



IUT Saint-Etienne

Département GEl

**SIN1 -
Systèmes d'Information
Numérique**

V. Fischer, F. Bernard

Février 2014



Informations de base sur le cours (1/2)

- ⊕ **Responsable :** *Viktor Fischer*
Professeur à l'Université de Saint-Etienne
- ⊕ **Contact :** *IUT, GEII*
fischer@univ-st-etienne.fr
- ⊕ **Consultations :** *Jeudi après-midi*
- ⊕ **Structure du cours :** *4 h CMs*
26 x 1,5 h TDs
2 x 3 h TP
- ⊕ **Partiel :** *QCM sur la première partie du cours*
- ⊕ **Examen :** *Composé d'un QCM (partie théorique) et d'exercices (langage VHDL) pour la partie pratique*

Informations de base sur le cours (2/2)

- ⊕ **Objectifs :**
 - *Connaître les fonctions de base de l'électronique numérique,*
 - *Familiariser les étudiants avec les différentes méthodes de conception des systèmes numériques simples.*
- ⊕ **Compétences minimales :**
 - *Savoir décomposer une fonction en blocs combinatoires et séquentiels,*
 - *Savoir choisir et mettre en oeuvre un circuit numérique conventionnel ou programmable,*
 - *Savoir utiliser une chaîne de développement (simulation et synthèse),*
 - *Savoir concevoir, simuler et tester un circuit logique programmable.*

Table des matières

- ⊕ **Introduction (CM1)**
- ⊕ **Chapitre 1 – Algèbre de Boole – fonctions logiques et leur réalisation dans le matériel (TD)**
- ⊕ **Chapitre 2 – Systèmes de numération (TD)**
- ⊕ **Chapitre 3 – Opérations arithmétiques sur des nombres exprimés en système binaire (TD)**
- ⊕ **Chapitre 4 – Conception de circuits logiques (CM1)**
- ⊕ **Chapitre 5 – Introduction au langage VHDL (CM1 + TD)**
- ⊕ **Chapitre 6 – Logique combinatoire et sa réalisation (TD)**
- ⊕ **Chapitre 7 – Logique séquentielle et sa réalisation (TD)**
- ⊕ **Chapitre 8 – Machines d'états (TD)**
- ⊕ **Chapitre 9 – Modularité et conception hiérarchique du design (TD)**
- ⊕ **Chapitre 10 – Familles technologiques de circ. intégrés logiques (CM2)**
- ⊕ **Chapitre 11 – Familles fonctionnelles de circ. intégrés logiques (CM3)**
- ⊕ **Chapitre 12 – Circuits logiques configurables (CM4)**



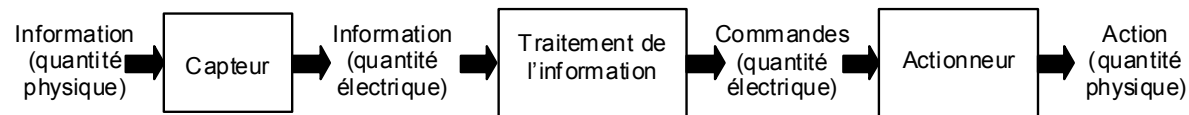
Introduction

Février 2014

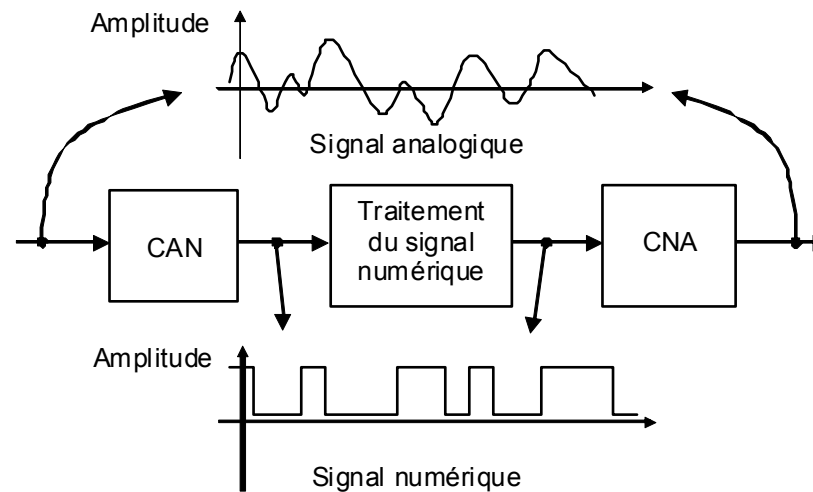
Electronique numérique

⊕ **Electronique**

- **Utilise les signaux électrique pour le traitement de l'information**



- **Electronique analogique** – la plupart des phénomènes physiques naturels est analogique (leur grandeur change en continu)
- **Electronique numérique** – permet de profiter de la puissance de calcul d'ordinateurs





Chapitre 1

Fonctions logiques et algèbre de Boole

Fonctions logiques

- ⊕ **Logique** – *partie de la philosophie qui cherche à résoudre les problèmes philosophiques : argument, sens et en particulier la vérité*



Ex.: problème philosophique de la phrase : “Je mens !”

- ⊕ **Les solutions mathématiques proposées par George Boole (1815 -1864) dans l'algèbre de Boole**



- *Utilisant des variables logiques pour construire et résoudre les équations*
- *Variable logique – variable qui contient les valeurs de vérité*
 - *Représentation logique : vrais et faux*
 - *Représentation algébrique : bits (binary digits) 1 et 0*
 - *Représentation physique : interrupteurs – ouverts et fermés*
 - *Représentation électrique : tension – haute (5V) et basse (0V)*

Opérations de base de l'algèbre de Boole



⊕ **AND** (*conjonction*)

Operateur	Equation logique	Table de vérité	Symbole logique															
AND	$L = A \cdot B$	<table><tr><td>A</td><td>B</td><td>L</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	L	0	0	0	0	1	0	1	0	0	1	1	1	ANSI 
		A	B	L														
		0	0	0														
		0	1	0														
		1	0	0														
1	1	1																
CEI 																		

⊕ **OR** (*disjonction*)

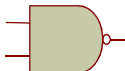

Operateur	Equation logique	Table de vérité	Symbole logique															
OR	L = A + B	<table><tr><td>A</td><td>B</td><td>L</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	L	0	0	0	0	1	1	1	0	1	1	1	1	ANSI 
		A	B	L														
		0	0	0														
		0	1	1														
		1	0	1														
		1	1	1														
CEI 																		

⊕ **NOT** (*négation*)

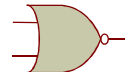
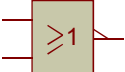
Operateur	Equation logique	Table de vérité	Symbole logique						
NOT	$L = \overline{A}$	<table><tr><td>A</td><td>L</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	L	0	1	1	0	<div>ANSI </div> <div>CEI </div>
A	L								
0	1								
1	0								

Autres opérations de l'algèbre de Boole


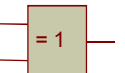
⊕ **NAND**

Opérateur	Equation logique	Table de vérité	Symbole logique															
NAND	$L = \overline{A \cdot B}$	<table><tr><th>A</th><th>B</th><th>L</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	L	0	0	1	0	1	1	1	0	1	1	1	0	<div><div>ANSI</div><div>CEI</div></div>
A	B	L																
0	0	1																
0	1	1																
1	0	1																
1	1	0																

⊕ **NOR**

Opérateur	Equation logique	Table de vérité	Symbole logique															
NOR	$L = \overline{A + B}$	<table><tr><th>A</th><th>B</th><th>L</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	L	0	0	1	0	1	0	1	0	0	1	1	0	<div>ANSI </div> <div>CEI </div>
A	B	L																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

⊕ **XOR (Exclusive OR)**

Opérateur	Equation logique	Table de vérité	Symbole logique															
XOR	$L = A \oplus B$	<table><tr><th>A</th><th>B</th><th>L</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	L	0	0	0	0	1	1	1	0	1	1	1	0	<div>ANSI </div> <div>CEI </div>
A	B	L																
0	0	0																
0	1	1																
1	0	1																
1	1	0																

Règles de l'algèbre de Boole

⊕ *Utilisées pour résoudre les équations logiques*

Propriétés	Sommes	Produits
Commutativité	$A + B = B + A$	$A \cdot B = B \cdot A$
Associativité	$A + B + C = A + (B + C)$	$A \cdot B \cdot C = (A \cdot B) \cdot C$
Distributivité	$A + (B \cdot C) = (A + B) \cdot (A + C)$	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
Éléments neutres	$A + 0 = A$	$A \cdot 1 = A$
Complémentation	$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
Idempotence	$A + A = A$	$A \cdot A = A$
Absorption d'un terme	$A + A \cdot B = A$	$A \cdot (A + B) = A$
Absorption d'un terme	$A + \overline{A} \cdot B = A + B$	$A \cdot (\overline{A} + B) = A \cdot B$
Absorption d'un élément	$A + 1 = 1$	$A \cdot 0 = 0$
Double complémentation	$A = \overline{\overline{A}}$	
Règles de De Morgan	$\overline{A + B} = \overline{A} \cdot \overline{B}$	$\overline{A \cdot B} = \overline{A} + \overline{B}$




Simplification d'équations logiques

- ⊕ ***Simplification = réduction de termes logiques dans l'équation
= réduction du coût***
- ⊕ ***Simplification par l'utilisation de règles de l'algèbre de Boole***
(voir l'écran précédent)
 - Seulement pour les équations simples
 - Un certain niveau d'expertise est nécessaire
 - Ne peut pas être automatisée
 - Résultats sans expertise non garantis
- ⊕ ***Simplification par les tableaux de Karnaugh***
 - ***Méthode :***
 - Regrouper les uns dans les tableaux de Karnaugh
 - Construire l'équation par l'expression du terme logique pour chaque groupe
 - ***Tableau de Karnaugh : une autre façon pour présenter les tables de vérité en utilisant les codes réfléchis (codes Gray)***

A vertical strip of six images showing the interior of a library. The images depict tall, dark wooden bookshelves filled with numerous books, creating a dense and organized environment. The lighting is warm, highlighting the spines of the books and the texture of the wood. The perspective is from within the aisles, looking down the length of the shelves.

- ## ⊕ Transformation de la table de vérité en tableau de Karnaugh



The diagram shows a 2x4 truth table for the Code Gray output BC. The input A has two rows: 0 and 1. The output BC has four columns: 00, 01, 11, and 10. The output values are 0, 1, 0, 0 for A=0 and 1, 1, 0, 0 for A=1. A red box labeled "Code Gray" with an arrow points to the 01 column.

	00	01	11	10
0	0	1	0	0
1	1	1	0	0

Règles de simplification avec tableaux de Karnaugh

- ⊕ Les groupes doivent contenir **2^n cellules** (puissance de deux)
- ⊕ Les groupes ne peuvent **pas contenir de zéros**
- ⊕ Les groupes doivent avoir **une forme rectangulaire** (pas en diagonale)
- ⊕ Chaque groupe doit être **aussi grand que possible**
- ⊕ Les groupes **peuvent se chevaucher**
- ⊕ Chaque cellule contenant un 1 doit **faire partie d'au moins un groupe**
- ⊕ Les groupes **peuvent passer** autour du tableau
- ⊕ Le nombre de groupes doit être **aussi petit que possible**, en respectant les règles précédentes

⊕ *Solution pour l'exemple précédent:*

- Deux groupes
- $Y = (A \cdot B) + (B \cdot C)$

		BC			
		00	01	11	10
A	0	0	1	0	0
	1	1	1	0	0



Chapitre 2

Systemes numériques

Nombres et systèmes de numération

- ⊕ **Les nombres sont classés dans les ensembles – systèmes de numération**
- ⊕ **Systèmes de base – naturel, entier, rationnel, réel, complexe, ...**
- ⊕ **Un système de numération naturel positionnel à n chiffres (incluant zéro) peut être exprimé en utilisant un polynôme d'ordre $(n - 1)$:**

$$A = a_{n-1}p^{n-1} + a_{n-2}p^{n-2} + \dots + a_1p + a_0$$

où p est la base du système et $0 \leq a_i \leq p - 1$

⊕ Exemples:

Système décimal ($p = 10, 0 \leq a_i \leq 9$): $A = 123_{(10)} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

Système binaire ($p = 2, 0 \leq a_i \leq 1$): $A = 101_{(2)} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5_{(10)}$

Système hexadécimal ($p = 16, 0 \leq a_i \leq 15$): $A = 123_{(16)} = 1 \cdot 16^2 + 2 \cdot 16^1 + 3 \cdot 16^0$
 $= 256 + 32 + 3 = 291_{(10)}$

Nombres équivalents en notation décimale, binaire et hexadécimale

Décimale	Binaire				Hexadécimale
	b3	b2	b1	b0	
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	1	0	1	0	A
11	1	0	1	1	B
12	1	1	0	0	C
13	1	1	0	1	D
14	1	1	1	0	E
15	1	1	1	1	F

Conversions entre les nombres décimaux, binaires et hexadécimaux (1/4)

✚ *Conversion d'une valeur binaire en décimale*

- Trouver la valeur décimale pour chaque position contenant le bit 1 et faire la somme.

- Exemple: Convertir $11011_{(2)}$ en décimal

- Solution:

1	1	0	1	1	
				→	$1 \cdot 2^0 = 1$
			→		$1 \cdot 2^1 = 2$
		→			$1 \cdot 2^3 = 8$
	→				$1 \cdot 2^4 = 16$
					<hr/>
					27

- La valeur décimale est donc $27_{(10)}$

Conversions entre les nombres décimaux, binaires et hexadécimaux (2/4)

⊕ *Conversion d'une valeur décimale en binaire*

- La même méthode peut être utilisée pour faire la conversion d'une valeur décimale vers n'importe quelle base. Celle-ci consiste en divisions successives par la base (ici par deux) jusqu'à ce que le dividende arrive à 0. A chaque division, le reste représente un chiffre du nombre recherché, en partant du chiffre le moins significatif.
- Exemple: Convertir $39_{(10)}$ en binaire
- Solution:
 $39 / 2 = 19$ reste **1** (bit le moins significatif)
 $19 / 2 = 9$ reste **1**
 $9 / 2 = 4$ reste **1**
 $4 / 2 = 2$ reste **0**
 $2 / 2 = 1$ reste **0**
 $1 / 2 = 0$ reste **1** (bit le plus significatif)
- Le code binaire résultant est $100111_{(2)}$

Conversions entre les nombres décimaux, binaires et hexadécimaux (3/4)

⊕ *Conversion d'une valeur binaire en hexadécimal*

- Diviser le nombre binaire en groupes de quatre chiffres (bits) en partant de droite. Si le nombre de bits n'est pas divisible par 4, compléter la valeur binaire de zéros à gauche pour que chaque groupe contienne 4 bits. Convertir la valeur de chaque groupe de 4 bits, en hexadécimal.
- Exemple: Convertir $100111011_{(2)}$ en hexadécimal
- Solution:

0001	0011	1011
└───┘	└───┘	└───┘
1	3	B
- La valeur hexadécimale correcte est $13B_{(16)}$

Conversions entre les nombres décimaux, binaires et hexadécimaux (4/4)

⊕ *Conversion d'une valeur hexadécimale en binaire*

- Pour convertir une valeur hexadécimale en binaire, convertir chaque chiffre hexadécimal en groupes de 4 bits.

