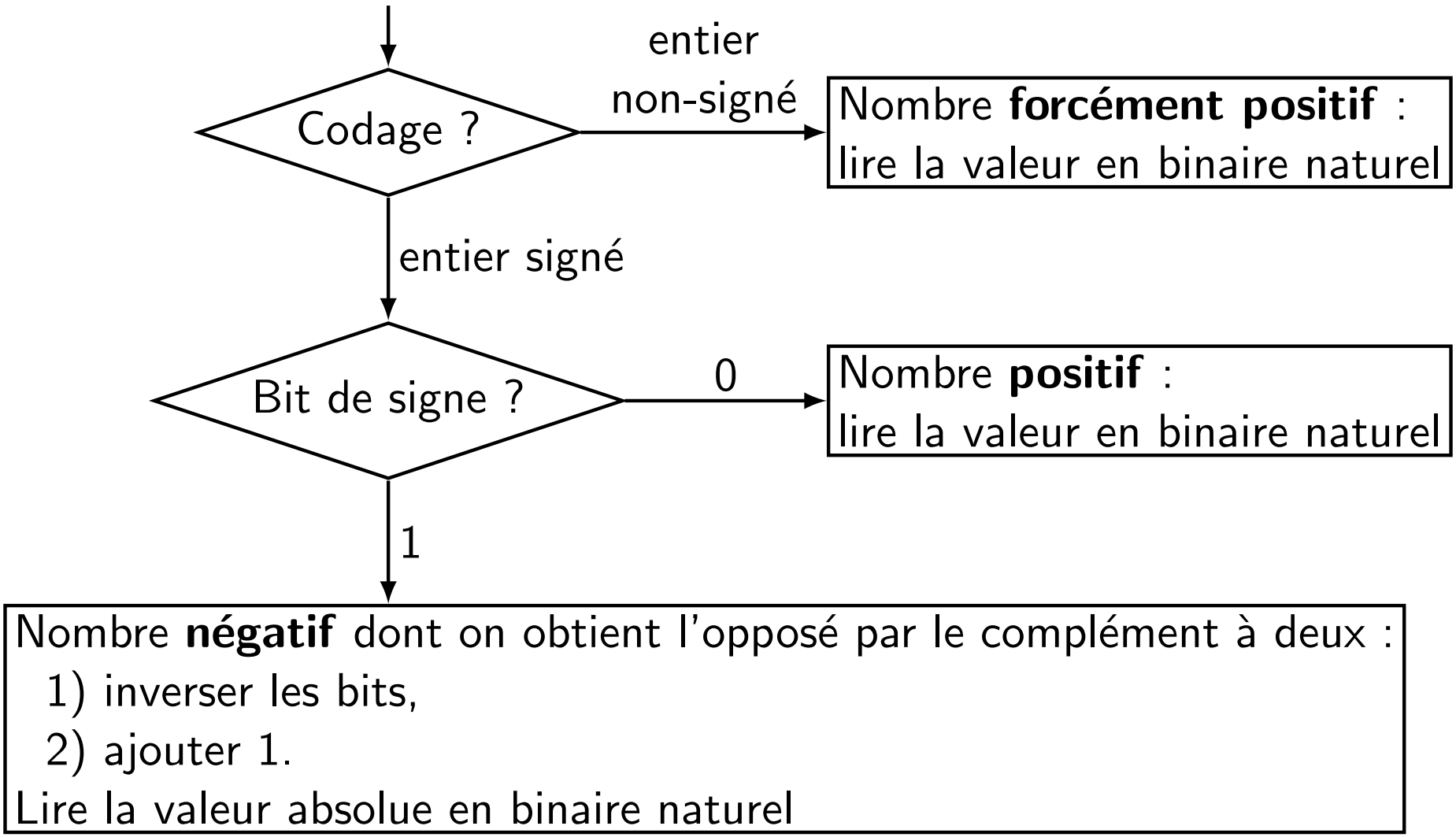
Reprenons l'exemple précédent :

$$\begin{array}{rcccc} & 0 & 1 & 0 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

- 0101
  - bit de signe à 0 : nombre
  - valeur absolue :
- 1010
  - bit de signe à 1 : nombre
  - inversion des bits restants +1 :
  - valeur absolue :
- 1111
  - bit de signe à 1 : nombre
  - inversion des bits restants +1 :
  - valeur absolue :

$$+ \quad \underline{\qquad}$$

Méthode : Interprétation d'un nombre binaire



Exercice : Méthode du complément à deux

Exercice 1 : Représenter **sur 4 bits**, si possible, les nombres signés suivants :

$5_{(10)} =$ $\quad -5_{(10)} =$ $\quad 12_{(10)} =$

$-3_{(10)} =$ $\quad -9_{(10)} =$ $\quad 4_{(10)} =$

Exercice 2 : Donner la valeur décimale des nombres signés binaires suivants :

$0010_{(2s)} =$ $\quad 1010_{(2s)} =$ $\quad 1111_{(2s)} =$

$11001000_{(2s)} =$ $\quad 00100010_{(2s)} =$ $\quad 11111111_{(2s)} =$

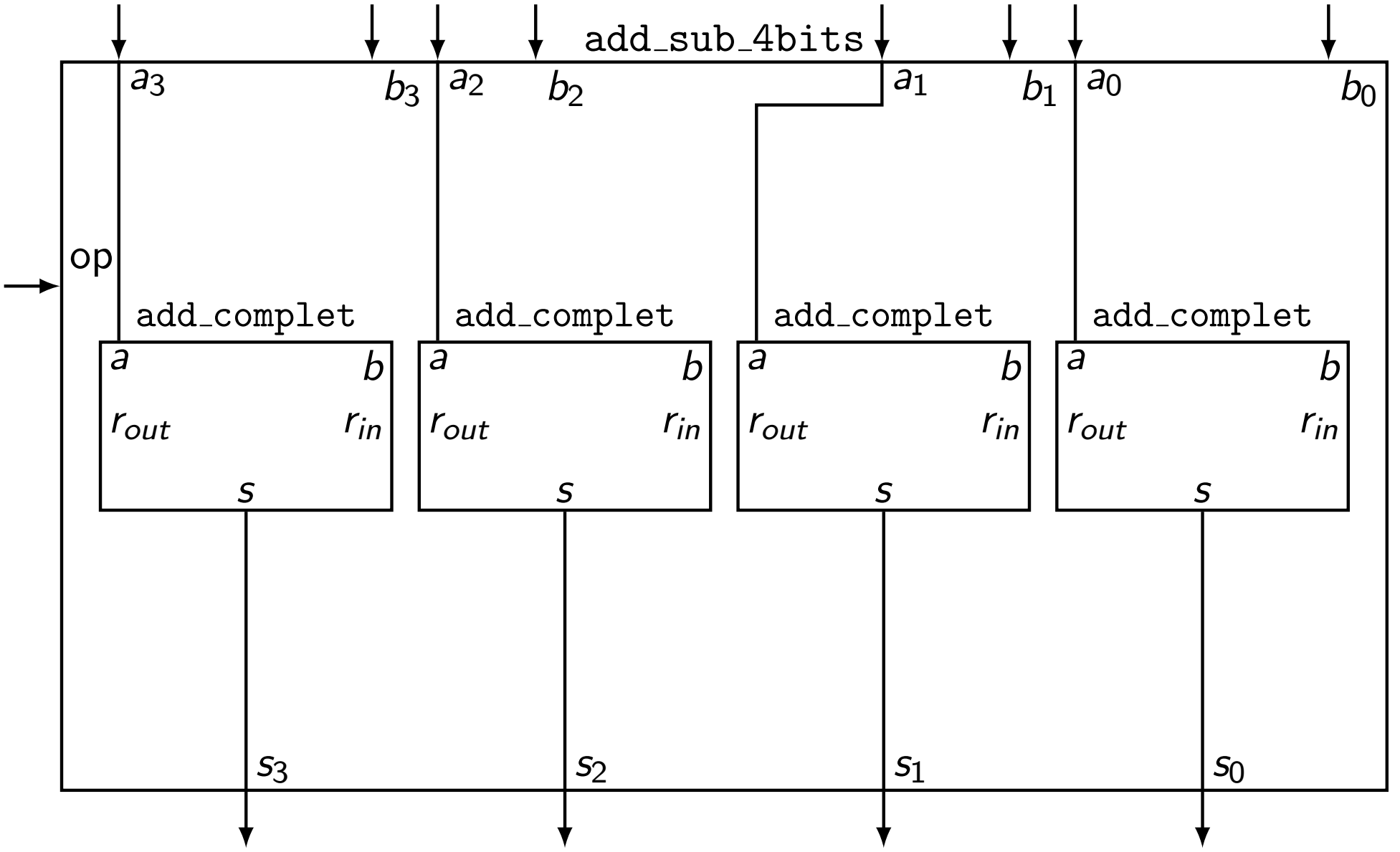
Fil rouge : Additionneur / soustracteur binaire sur 4 bits

On ajoute une entrée (op) permettant de choisir l'opération à effectuer :

- 0 : addition
- 1 : soustraction

L'obtention de **l'opposé** (par la méthode du complément à deux) pour réaliser la **soustraction** sera faite **en interne**.

Fil rouge : Additionneur/soustracteur binaire sur 4 bits



Cas corrects / incorrects

Attention

Les opérations sont réalisées sur n bits.

non-signé **signé**

$$\begin{array}{rcccc} & 1 & 1 & 0 & 0 \\ + & 0 & 1 & 0 & 1 \\ \hline & 0 & 0 & 0 & 1 \end{array}$$

non-signé **signé**

$$\begin{array}{rcccc} & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 \end{array}$$

Drapeaux indicateurs d'état

Lorsqu'un processeur exécute un programme (et utilise son ALU), il doit pouvoir obtenir très rapidement les informations suivantes :

`Z` (**Z**ero) Résultat nul

Égalité vraie (`a == b`)

`N` (**N**egative) Résultat négatif

Comparaison vraie (`a > b`)

`C` (**C**arry) Retenue sortante

Somme non-signée fausse **sur le mot machine de n bits**

`V` (o**V**erflow)

Somme signée fausse **sur le mot machine de n bits**

Carry et Overflow

Les opérations sont **fausses** sur n bits, mais seraient **correctes** sur $n+1$ bits.

Carry

1111	15
1110	14
...	
0001	1
0000	0

Overflow

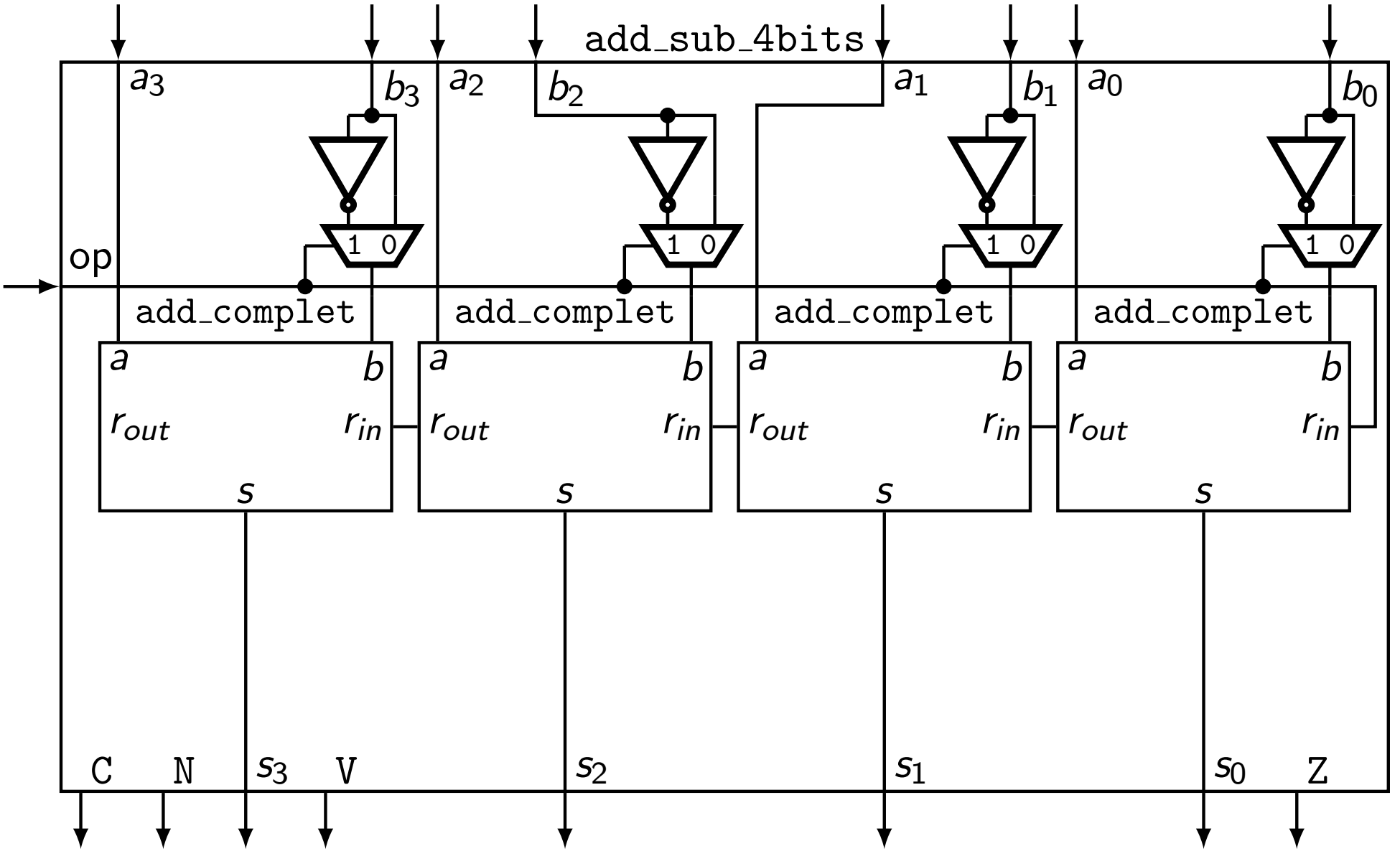
0111	7
...	
0000	0
1111	-1
...	
1000	-8

Overflow

Table de vérité de l'additionneur complet gérant le bit de signe :

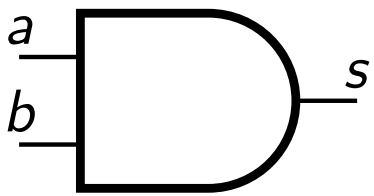
a	b	c_{in}	c_{out}	s		V
0	0	0			$(\ ) + (\ ) = (\ )$	
0	0	1			$(\ ) + (\ ) = (\ )$	
0	1	0			$(\ ) + (\ ) = (\ )$	
0	1	1			$(\ ) + (\ ) = (\ )$	
1	0	0			$(\ ) + (\ ) = (\ )$	
1	0	1			$(\ ) + (\ ) = (\ )$	
1	1	0			$(\ ) + (\ ) = (\ )$	
1	1	1			$(\ ) + (\ ) = (\ )$	

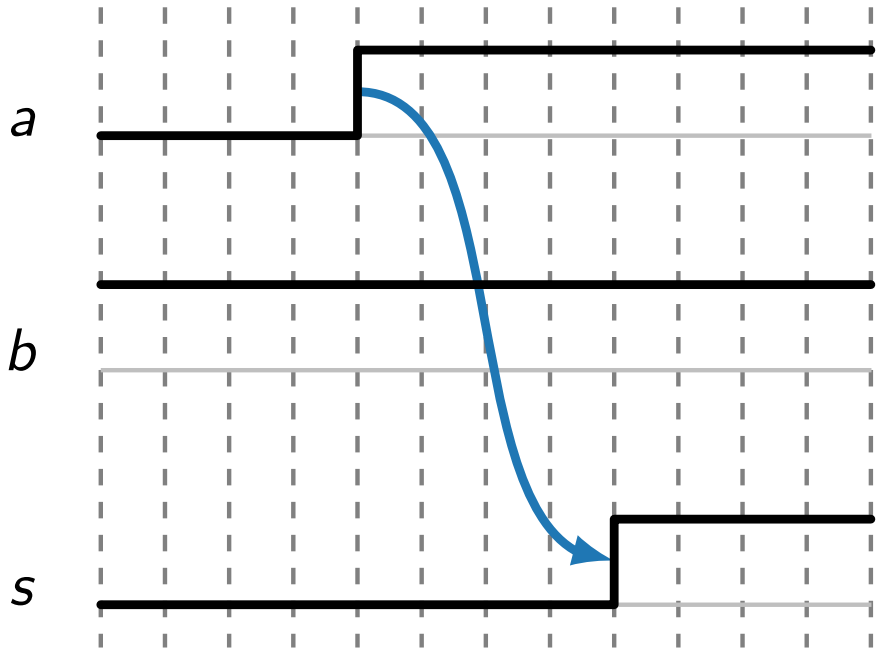
Fil rouge : Add/sub avec drapeaux indicateurs d'état



Délai de propagation

Chaque porte logique possède un **délai de propagation**.
Un changement aux entrées met **un certain temps** à se propager à la sortie.

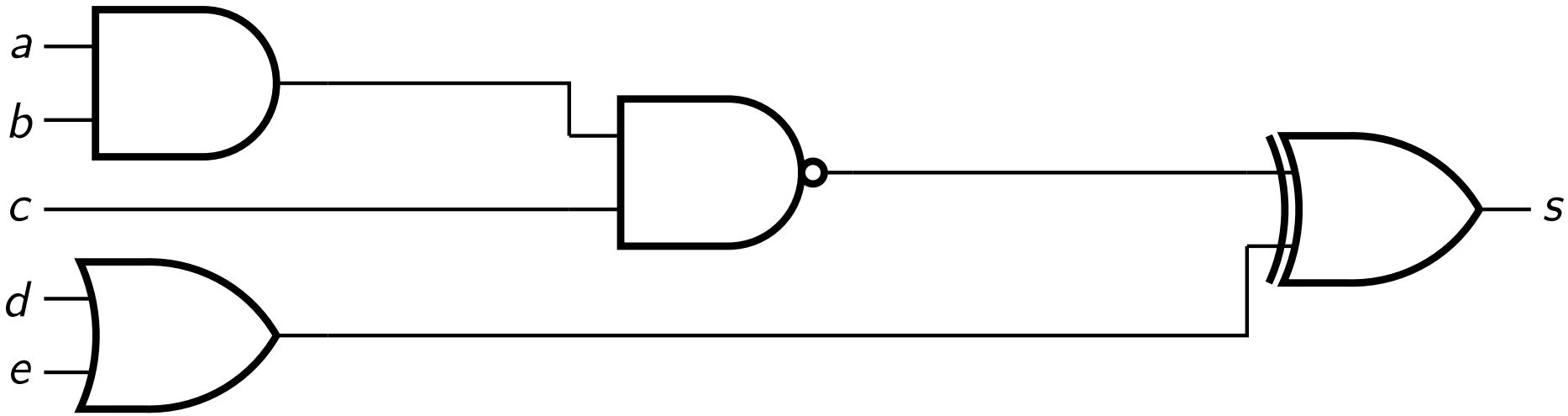




Ces délais sont de l'ordre de la nanoseconde (10) à la picoseconde (10).

Chemin critique

Le **chemin critique** est le chemin, passant par une séquence de portes logiques, ayant le **plus long délai** de propagation dans un circuit numérique.

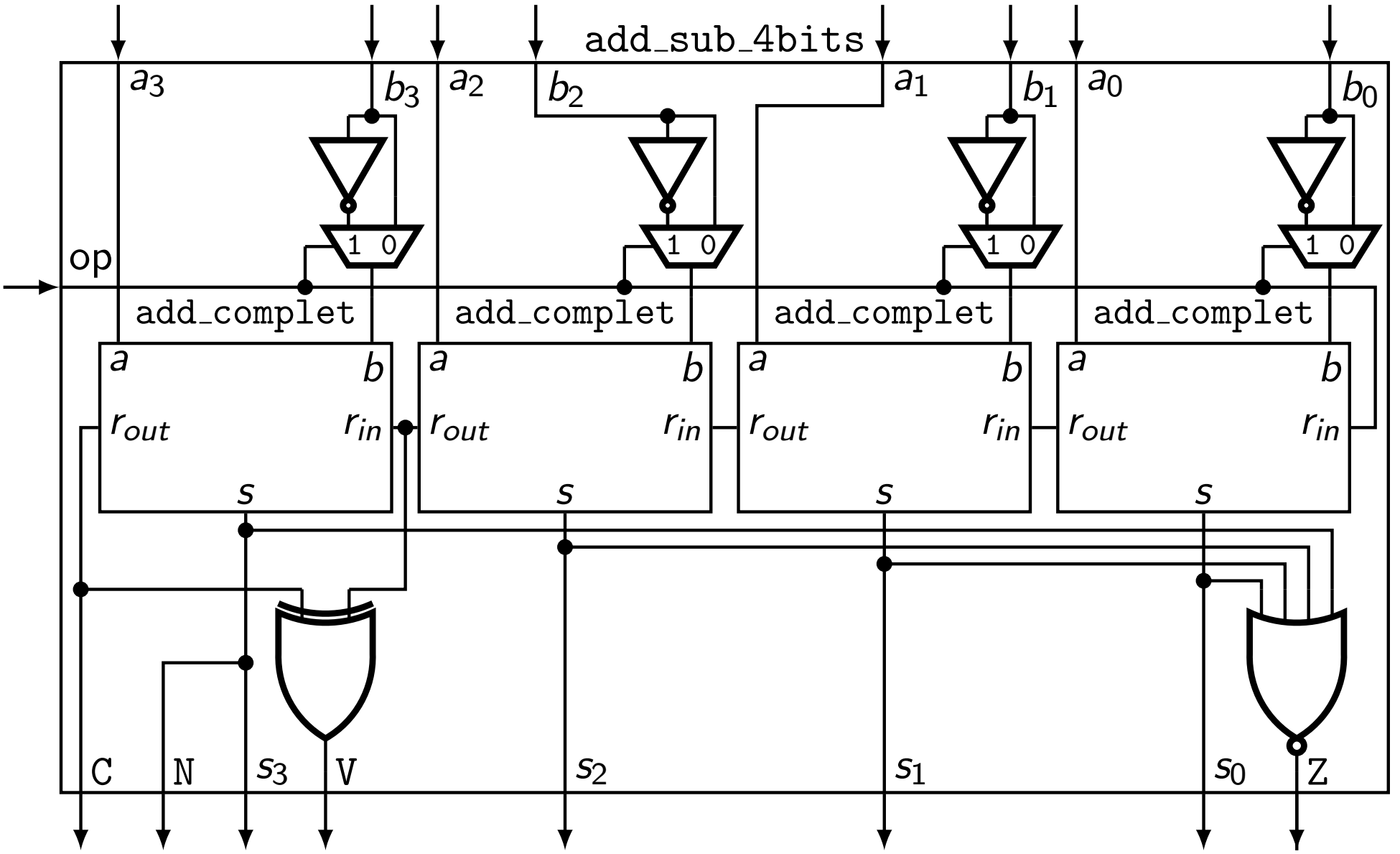


$$t_{critique} =$$

La **fréquence maximale** de fonctionnement dépend du chemin critique :

$$f_{max} = \frac{1}{t_{critique}}$$

Fil rouge : Chemin critique



Bilan du chapitre

Vous êtes à présent capables de :

- ✔ représenter des nombres signés ou non en binaire,
- ✔ concevoir une ALU basique (add / sub), et comprendre ses limites,
- ✔ déterminer la fréquence maximale de fonctionnement d'un circuit.

Limite

La conception de circuits numériques complexes en schéma est :

- fastidieuse,
- difficile à vérifier,
- inefficace.

Chapitre 5

—

Description de circuits numériques combinatoires en VHDL

Motivation

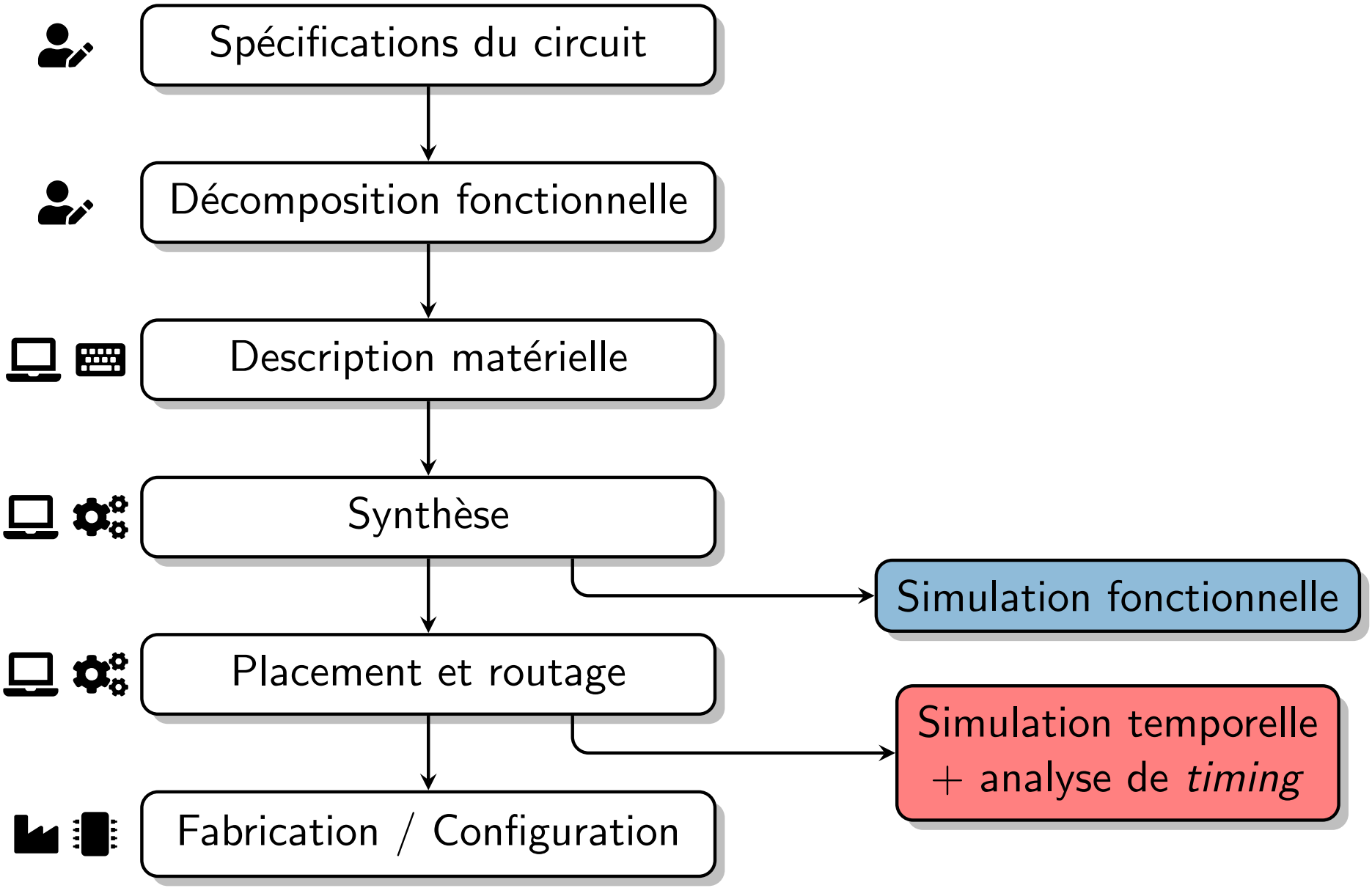
Pour des circuits complexes, il est rapidement **difficile** (ou **pénible**) de :

- Concevoir en traçant des logigrammes,
- Câbler des portes logiques en boîtier,
- Vérifier que le circuit fonctionne correctement,
- Trouver et corriger les erreurs.

Un **langage de description matérielle** permet :

- de **décrire** des circuits sous la forme de **fichiers texte** légers,
- d'exploiter de puissants **logiciels d'optimisation**,
- de **simuler** le circuit conçu avec des logiciels dédiés pour **vérifier** qu'il fonctionnera selon les spécifications.

Flot de conception numérique



VHDL

Le langage de description matérielle que nous utiliserons est le VHDL :

- **V**HSIC **H**ardware **D**escription **L**anguage
  - **V**ery **H**igh-**S**peed **I**ntegrated **C**ircuits
- Langage de description matérielle pour les circuits intégrés très rapides

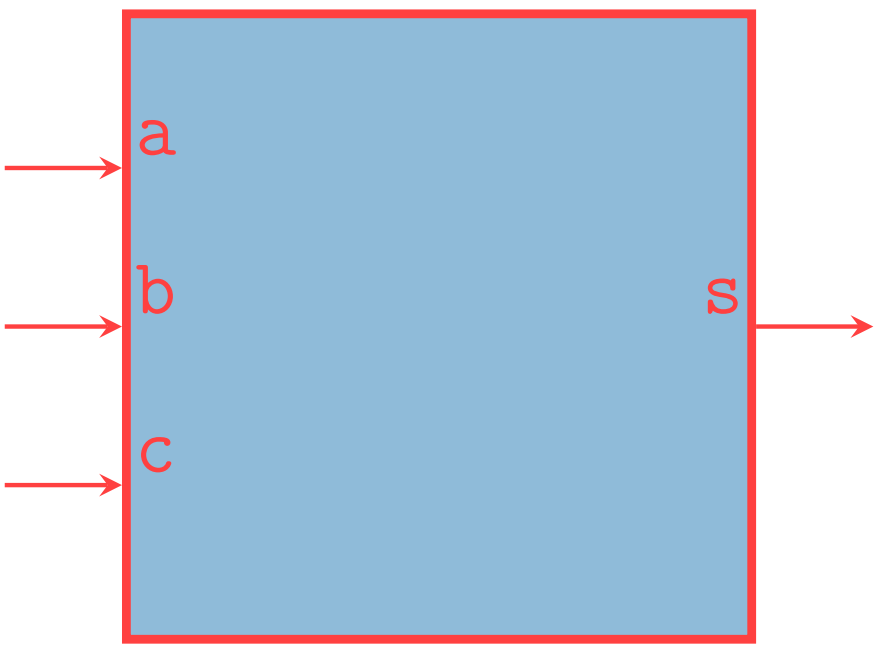
Langage normalisé par l'IEEE

- **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers

En 1987, 1993, 2000, 2002, 2007, 2008 et 2019

Concurrent : langage SystemVerilog.

Organisation d'un fichier .vhd



composant.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

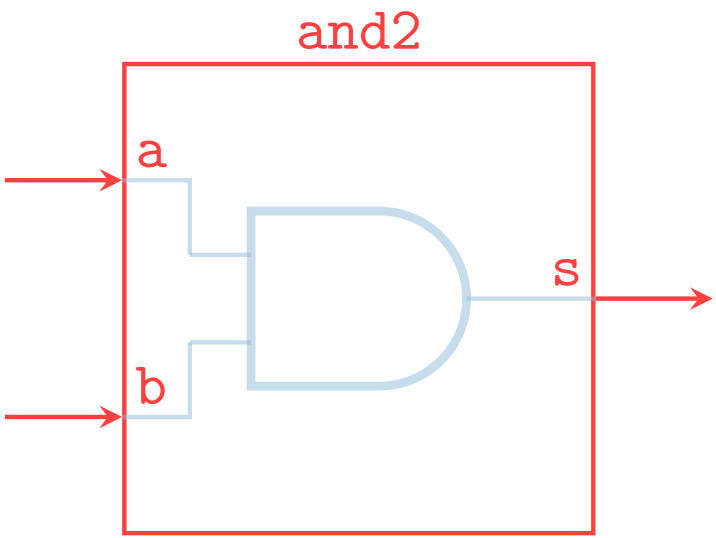
ENTITY

ARCHITECTURE

Entité

L'**entité** (ENTITY) décrit le composant tel qu'on le voit **de l'extérieur** :

- nom,
- entrées,
- sorties.



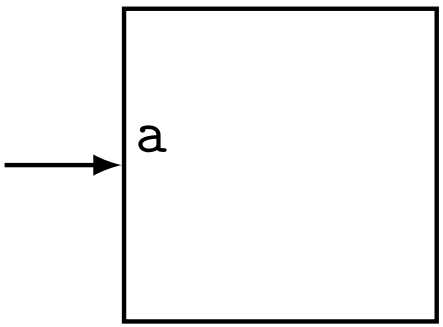
and2.vhd

```
ENTITY and2 IS

  PORT (
    a : IN  STD_LOGIC;
    b : IN  STD_LOGIC;
    s : OUT STD_LOGIC);

END and2;
```

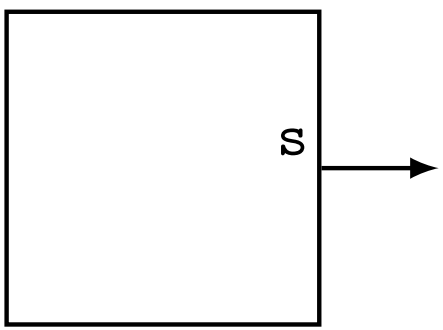
Modes de port

Les **ports** déclarés dans **l'entité** peuvent être de l'un des **deux modes** :

- `IN` : port d'**entrée**
  Les données sur ce port ne peuvent être **que lues, pas écrites**.



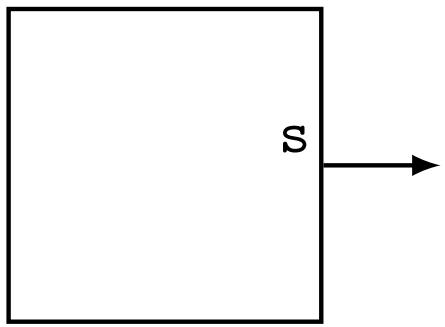
- `OUT` : port de **sortie**
  Les données sur ce port ne peuvent être **qu'écrites, pas lues**.



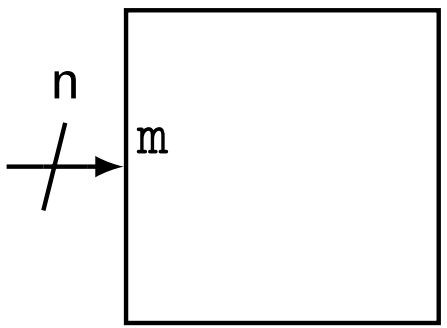
Types de port

Les **ports** déclarés dans **l'entité** peuvent être de l'un des **deux types** :

- `STD_LOGIC` : port sur **un seul bit**



- `STD_LOGIC_VECTOR` : port sur **plusieurs bits**, ou **bus**



Type STD_LOGIC_VECTOR

Le type `STD_LOGIC_VECTOR` décrit un **bus**, c'est à dire un groupement de plusieurs signaux d'un bit.

Un bus de n bits s'écrit : `STD_LOGIC_VECTOR(n - 1 DOWNTO 0)`

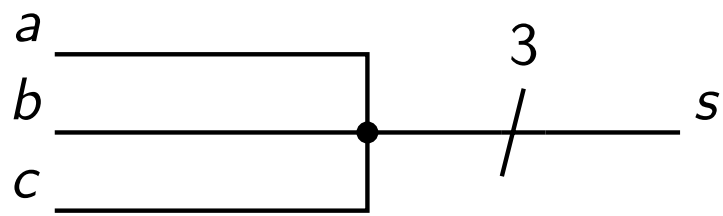
Par exemple, pour un bus de 8 bits : `STD_LOGIC_VECTOR(7 DOWNTO 0)`

On accède aux **sous-groupes** ou aux **sous-signaux** par des **parenthèses** :

- `s(0)` : accès au bit d'indice 0,
- `s(7 DOWNTO 4)` : accès aux bits d'indice 7 à 4.

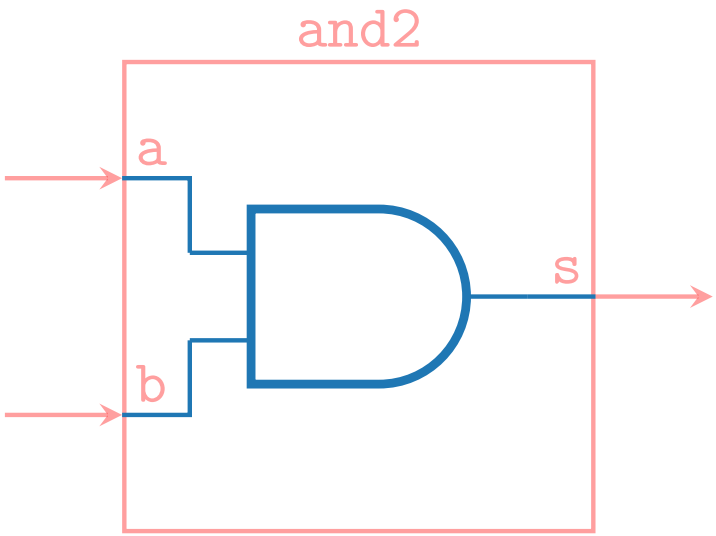
L'opération inverse, de **concaténation**, utilise l'opérateur `&` :

- `s <= a & b & c`



Architecture

L'**architecture** (ARCHITECTURE) décrit **l'intérieur** du composant, la manière dont les entrées sont "transformées" pour obtenir les sorties.



and2.vhd

```
ARCHITECTURE archi OF and2 IS

BEGIN

  s <= a AND b;

END archi;
```

L'architecture est toujours **nommée** (ici archi).

Affectation inconditionnelle

Pour décrire les connexions des signaux à l'intérieur de l'architecture, on utilise une **affectation inconditionnelle**, avec le symbole `<=`

Tous les opérateurs logiques vus précédemment sont disponibles :

- `NOT`
- `AND`
- `NAND`
- `OR`
- `NOR`
- `XOR`
- `XNOR`

Exemples :

```
a <= b XOR c;
s <= a AND b AND c;
y <= (a AND b) OR (c AND d);
```

Attention

Utiliser des parenthèses, sinon **pas de priorité**.

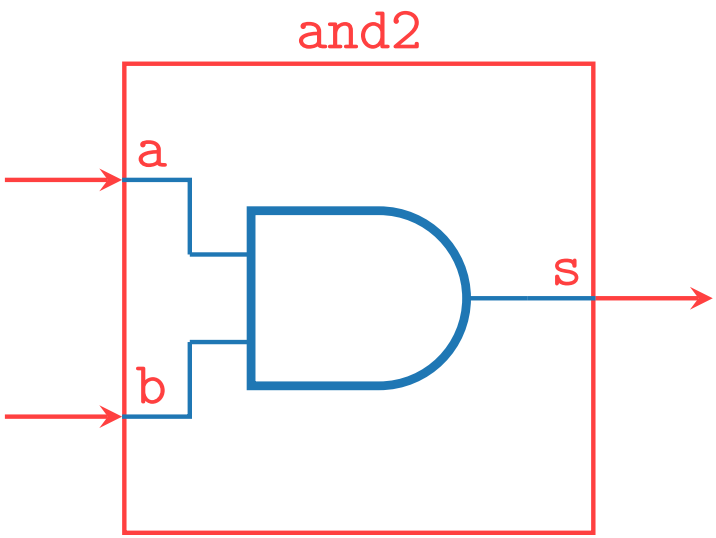
Affectation inconditionnelle et priorité

Comment écrire en VHDL : $y = a \cdot b \vee c \cdot d$

```
y <= (a AND b) OR (c AND d);
```

```
y <= a AND b OR c AND d;
```

Description complète



and2.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and2 IS

  PORT (
    a : IN  STD_LOGIC;
    b : IN  STD_LOGIC;
    s : OUT STD_LOGIC);

END and2;

ARCHITECTURE archi OF and2 IS

BEGIN

  s <= a AND b;

END archi;
```

Fil rouge : Vote majoritaire

Implémenter en VHDL le vote majoritaire à 3 entrées : $m = t \cdot s \vee p \cdot t \vee p \cdot s$

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY maj3 IS









END maj3;

ARCHITECTURE archi OF maj3 IS

BEGIN




END archi;
```

Limite

On a pas gagné grand chose en **lisibilité** par rapport au logigramme...