- Exemple : Convertir $2AB_{(16)}$ en binaire

- Solution :

2	A	B
└─┬─┘	└─┬─┘	└─┬─┘
0010	1010	1011

- Le code binaire résultant est $1010101011_{(2)}$

Nombres décimaux codés binaire

- ⊕ **Les nombres BCD sont représentés en décimal, mais chaque chiffre décimal est codé en utilisant une valeur binaire de 4 bits**
- ⊕ Pour convertir un nombre décimal en BCD, il faut convertir chaque chiffre décimal en groupes de 4 bits.
 - Exemple: Convertir $189_{(10)}$ en BCD
 - Solution:
$$\begin{array}{ccc} 1 & 8 & 9 \\ \hline 0001 & 1000 & 1001 \end{array} \rightarrow 000110001001_{(BCD)}$$
- ⊕ Pour convertir une valeur BCD en décimal, il faut diviser le nombre en groupes de 4 bits et convertir chaque groupe en décimal.
 - Exemple: Convertir $10001110101_{(BCD)}$ en binaire
 - Solution:
$$\begin{array}{ccc} 0100 & 0111 & 0101 \\ \hline 4 & 7 & 5 \end{array} \rightarrow 475_{(10)}$$

Nombres entiers signés (arithmétiques) et non signés (logiques) en binaire

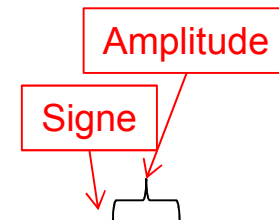
⊕ Le **type** de la représentation doit être **connu à l'avance**

⊕ Le **nombre de bits** doit être **connu à l'avance**

⊕ Les représentations de nombres signés

- Représentation en signe et amplitude

- Bit à gauche réservé pour le signe
- Signe codé en binaire: $+\leftrightarrow 0, -\leftrightarrow 1$
 - Exemple: Représentation de -5 sur 4 bits: $1101_{(2)}$



- Représentation en complément à un

- Les nombres positifs sont codés simplement en binaire
- Les nombres négatifs sont créés à partir de leur valeur absolue, par inversion bit par bit
 - Exemple : Représentation de -5 sur 4 bits: $1010_{(2)}$

Inversé bit par bit
 $0101_{(2)} = 5_{(10)}$

- Représentation en complément à deux

- Comme complément à un, puis incrémenté par un
 - Exemple : Représentation de -5 sur 4 bits: $1011_{(2)}$

Complément à 1
plus un

Nombres binaires signés – plusieurs interprétations possibles (le type doit être connu à l'avance)

Valeur binaire				Décimal non signé	En signe et amplitude	En complément à un	En complément à deux
b3	b2	b1	b0				
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	2	2	2	2
0	0	1	1	3	3	3	3
0	1	0	0	4	4	4	4
0	1	0	1	5	5	5	5
0	1	1	0	6	6	6	6
0	1	1	1	7	7	7	7
1	0	0	0	8	-0	-7	-8
1	0	0	1	9	-1	-6	-7
1	0	1	0	10	-2	-5	-6
1	0	1	1	11	-3	-4	-5
1	1	0	0	12	-4	-3	-4
1	1	0	1	13	-5	-2	-3
1	1	1	0	14	-6	-1	-2
1	1	1	1	15	-7	-0	-1



Chapitre 3

Opérations arithmétiques sur les nombres binaires

Addition de deux nombres binaires non signés

⊕ **Principe** – similaire à l'addition de nombres décimaux

**Rappel : addition de deux nombres décimaux de 4 chiffres
(addition chiffre par chiffre en partant de droite)**

Ordre (i)	4	3	2	1	0	
Retenue intermédiaire entrante	1	1	1	0	0	Trois entrées par chiffre
Addende a		5	2	8	2	
Addende b	+	7	8	6	5	
Somme		3	1	4	7	Deux sorties par chiffre
Retenue intermédiaire sortante	1	1	1	0		

⊕

Retenue sortante

Addition de deux nombres binaires non signés

⊕ Table de vérité pour l'addition binaire de trois chiffres (ci, a, b)

= Table de vérité d'un additionneur complet sur 1 bit – Full Adder (FA)

ci	a	b	Decim.	co	sum
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

$$1 + 1 + 1 = 3_{(10)} = 11_{(2)}$$

⊕ Addition de deux nombres binaires de 4 chiffres

Ordre (i)

Retenue intermédiaire entrante (ci)

Addende a

Addende b

Somme

Retenue intermédiaire sortante (co)

	4	3	2	1	0
Retenue intermédiaire entrante (ci)	1	1	1	0	0
Addende a		1	0	1	1
Addende b		1	1	1	0
Somme	1	1	0	0	1
Retenue intermédiaire sortante (co)		1	1	1	0

Retenue sortante

$$\begin{array}{r} 11 \\ + 14 \\ \hline 25 \end{array}$$

Addition de deux nombres binaires signés

- ⊕ Les opérations arithmétiques se font **en complément à deux**
- ⊕ L'addition bit par bit est la même, seul le **résultat est interprété différemment**
- ⊕ **Addition de deux nombres de 8 bits et interprétation du résultat pour les entiers signés et non signés**

i		8	7	6	5	4	3	2	1	0
ci		1	1	1	1	1	1	1	0	0
a	- 101		1	0	0	1	1	0	1	1
b	+ 126		0	1	1	1	1	1	1	0
s	25	1	0	0	0	1	1	0	0	1
co			1	1	1	1	1	1	1	0

Retenue sortante

La somme s n'est pas zéro

Drapeaux (flags):

$$Z \text{ (Zero)} = \overline{s(7)} \cdot \overline{s(6)} \cdot \overline{s(5)} \cdot \overline{s(4)} \cdot \overline{s(3)} \cdot \overline{s(2)} \cdot \overline{s(1)} \cdot \overline{s(0)} = 0$$

$$N \text{ (Negative)} = s(7) = 0$$

$$C \text{ (Carry)} = co(7) = 1$$

$$V \text{ (oVerflow)} = co(7) \oplus co(6) = 0$$

La somme s n'est pas négative

La somme s est incorrecte pour les nombres non signés (logiques)

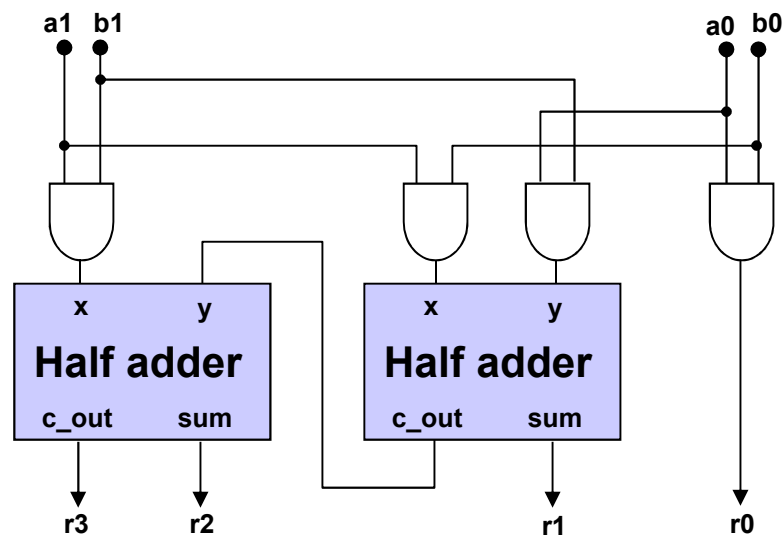
La somme s est correcte pour les nombres signés (arithmétiques)

Multiplication de nombres binaires non signés

- ⊕ **Principe** – décalages conditionnels à gauche et additions
 - *Multiplication de deux nombres non signés de 2 bits*

Facteur a	a1	a0		
Facteur b	x	b1	b0	
		a1b0	a0b0	
	a1b1	a0b1		
Produit	r3	r2	r1	r0

- Résultat sur 4 bits



Half adder = demi-additionneur
(il a seulement deux entrées),
il faut deux demi-additionneurs pour
faire un additionneur complet



Chapitre 4

Méthodes de conception de circuits logiques



Méthodes de conception de circuits logiques

⊕ **Synthèse logique**

- Procédure de la conception de systèmes logiques : de la spécification jusqu'à l'implantation du système dans le matériel

⊕ **Outil classique de l'électronicien pour la conception d'un circuit logique : le schéma électrique**

- Utilise des portes ou des circuits élémentaires
- Peut comporter plusieurs niveaux hiérarchiques: peu performant dans le cas de systèmes complexes
- Difficile à déboguer sans réaliser de prototype

⊕ **Les langages de description (HDL – Hardware Description Language) - outils de type textuel offrant :**

- Une plus grande souplesse d'emploi
- Une plus grande diversité des techniques de description (niveaux d'abstraction)
- La simulation permet de simuler la logique avant de l'implanter dans le matériel – gain de temps considérable



Implantation de fonctions logiques dans le matériel

⊕ **Implantations basées sur la logique câblée**

- **Étapes de la conception :**

En TP

- **Spécification de la fonction à réaliser**
 - **Comportement (par exemple par le tableau de vérité)**
 - **Nombre d'entrées/sorties**
 - **Vitesse, surface (coût) et consommation, etc.**
- **Construction de tableaux de Karnaugh**
- **Minimisation de fonctions en utilisant les tableaux de Karnaugh**
- **Conversion éventuelle de la fonction en logique NAND, NOR, ...**
- **Choix de la technologie (CMOS ou TTL) et du type de circuits**
- **Réalisation du prototype (câblage)**
- **Débogage du prototype**
- **Réalisation d'une carte imprimée**



Implantation de fonctions logiques dans le matériel

(suite)

⊕ **Implantations basées sur les circuits logiques configurables**

- **Étapes de la conception :**

En TD,
TP et
TR

- **Spécification de la fonction à réaliser**
- **Description de la logique en schéma ou en HDL**
- **Simulation fonctionnelle**
- **Choix de la technologie et de la famille**
- **Synthèse logique**
- **Placement – routage**
- **Simulation temporelle (éventuelle)**

En TP
et TR

- **Réalisation d'une carte imprimée**
- **Configuration du circuit**

Langage VHDL

⊕ *Langage VHDL*

- VHDL = VSIC HDL (VSIC = Very High Speed Integrated Circuits)
- Conçu pour le DoD (Department of Defence) aux USA par Intermetrics, IBM, Texas Instruments
- Normalisé par IEEE
 - norme IEEE 1076 en 1987, puis 1993
- Très répandu, surtout en Europe et dans le milieu universitaire
- Langage complexe avec haut niveau d'abstraction, inspiré par l'ADA (destiné à la programmation de systèmes parallèles)
- Utilisé pour :
 - Spécification
 - Modélisation
 - Synthèse

Problème : Les systèmes qui peuvent être modélisés, ne sont pas forcément synthétisables !

⊕ *Concurrent : Langage Verilog HDL*

- Plus proche du matériel, étendu aux USA



Modélisation = Description + simulation

⊕ Changements importants des méthodes de travail des électroniciens liés à l'évolution de la complexité des systèmes :

- Étapes classiques de prototypage - remplacées par des **étapes de simulation** permettant de diminuer notablement les coûts et les délais de développement
- La simulation est présente **dans toutes les phases de la conception** d'un système logique
- Rôle de la simulation : **simuler le modèle** avant de le synthétiser
- **Deux types de simulation** : simulation fonctionnelle et simulation physique (temporelle)



Type de simulation

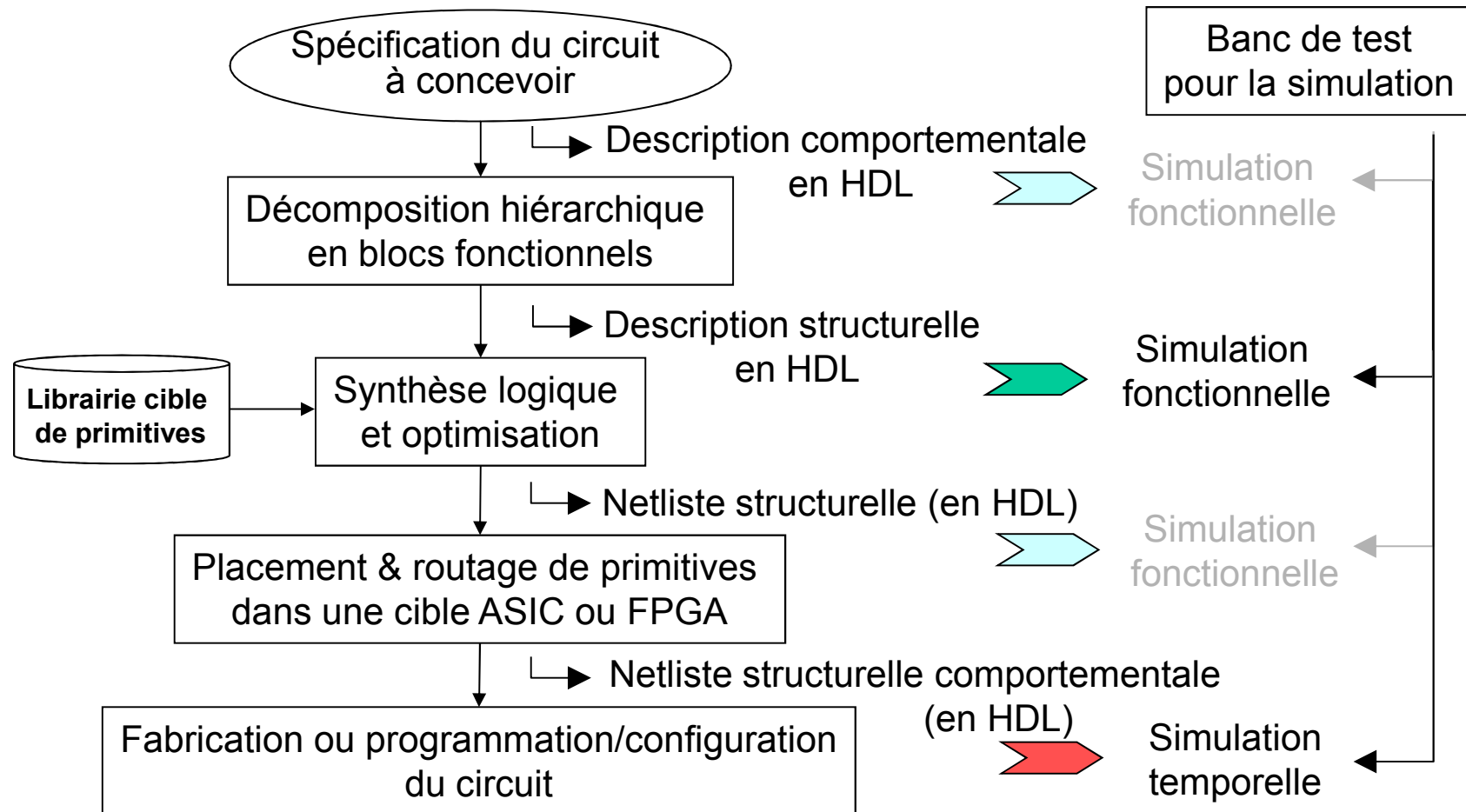
⊕ Simulation fonctionnelle :

- Beaucoup plus rapide, moins précise
- Objectif - la preuve formelle du fonctionnement pour chaque élément du système
- Aucun délai de propagation n'est pris en compte, le système physique est considéré comme parfait

⊕ Simulation physique (simulation temporelle) :

- Beaucoup plus précise, beaucoup plus lente
- Objectif - obtenir une simulation la plus proche possible du système physique réel
- Prend en compte des délais physiques de propagation à l'intérieur du système (circuit)

Flot de conception d'un circuit logique en HDL





Chapitre 5

Langage VHDL - Introduction



Préambule

⊕ Caractéristiques du langage VHDL :

- Un vocabulaire très volumineux
- Des contextes d'utilisation différents (spécification, modélisation, synthèse)

⊕ Deux aspects importants du langage VHDL :

- VHDL est un **langage de description** des structures matérielles et non pas un langage de programmation classique de l'informatique scientifique (attention aux réflexes d'informaticien)
- Certains éléments de ce langage **ne sont pas utilisables dans tous les contextes** d'application

⊕ A noter :

Dans les chapitres suivants nous allons utiliser **un sous-ensemble** du langage VHDL – structures de base utilisées pour la synthèse logique !