Multiplexeur

Implémenter en VHDL un multiplexeur 2-vers-1.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux2x1 IS

  PORT (
    a : IN  STD_LOGIC;
    b : IN  STD_LOGIC;
    s : IN  STD_LOGIC;
    q : OUT STD_LOGIC);

END mux2x1;

ARCHITECTURE archi OF mux2x1 IS
```

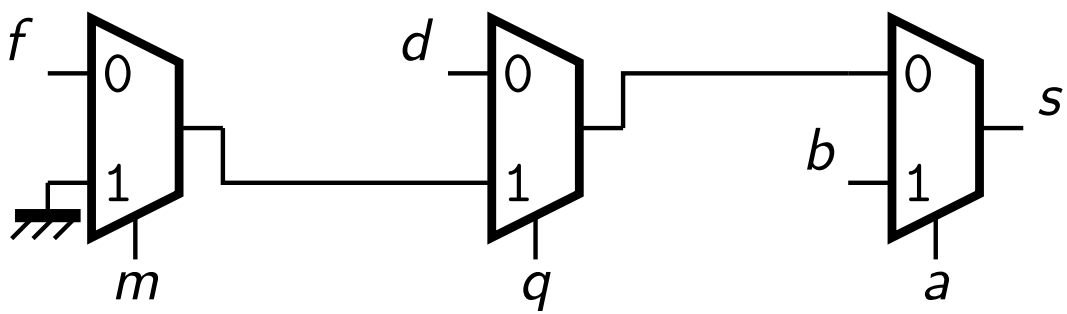

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux2x1 IS

  PORT (
    a : IN  STD_LOGIC;
    b : IN  STD_LOGIC;
    s : IN  STD_LOGIC;
    q : OUT STD_LOGIC);

END mux2x1;

ARCHITECTURE archi OF mux2x1 IS
```

Affectation conditionnelle

L'**affectation conditionnelle** permet de décrire de manière plus "naturelle" une fonction logique, en spécifiant les **différentes valeurs prises** en fonction de **conditions explicites**.

```
s <= b   WHEN a = '1' ELSE
     d   WHEN q = '0' ELSE
     '0' WHEN m = '1' ELSE
     f;                          -- cas par défaut INDISPENSABLE
```

Question : que vaut la sortie s si a est à l'état haut et q à l'état bas ?
Réponse : s vaut b, car les conditions sont évaluées **dans l'ordre d'écriture**.



L'affectation sélective instancie **une suite de multiplexeurs**.

Buffer tri-state

L'affectation conditionnelle est très utile pour décrire un **buffer tri-state**.
Ce composant permet de **déconnecter** un signal **de manière contrôlée**.

tri-state: **trois** états:

- niveau logique haut 1,
- niveau logique bas 0,
- état .

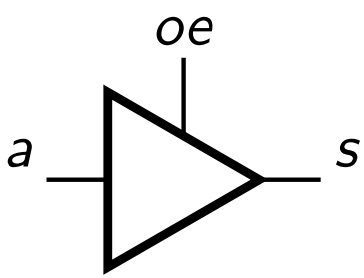


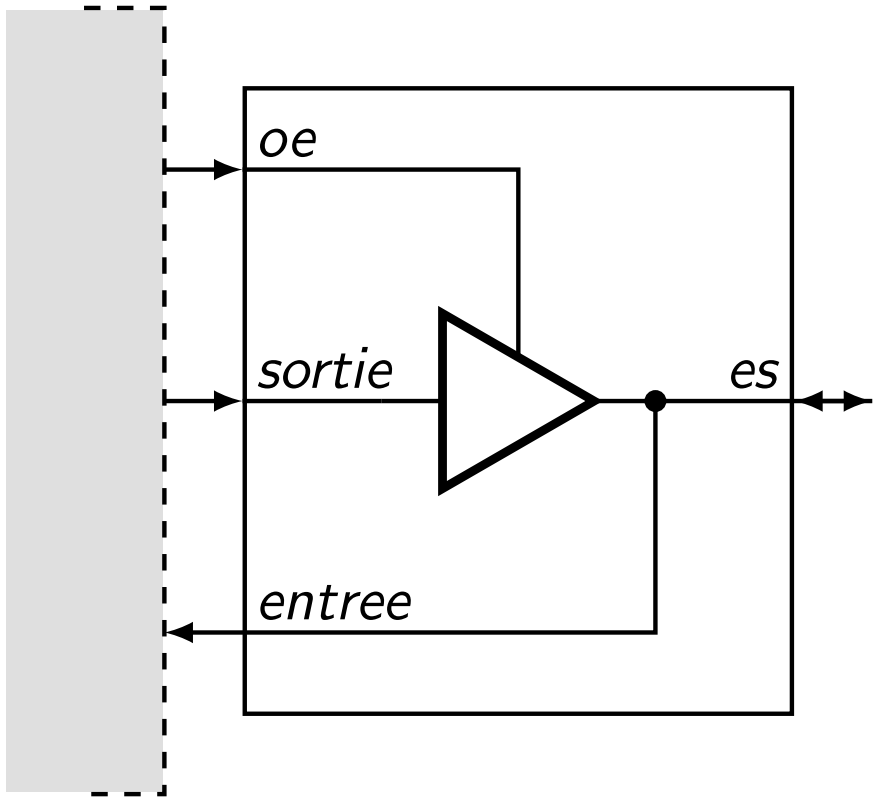
Table de vérité

a	oe	s
0	0	
1	0	
0	1	
1	1	

- Si $oe = 0$:
- Si $oe = 1$:

Entrée/sortie bi-directionnelle

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY es_bidir IS

  PORT (
    es     : INOUT STD_LOGIC;
    oe     : IN    STD_LOGIC;
    sortie : IN    STD_LOGIC;
    entree : OUT   STD_LOGIC);

END es_bidir;

ARCHITECTURE archi OF es_bidir IS

BEGIN

  es <= sortie WHEN oe = '1' ELSE
        'Z';
  entree <= es;

END archi;
```



Attention

N'utiliser `INOUT` que dans les cas **strictement nécessaires** ! Il y a du **matériel** derrière le VHDL !

Codeur

Que fait le design décrit ci-dessous ?

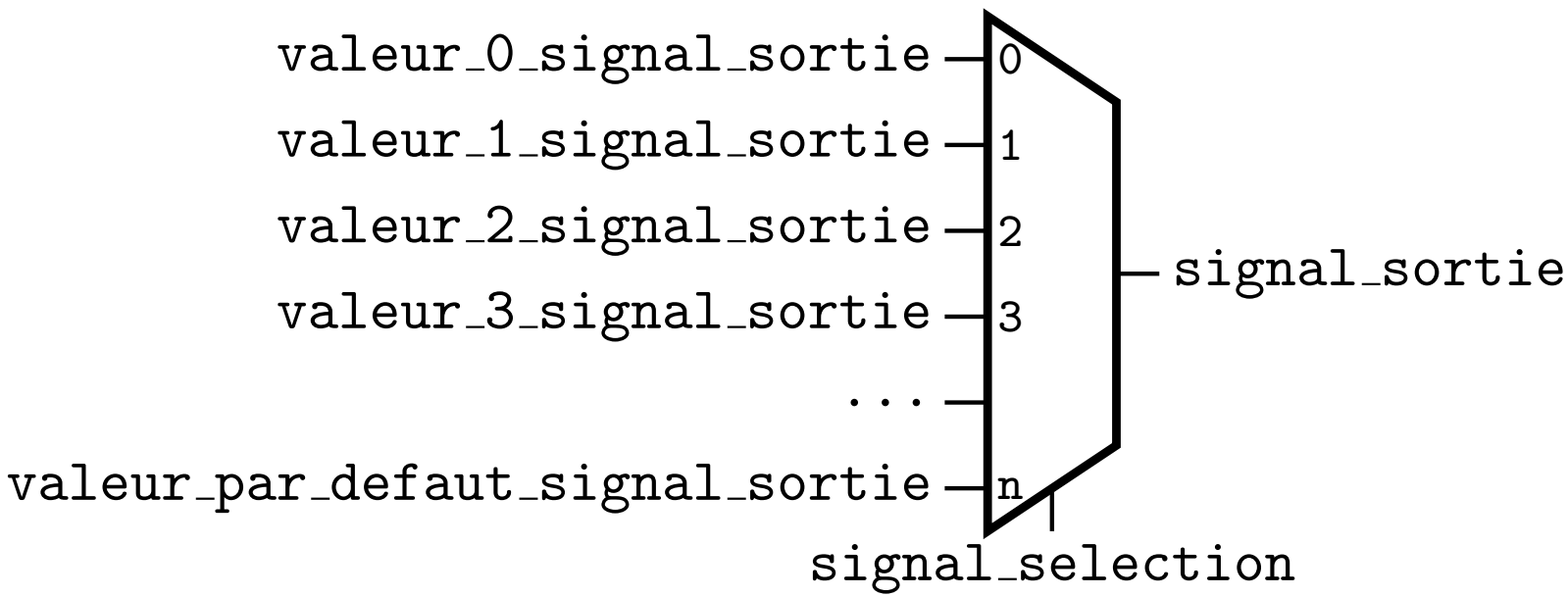
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY codeur IS

  PORT (
    entree : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
    sortie : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));

END codeur;
```

```
ARCHITECTURE archi OF codeur IS

BEGIN

  WITH entree SELECT
    sortie <=
    "00000001" WHEN "000",
    "00000010" WHEN "001",
    "00000100" WHEN "010",
    "00001000" WHEN "011",
    "00010000" WHEN "100",
    "00100000" WHEN "101",
    "01000000" WHEN "110",
    "10000000" WHEN "111",
    "00000000" WHEN OTHERS;

END archi;
```

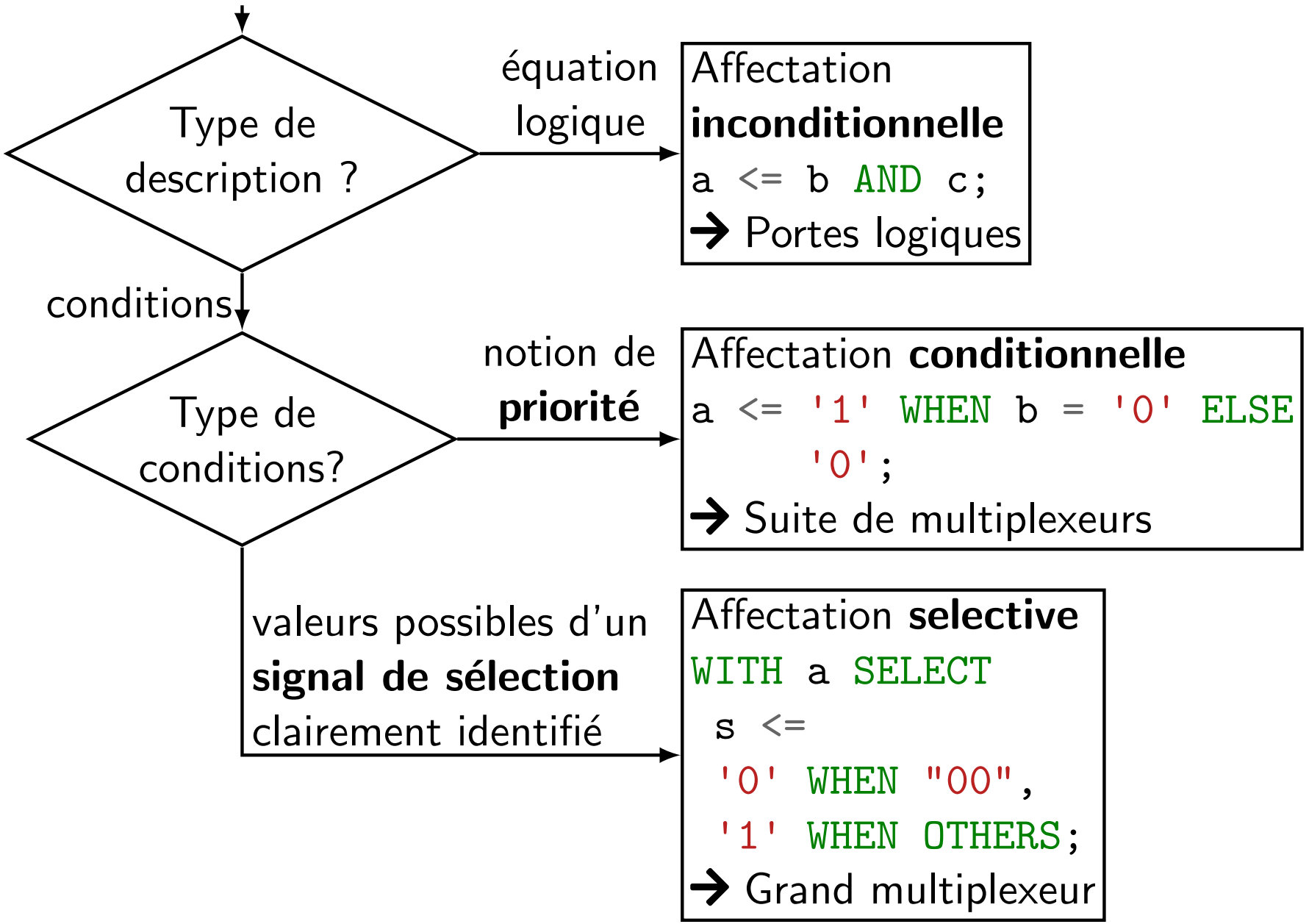
C'est un .

Affectation sélective

```
WITH signal_selection SELECT
signal_sortie <=
valeur_0_signal_sortie WHEN valeur_0_signal_selection,
valeur_1_signal_sortie WHEN valeur_1_signal_selection,
valeur_2_signal_sortie WHEN valeur_2_signal_selection,
valeur_3_signal_sortie WHEN valeur_3_signal_selection,
...
valeur_par_defaut_signal_sortie WHEN OTHERS;
```



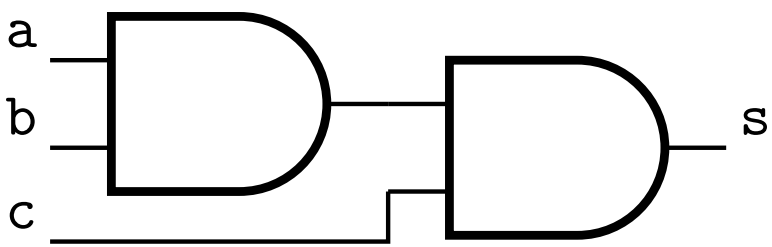
Méthode : Circuits combinatoires en VHDL



Signaux internes

On a souvent besoin de **signaux internes** pour décrire les connexions.

Non-visibles depuis l'entité, ils sont **déclarés dans l'architecture**.




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY and3 IS

  PORT (
    a : IN  STD_LOGIC;
    b : IN  STD_LOGIC;
    c : IN  STD_LOGIC;
    s : OUT STD_LOGIC);

END and3;

ARCHITECTURE archi OF and3 IS

  SIGNAL a_and_b : STD_LOGIC;

BEGIN

  a_and_b <= a AND b;
  s       <= a_and_b AND c;

END archi;
```

Fil rouge : Vote majoritaire

Implémenter en VHDL le vote majoritaire à 3 entrées. Grouper les trois signaux p, s et t en un seul SIGNAL avec l'opérateur de **concaténation**.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY maj3 IS




END maj3;
```

```
ARCHITECTURE archi OF maj3 IS



BEGIN




END archi;
```

Conception structurelle

Pour décrire un circuit **complexe**, on réutilise souvent des **blocs existants**.

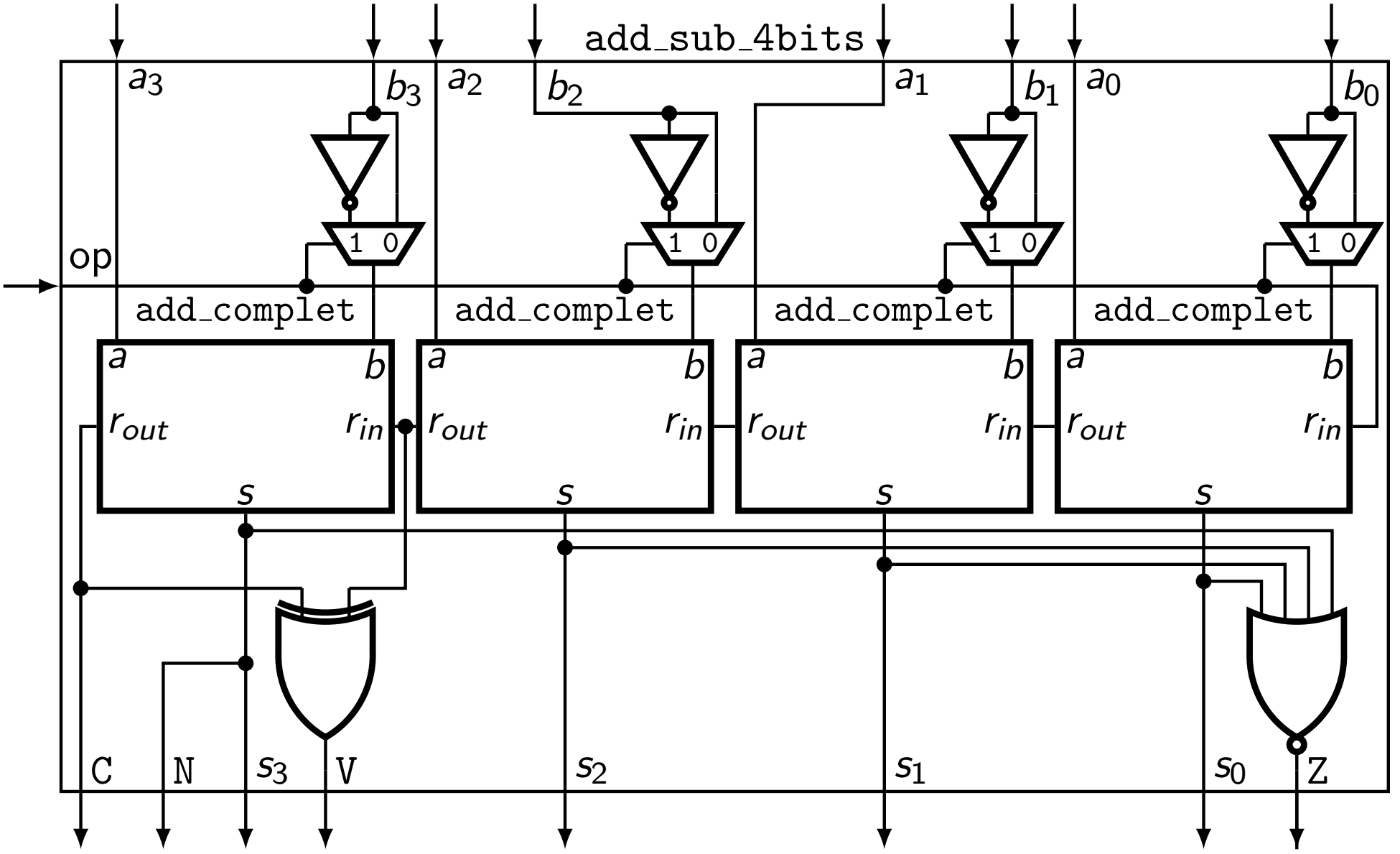
Par exemple, pour l'additionneur / soustracteur sur 4 bits :

- 4× additionneurs complets

En VHDL, on utilise les constructions suivantes :

- `COMPONENT` : bloc **existant** qu'on **déclare** puis qu'on **instancie**,
- `SIGNAL` : signal **interne**, non visible depuis l'`ENTITY`.

Add/sub avec drapeaux indicateurs d'état : blocs existants



Bloc élémentaire : additionneur complet

add_complet.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY add_complet IS

  PORT (
    a    : IN  STD_LOGIC;
    b    : IN  STD_LOGIC;
    rin  : IN  STD_LOGIC;
    s    : OUT STD_LOGIC;
    rout : OUT STD_LOGIC);

END add_complet;

ARCHITECTURE archi OF add_complet IS

BEGIN

  s    <= a XOR b XOR rin;
  rout <= (a AND b) OR (a AND rin) OR (b AND rin);

END archi;
```

Conception structurelle

add_sub_4bits.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY add_sub_4bits IS

  PORT (
    a  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    b  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    op : IN  STD_LOGIC;
    s  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
    Z  : OUT STD_LOGIC;
    N  : OUT STD_LOGIC;
    C  : OUT STD_LOGIC;
    V  : OUT STD_LOGIC);

END add_sub_4bits;

ARCHITECTURE archi OF add_sub_4bits IS
```

...

L'ENTITY est **classique** :

- nom,
- entrées,
- sorties.

Tout a lieu à l'intérieur de l'ARCHITECTURE, car ce n'est **pas visible de l'extérieur**.

Déclaration

On **déclare** les blocs basiques qu'on va utiliser avec le mot clé `COMPONENT`.

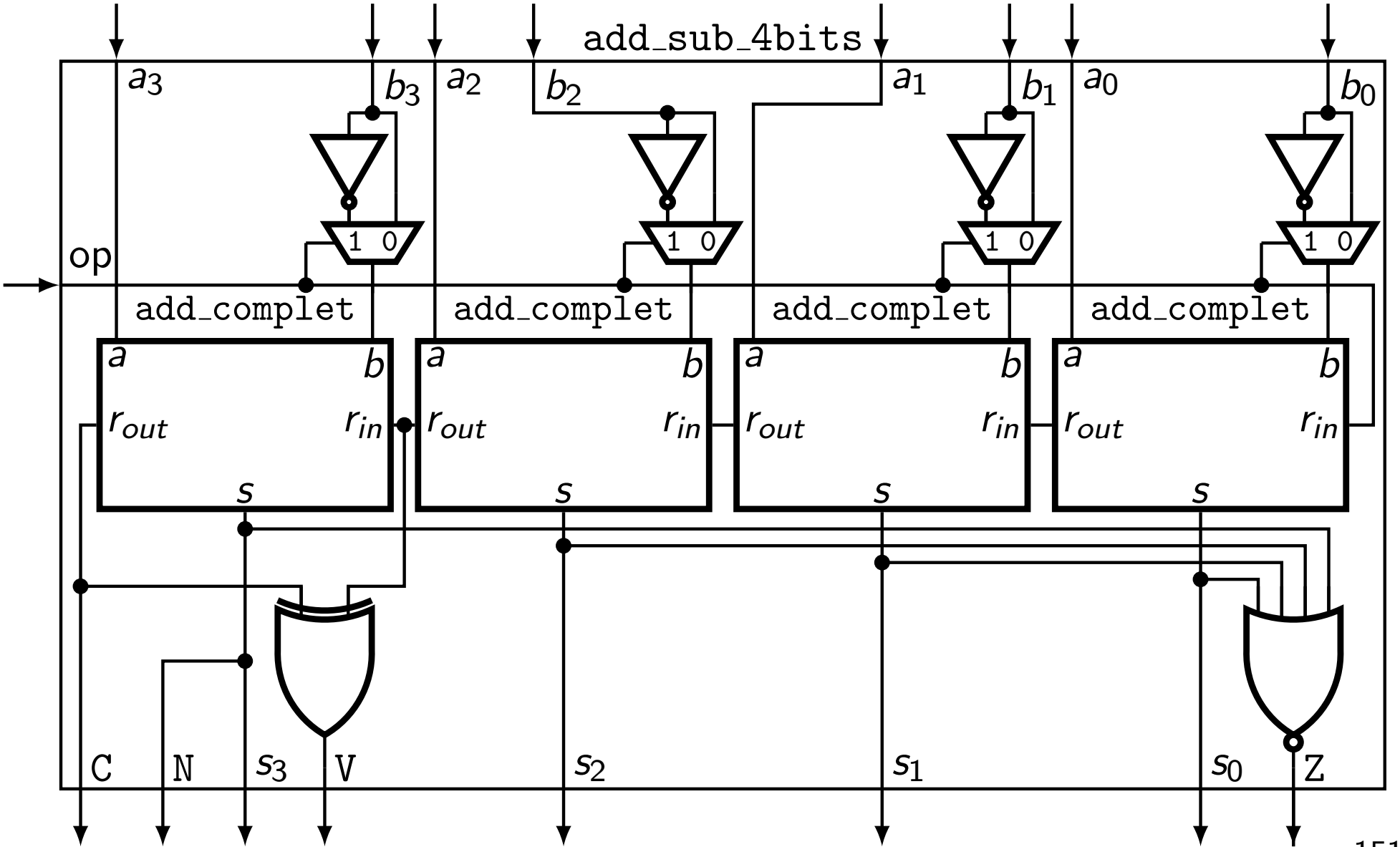
add_sub_4bits.vhd (suite)

```
ARCHITECTURE archi OF add_sub_4bits IS

  COMPONENT add_complet
    PORT (
      a    : IN  STD_LOGIC;
      b    : IN  STD_LOGIC;
      rin  : IN  STD_LOGIC;
      s    : OUT STD_LOGIC;
      rout : OUT STD_LOGIC);
  END COMPONENT;
```

Les `PORT` des blocs existants sont **recopiés à l'identique** à partir de l'entité originale.

Add/sub avec drapeaux indicateurs d'état : signaux internes

On identifie les **signaux internes** :



Signaux internes (suite)

On utilise le mot-clé SIGNAL pour déclarer les signaux internes.

Ils peuvent être de même type que les entrées et sorties :

- STD_LOGIC
- STD_LOGIC_VECTOR

add_sub_4bits.vhd (suite)

```
  END COMPONENT;

  -- Signaux internes (ni entree, ni sortie)
  SIGNAL rout0_rin1 : STD_LOGIC;
  SIGNAL rout1_rin2 : STD_LOGIC;
  SIGNAL rout2_rin3 : STD_LOGIC;
  SIGNAL inverse_B  : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL B_ou_inv_B : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL s_interne  : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL rout_3     : STD_LOGIC;

BEGIN
```

Instanciation

Après le BEGIN de l'ARCHITECTURE, on instancie les COMPONENT, une ou plusieurs fois, en **nommant** chaque instance pour **l'identifier**.

add_sub_4bits.vhd (suite)

```
add_complet_0 : add_complet
  PORT MAP(
    a    => a(0),
    b    => B_ou_inv_B(0),
    rin  => op,
    s    => s_interne(0),
    rout => rout0_rin1
    );

add_complet_1 : add_complet
  PORT MAP(
    a    => a(1),
    b    => B_ou_inv_B(1),
    rin  => rout0_rin1,
    s    => s_interne(1),
    rout => rout1_rin2
    );
```

On **instancie** un COMPONENT en nommant l'instance et on le "connecte" grâce au mot-clé PORT MAP.

```
nom_instance : nom_component
  PORT MAP (
    ES_component => connexion,
    ES_component => connexion,
    ES_component => connexion,
    ES_component => connexion);
```

La connexion se fait soit à :

- un signal interne,
- une entrée / sortie de l'entité.

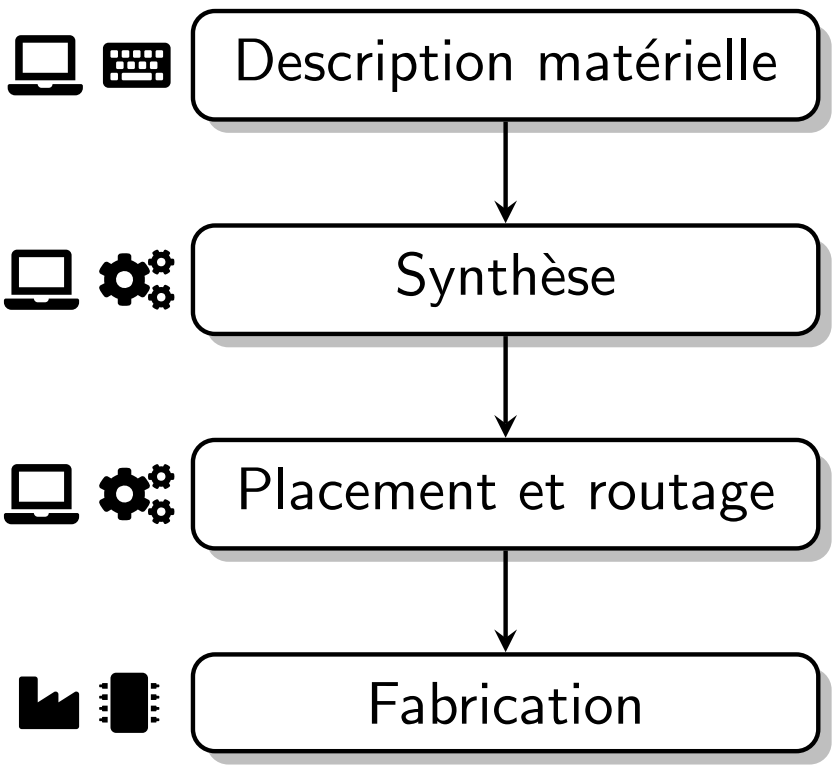
Affectations

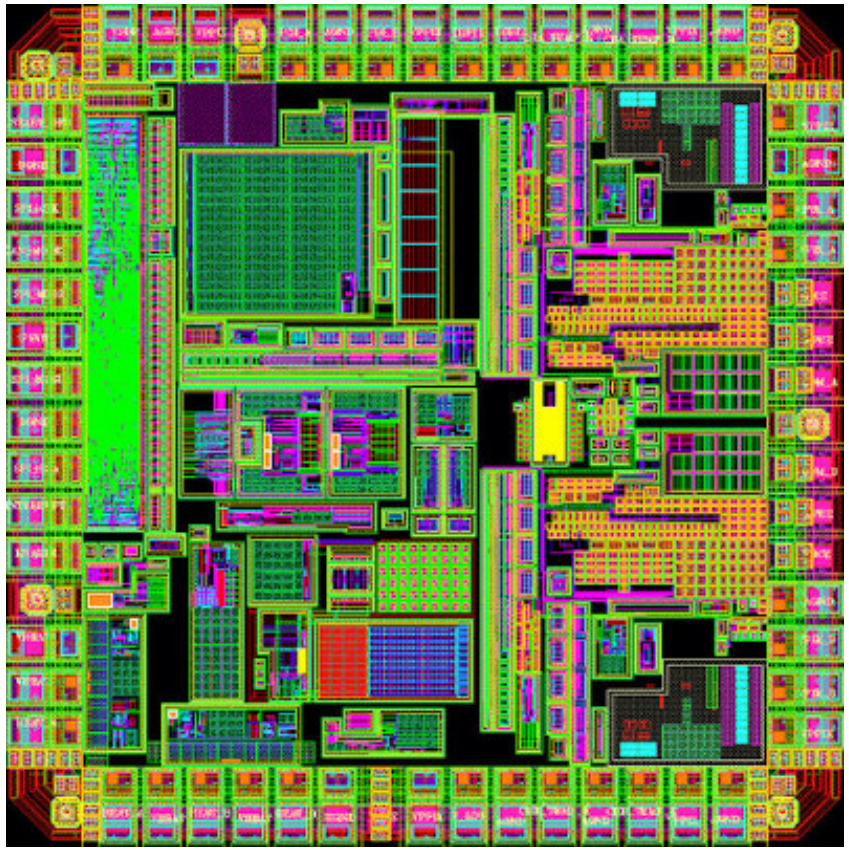
Après avoir instancié tous les COMPONENT nécessaires, on réalise si nécessaire les affectations supplémentaires.

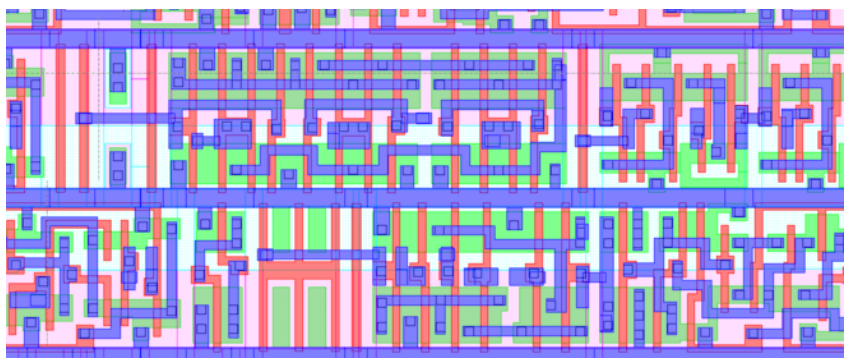
```
      a    => a(3),
      b    => B_ou_inv_B(3),
      rin  => rout2_rin3,
      s    => s_interne(3),
      rout => rout_3
      );

  s           <= s_interne;
  inverse_B   <= NOT(b);
  B_ou_inv_B <= inverse_B WHEN op = '1' ELSE b;

  Z <= NOT(s_interne(3) OR s_interne(2) OR s_interne(1) OR s_interne(0));
  N <= s_interne(3);
  C <= rout_3;
  V <= rout_3 XOR rout2_rin3;

END archi;
```

Implantation sur ASIC

Un ASIC (*Application-Specific Integrated Circuit*[4]) est un circuit conçu pour **une seule application**, décidée lors de la conception.



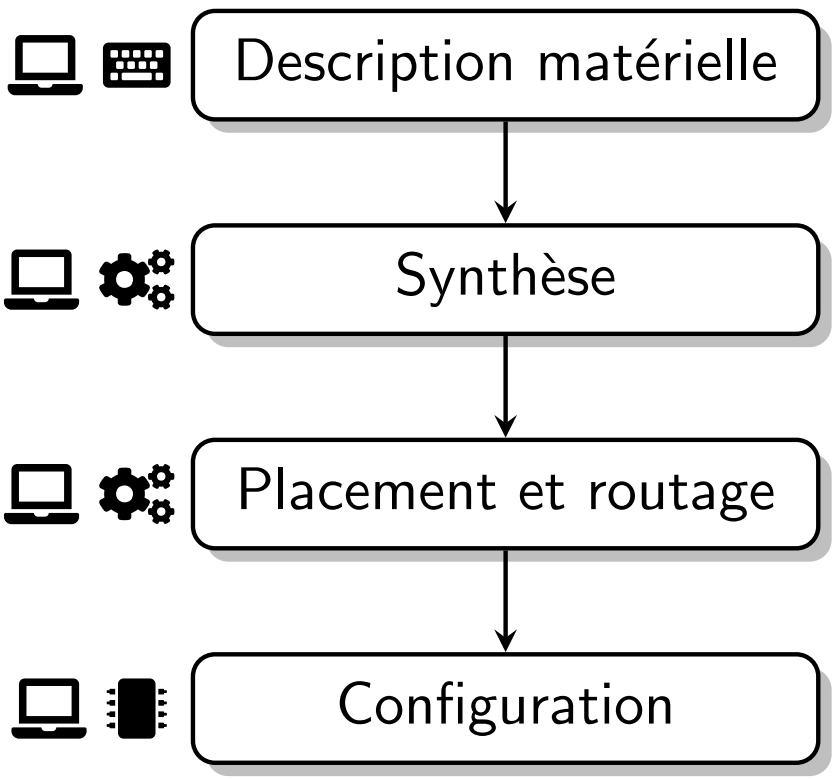


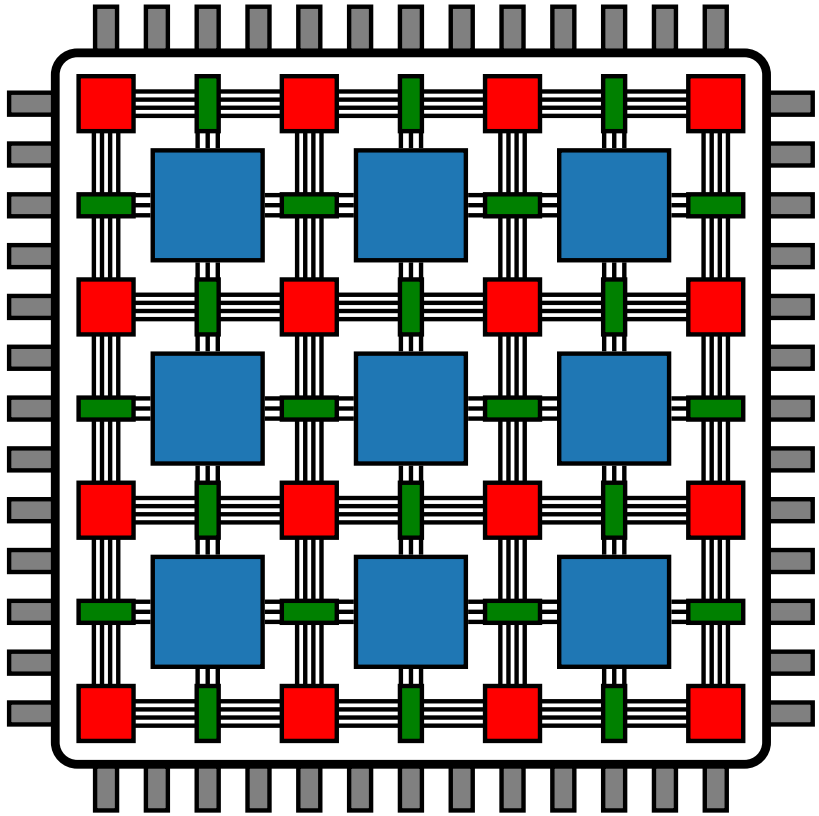


[4] Circuit Intégré Spécifique à une Application

Implantation sur FPGA

Un FPGA (*Field-Programmable Gate Array*[5]) est un circuit **reconfigurable**. Il inclut des éléments **génériques**, qui permettent de réaliser **n'importe quelle fonction logique**.





"**Configurer**" le FPGA consiste à fixer les interconnections et et à remplir les LUTs avec les bonnes valeurs.

[5] Matrice de Portes Programmable sur le Terrain

LUT

Une LUT (*Look-Up Table*[6]) est une **mémoire** de très petite taille stockant la **table de vérité** d'une fonction logique.

En écrivant des **valeurs différentes** dans la mémoire, on **change la table de vérité**, et donc la **fonction logique réalisée**.

Exemple :

$$s = a \cdot (b \vee c)$$

a, b et c sont reliés aux bits d'adresse de la mémoire :

- $a = addr(2)$
- $b = addr(1)$
- $c = addr(0)$

Adresse	Donnée
000	
001	
010	
011	
100	
101	
110	
111	

[6] Table de correspondance

Comparaison entre ASIC et FPGA

Critère	**ASIC**	**FPGA**
Coût de conception	élevé	moyen
Coût de fabrication / configuration	très élevé	nul[7]
Coût d'une pièce unique	très élevé	faible
Coût d'un lot	amorti	élevé
Délai entre conception et démarrage	mois	minutes
Performances	élevées	moyennes[8]

Table: Comparaison entre ASIC et FPGA selon différents critères

[7] Hormis le coût du programmateur

[8] Environ 10× moindres

Limite

Aucun des circuits conçus jusqu'à présent n'a de **mémoire**.

Les **sorties** sont **déterminées de manière unique par les entrées**.

Pour la plupart des circuits numériques, ce n'est pas le cas !

Chapitre 6

—

Logique séquentielle

Logique combinatoire VS séquentielle

Système **combinatoire**

Les **sorties** du système dépendent :

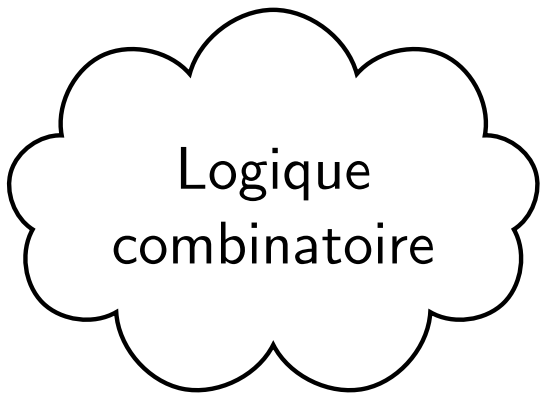
.

Exemples :

- Sur une télécommande :
  - 1, 2, 3, 4, ..., 9
- Boîte de vitesse manuelle.

Composants matériels :

- Portes logiques



Logique combinatoire VS séquentielle

Système **séquentiel**

Les **sorties** du système dépendent :

- de l'état .

L'état **suivant** du système dépend :

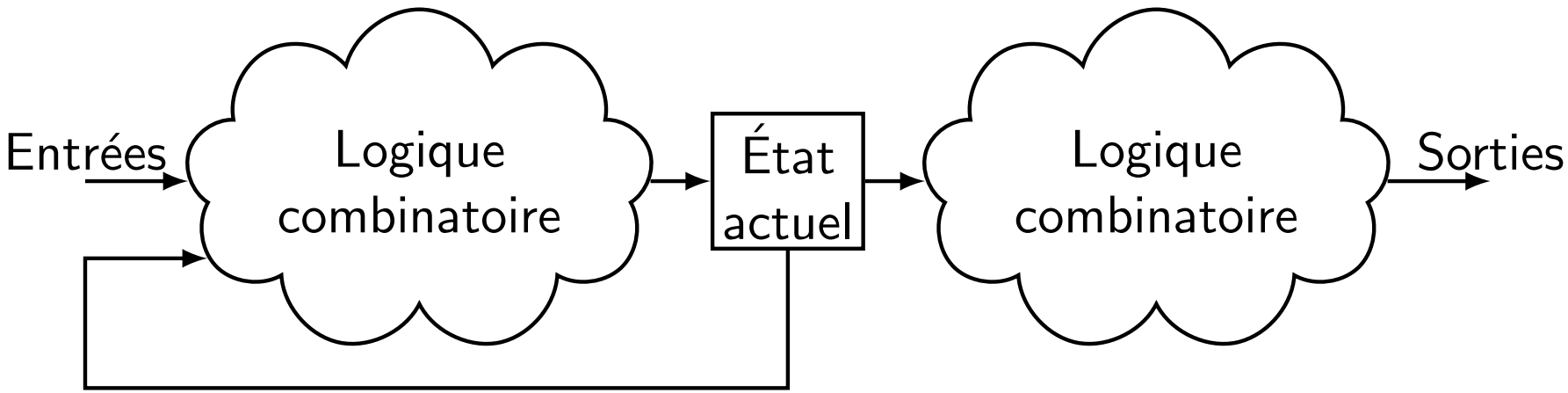
- des ,
- de l'état du système.

Exemples :

- Sur une télécommande :
  - ⏮ ⏭
- Palettes au volant.