Unités de conception

Une description VHDL comporte un certain nombre d'unités de conception. Une unité de conception constitue un sous ensemble de la structure logique pouvant être compilé séparément, stocké dans un fichier indépendant et sauvegardé dans une librairie. Une unité de conception peut se trouver :

- Dans un fichier *.vhd (une ou plusieurs unités dans un fichier)
- Dans un répertoire (de travail), dans plusieurs fichiers *.vhd
- Dans une librairie (un paquetage)

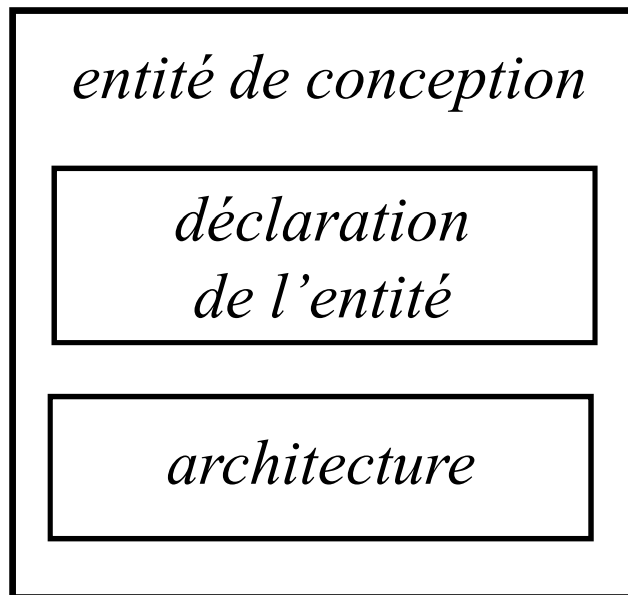
← Méthode appliquée en ENSL1

⊕ Unités de conception :

- **Entité** – élément de base (composant, module) définie par la :
 - spécification d'entité (= interface externe ⇒ symbole)
 - architecture (= structure interne ⇒ schéma)
- **Paquet** – regroupement d'éléments défini par la
 - spécification de paquet
 - corps de paquet
- **Configuration** – association architecture - entité

} Non utilisé en ENSL1

Entité de conception



⊕ **Entité de conception** – élément de base de construction :

- Vue de l'extérieur par l'intermédiaire de **signaux d'entrées/sorties** (boîte noire)
- A l'intérieur spécifiée par **l'architecture**

⊕ **Déclaration de l'entité**

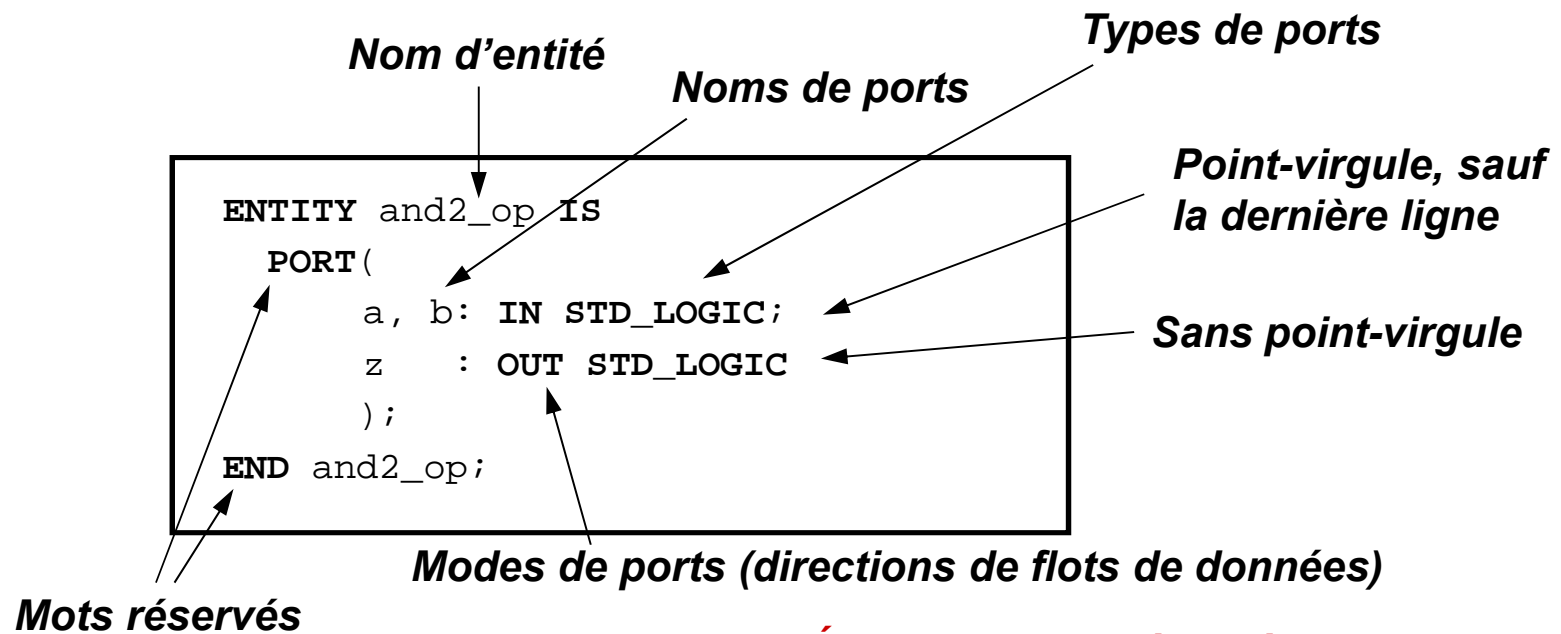
- Nom
- Paramètres (optionnels)
- Entrées/sorties

⊕ **Architecture**

- Structure interne de l'entité

Déclaration d'entité

- ⊕ Décrit le nom et l'interface du composant (signaux d'entrées et sorties)



Notre convention : Écrire les mots réservés en majuscules
But : Lisibilité du code (même en noir et blanc)

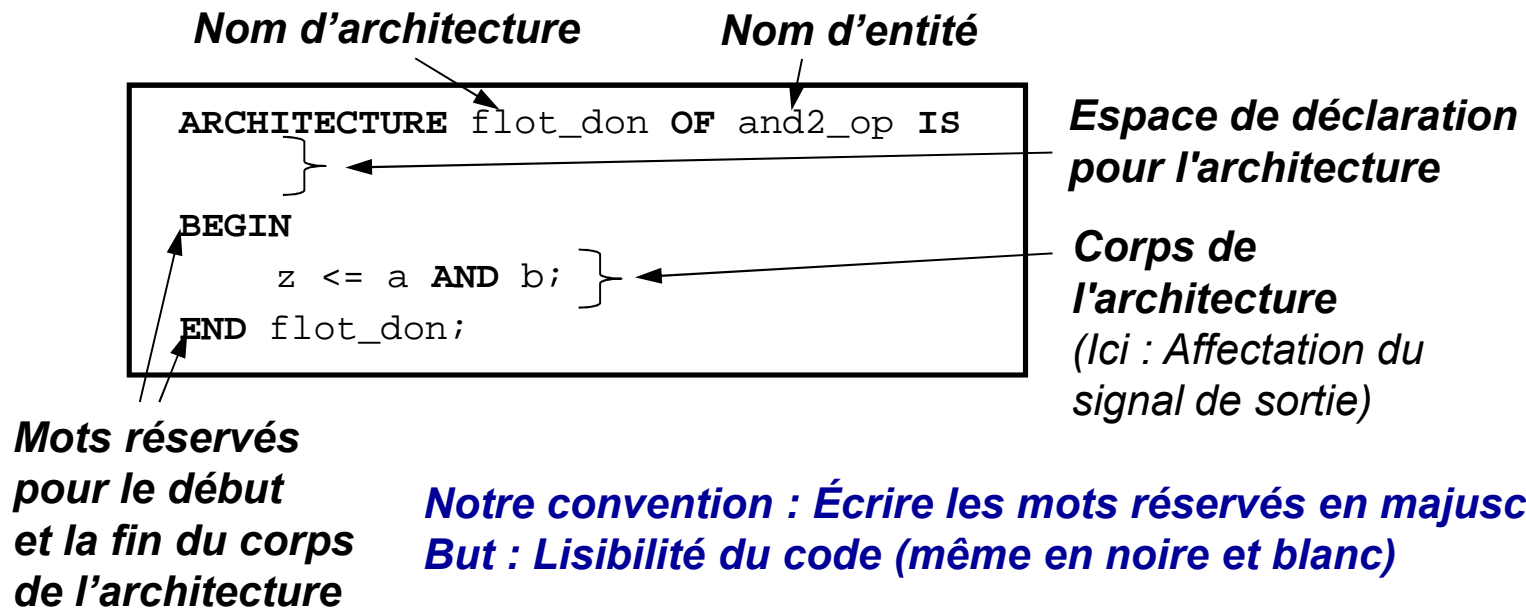
A noter :

Nous n'utiliserons pas les possibilités de paramétrer des modules.

Description d'architecture

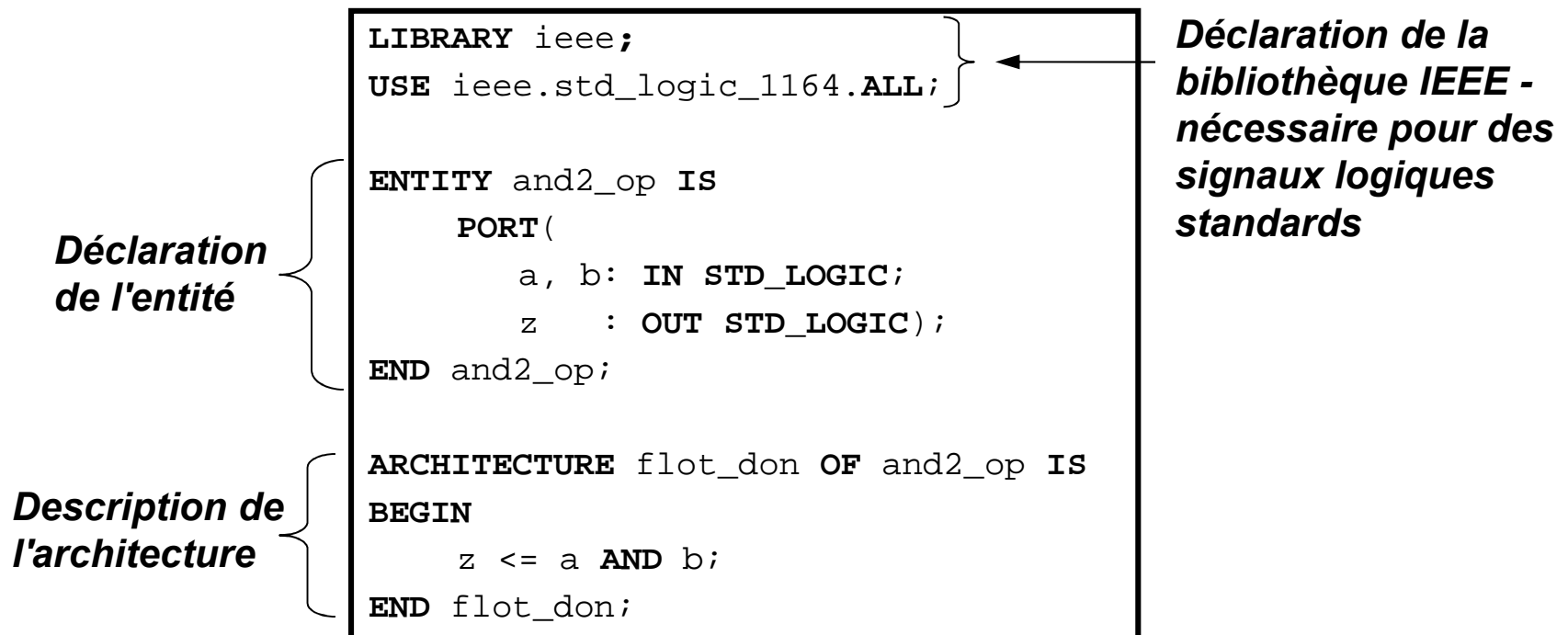
⊕ **Architecture** - décrit la structure ou le comportement du composant

- Plusieurs architectures pour la même entité (module) peuvent exister. Ce cas est similaire à la possibilité d'implanter une fonction logique soit en utilisant des portes NAND ou en utilisant des portes NOR ou d'autres portes logiques.



Vue d'ensemble : déclaration d'entité et architecture

⊕ *Code VHDL complet d'une entité*



Modes de ports

⊕ *Ils spécifient la direction du transfert des données vis-à-vis du composant*

- **IN** : port d'entrée (unidirectionnel) - les données arrivant sur ce port peuvent être lus à l'intérieur du composant, elles peuvent donc se trouver **seulement à droite** dans l'expression d'affectation d'un signal ou d'une variable.
- **OUT** : port de sortie (unidirectionnel) - les données sortant peuvent être seulement mises à jour (et non pas lus) à l'intérieur du composant, elles peuvent se trouver **seulement à gauche** de l'expression d'affectation.
- **INOUT** : port d'entrée/sortie (bi-directionnel) – les données peuvent être mises à jour et lus à l'intérieur du composant, elles peuvent se trouver **à gauche ou à droite** de l'expression d'affectation.

A noter : Il existe un autre mode (Buffer) que nous n'utiliserons pas

Classes, types et composition de données

⊕ **Classes de données**

- **Signaux** (SIGNAL)
- **Variables** (VARIABLE)
- **Constantes** (CONSTANT)

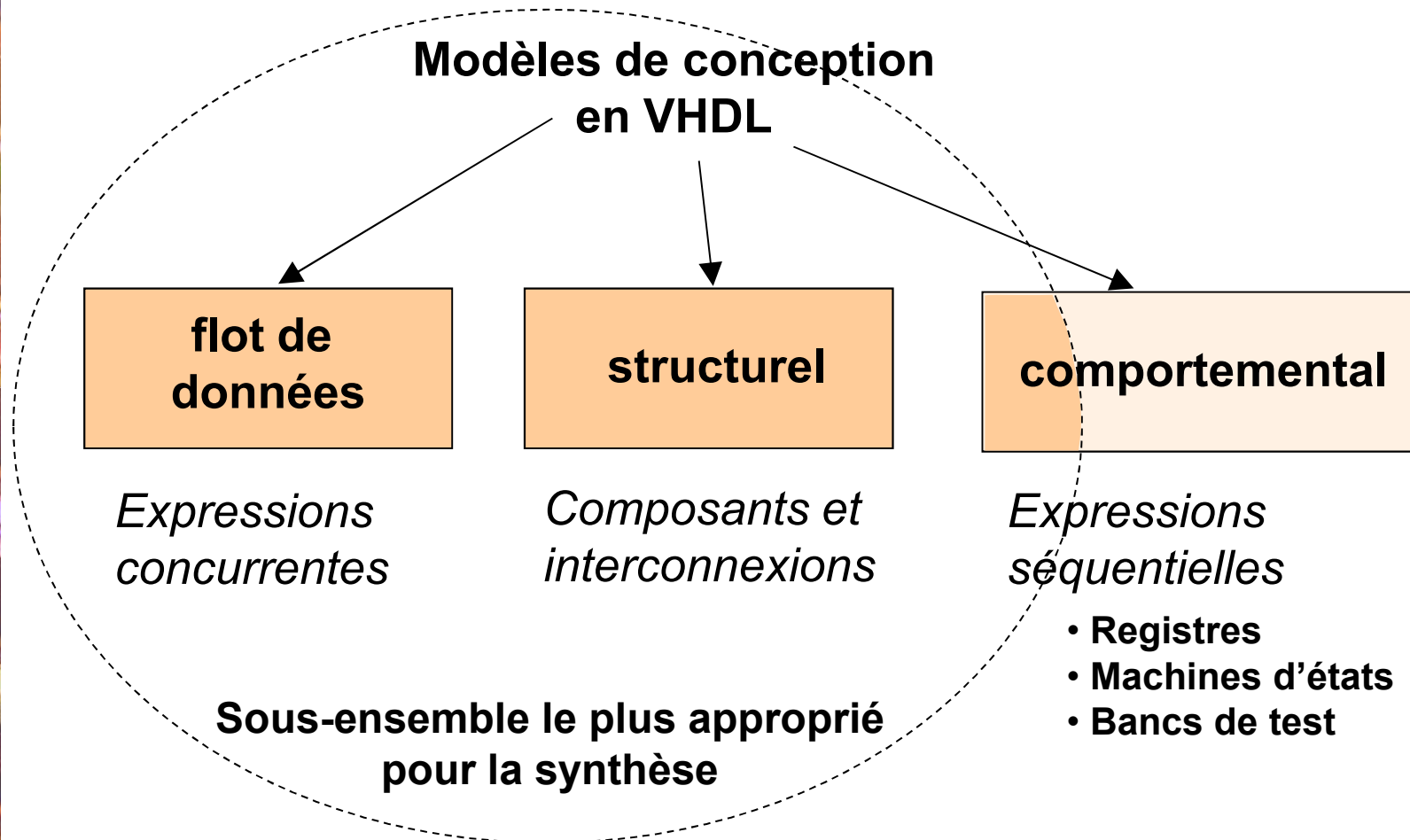
⊕ **Types de données**

- **Entiers** (INTEGER)
 - **Bits** (BIT)
 - **Booléens** (BOOLEAN)
 - **Caractères** (CHARACTER)
 - **Réels** (REAL)
 - **Physiques** (TIME)
 - **Énumérés** (ENUMERATED)
 - **Logique standard** (STD_LOGIC)
- Bibliothèque standard**
- Bibliothèque IEEE**