Composants matériels :

- Portes logiques,
- Éléments de **mémorisation**.

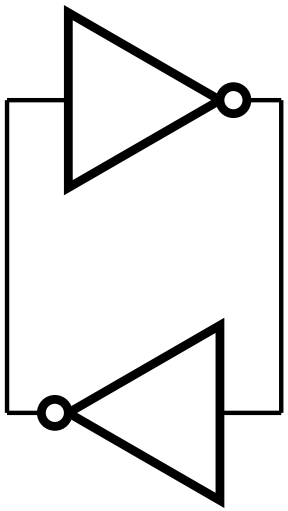


On peut ensuite **chaîner** ces éléments pour créer un système plus complexe.

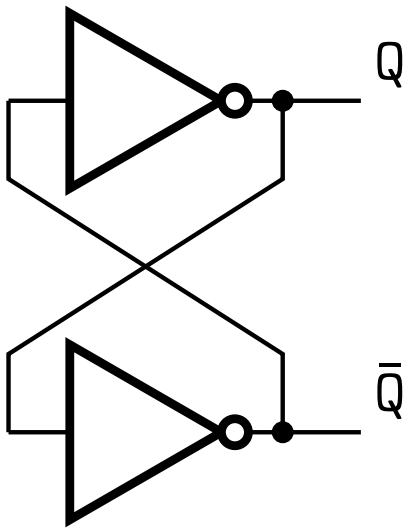
Élément de mémorisation basique

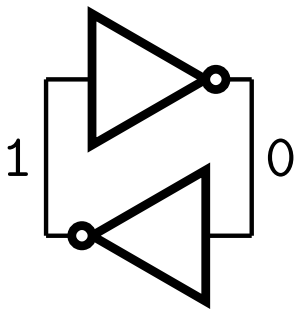
On veut un circuit **stable**, aussi **simple** que possible, **stockant** une valeur logique donnée sans avoir besoin de la **maintenir** par un moyen extérieur.

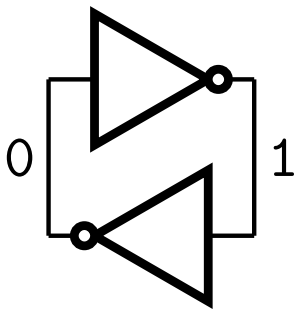
Idée initiale



Version plus lisible







Limite

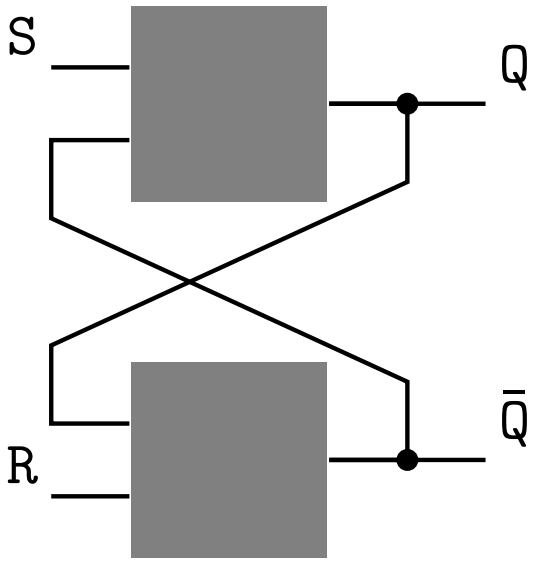
La valeur est bien **mémorisée**, mais on ne peut pas la **choisir**.

Le circuit n'a **pas d'entrée** pour fixer la valeur mémorisée.

Élément de mémorisation avec entrées de mise à 0/1

Table de vérité souhaitée

R	S	Q	$\bar{Q}$
0	0		
1	0		
0	1		



- R : reset (mise à 0),
- S : set (mise à 1),
- Q^- : valeur **précédente** de Q.

S	$\bar{Q}$	Q
0	0	
0	1	
1	0	
1	1	

R	Q	$\bar{Q}$
0	0	
0	1	
1	0	
1	1	

Fonction logique mystère

S / R	$\bar{Q}$	Q
0	0	
0	1	
1	0	
1	1	

$\bar{S}$ / $\bar{R}$	$\bar{Q}$	Q
0	0	
0	1	
1	0	
1	1	

Fonction

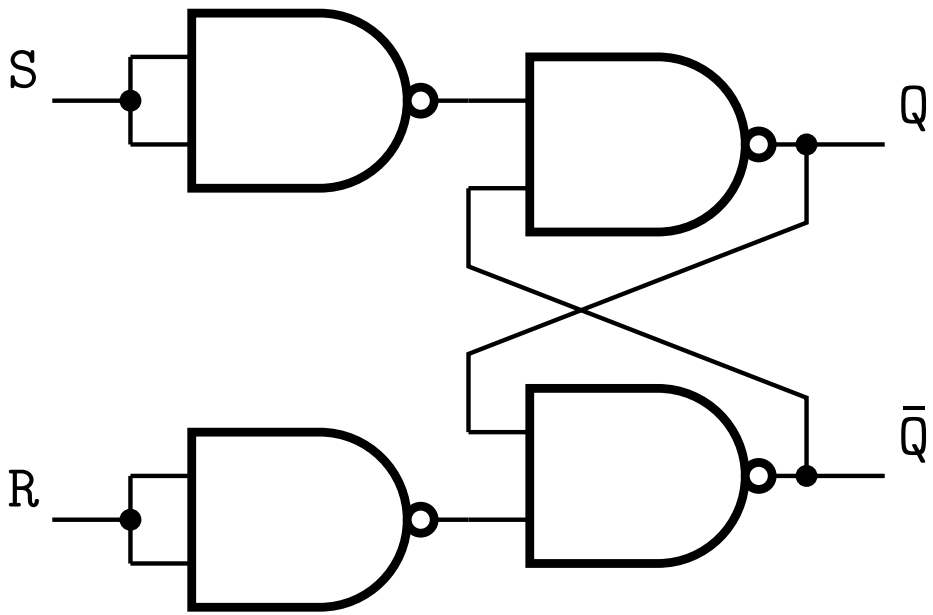
Verrou RS

R	S	Q	$\bar{Q}$
0	0	Q^-	$\bar{Q}^-$
1	0	0	1
0	1	1	0

→ Mémorisation de l'état précédent

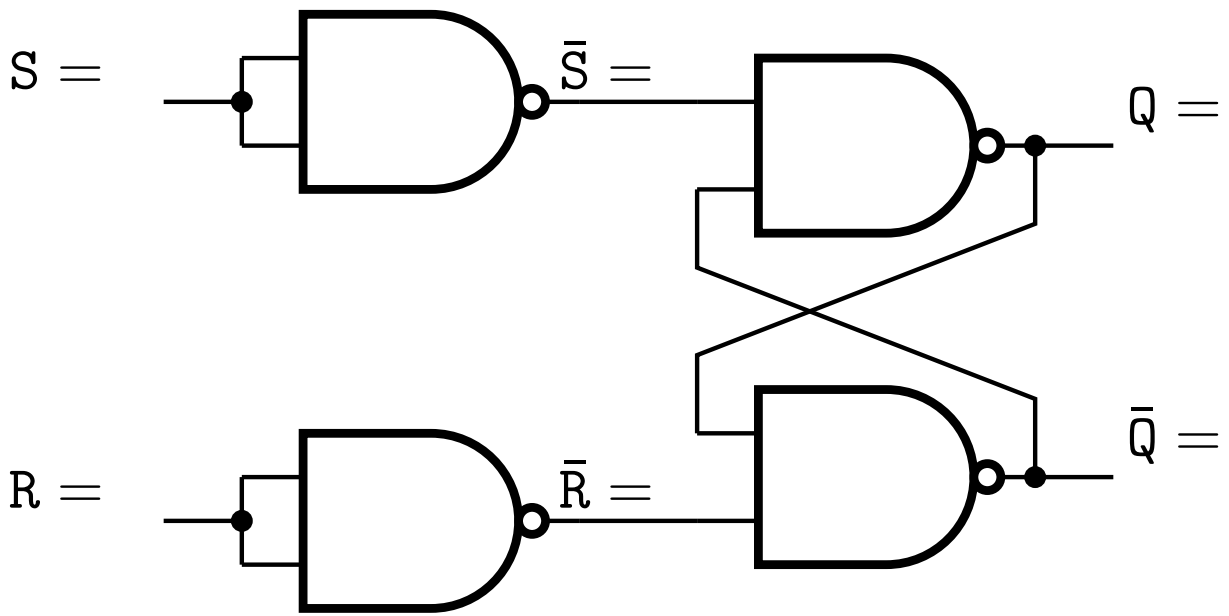
→ Forçage à 0

→ Forçage à 1



Limite

Le verrou RS possède un **état ambigu**, si $R = 1$ ET $S = 1$.

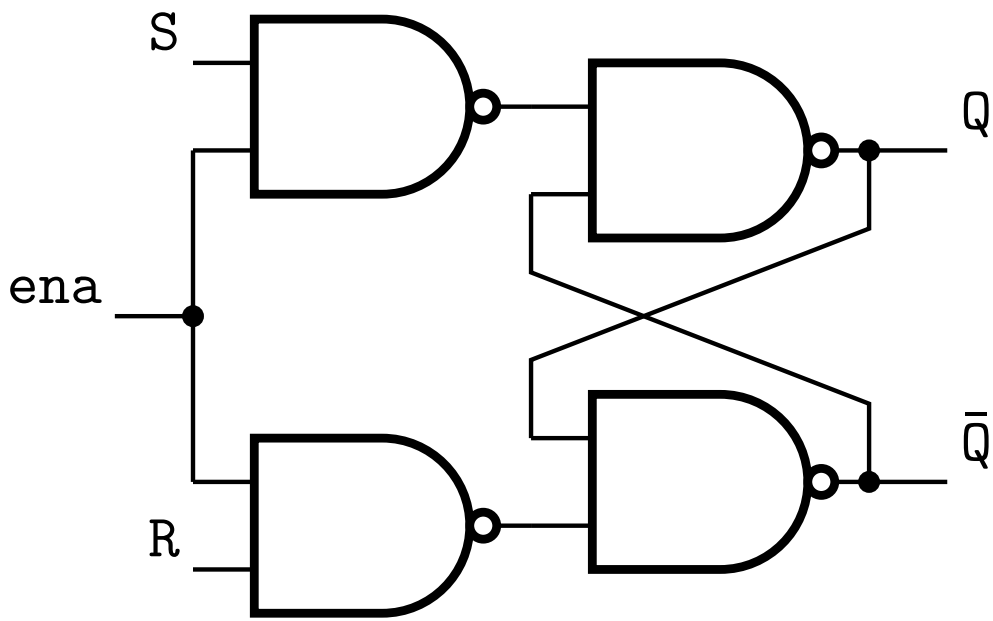


Contradiction !

Verrou RS à autorisation

On ajoute une entrée pour **autoriser** la prise en compte des signaux R et S.

ena	S	R	Q	$\bar{Q}$
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



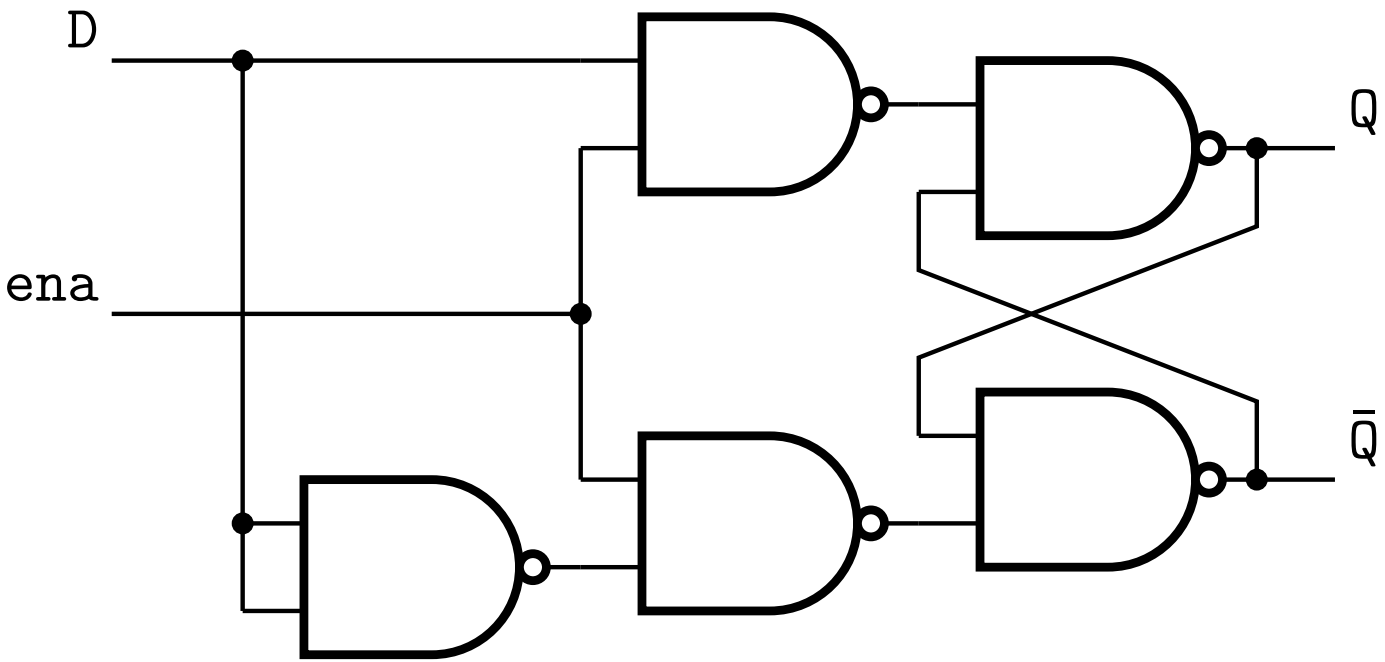
Limite

Il y a **toujours un état ambigu** ($ena = 1$, $R = 1$, $S = 1$)...

On peut faire en sorte que la condition $R \neq S$ soit **toujours vraie**[9] !

[9] La condition $R = S$ n'a pas de sens : pourquoi forcer à 0 et 1 simultanément ?

Verrou D à autorisation



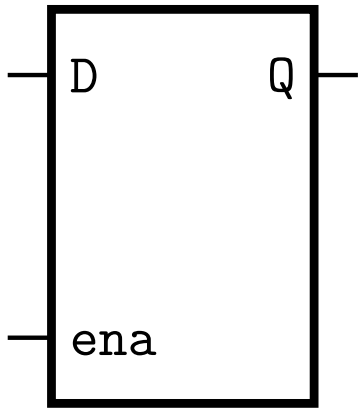
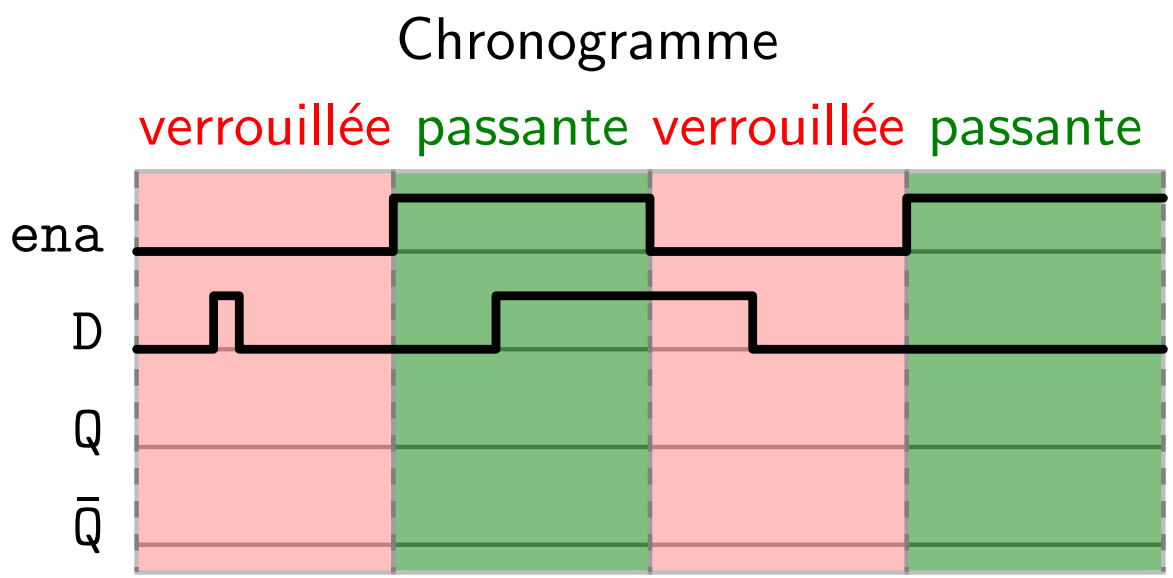


Table de vérité

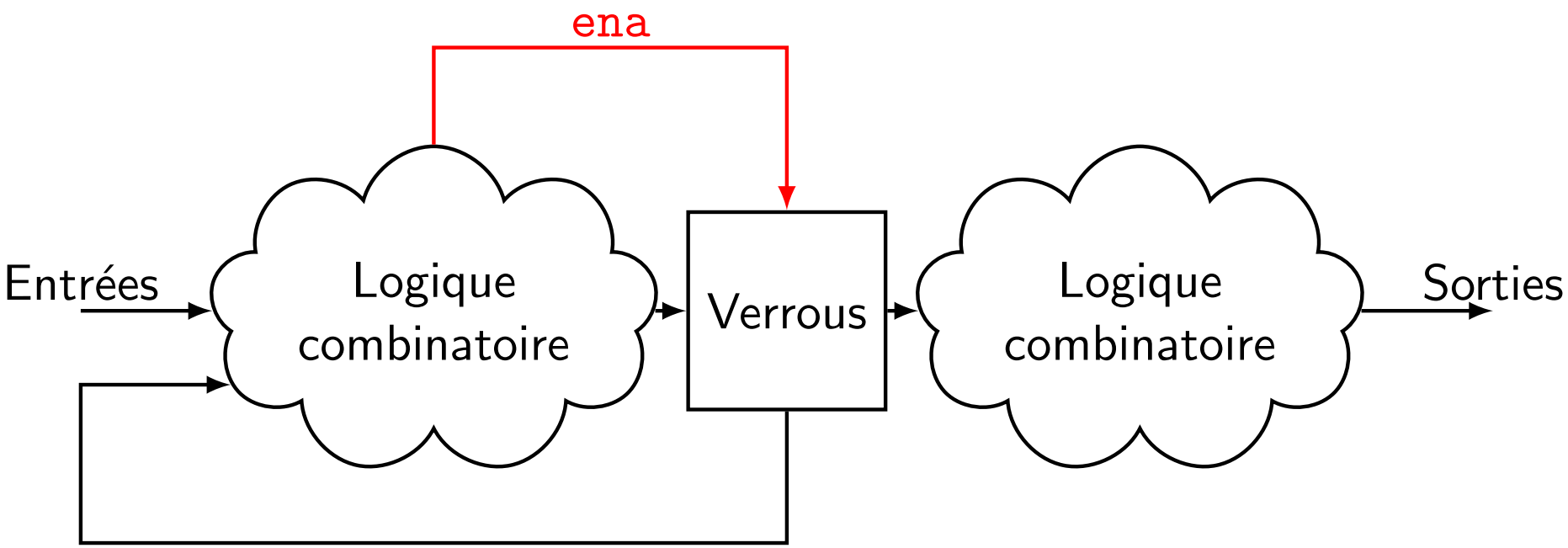
ena	D	Q	$\bar{Q}$
0	x		
1	x		

Chronogramme



Logique asynchrone

La logique **asynchrone** utilise les **verrous** comme éléments de mémorisation.



Il est **très difficile** de générer correctement les signaux `ena` :

- **quand** passer de 0 à 1 ?
- **combien de temps** rester à 1 ?

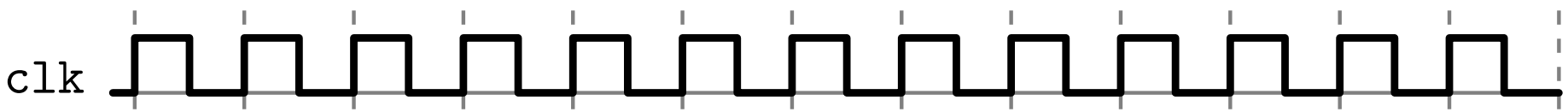
Nombreux **paramètres** : délais de propagation, température, tension, etc...

Logique synchrone

La logique **synchrone** résoud ces problèmes en activant **tous** les éléments de mémorisation du système **simultanément**, et pour une **durée très courte**.

On dispose pour cela d'un **signal d'horloge** `clk` :

- Fréquence fixée et stable,
- Rapport cyclique .

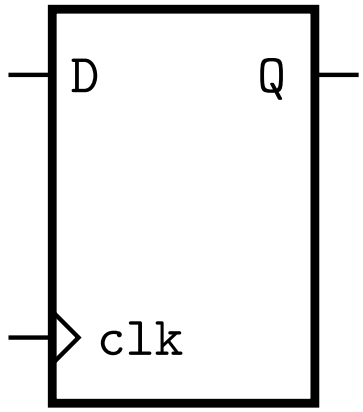


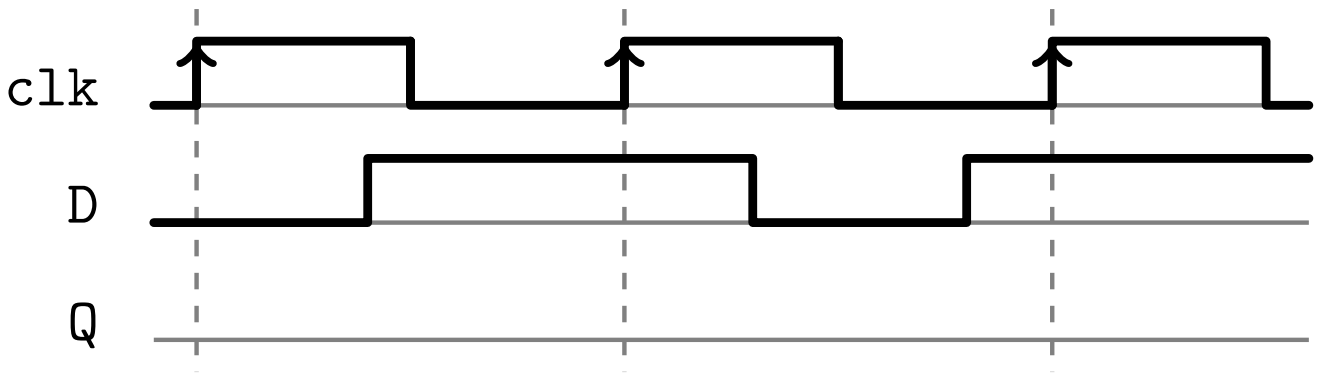
Bascule D

Le **verrou** D était sensible au **niveau** du signal `ena`.
La **bascule** D est sensible aux **fronts montants** du signal `clk`.

Table de vérité

clk	D	Q
0	x	
1	x	
↓	x	
↑	0	
↑	1	

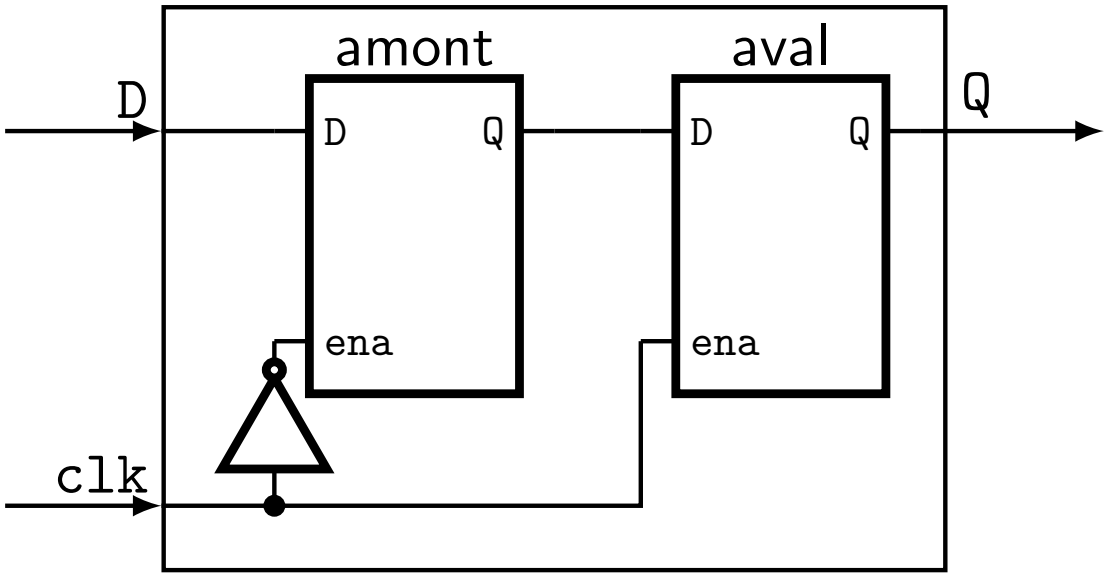


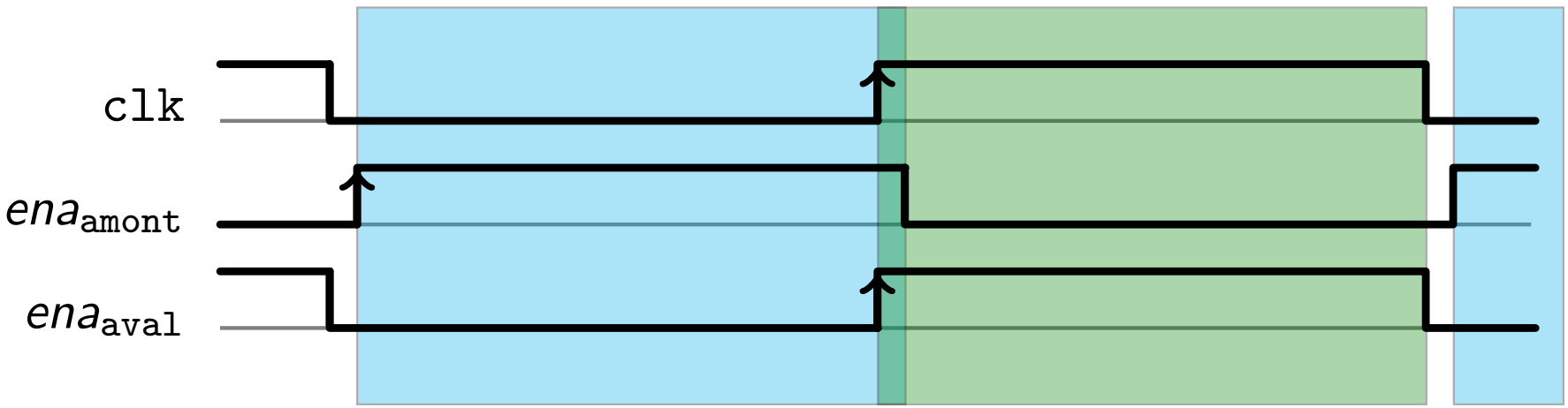


À **chaque front montant**, l'entrée `D` est **recopiée** sur la sortie `Q`.

Architecture interne d'une bascule D

Une bascule D est constituée de **deux verrous D** en série et d'un **inverseur**.





enregistrement dans le verrou amont

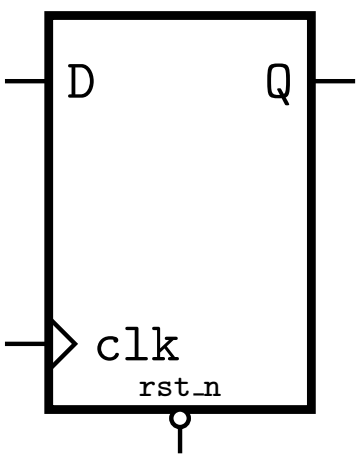
enregistrement dans le verrou aval

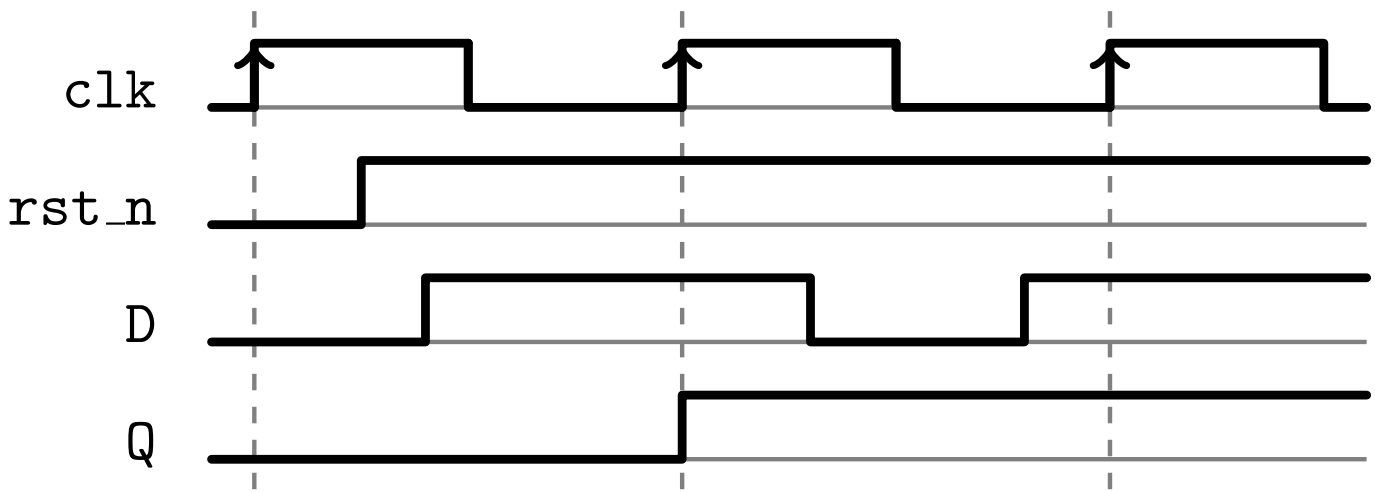
État initial

Afin de partir d'un **état intial connu**, on utilise un signal spécifique : **reset**. Le plus souvent, ce signal est :

- asynchrone,
- actif à l'état bas (d'où la notation `rst_n`).

Il est **prioritaire** sur toutes les autres entrées de la bascule `D`.

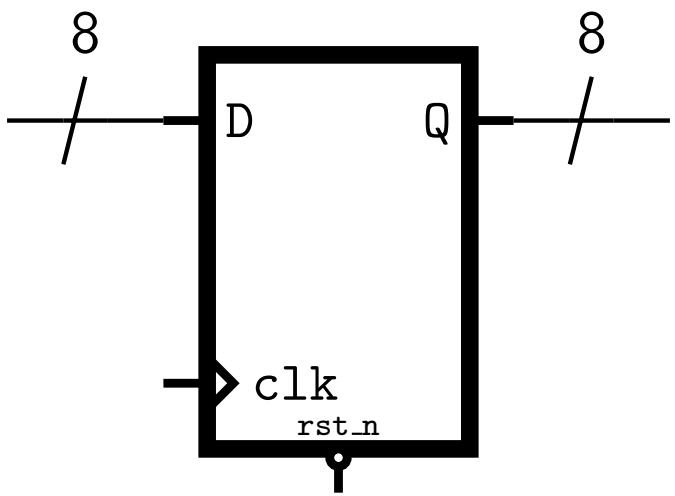




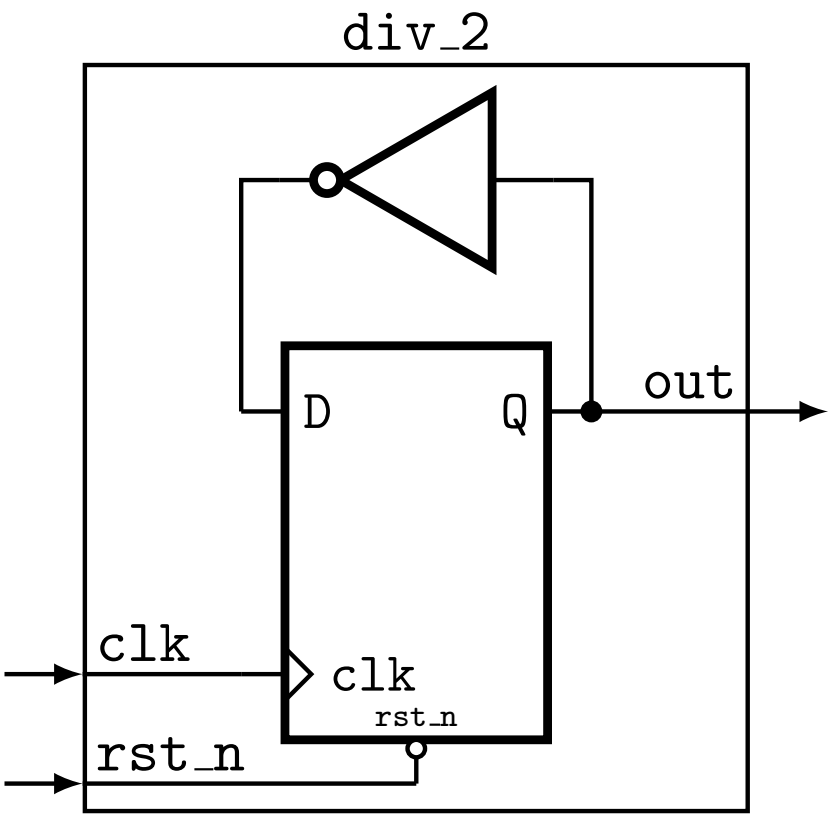
Registre

En accolant plusieurs bascules D, on crée un **registre**.
Cela permet de stocker une information de **plusieurs bits**.

Exemple : registre de 8 bits



Diviseur de fréquence par 2

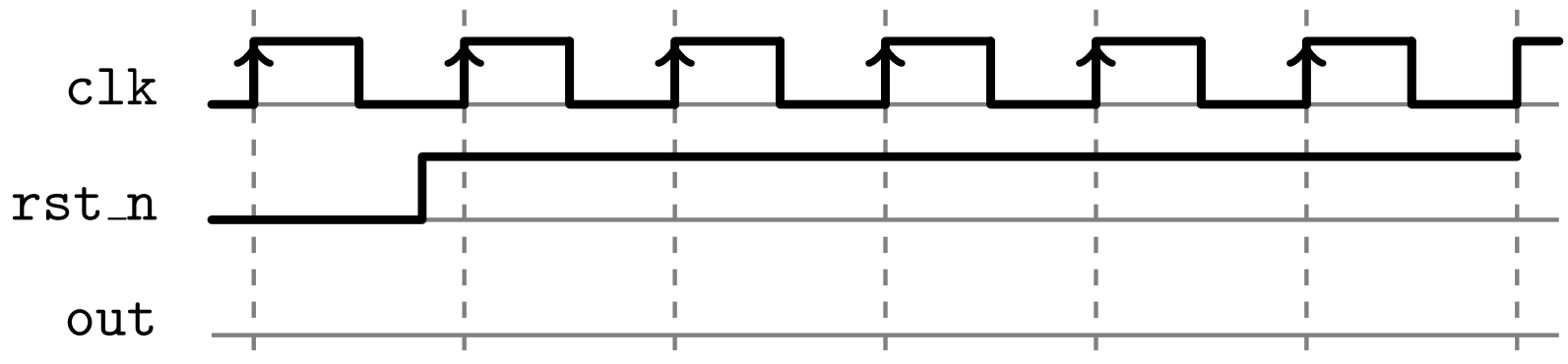


états possibles

À chaque **front montant d'horloge**, l'état actuel est **mis à jour**.

État actuel	**État futur**
Q = 0	Q =
Q = 1	Q =

Q	D
0	
1	

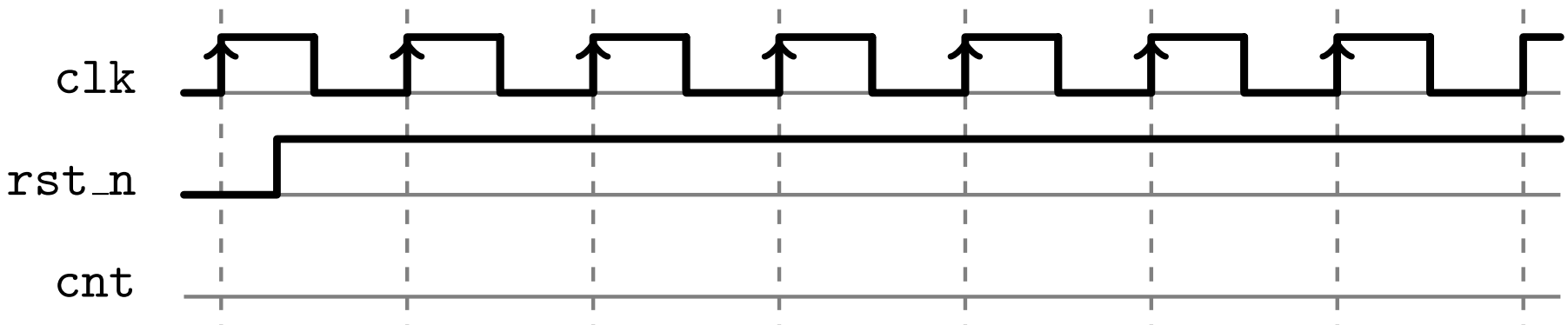


Compteur synchrone cyclique de 0 à 3

états possibles : bits.

Table de vérité

État actuel	État futur



Compteur synchrone cyclique de 0 à 4

états possibles : bits

Compteurs : variantes

Le compteur est un bloc basique essentiel en électronique numérique.
De nombreuses variantes existent pour des applications spécifiques :

- Compteur à autorisation,
- Compteur mono-coup à déclenchement,
- Compteur bi-directionnel,
- Compteur cyclique indiquant que le comptage est terminé,
- Compteur à maximum variable,
- Compteur à pas variable,
- etc.

Limite

Décrire des circuits séquentiels devient vite difficile en schéma...

Chapitre 7

—

Logique séquentielle en VHDL

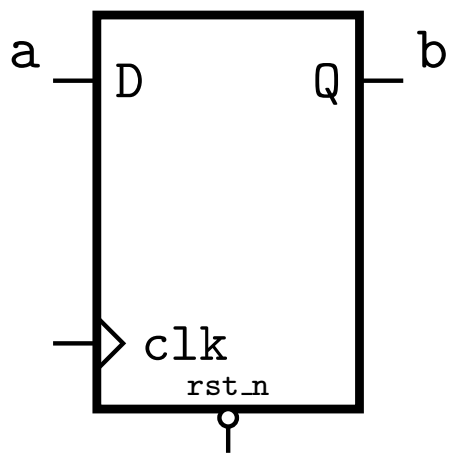
Description comportementale en VHDL

Contrairement à ce qu'on a fait jusqu'à présent pour la logique combinatoire, on **n'instancie pas** de bascules D.

On utilise les PROCESS du VHDL, pour décrire le **comportement** d'une partie du circuit : c'est une **description comportementale**.

C'est le synthétiseur (étape de synthèse) qui **infèrera** les bascules nécessaires.

```
PROCESS (clk, rst_n)
BEGIN
  IF rst_n = '0' THEN
    b <= '0';
  ELSE
    IF rising_edge(clk) THEN
      b <= a;
    END IF;
  END IF;
END PROCESS;
```



Construction d'un PROCESS

```
PROCESS (clk, rst_n)
BEGIN
  IF rst_n = '0' THEN
    b <= '0';
  ELSE
    IF rising_edge(clk) THEN
      b <= a;
    END IF;
  END IF;
END PROCESS;
```

```
PROCESS (liste de sensibilité)
BEGIN
  -- corps du process
END PROCESS;
```

Un PROCESS peut être **vu comme un petit "programme"**, exécuté dès qu'un des signaux de la **liste de sensibilité** change de **valeur**.

Attention

On ne fera que des PROCESS **séquentiels** dans ce cours, dont la liste de sensibilité incluera seulement :

- un signal d'horloge
- un signal de reset

On ne fera **pas de** PROCESS **combinatoires !**

“Exécution” d’un process

Si l’un des signaux de la liste de sensibilité **change**, le PROCESS est “exécuté”.

Les signaux affectés prennent leur valeur **à la fin** du PROCESS.

Les deux descriptions suivantes sont donc **équivalentes** :

```
PROCESS (clk, rst_n)
BEGIN
  IF rst_n = '0' THEN
    a <= '0';
  ELSE
    IF rising_edge(clk) THEN
      a <= b;
      a <= c;
    END IF;
  END IF;
END PROCESS;
```

```
PROCESS (clk, rst_n)
BEGIN
  IF rst_n = '0' THEN
    a <= '0';
  ELSE
    IF rising_edge(clk) THEN
      a <= c;
    END IF;
  END IF;
END PROCESS;
```

Compteur synchrone de 0 à 3 en VHDL

```
ARCHITECTURE comportementale OF compteur IS

  SIGNAL val : STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN

  PROCESS (clk, rst_n)
  BEGIN
    IF rst_n = '0' THEN
      val <= (OTHERS => '0');
    ELSE
      IF rising_edge(clk) THEN
        val(0) <= NOT(val(0));
        val(1) <= val(0) XOR val(1);
      END IF;
    END IF;
  END PROCESS;

  c <= val;

END comportementale;
```

Limite

Difficile de reconnaître un compteur ici...