⊕ **Composition de données**

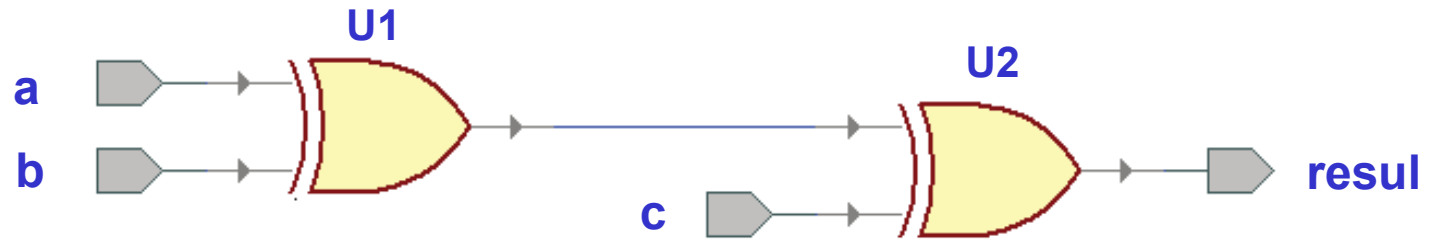
- **Scalaires** - donnée composée d'un seul élément
- **Composites** - donnée composée de plusieurs éléments (VECTOR)

Modèles de conception en VHDL



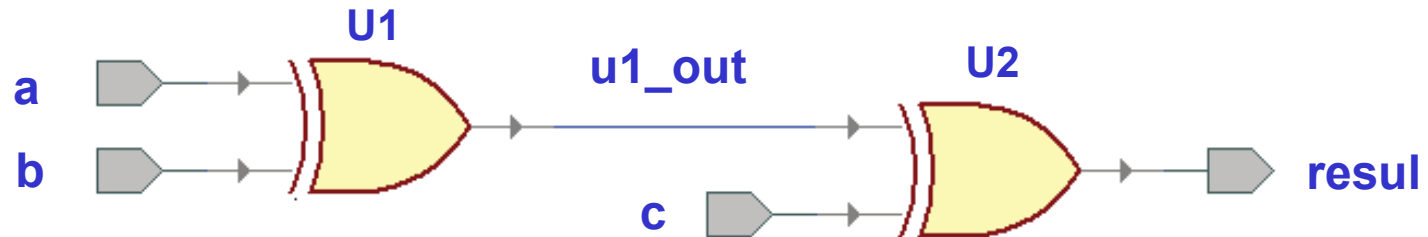
Critère du choix d'un modèle : la lisibilité du code !

Exemple : XOR3



```
ENTITY xor3 IS
  PORT (
    a      : IN STD_LOGIC;
    b      : IN STD_LOGIC;
    c      : IN STD_LOGIC;
    resul  : OUT STD_LOGIC
  );
END xor3;
```


Architecture « Flot de données » (Dataflow)



Nom d'architecture

Nom d'entité

```
ARCHITECTURE xor3_flotdon OF xor3 IS
    SIGNAL u1_out: STD_LOGIC;
BEGIN
    u1_out <= a XOR b;
    resul  <= u1_out XOR c;
END xor3_flotdon;
```

*Déclaration
du signal interne*



Description d'architecture par un « Flot de données »

- ⊕ Le modèle « Flot de données » décrit les relations internes entre les données dans le module.
- ⊕ La description de l'architecture basée sur ce modèle utilise les expressions concurrentes pour réaliser la logique. Ces expressions sont évaluées en même temps (en parallèle), donc **leur ordre n'est pas important !**
- ⊕ Le modèle « Flot de données » est le plus utile, si la logique peut être représentée par des fonctions booléennes.

Architecture utilisant le modèle structurel

```
ARCHITECTURE xor3_struct OF xor3 IS
```

```
  SIGNAL u1_out: STD_LOGIC;
```

```
  COMPONENT xor2 IS
```

```
    PORT(
```

```
      i1 : IN STD_LOGIC;
```

```
      i2 : IN STD_LOGIC;
```

```
      y  : OUT STD_LOGIC
```

```
    );
```

```
  END COMPONENT;
```

```
BEGIN
```

```
  u1: xor2 PORT MAP (i1 => a,  
                    i2 => b,  
                    y  => u1_out);
```

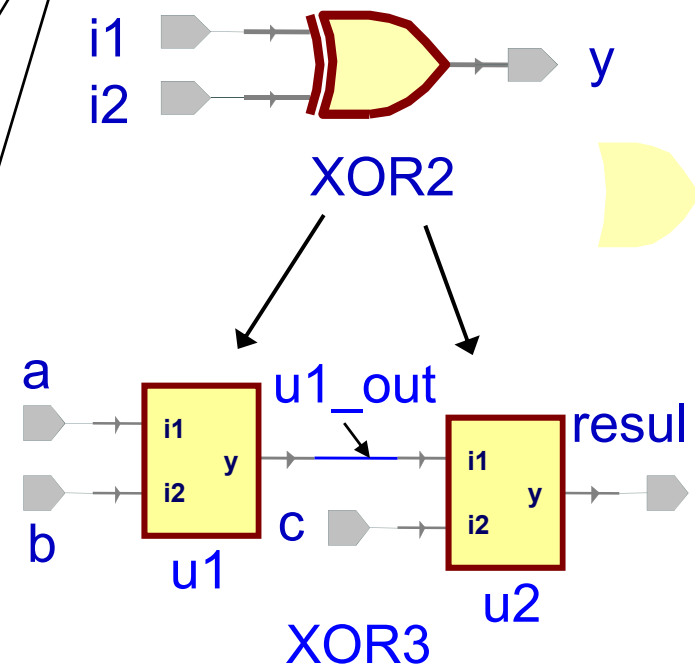
```
  u2: xor2 PORT MAP (i1 => u1_out,  
                    i2 => c,  
                    Y => resul);
```

```
END xor3_struct;
```

Déclaration du signal interne

*Déclaration du composant
(qui est défini ailleurs)*

Instanciations du composant





Architecture utilisant le modèle structurel (suite)

- ⊕ Le modèle structurel est le plus **simple à comprendre**. Il est le plus **proche à la saisie du schéma** : il utilise les blocs simples pour composer des fonctions logiques.
- ⊕ Le modèle structurel **remplace le schéma bloc**
- ⊕ Les composants peuvent être interconnectés d'une **manière hiérarchique** – la structure logique peut avoir plusieurs niveaux, chaque niveau étant décrit dans une entité indépendante en utilisant les trois modèles (flot de données, structurel ou comportemental) ou leur combinaison
- ⊕ Dans le modèle structurel nous pouvons connecter des portes logiques simples (déclarés dans une bibliothèque) ou des composants complexes décrits dans des entités séparées

Déclaration et instantiation du composant

- ⊕ **Instantiation** – la mise en place physique en associant les entrées/sorties du composant avec les signaux de l'architecture
- ⊕ **Association des connexions** **par leur noms** (recommandée)

```
COMPONENT xor2 IS
  PORT (
    i1 : IN STD_LOGIC;
    i2 : IN STD_LOGIC;
    y  : OUT STD_LOGIC
  );
END COMPONENT;
```

*Déclaration du composant
(dans la partie « déclarations »
de l'architecture)*

```
u1: xor2 PORT MAP (i1 => a,
                  i2 => b,
                  y  => u1_out);
```

*Instantiation du composant
(dans le corps de l'architecture)*

Nom de l'instanciation *Nom du composant*

Architecture utilisant le modèle comportemental

```
ARCHITECTURE xor3_comp OF xor3 IS
BEGIN
  xor3_proc: PROCESS (a, b, c)
  BEGIN
    IF ((a XOR b XOR c) = '1') THEN
      resul <= '1';
    ELSE
      resul <= '0';
    END IF;
  END PROCESS xor3_proc;
END xor3_comp;
```

- ⊕ Le modèle comportemental décrit ce qui se passe à la sortie du module (en fonction des entrées) sans préciser la structure interne du module (qui représente une « boîte noire »)
- ⊕ Ce modèle utilise une structure VHDL appelée *PROCESS*
- ⊕ Si possible, ne pas utiliser ce modèle pour synthétiser la logique combinatoire



Chapitre 6

Réalisation de fonctions logiques combinatoires dans le matériel

Fonctions logiques combinatoires

⊕ *Fonction logique combinatoire - définition*

- La valeur à la sortie d'une fonction logique combinatoire **ne dépend que des valeurs de signaux des entrées** (leur combinaison) et (contrairement à la logique séquentielle) ne dépend pas de l'état interne de la fonction
- Exemple : $Y = f(A, B, C) = A + B + C$

Y dépend que de A, B, et C

⊕ *Fonctions combinatoires de base*

- Fonctions logiques simples
- Générateurs de parité
- Multiplexeurs, démultiplexeurs
- Codeurs, décodeurs
- Fonctions arithmétiques simples (addition, soustraction, etc.)
- Compareurs
- Fonctions d'entrées/sorties avec la logique trois états

Implantation des fonctions logiques combinatoires en VHDL - structures concurrentes

⊕ **Instructions concurrentes**

- Affectation **inconditionnelle** d'un signal
signal <= expression (avec les signaux et/ou avec les constantes);
- Affectation **conditionnelle** d'un signal
signal <= expression1 WHEN condition ELSE expression2;
- Affectation **sélective** d'un signal
WITH *selecteur* **SELECT**
signal <= expression1 WHEN valeur_selecteur, ...;

⊕ **Instanciation de composant**

⊕ **Duplication multiple du matériel**

étiquette : **FOR** *variable_de_boucle* **IN** *intervalle* **GENERATE**
{instruction(s) concurrente(s)}
END GENERATE *étiquette*;

⊕ **Duplication conditionnelle du matériel**

étiquette : **IF** *condition* **GENERATE**
{instruction(s) concurrente(s)}
END GENERATE *étiquette*;

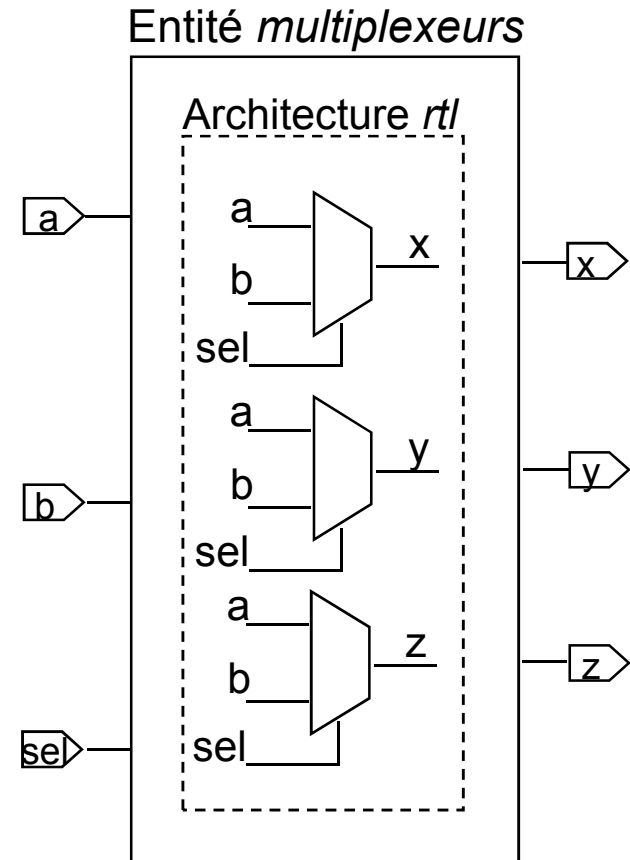
Instructions concurrentes – Exemple

```
ENTITY multiplexeurs IS
PORT (a, b, sel : IN bit;
      x, y, z   : OUT bit);
END multiplexeurs;

ARCHITECTURE rtl OF multiplexeurs IS
BEGIN
  -- affectation inconditionnelle
  x <= (a AND NOT sel) OR (b AND sel);

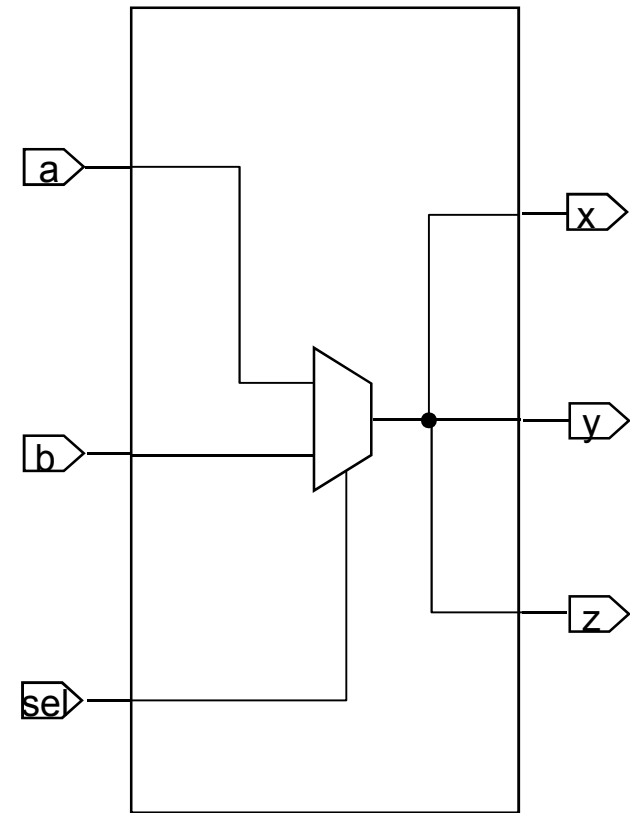
  -- affectation conditionnelle
  y <= a WHEN sel='0' ELSE
      b;

  -- affectation sélective
  WITH sel SELECT
    z <= a  WHEN '0',
        b  WHEN OTHERS;
END rtl;
```



Instructions concurrentes – Exemple (suite)

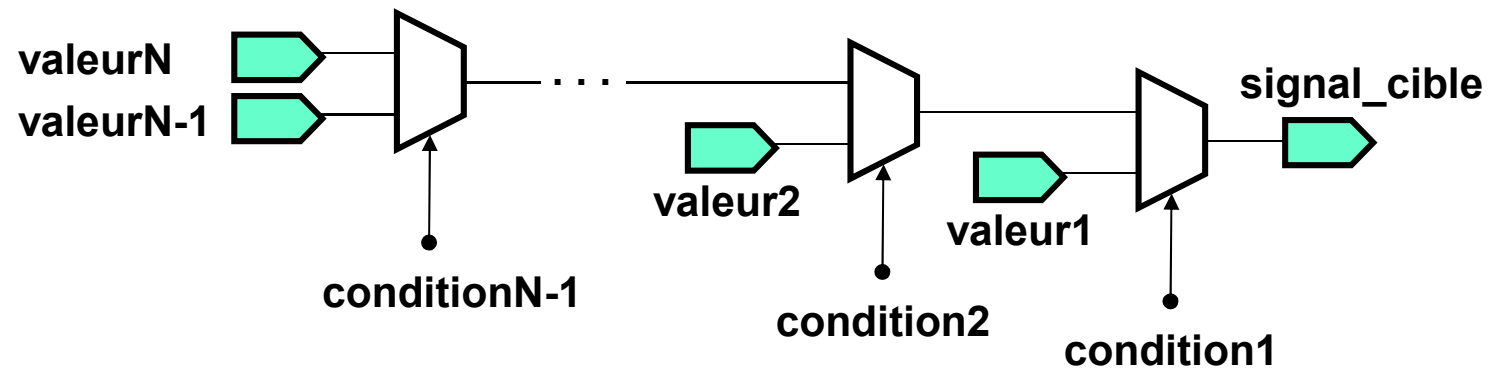
- ⊕ *Puisque les **trois descriptions décrivent finalement la même structure** logique, l'architecture **rtl** sera implanté dans le matériel de la façon suivante - deux structures redondantes **seront supprimées***



Instructions concurrentes (suite)

⊕ **Affectation conditionnelle**

```
signal_cible <= valeur1 WHEN condition1 ELSE  
valeur2 WHEN condition2 ELSE  
...  
valeurN-1 WHEN conditionN-1 ELSE  
valeurN;
```

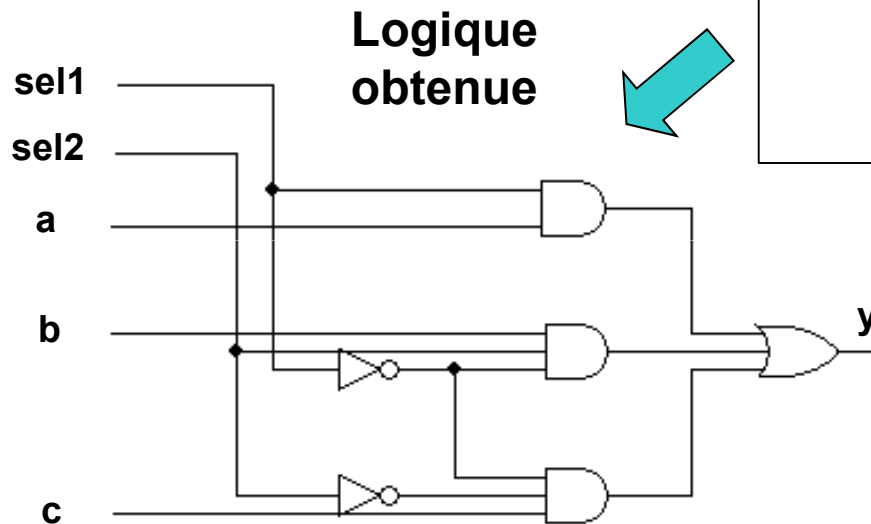


Attention : La première condition (condition1) est prioritaire devant les conditions suivantes – Codage par priorité !