On aimerait, comme en programmation classique (comme en C), disposer de quelques fonctions “standard”.

Opérateurs arithmétiques en VHDL

En tête :

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
```

La bibliothèque `numeric_std` fournit les opérateurs suivants :

- `+`
- `-`
- `*`
- `=`
- `>`
- `>=`
- `/=`
- `<`
- `<=`

On les utilise en "convertissant" un signal de type `STD_LOGIC_VECTOR` en :

- `UNSIGNED`
- `SIGNED`

en fonction de la manière dont on souhaite **interpréter** la donnée binaire.
On **re-convertit** ensuite le résultat en `STD_LOGIC_VECTOR`.

Compteur synchrone de 0 à 3 en VHDL

```
BEGIN

  PROCESS (clk, rst_n)
  BEGIN
    IF rst_n = '0' THEN
      val <= (OTHERS => '0');
    ELSE
      IF rising_edge(clk) THEN
        val <= STD_LOGIC_VECTOR(UNSIGNED(val) + 1);
      END IF;
    END IF;
  END PROCESS;

  c <= val;

END comport;
```

Compteurs

On peut imaginer de nombreuses variations autour d'un compteur simple :

- valeur maximale constante,
- valeur maximale donnée en entrée,
- commande de remise à zéro,
- direction de comptage : incrémenter ou décrémenter,
- arrêt en fin de cycle ou rebouclage,
- autorisation de comptage.

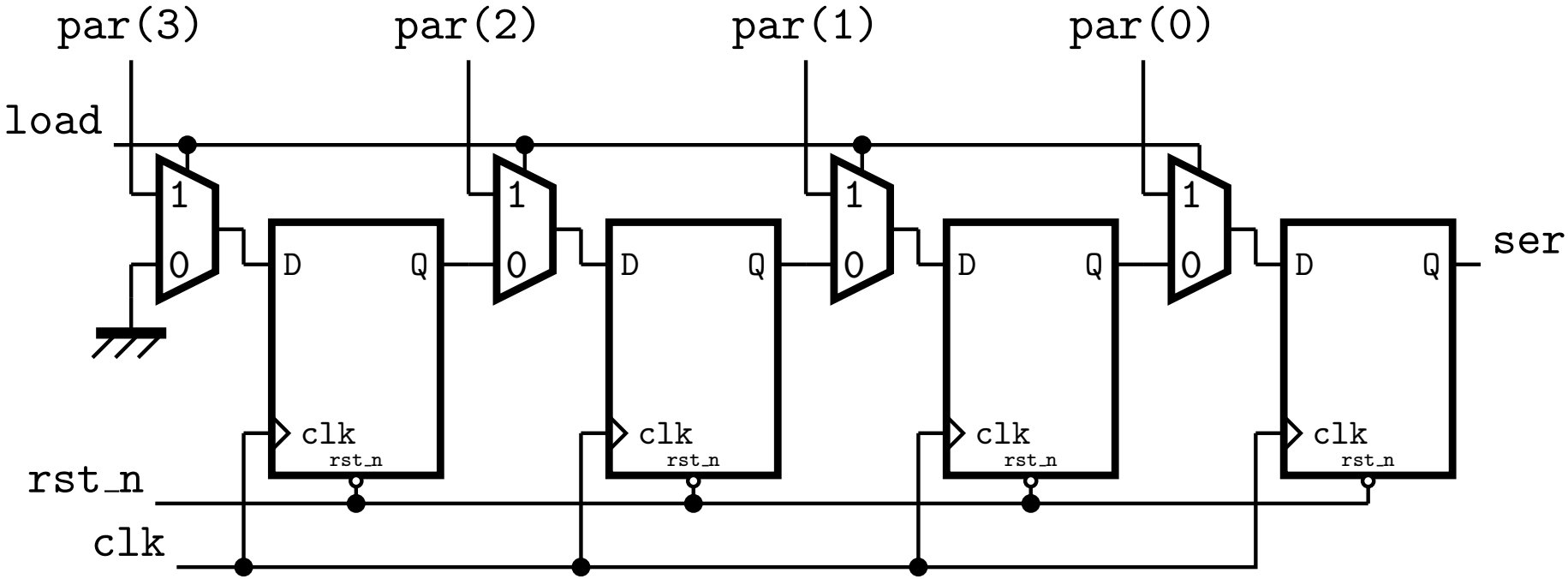
Compteur cyclique avec autorisation et direction

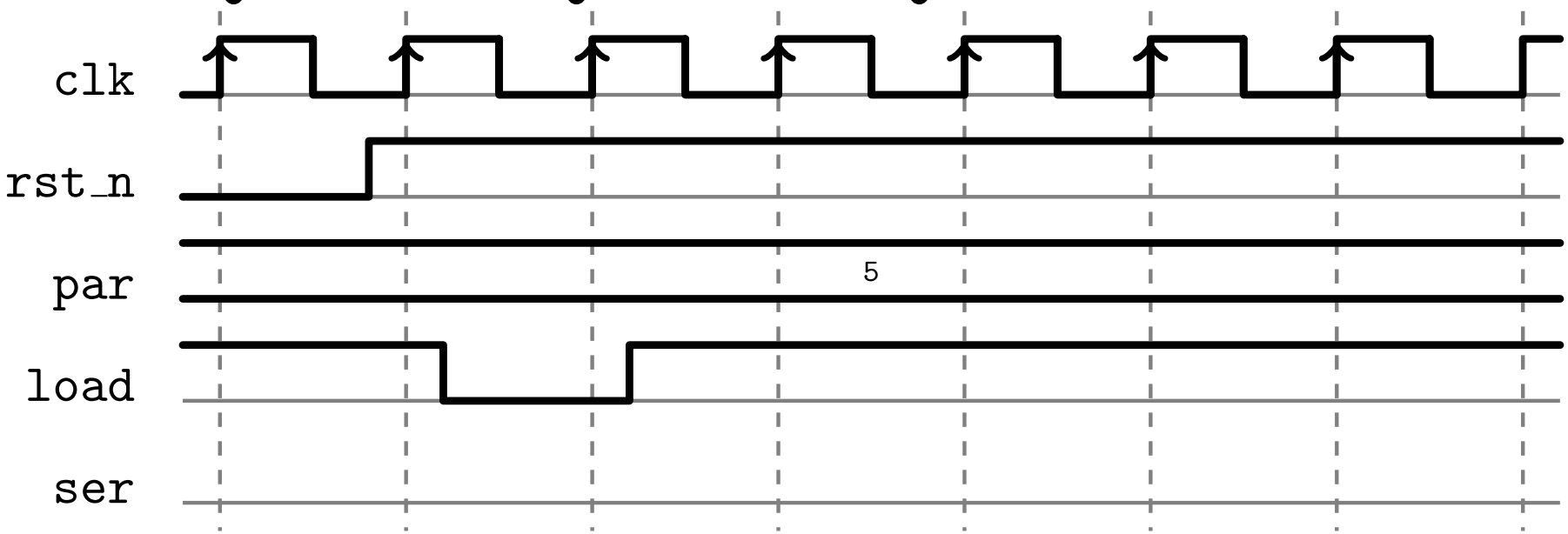
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY compteur_ena_dir IS

  PORT (
    clk   : IN  STD_LOGIC;
    rst_n : IN  STD_LOGIC;
    ena   : IN  STD_LOGIC;
    dir   : IN  STD_LOGIC;
    val   : OUT STD_LOGIC);

END compteur_ena_dir;

ARCHITECTURE comport OF compteur_ena_dir IS

  CONSTANT max : STD_LOGIC_VECTOR(3 DOWNTO 0) := "1010";

  SIGNAL val_interne : STD_LOGIC_VECTOR(3 DOWNTO 0);
```

Compteur cyclique avec autorisation et direction (suite)

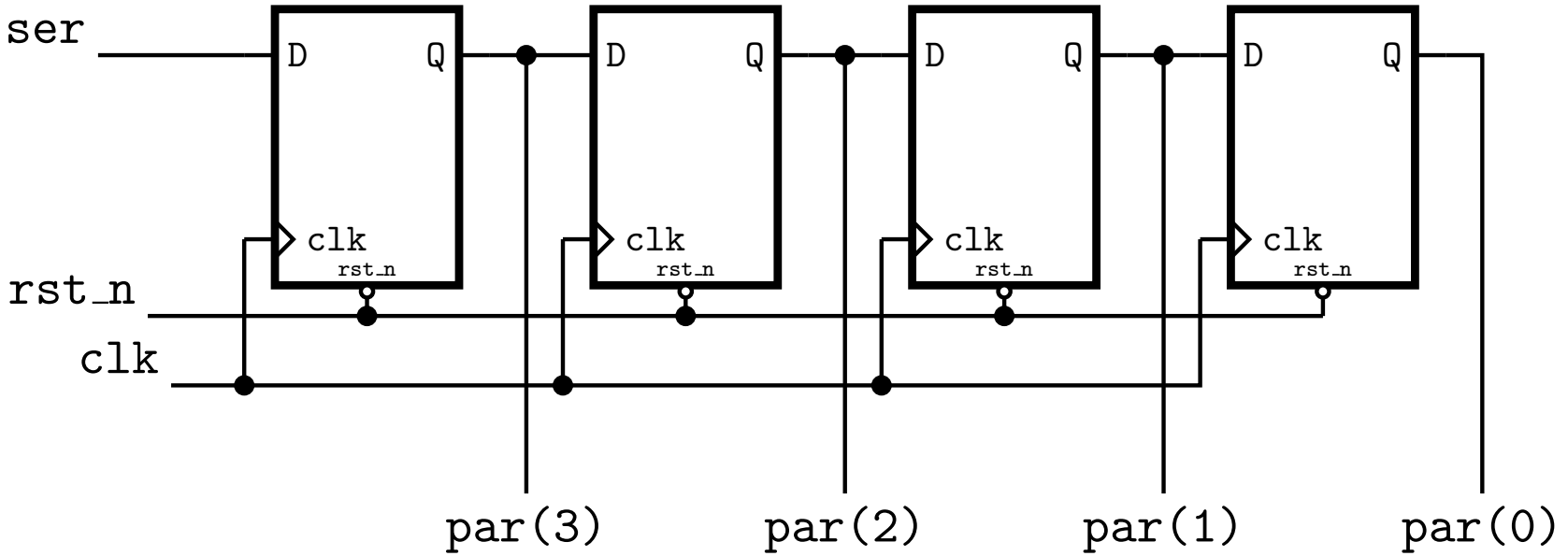
```
BEGIN

  PROCESS (clk, rst_n)
  BEGIN
    IF rst_n = '0' THEN
      val_interne <= (OTHERS => '0');
    ELSE
      IF rising_edge(clk) THEN
        IF ena = '1' THEN
          IF dir = '1' THEN
            IF val_interne = max THEN
              val_interne <= (OTHERS => '0');
            ELSE
              val_interne <= STD_LOGIC_VECTOR(UNSIGNED(val_interne) + 1);
            END IF;
          ELSE
            IF UNSIGNED(val_interne) = 0 THEN
              val_interne <= max;
            ELSE
              val_interne <= STD_LOGIC_VECTOR(UNSIGNED(val_interne) - 1);
            END IF;
          END IF;
        END IF;
      END IF;
    END IF;
  END PROCESS;

  val <= val_interne;

END comport;
```

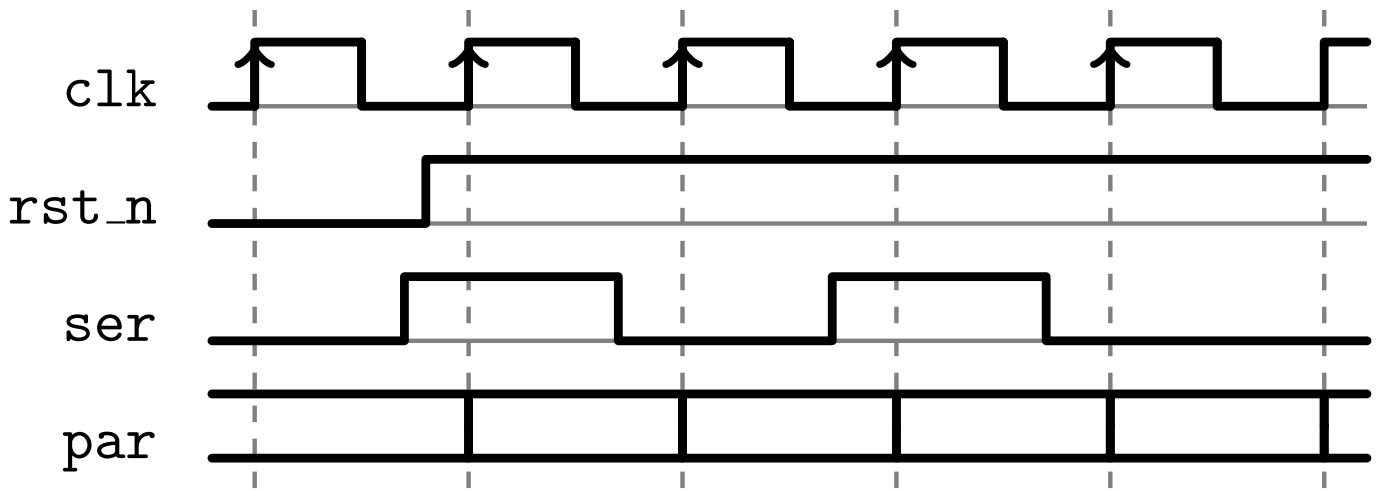
Convertisseur parallèle-série (*sérialiseur*)





Convertisseur série-parallèle (*désérialiseur*)





Sérialiseur en VHDL

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY serialiseur IS

  PORT (
    clk   : IN  STD_LOGIC;
    rst_n : IN  STD_LOGIC;
    par   : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    load  : IN  STD_LOGIC;
    ser   : OUT STD_LOGIC);

END serialiseur;

ARCHITECTURE comportement OF serialiseur IS

  SIGNAL val : STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```vhdl
BEGIN

  PROCESS (clk, rst_n)
  BEGIN
    IF rst_n = '0' THEN
      val <= (OTHERS => '0');
    ELSE
      IF rising_edge(clk) THEN
        IF load = '0' THEN
          val <= par;
        ELSE
          val <= '0' & val(3 DOWNTO 1);
        END IF;
      END IF;
    END IF;
  END PROCESS;

  ser <= val(0);

END comportement;
```

Désérialiseur en VHDL

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY deserialiseur IS

  PORT (
    clk   : IN  STD_LOGIC;
    rst_n : IN  STD_LOGIC;
    ser   : IN  STD_LOGIC;
    par   : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));

END deserialiseur;

ARCHITECTURE comportement OF deserialiseur IS

  SIGNAL val : STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```vhdl
BEGIN

  PROCESS (clk, rst_n)
  BEGIN
    IF rst_n = '0' THEN
      val <= (OTHERS => '0');
    ELSE
      IF rising_edge(clk) THEN
        val <= ser & val(3 DOWNTO 1);
      END IF;
    END IF;
  END PROCESS;

  par <= val;

END comportement;
```

Chapitre 8

—

Machine d'états

Limite

Tous les designs séquentiels vus jusqu'à présent
sont assez "rigides" et limités.

On aimerait avoir une méthode de conception généraliste
applicable à beaucoup de systèmes séquentiels.

On aimerait que cette méthode soit facile à mettre en œuvre en VHDL.

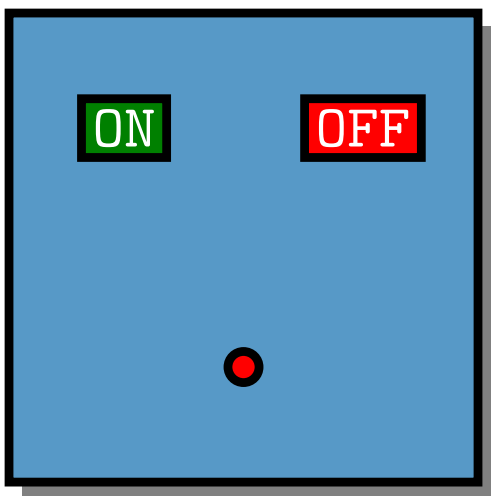
Machines d'états : description graphique

Un système séquentiel peut être décrit par :

1. des **états** possibles, **nommés**,
2. des **transisitions** possibles entre certains états,
3. des **conditions** sous lesquelles la transition entre deux états a lieu,
4. des **valeurs de sortie** associées à chaque état.

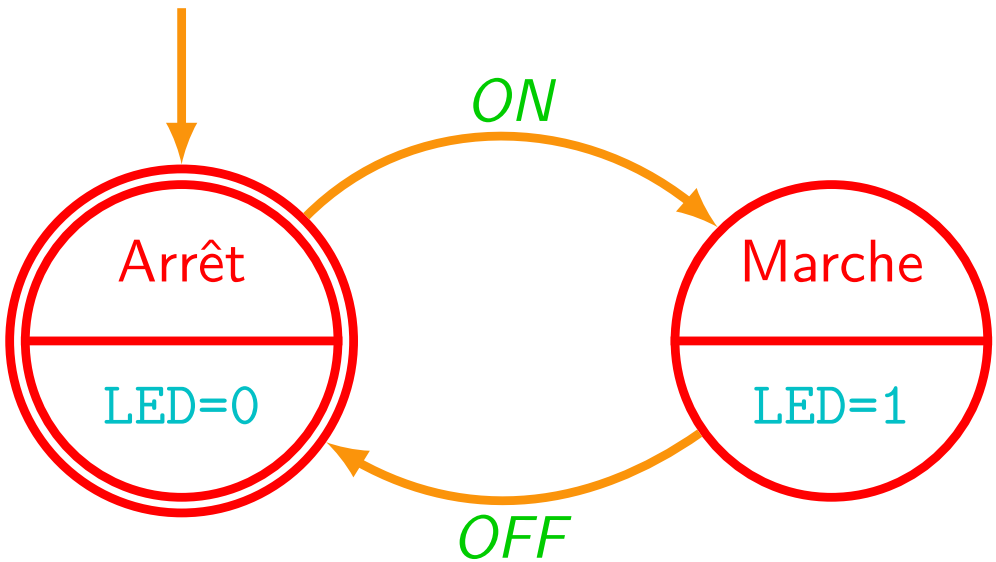
On peut représenter cela sous la forme d'un **diagramme états-transitions**.

Exemple : appuyer sur `ON` allume l'apapreil, appuyer sur `OFF` l'éteint.
La LED est allumée lorsque l'appareil est en marche.



Transitions implicites

La plupart du temps, on ne représente que les transitions **"utiles"**.
Les autres sont **implicites**, et signifient que l'on reste dans **l'état actuel**.



Exercice : Contrôleur pour vélo électrique

On souhaite concevoir un **contrôleur pour le moteur** d'un vélo électrique. Le moteur dispose d'une **entrée V**, sur 2 bits, définissant les niveaux d'assistance : aucune (00), basse (01), haute (10) ou maximale (11).

Le contrôleur dispose de **quatre modes** différents :

- **Off** : pas d'assistance,
- **Coast** : assistance basse,
- **Speed** : assistance haute,
- **Boost** : assistance maximale.

On contrôle le mode par l'intermédiaire de **deux boutons** connectés à deux entrées :

- `P` : passer au mode plus puissant,
- `M` : passer au mode moins puissant.