Instructions concurrentes (suite)

Codage par priorité d'une affectation conditionnelle :

```
-- affectation conditionnelle  
y <= a WHEN sel1 = '1' ELSE  
  b WHEN sel2 = '1' ELSE  
  c;
```



La sélection du signal *a* avec le signal *sel1* est *prioritaire* devant *b* et *c* !

```
y = (sel1 AND a)  
  OR ((NOT sel1) AND sel2 AND b)  
  OR ((NOT sel1) AND (NOT sel2) AND c)
```

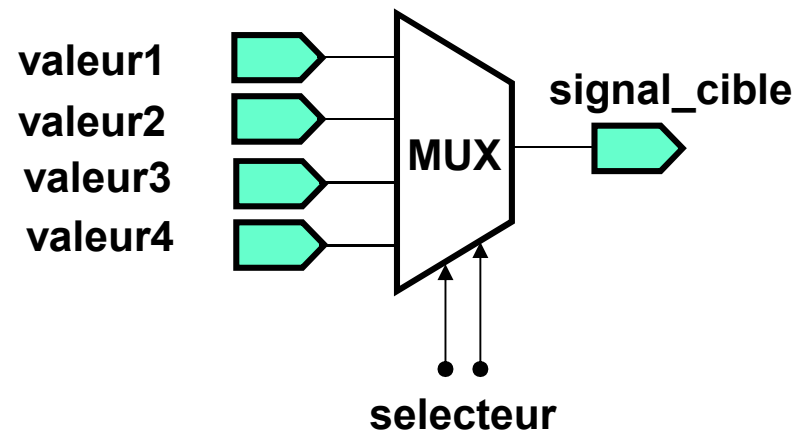
Instructions concurrentes (suite)

⊕ *Affectation sélective*

WITH *selecteur* **SELECT**

```
signal_cible <= valeur1 WHEN valeur_selecteur1,  
                valeur2 WHEN valeur_selecteur2,  
                . . .  
                valeurN WHEN OTHERS;
```

Exemple d'application : **multiplexeur**




Instructions concurrentes (suite)

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

ENTITY multiplexeur IS
PORT (a, b, c, d : IN std_logic;
      sel          : IN std_logic_vector(1 DOWNTO 0);
      y           : OUT std_logic);
END multiplexeur;

ARCHITECTURE rtl OF multiplexeur IS
BEGIN
  -- affectation sélective
  WITH sel SELECT
    y <= a  WHEN "00",
         b  WHEN "01",
         c  WHEN "10",
         d  WHEN OTHERS;
END rtl;
```

Sélection sans priorité !



```
y = (a AND (NOT sel(1)) AND (NOT sel(0)))
     OR (b AND (NOT sel(1)) AND sel(0))
     OR (c AND sel(1) AND (NOT sel(0)))
     OR (d AND sel(1) AND sel(0))
```

Implantation des fonctions logiques simples

- ✦ **Implantation basée sur l'utilisation d'opérateurs de base**
AND, OR, NOT, XOR, NAND, NOR
- ✦ **Quelques exemples d'implantation de la fonction ET logique en VHDL**

- **Ver 1 :**

```
y <= a AND b;
```

- **Ver 2 :**

```
y <= '1' WHEN (a='1' AND b='1') ELSE '0';
```

- **Ver 3 :**

```
y <= '0' WHEN (a='0' OR b='0') ELSE '1';
```

- **Ver 4 :**

```
s <= a & b;
```

```
WITH s SELECT
```

```
y <= '1' WHEN "11",  
      '0' WHEN OTHERS;
```

**Concaténation
de a et b**

**N'oubliez pas les
branches alternatives !!!**

Implantation des fonctions logiques évoluées

- ⊕ **Utilisation de l'affectation conditionnelle si certains signaux sont prioritaires (sélecteurs avec priorité)**

- **Ex. :**

```
y <= a WHEN sel(0)='1' ELSE  
    b WHEN sel(1)='1' ELSE  
    c;
```

- ⊕ **Utilisation de l'affectation sélective pour obtenir une structure régulière sans priorité (multiplexeurs)**

- **Ex. :**

```
WITH sel SELECT  
    y' <= a WHEN "00",  
    b WHEN "01",  
    c WHEN OTHERS;
```

**N'oubliez pas les
branches alternatives !!!**

Fonctions combinatoires : Générateurs de parité

- ⊕ **Générateur de parité** – génèrent le bit de parité qui permet de détecter les erreurs survenues pendant la transmission ou la sauvegarde (dans la mémoire) de données
 - ⊕ **Principe** – le bit de parité est égal à un si le nombre de uns dans la donnée est impaire – pour une donnée de N bits on peut le réaliser avec une fonction logique OU EXCLUSIF à N entrées
- A noter : un nombre paire d'erreurs n'est pas détecté ...*

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

ENTITY parity IS
PORT (a      : IN std_logic_vector(7 DOWNT0 0);
      par_out : OUT std_logic);
END parity;

ARCHITECTURE rtl OF parity IS
BEGIN
-- assignation inconditionnelle
    par_out <= a(7) XOR a(6) XOR a(5) XOR a(4)
              XOR a(3) XOR a(2) XOR a(1) XOR a(0);
END rtl;
```

Fonctions combinatoires : Générateur de parité paramétrable

- ⊕ **Fonction générique** – fonction universelle, adaptable aux besoins – la largeur du générateur peut être définie par le paramètre *N*
- ⊕ **Exemple de la réalisation** – structure **FOR ... GENERATE** :

étiquette : **FOR** *variable_de_boucle* **IN** *intervalle* **GENERATE**

[déclarations]

BEGIN

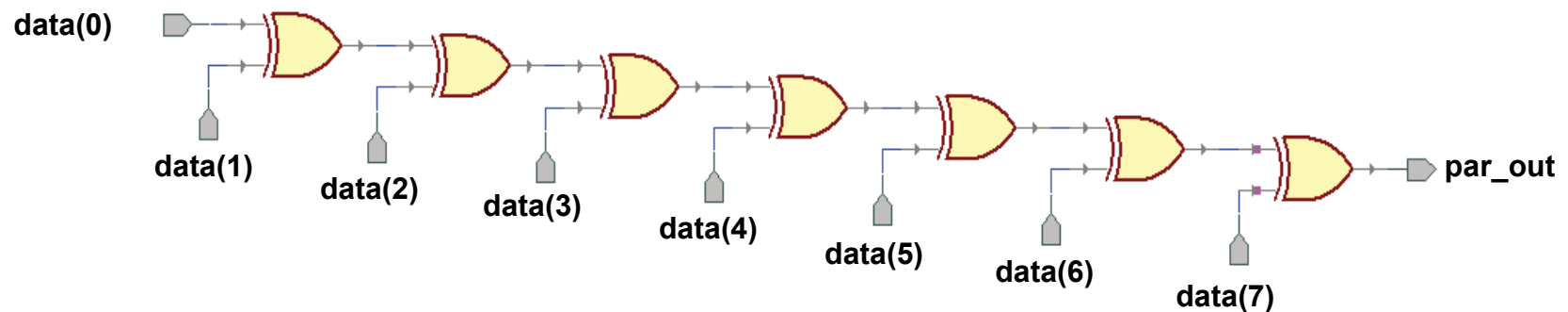
{*assignation(s) concurrente(s)*}

END GENERATE *étiquette*;

Optionnelles

Étiquette
obligatoire

Structure à réaliser pour *N* = 8 :



Fonctions combinatoires : Générateur de parité paramétrable (suite)

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

ENTITY parity IS
  GENERIC (WIDTH : INTEGER := 8);
  PORT (data      : IN std_logic_vector(WIDTH-1 DOWNT0 0);
        par_out   : OUT std_logic);
END parity;

ARCHITECTURE rtl OF parity IS
  SIGNAL par_int : std_logic_vector(WIDTH-1 DOWNT0 0);

  BEGIN
    par_int(0) <= data(0);
    par_out <= par_int(WIDTH-1);
    Parity_gen:
      FOR i IN 1 TO (WIDTH-1) GENERATE
        par_int(i) <= data(i) XOR par_int(i-1);
      END GENERATE Parity_gen;
  END rtl;
```

On peut modifier la largeur
du générateur en modifiant
le paramètre WIDTH

Nous pouvons utiliser le signal
par_int(WIDTH-1) avant qu'il soit
affecté (structure concurrente)

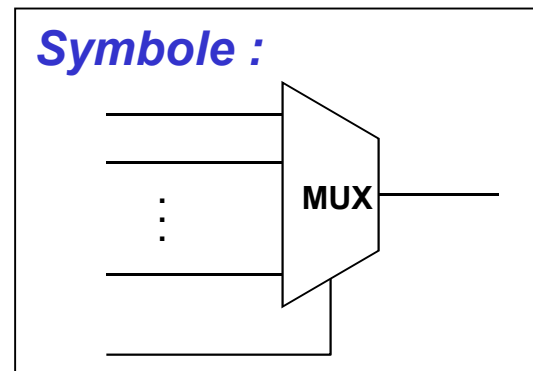
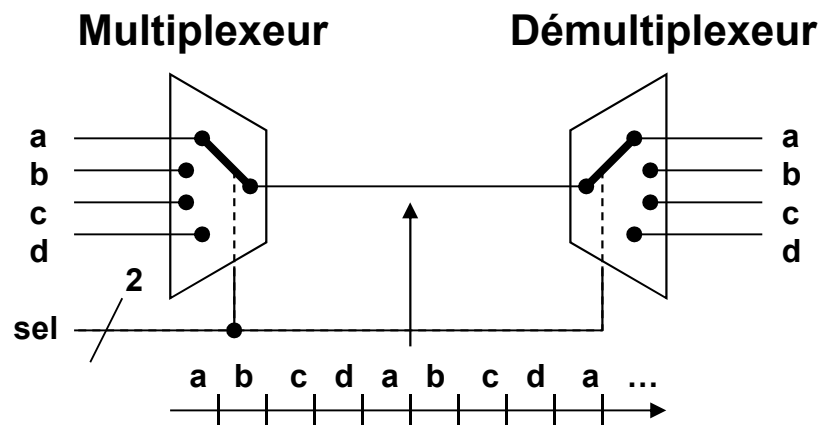
↓

```
par_out = data(0) XOR data(1) XOR data(2) XOR data(3)
          XOR data(4) XOR data(5) XOR data(6) XOR data(7)
```

Fonctions combinatoires : Multiplexeurs et démultiplexeurs

- ✚ **Multiplexeurs** – permettent de sélectionner un parmi plusieurs signaux d'entrée
- ✚ **Démultiplexeurs** – permettent de rediriger la valeur présente à l'entrée vers une parmi plusieurs sorties

Schéma du principe :



- ✚ **Exemple d'utilisation** – transfert de plusieurs données (4 dans notre cas) par une liaison série – une tranche de temps est attribuée périodiquement à chaque canal

Fonctions combinatoires :


Implantation d'un multiplexeur à 4 entrées

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

ENTITY multiplexer IS
PORT (a, b, c, d : IN std_logic;
      sel          : IN std_logic_vector(1 DOWNTO 0);
      y           : OUT std_logic);
END multiplexer;

ARCHITECTURE rtl OF multiplexer IS
BEGIN
  -- assignation sélective
  WITH sel SELECT
    y <= a  WHEN "00",
         b  WHEN "01",
         c  WHEN "10",
         d  WHEN OTHERS;
END rtl;
```

Sélection sans priorité !

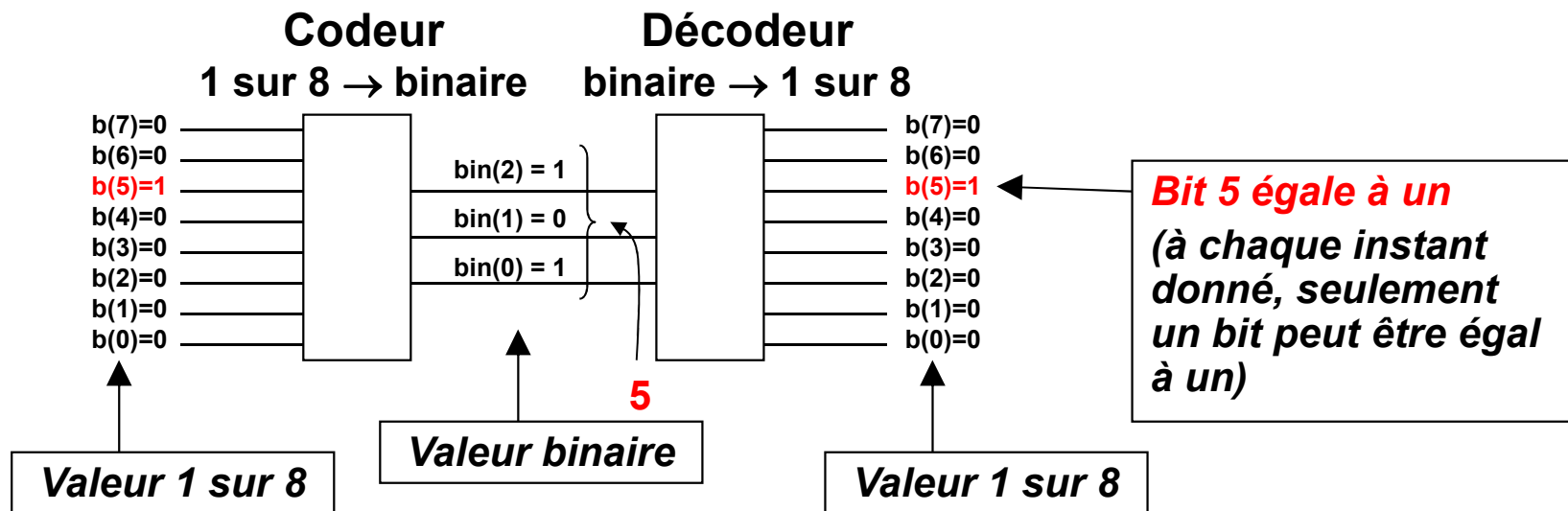


$$y = (a \text{ AND } (\text{NOT } sel(1)) \text{ AND } (\text{NOT } sel(0)))$$
$$\text{OR } (b \text{ AND } (\text{NOT } sel(1)) \text{ AND } sel(0))$$
$$\text{OR } (c \text{ AND } sel(1) \text{ AND } (\text{NOT } sel(0)))$$
$$\text{OR } (d \text{ AND } sel(1) \text{ AND } sel(0))$$

Fonctions combinatoires : Codeurs et décodeurs

- ⊕ **Codeurs** – permettent de coder une donnée (binaire) d'une manière plus efficace ou d'une manière plus avantageuse
- ⊕ **Décodeurs** – réalisent une opération inverse par rapport au codage et permettent de représenter une information d'une manière mieux compréhensible ou plus facile à utiliser

Exemple : Codeur 1 sur N → binaire



Fonctions combinatoires : Codeurs et décodeurs (suite)

- ⊕ **Codeur avec priorité** – donne le numéro (en binaire) du canal égal à un ; si plusieurs canaux sont égaux à un, c'est le numéro du canal prioritaire, qui est donné à la sortie