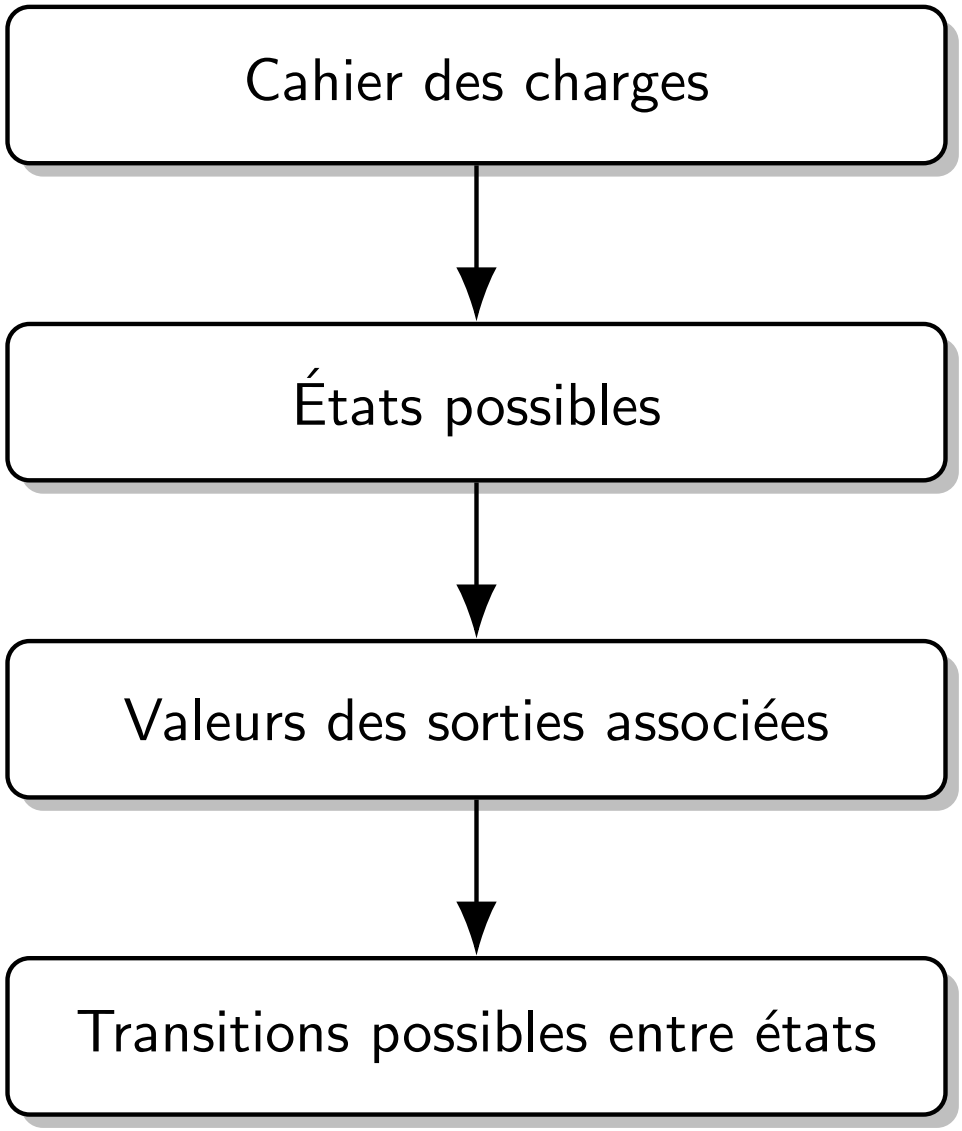
On dispose en outre d'un bouton d'arret d'urgence connecté à une entrée `S` permettant, depuis n'importe quel mode, de couper l'assistance, c'est à dire de repasser au mode **Off**.

Exercice : représenter le contrôleur, le moteur et leur connexion, puis dessiner le diagramme états-transitions du contrôleur.

Exercice : Contrôleur pour vélo électrique

Exercice : Contrôleur pour vélo électrique

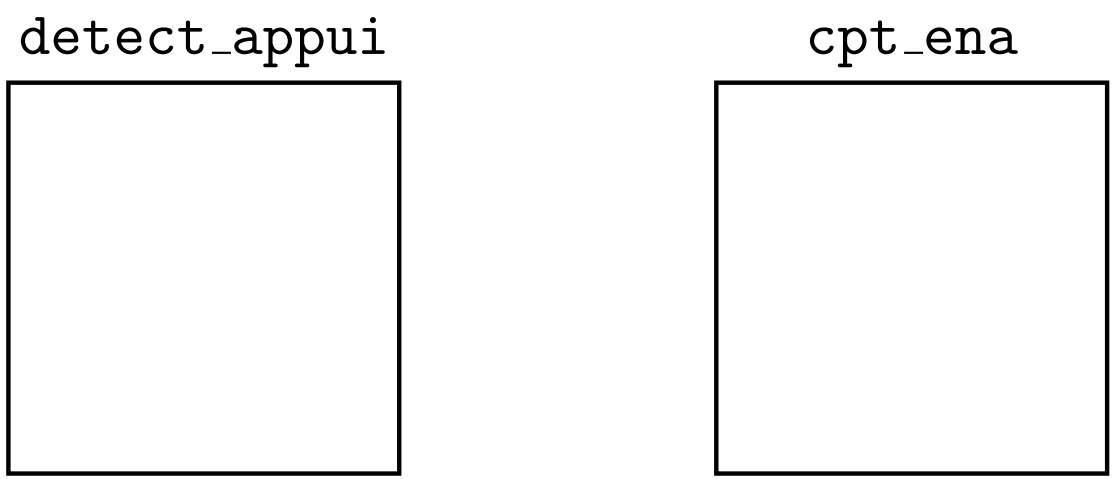
Méthode : Description graphique d'une FSM



Exercice : Détecteur d'appui

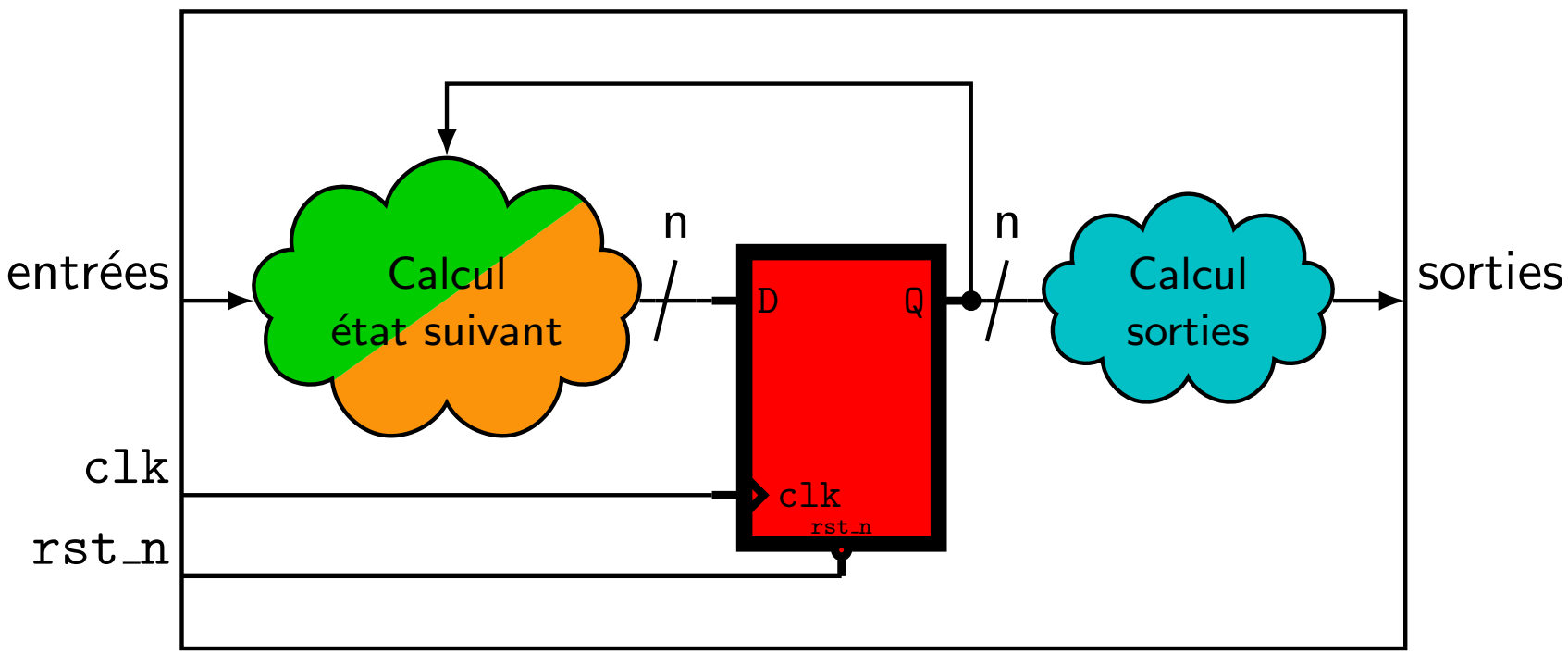
Le bouton start/stop d'un chronomètre démarre ou suspend le comptage. (C'est donc bien un système **séquentiel**)
On souhaite concevoir un détecteur d'appui sur le bouton start/stop, qui autorisera ou non le comptage.

1. dessiner le schéma de connexion entre deux blocs :
   - le détecteur d'appui,
   - le compteur avec autorisation de comptage.
2. proposer un diagramme état-transitions pour ce détecteur.

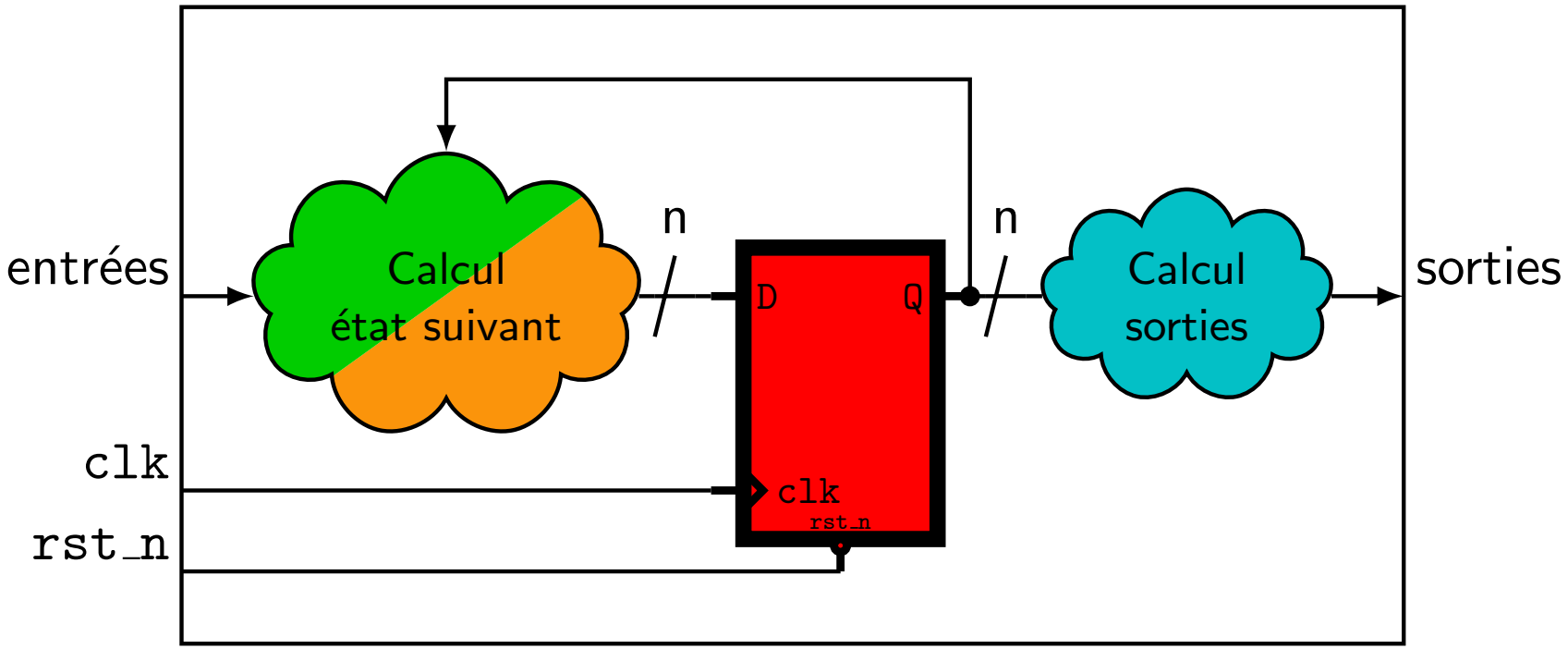


Architecture matérielle d'une machine d'états

1. des **états** possibles, **nommés**,
2. des **transisitions** possibles entre certains états,
3. des **conditions** sous lesquelles la transition entre deux états a lieu,
4. des **valeurs de sortie** associées à chaque état.



Architecture matérielle d'une machine d'états



- Le **registre d'état** stocke l'état **actuel**, sur n bits
- La **logique combinatoire de calcul de l'état suivant** détermine l'état **suivant** à partir de l'état **actuel** et des **entrées**,
- La **logique combinatoire de calcul des sorties** calcule les **sorties** à partir de l'état **actuel**.
- Le signal `rst_n` place la machine d'états dans son **état initial**,
- Le signal `clk` commande la **mise à jour périodique de l'état actuel**.

Chapitre 9

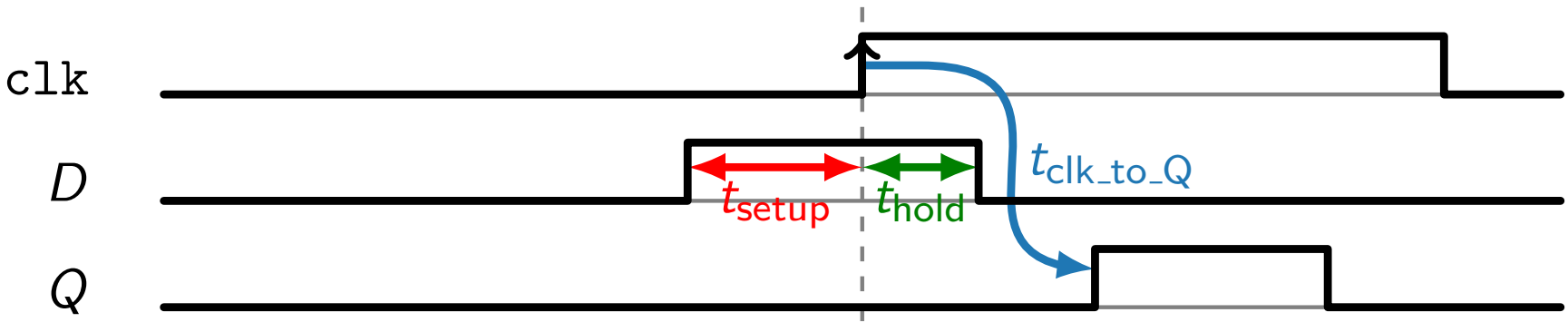
—

Machine d'états en VHDL

Contraintes temporelles pour le fonctionnement des bascules

Pour qu'une bascule D fonctionne correctement, deux **contraintes temporelles** doivent être respectées :

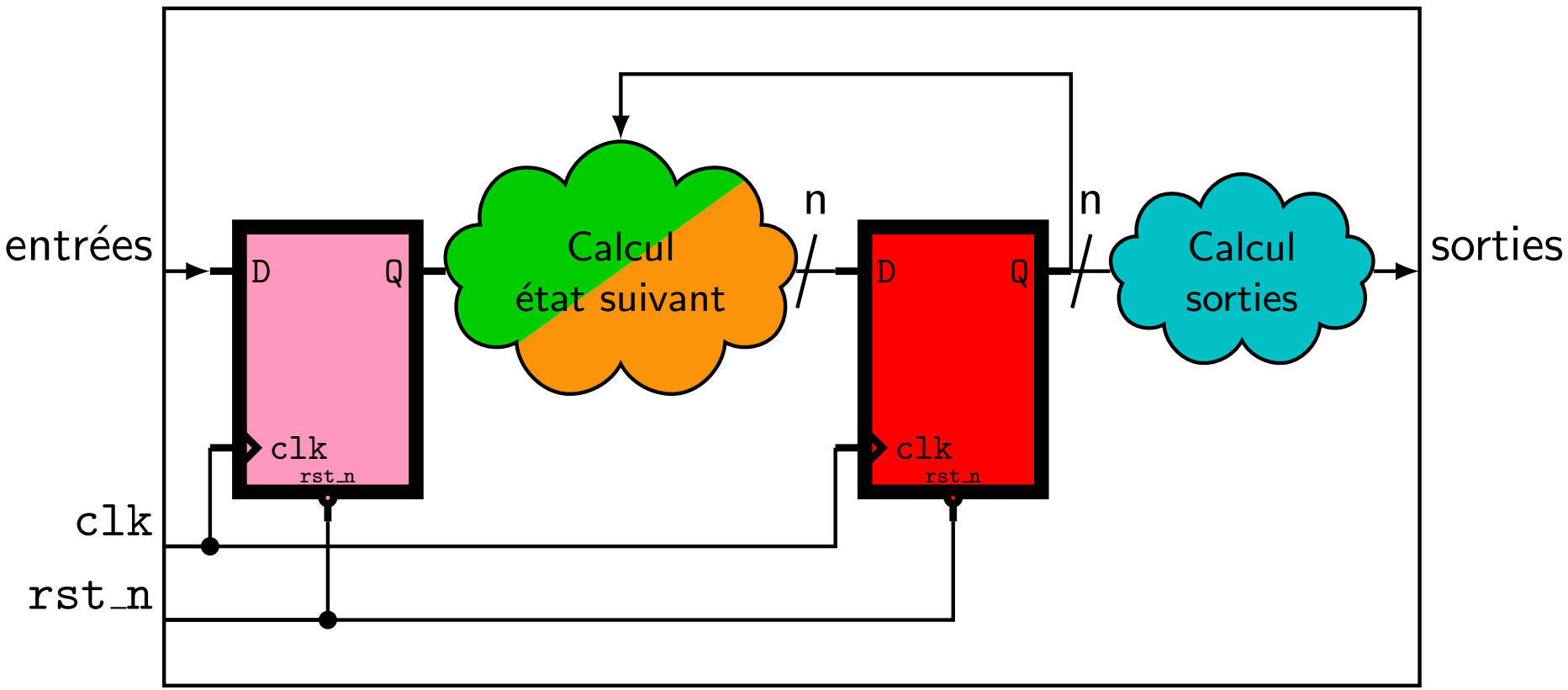
- t_{setup} : durée pendant laquelle la donnée présente sur l'entrée D doit **être stable avant** le front montant de l'horloge.
- t_{hold} : durée pendant laquelle la donnée présente sur l'entrée D soit **rester stable après** le front montant de l'horloge.



Ces valeurs sont données par le **concepteur / fabriquant** de la bascule D.

Contraintes temporelles pour l'implantation d'une FSM

Pour éviter les erreurs sur le registre d'état, il faut **synchroniser les entrées**.



Entité

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bike_ctrl IS

  PORT (
    clk   : IN  STD_LOGIC;
    rst_n : IN  STD_LOGIC;
    p     : IN  STD_LOGIC;
    m     : IN  STD_LOGIC;
    s     : IN  STD_LOGIC;
    v     : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));

END bike_ctrl;
```

Type énuméré pour le codage des états

On utilise un **type énuméré** pour coder les états :

```
TYPE nom_du_type IS (valeur_1, valeur_2, ..., valeur_n);
```

Puis on déclare le signal qui aura ce nouveau type :

```
SIGNAL nom_du_signal : nom_du_type;
```

```
ARCHITECTURE comport OF bike_ctrl IS

  TYPE etat IS (off, coast, speed, boost);
  SIGNAL etat_actuel : etat;

  -- Signaux pour la synchronisation des entrées
  SIGNAL p_s : STD_LOGIC;
  SIGNAL m_s : STD_LOGIC;
  SIGNAL s_s : STD_LOGIC;

BEGIN
```

Mise à jour de l'état et calcul de l'état suivant

Une structure CASE énumère tous les états actuels possibles.
Des structures IF décrivent les **transitions** et les **conditions associées**.

```
PROCESS (clk, rst_n)
BEGIN
  IF rst_n = '0' THEN
    etat_actuel <= off;
    p_s <= '0';
    m_s <= '0';
    s_s <= '0';
  ELSE
    IF rising_edge(clk) THEN
      p_s <= p;
      m_s <= m;
      s_s <= s;
      CASE etat_actuel IS
        WHEN off =>
          IF p_s = '1' THEN
            etat_actuel <= coast;
          END IF;
        WHEN coast =>
          IF m_s = '1' OR s_s = '1' THEN
            etat_actuel <= off;
          ELSE
            IF p_s = '1' THEN
              etat_actuel <= speed;
            END IF;
          END IF;
        WHEN speed =>
          IF s_s = '1' THEN
            etat_actuel <= off;
          ELSE
            IF m_s = '1' THEN
              etat_actuel <= coast;
            ELSE
              IF p_s = '1' THEN
                etat_actuel <= boost;
              END IF;
            END IF;
          END IF;
```

Calcul des sorties

Une structure WITH ... SELECT donne les valeurs des sorties pour chaque état possible.

```
        END CASE;
      END IF;
    END IF;
  END PROCESS;

  WITH etat_actuel SELECT
    v <=
    "00" WHEN off,
    "01" WHEN coast,
    "10" WHEN speed,
    "11" WHEN boost,
    "00" WHEN OTHERS;

END comport;
```

On aurait aussi pu utiliser une structure WHEN ... ELSE.