Exemple – Codeur « 1 sur 3 → binaire » avec priorité

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

ENTITY priority_encod IS
PORT (a, b, c : IN std_logic;
      bin_out : OUT std_logic_vector(1 DOWNTO 0));
END priority_encod;

ARCHITECTURE rtl OF priority_encod IS
BEGIN
  -- assignation conditionnelle
  bin_out <= "01" WHEN a = '1' ELSE
             "10" WHEN b = '1' ELSE
             "11" WHEN c = '1' ELSE
             "00";
END rtl;
```

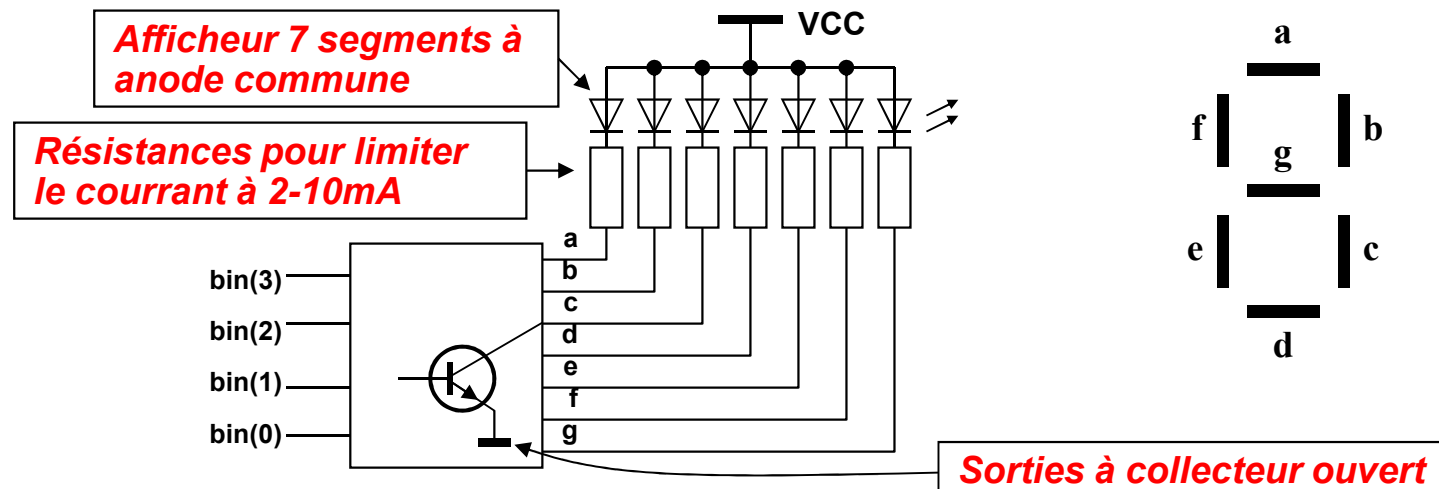
**a est prioritaire
devant b, qui
est prioritaire
devant c**

Fonctions combinatoires : Codeurs et décodeurs (suite)

- ❖ **Décodeur « code binaire → code afficheur sept segments »**
 - permet d'afficher une valeur binaire de 4 bits en hexadécimal sur un afficheur (avec les diodes LED) de sept segments

Schéma d'interconnexion :

Codage de segments :



Affichage de caractères hexadécimaux :

0 1 2 3 4 5 6 7 8 9 A b c d e f

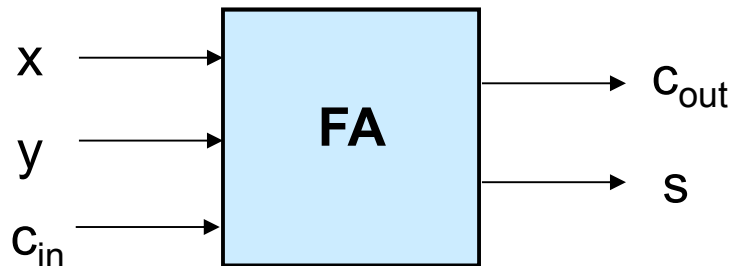
Fonctions combinatoires : Blocs arithmétiques

⊕ Opérations arithmétiques peuvent être réalisées en VHDL

- **Au niveau bits** (niveau bas d'abstraction)
- **Au niveau d'opérateurs arithmétiques** (seulement les opérateurs d'addition et de soustraction sont supportés pour la synthèse)

⊕ Opérateur d'addition bit par bit – additionneur complet **Full Adder - FA**

Symbole



Somme :

$$s = x \text{ XOR } y \text{ XOR } c_{in}$$

Retenue :

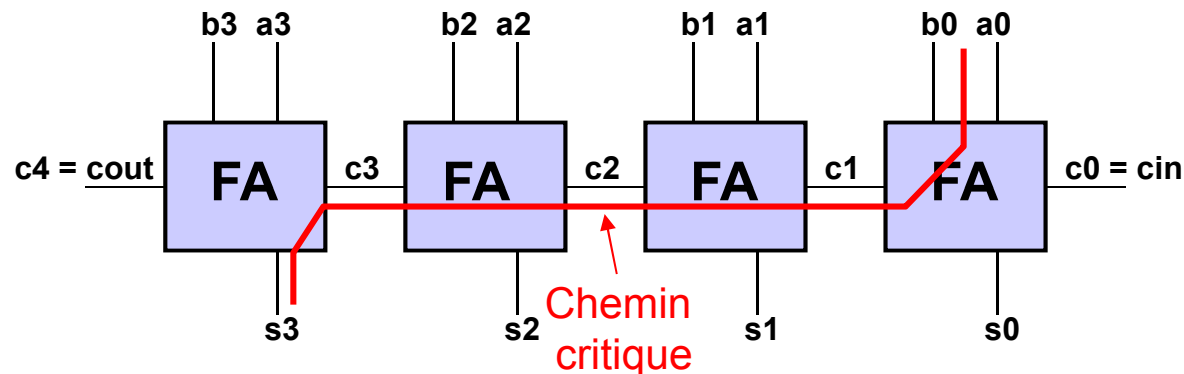
$$c_{out} = (x \text{ AND } y) \text{ OR } (c_{in} \text{ AND } x) \\ \text{OR } (c_{in} \text{ AND } y)$$

Tableau de vérité

x	y	c _{in}	C _{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fonctions combinatoires : Blocs arithmétiques (suite)

- ✦ **Réalisation d'un additionneur de quatre bits basé sur un additionneur complet**



- ✦ **Chemin critique** – le chemin le plus long dans le flot de données (entre une entrée et une sortie quelconque) – il correspond à la fonction logique du bloc la plus complexe
 - Ici – nous avons 5 sorties, donc 5 fonctions logiques ($c_{out}, s_0, s_1, s_2, s_3$), les fonctions les plus complexes sont c_{out} et s_3 – elles représentent les chemins critiques du bloc

Fonctions combinatoires : Blocs arithmétiques (suite)

- ✦ **Réalisation d'un additionneur de quatre bits** en utilisant un opérateur arithmétique (haut niveau d'abstraction)

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

ENTITY addition IS
PORT (a, b      : IN  std_logic_vector(3 DOWNTO 0);
      s         : OUT std_logic_vector(3 DOWNTO 0);
      c_out     : OUT std_logic);
END addition;

ARCHITECTURE rtl OF addition IS
    SIGNAL s_int : std_logic_vector(4 DOWNTO 0);
BEGIN
    s_int <= ('0' & a) + ('0' & b);
    s      <= s_int (3 DOWNTO 0);
    c_out  <= s_int(4);
END rtl;
```

**Paquetage nécessaire pour
les opérations arithmétiques**

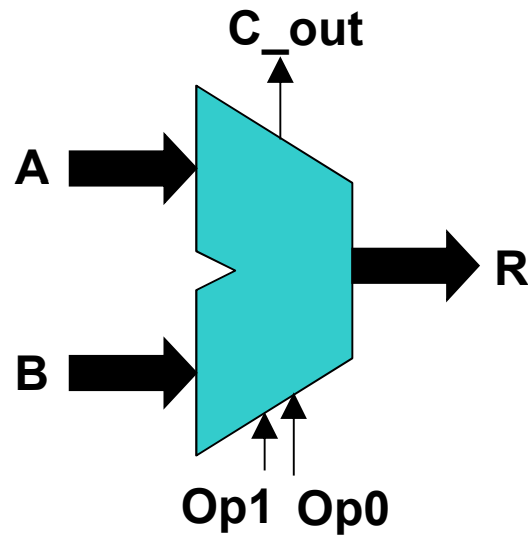
**Extension du a et b sur 5 bits
(le cinquième bit sera la retenue)**

A noter :

En réalité, le compilateur traduit la description de haut niveau d'abstraction (abstrait du matériel) en une structure d'addition bit par bit (probablement en utilisant l'additionneur complet), qui correspond à la structure logique disponible dans le matériel

Fonctions combinatoires : Blocs arithmétiques (suite)

⊕ *Unité arithmétique et logique*



Op1	Op0	Opération
0	0	$R = A + B$
0	1	$R = A - B$
1	0	$R = A \text{ and } B$
1	1	$R = A \text{ or } B$

Fonctions combinatoires : Blocs arithmétiques (suite)

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

ENTITY alu IS
PORT (a, b      : IN std_logic_vector(7 DOWNTO 0);
      op1, op0  : IN std_logic;
      r        : OUT std_logic_vector(7 DOWNTO 0);
      c_out    : OUT std_logic);
END alu;

ARCHITECTURE rtl OF alu IS
SIGNAL oper      : std_logic_vector(1 DOWNTO 0);
SIGNAL int       : std_logic_vector(8 DOWNTO 0);

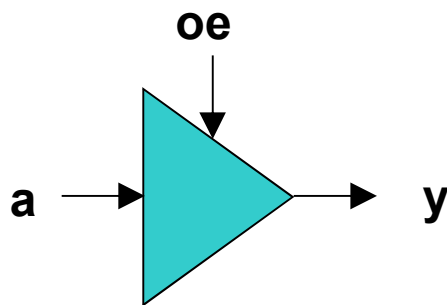
BEGIN
    oper    <= op1 & op0;  -- code operation
    c_out   <= int(8);
    r      <= int(7 DOWNTO 0);
    WITH oper SELECT
        int <= (('0' & a) + ('0' & b)) WHEN "00",
              (('0' & a) - ('0' & b)) WHEN "01",
              ('0' & (a AND b))      WHEN "10",
              ('0' & (a OR b))       WHEN OTHERS;

END rtl;
```

**Paquetage nécessaire pour les
opérations arithmétiques signées**

Fonctions combinatoires : Sorties commandées

⊕ *Sorties trois états*



a	oe	y
0	0	Z
0	1	0
1	0	Z
1	1	1

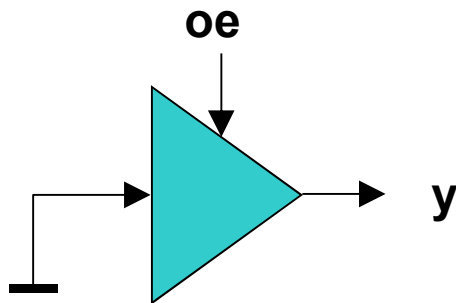
```
-- assignation conditionnelle  
y <= a WHEN oe = '1' ELSE  
    'Z';
```

y doit être déclaré comme un
signal de la logique standard et
non pas comme un bit !

Fonctions combinatoires : Sorties commandées (suite)

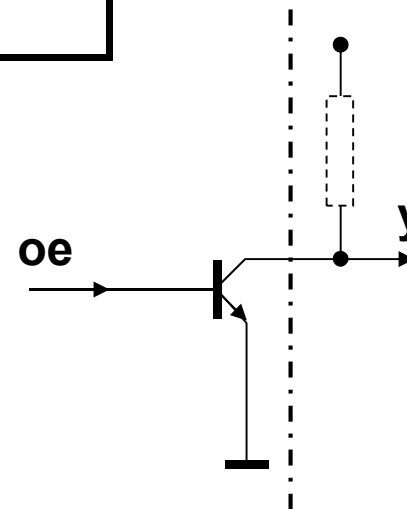
⊕ **Collecteur ouvert**

seulement deux états à la sortie sont possibles : Z et 0



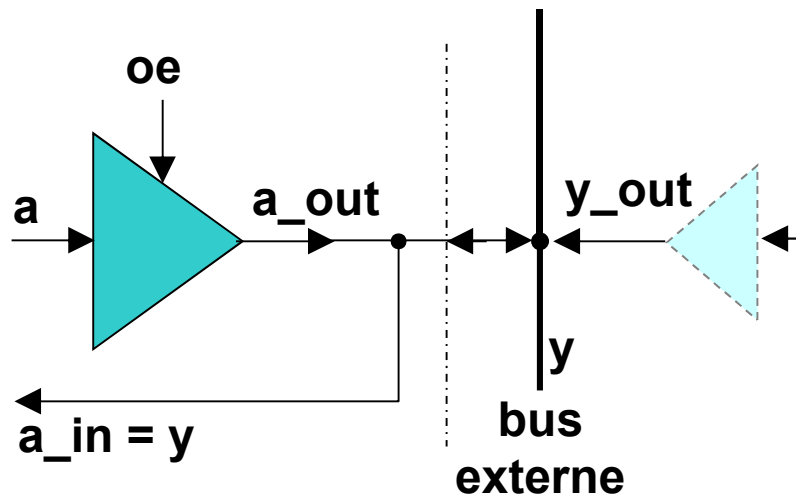
oe	y
0	Z
1	0

```
-- assignation conditionnelle  
y <= '0' WHEN oe = '1' ELSE  
    'Z';
```



Fonctions combinatoires : Sorties commandées (suite)

⊕ *Entrée sortie bidirectionnelle*



a_out et a_in peuvent avoir des valeurs différentes
(\Rightarrow deux signaux nécessaires pour la simulation)

Conflits sur le bus !

a	oe	a_out	y a_in	y_out
0	0	Z	Z	Z
0	1	0	\rightarrow 0	Z
1	0	Z	Z	Z
1	1	1	\rightarrow 1	Z
0	0	Z	0 \leftarrow	0
0	1	0	\rightarrow 0 \leftarrow	0
1	0	Z	0 \leftarrow	0
1	1	1	\rightarrow X \leftarrow	0
0	0	Z	1 \leftarrow	1
0	1	0	\rightarrow X \leftarrow	1
1	0	Z	1 \leftarrow	1
1	1	1	\rightarrow 1 \leftarrow	1

Fonctions combinatoires : Sorties commandées (suite)

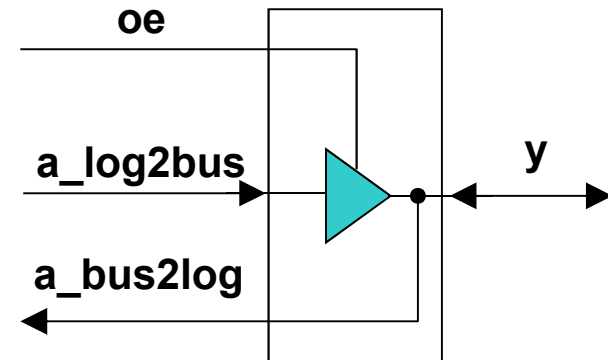
```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY es_bidir IS
PORT (y          : INOUT std_logic;
      oe         : IN std_logic;
      a_log2bus  : IN std_logic;
      a_bus2log  : OUT std_logic);
END es_bidir;

ARCHITECTURE rtl OF es_bidir IS
BEGIN
    y <= a_log2bus WHEN oe = '1' ELSE
        'Z';
    a_bus2log <= y;
END rtl;

```

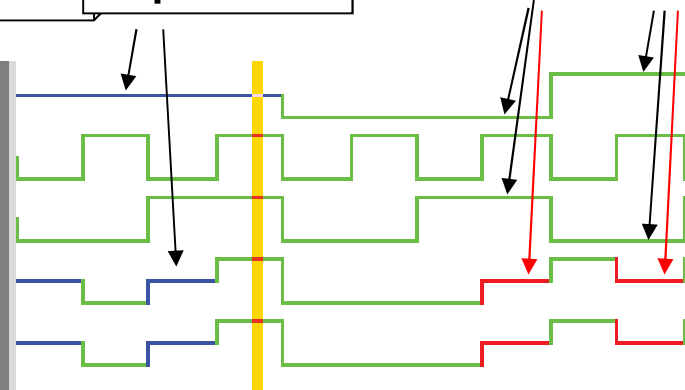


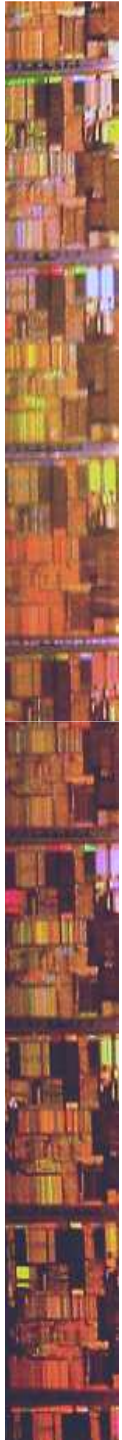
Stimulateur

/tb_es_bidir/s_y_in	Z
/tb_es_bidir/s_oe	1
/tb_es_bidir/s_a_log2bus	1
/tb_es_bidir/s_a_bus2log	1
/tb_es_bidir/s_y	1

Haute
impédance

Conflits sur le bus





Chapitre 7

Fonctions de base de la logique séquentielle

Fonctions logiques séquentielles

⊕ **Fonction logique séquentielle - définition**

- La valeur à la sortie d'une fonction logique séquentielle **dépend de l'état actuel des entrées ET des états passés** – ceci implique l'utilisation d'un élément de mémorisation

⊕ **Deux types principaux de logique séquentielle**

- **Logique asynchrone** – l'état de la logique peut **changer à tout instant**
- **Logique synchrone** – l'état de la logique peut changer **qu'aux instants donnés** – front montant ou front descendant d'un signal d'horloge

⊕ **Fonctions séquentielles de base**

- **Bascules asynchrones** – RS, D à verrouillage (D latch)
- **Bascules synchrones** - D, T, RS, JK
- **Compteurs asynchrones et compteurs synchrones**
- **Registres, registres à décalage**
- **Machines d'états**

Blocs séquentiels élémentaires

Bascules asynchrones

- ⊕ **Bascule** – circuit bi-stable, capable de mémoriser un bit
 - L'élément de mémorisation est souvent réalisé par une **contre-réaction** (la sortie de la fonction est redirigée vers les entrées)
- ⊕ **Bascule RS asynchrone** – bascule constituant la base de toutes les autres bascules (**S = Set = mise à un**, **R = Reset = remise à zéro**)
- ⊕ **Bascule RS avec les portes NAND**

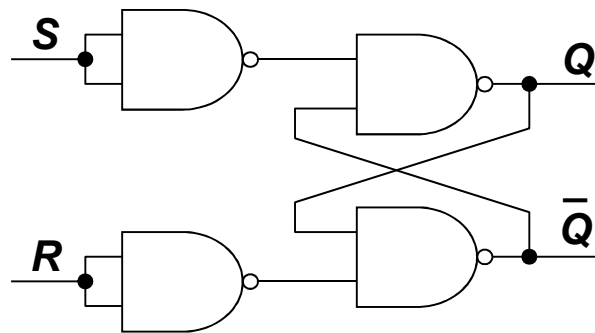


Tableau de vérité

S	R	Q^+	nQ^+	État suivant
0	0	$\neg Q$	$\neg nQ$	État précédent
0	1	0	1	Mise à zéro
1	0	1	0	Mise à un
1	1	Ambiguïté		Interdit

Blocs séquentiels élémentaires

Bascules asynchrones (suite)

⊕ *Basculer RS à verrouillage*

Si l'entrée $Ena = 0$, la bascule est verrouillée

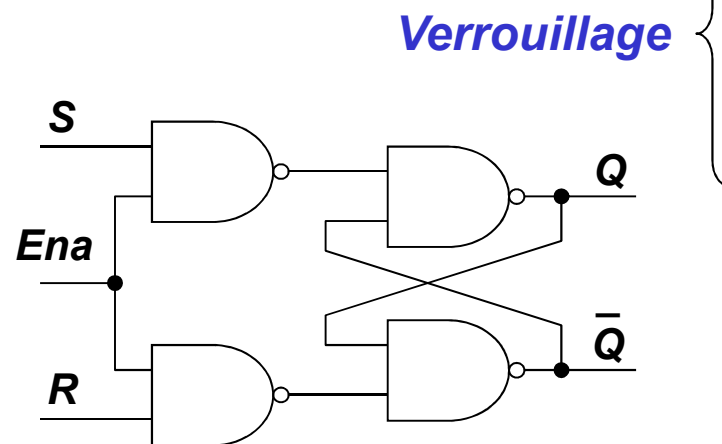


Tableau de vérité

Ena	S	R	Q^+	nQ^+
0	0	0	$\neg Q$	$\neg nQ$
0	0	1	$\neg Q$	$\neg nQ$
0	1	0	$\neg Q$	$\neg nQ$
0	1	1	$\neg Q$	$\neg nQ$
1	0	0	$\neg Q$	$\neg nQ$
1	0	1	0	1
1	1	0	1	0
1	1	1	Ambiguïté	

Blocs séquentiels élémentaires

Basculas asynchrones (suite)

⊕ Bascule D à verrouillage – « D Latch »

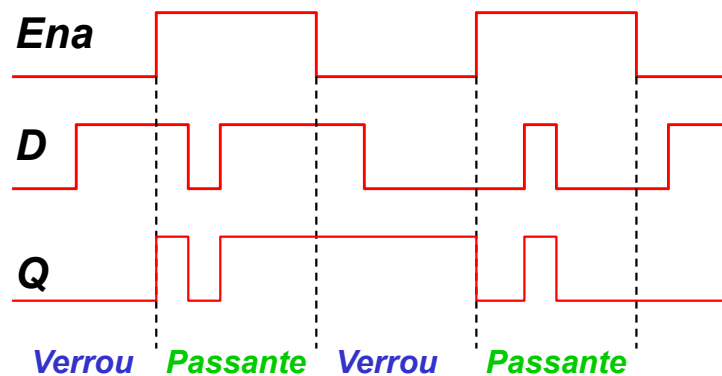
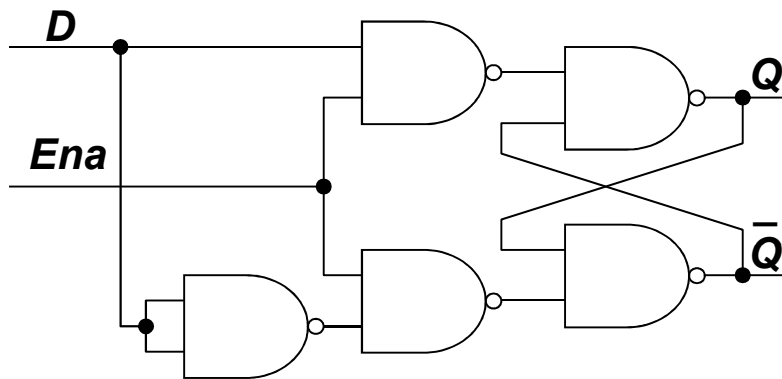


Tableau de vérité

Ena	D	Q^+	nQ^+
0	0	$\neg Q$	$\neg nQ$
0	1	$\neg Q$	$\neg nQ$
1	0	0	1
1	1	1	0

Blocs séquentiels élémentaires

Bascules asynchrones (suite)

⊕ *Bascule D à verrouillage – « D Latch » en flot de données*

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

ENTITY d_latch IS
PORT ( d      : IN std_logic;
      ena     : IN std_logic;
      q       : OUT std_logic
    );
END d_latch;

ARCHITECTURE data_flow OF d_latch IS
  SIGNAL n_s, n_r      : std_logic;
  SIGNAL q_int, n_q_int : std_logic;
BEGIN
  n_s      <= NOT (d AND ena);
  n_r      <= NOT ((NOT d) AND ena);
  --
  n_q_int <= NOT (q_int AND n_r);
  q_int   <= NOT (n_q_int AND n_s);
  --
  q       <= q_int;
END data_flow;
```

Description flot de données basée sur la figure précédente, difficile d'estimer son comportement

Solution :
Description comportementale, voir la suite

Structures VHDL séquentielles de base

⊕ **Utilisées seulement à l'intérieur d'un processus (PROCESS), fonction (FUNCTION) et procédure (PROCEDURE) !**

⊕ **Quatre structures séquentielles de base :**

- **Affectation d'un signal**

signal <= expression (avec les signaux);

- **Structure conditionnelle**

IF *condition1* **THEN**

{instruction(s) séquentielle(s)}

[ELSIF *condition2* **THEN**

{instruction(s) séquentielle(s)}]

[ELSE

{Instruction(s) séquentielle(s)}]

...

END IF;

**Parties
optionnelles**

Structures VHDL séquentielles de base (suite)

- **Structure sélective**

CASE *selecteur* **IS**

WHEN *valeur_selecteur1* **=>**

{instruction(s) séquentielle(s)}

WHEN *valeur_selecteur2* **=>**

{instruction(s) séquentielle(s)}

WHEN *valeur_selecteur3* **=>**

{instruction(s) séquentielle(s)}

...

[WHEN OTHERS =>

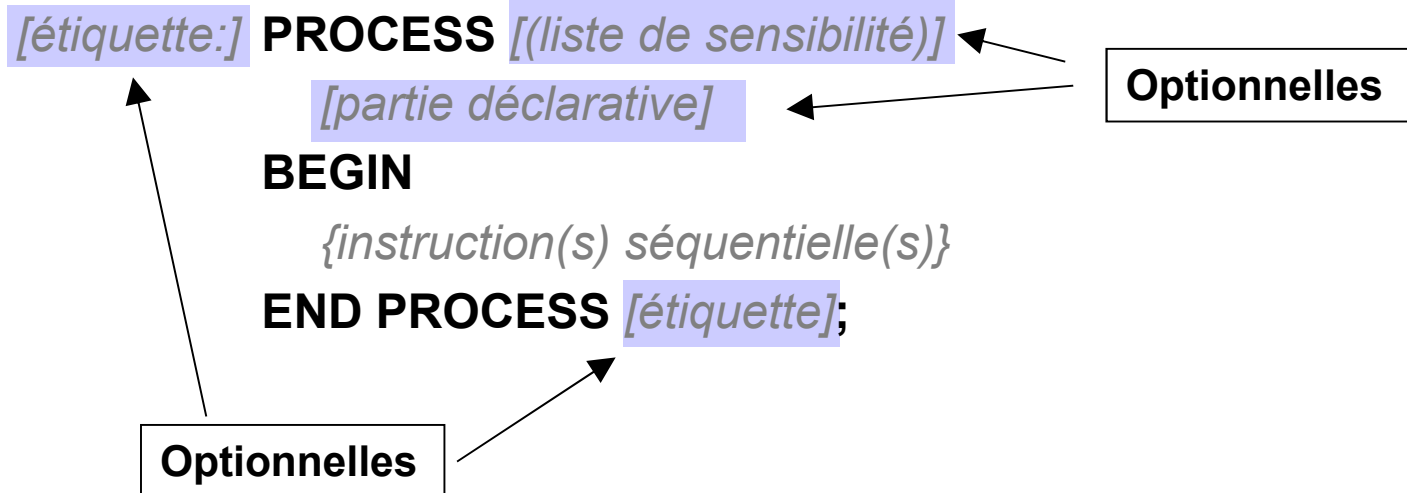
{instruction(s) séquentielle(s)}]

END CASE;

← **Partie optionnelle,
mais conseillée**

PROCESS

- ⊕ Suite d'instructions VHDL avec un **comportement séquentiel**
- ⊕ **Ordre d'instructions à l'intérieur du PROCESS - important !**
- ⊕ Trois phases du PROCESS :
 - Repos, activation, exécution
- ⊕ **Syntaxe :**



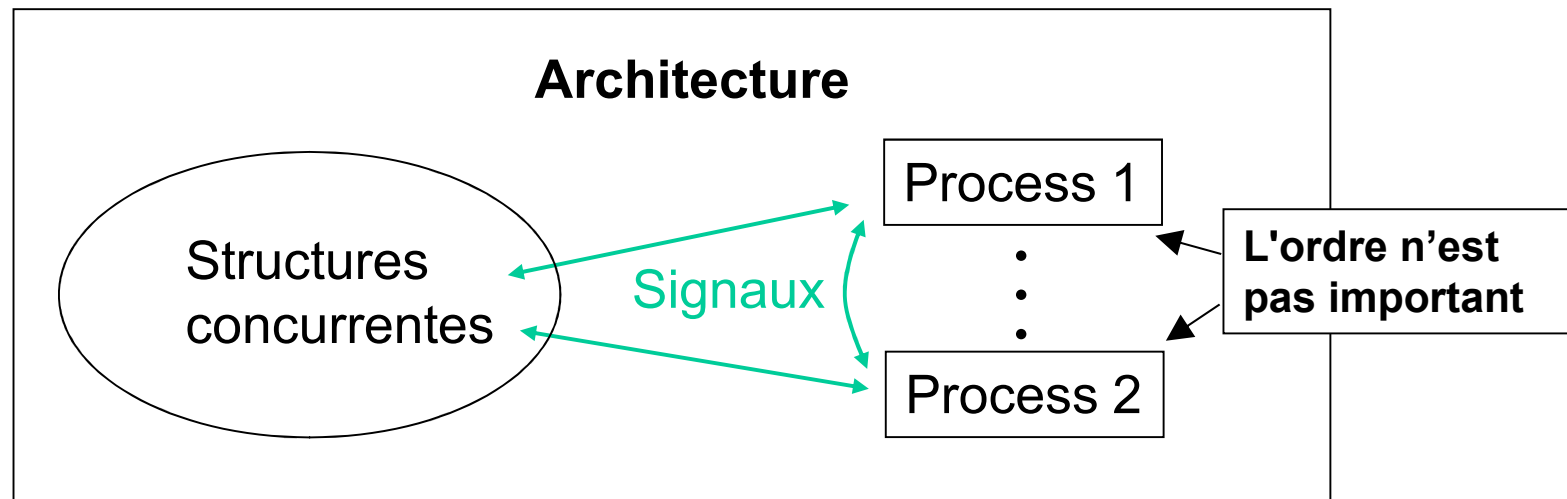


Déclenchement du PROCESS et mise à jour des valeur des signaux

- ⊕ Deux possibilités de déclenchement :
 - **A chaque changement d'état d'un de ses signaux d'activation** donnés dans la liste de sensibilité (plusieurs signaux d'activation peuvent être suivis), ce type de déclenchement est utilisé plutôt pour réaliser les :
 - Latches,
 - Registres
 - Machines d'états
 - Après un temps d'attente limité par un événement (WAIT UNTIL) ou par la durée spécifié - WAIT FOR (un seul paramètre peut être suivi), ce type de déclenchement est utilisé plutôt dans les :
 - Bancs de test
- ⊕ Les **signaux** sont **évalués pendant** le PROCESS, mais **miss à jour à la fin** de PROCESS
- ⊕ Les variables sont évaluées et mises à jour immédiatement

Utilisation de plusieurs PROCESS

- ⊕ Une architecture peut contenir **plusieurs PROCESS**
- ⊕ Les PROCESS sont **exécutés en parallèle** – ils se trouvent dans la partie concurrente de l'architecture
- ⊕ A l'intérieur du PROCESS les instructions sont **exécutés séquentiellement**
- ⊕ La réduction du nombre de PROCESS **améliore la lisibilité**



Blocs élémentaires de la logique séquentielle

Bascules asynchrones (suite)

⊕ *Bascule D à verrouillage – « D Latch » en description comportementale*

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

ENTITY d_latch IS
PORT ( d    : IN std_logic;
      ena   : IN std_logic;
      q     : OUT std_logic
    );
END d_latch;

ARCHITECTURE behavior OF d_latch IS
BEGIN
    PROCESS (ena, d)
    BEGIN
        IF ena = '1' THEN
            q <= d;
        END IF;
    END PROCESS;
END behavior;
```

La liste de sensibilité contient
les deux entrées

Blocs séquentiels élémentaires

Bascules synchrones (suite)

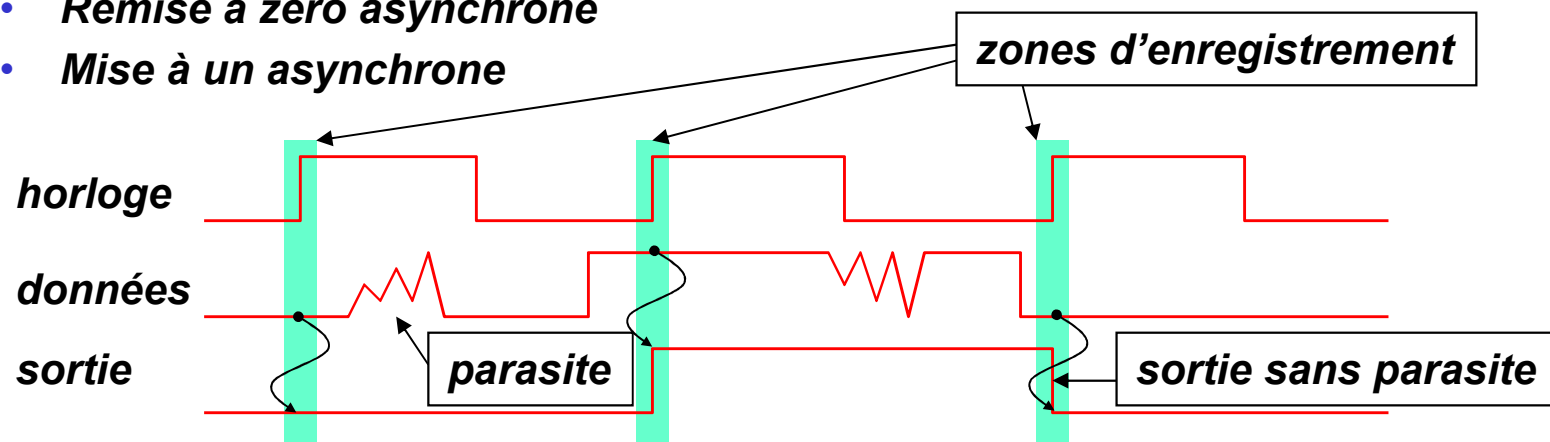
⊕ **Bascule synchrone** – circuit bi-stable, capable de changer son état sur le front montant (ou descendant) d'un signal d'horloge

⊕ **Types de base**

- **Bascule D**
- **Bascule JK**
- **Bascule T**
- **Bascule RS**

⊕ A part les entrées de données synchrones, une ou deux **entrées asynchrones de contrôle** peuvent être employées

- Remise à zéro asynchrone
- Mise à un asynchrone



Blocs séquentiels élémentaires

Bascules synchrones (suite)

- ⊕ **Bascule D synchrone** – la valeur qui se trouve à l'entrée D pendant le front montant du signal d'horloge est recopiée à la sortie, puis mémorisée jusqu'au prochain front montant

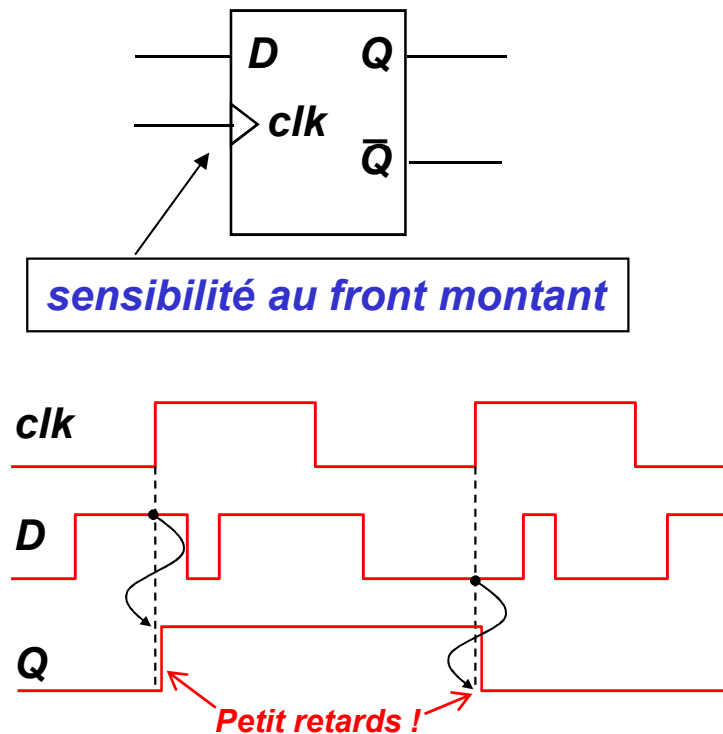


Tableau de vérité

clk	D	Q^+	nQ^+
0	x	\bar{Q}	\bar{nQ}
1	x	\bar{Q}	\bar{nQ}
↓	x	\bar{Q}	\bar{nQ}
↑	0	0	1
↑	1	1	0

Blocs séquentiels élémentaires

Bascules synchrones (suite)

⊕ *Bascule D synchrone en description comportementale*

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

ENTITY d_ff IS
PORT ( d    : IN std_logic;
      clk   : IN std_logic;
      q     : OUT std_logic
    );
END d_ff;

ARCHITECTURE behavior OF d_ff IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF rising_edge(clk) THEN
      q <= d;
    END IF;
  END PROCESS;
END behavior;
```

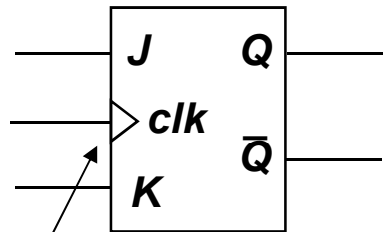
La bibliothèque `std_logic` est nécessaire pour utiliser la fonction ***rising_edge*** ou ***falling_edge***

L'affectation dans la partie synchrone du ***PROCESS*** implique la création d'une bascule

Blocs séquentiels élémentaires

Bascules synchrones (suite)

- ⊕ **Bascule JK synchrone** – comportement similaire à la bascule RS (l'entrée J fonctionne comme S et K comme R), sauf si J et K sont égaux à un, la sortie est inversée



sensibilité au front montant

sensibilité au front descendant

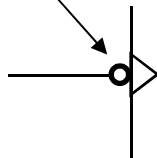


Tableau de vérité

clk	J	K	Q ⁺	nQ ⁺	État suivant
0	x	x	$\neg Q$	$\neg nQ$	État précédent
1	x	x	$\neg Q$	$\neg nQ$	État précédent
↓	x	x	$\neg Q$	$\neg nQ$	État précédent
↑	0	0	$\neg Q$	$\neg nQ$	État précédent
↑	1	0	1	0	Mise à un
↑	0	1	0	1	Mise à zéro
↑	1	1	$\neg nQ$	$\neg Q$	Inversion

Blocs séquentiels élémentaires

Bascules synchrones (suite)

⊕ *Bascule JK synchrone en description comportementale*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY jk_ff IS
PORT ( j, k, clk : IN std_logic;
      q           : OUT std_logic
      );
END jk_ff;
ARCHITECTURE behavior OF jk_ff IS
    SIGNAL sel    : std_logic_vector(1 DOWNTO 0);
    SIGNAL q_int  : std_logic;
BEGIN
    sel <= j & k;
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            CASE sel IS
                WHEN "10" => q_int <= '1';
                WHEN "01" => q_int <= '0';
                WHEN "11" => q_int <= NOT q_int;
                WHEN OTHERS => q_int <= q_int;
            END CASE;
        END IF;
    END PROCESS;
    q <= q_int;
END behavior;
```

mise à un

mise à zéro

inversion

mémorisation

Blocs séquentiels élémentaires

Bascules synchrones (suite)

- ⊕ **Basculer T synchrone** – la valeur à la sortie Q est inversée à chaque coup d'horloge (ici le front montant du signal d'horloge), si l'entrée T est égale à un, sinon la bascule mémorise la dernière valeur

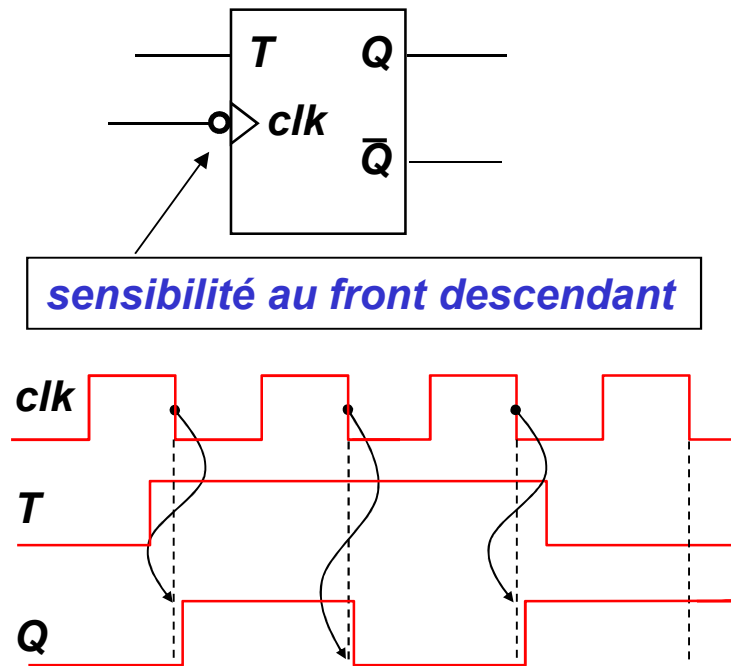


Tableau de vérité

clk	T	Q^+	$\bar{n}Q^+$
0	x	\bar{Q}	$\bar{n}Q$
1	x	\bar{Q}	$\bar{n}Q$
↑	x	\bar{Q}	$\bar{n}Q$
↓	0	\bar{Q}	$\bar{n}Q$
↓	1	$\bar{n}Q$	\bar{Q}

Blocs séquentiels élémentaires

Bascules synchrones (suite)

⊕ *Bascule T synchrone en description comportementale*

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

ENTITY t_ff IS
PORT ( t      : IN std_logic;
      clk     : IN std_logic;
      q       : OUT std_logic
      );
END t_ff;

ARCHITECTURE behavior OF t_ff IS
  SIGNAL qi : std_logic;
BEGIN
  PROCESS (clk)
  BEGIN
    IF falling_edge(clk) THEN
      IF t = '1' THEN
        qi <= NOT qi;
      END IF;
    END IF;
    q <= qi;
  END PROCESS;
END behavior;
```

La bibliothèque *std_logic* est nécessaire pour utiliser la fonction *rising_edge* ou *falling_edge*

L'affectation dans la partie synchrone du *PROCESS* implique la création d'une bascule

Mémorisation implicite si *t = 0* (il manque *ELSE*)

Blocs séquentiels élémentaires

Bascules synchrones (suite)

- ⊕ **Entrées asynchrones d'une bascule synchrone** – ces entrées sont prioritaires par rapport au signal d'horloge
- ⊕ Deux **types d'entrées** asynchrones optionnelles
 - Remise à zéro asynchrone
 - Remise à un asynchrone
- ⊕ Deux **niveau actifs** possibles
 - Niveau actif zéro
 - Niveau actif un
- ⊕ **Exemple – bascule D avec RAZ asynchrone**

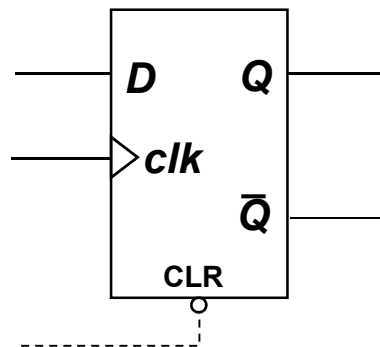


Tableau de vérité

CLR	clk	D	Q ⁺	nQ ⁺
1	0	x	$\neg Q$	$\neg nQ$
1	1	x	$\neg Q$	$\neg nQ$
1	↓	x	$\neg Q$	$\neg nQ$
1	↑	0	0	1
1	↑	1	1	0
0	x	x	0	1

Blocs séquentiels élémentaires

Bascules synchrones (suite)

⊕ *Bascule D synchrone avec remise à zéro asynchrone*

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

ENTITY d_ff IS
PORT ( d, clk : IN std_logic;
      clr      : IN std_logic;
      q       : OUT std_logic
      );
END d_ff;

ARCHITECTURE behavior OF d_ff IS
BEGIN
  PROCESS (clk, clr)
  BEGIN
    IF (clr = '0') THEN
      q <= '0';
    ELSIF rising_edge(clk) THEN
      q <= d;
    END IF;
  END PROCESS;
END behavior;
```

*Codage avec priorité –
la remise à zéro est
prioritaire par rapport
au signal d'horloge*

Blocs séquentiels de base : Registres

⊕ **Registres** – blocs logiques permettant de mémoriser la valeur binaire sur un ou plusieurs bits

Exemple d'un registre de 8 bits en VHDL :

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

ENTITY reg8 IS
PORT (clk, rst, ena: IN std_logic;
      reg_in      : IN std_logic_vector(7 DOWNTO 0);
      reg_out      : OUT std_logic_vector(7 DOWNTO 0));
END reg8;

ARCHITECTURE rtl OF reg8 IS
BEGIN
  PROCESS(clk, rst)
  BEGIN
    IF rst = '0' THEN
      reg_out <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      IF ena = '1' THEN
        reg_out <= reg_in;
      END IF;
    END IF;
  END PROCESS;
END rtl;
```

*Condition pour le
chargement du registre
évaluée aux fronts
montants de clk*

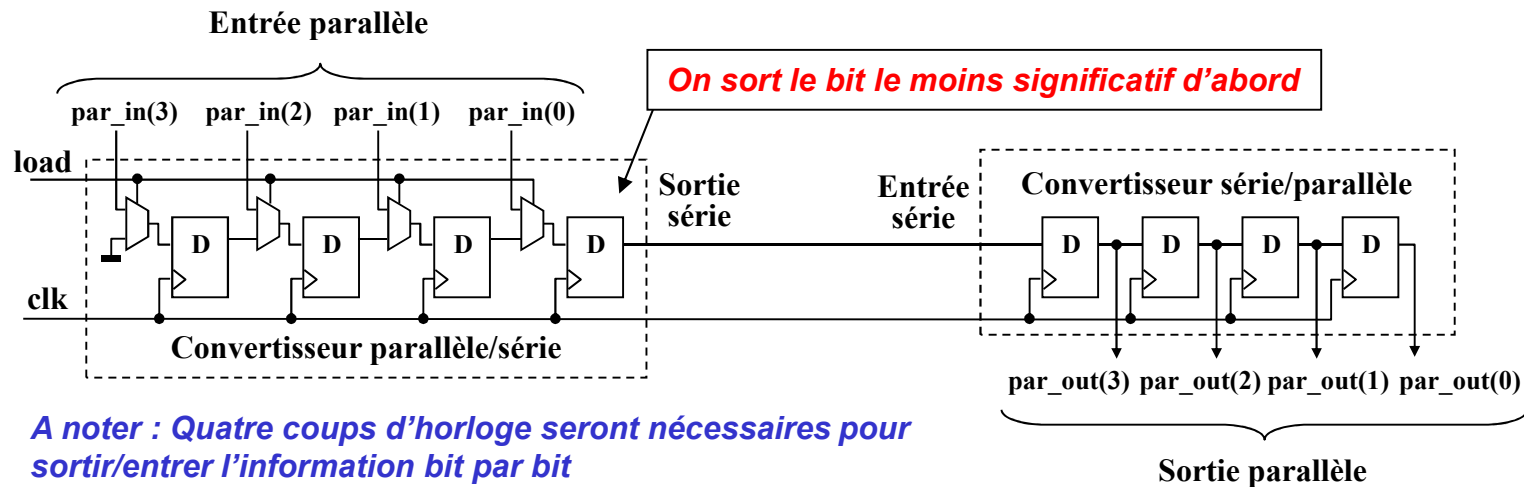
*Sans ELSE, donc
mémorisation implicite*

Blocs séquentiels de base : Registres à décalage

- ⊕ **Registre à décalage** – une suite de bascules connectées en série, utilisée pour décaler les bits à gauche ou à droite à chaque coup d'horloge
- ⊕ Deux types de registres à décalage : avec décalage à gauche et décalage à droite

Exemple d'une application :

Convertisseur du code parallèle/série et série/parallèle



Blocs séquentiels de base : Convertisseur parallèle/série

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

ENTITY par2ser IS
PORT (clk, load : IN std_logic;
      par_in      : IN std_logic_vector(7 DOWNT0 0);
      ser_out     : OUT std_logic);
END par2ser;

ARCHITECTURE rtl OF par2ser IS
  SIGNAL int_q : std_logic_vector(7 DOWNT0 0);
BEGIN
  PROCESS(clk)
  BEGIN
    IF rising_edge(clk) THEN
      IF load = '1' THEN
        int_q <= par_in;
      ELSE
        int_q(7) <= '0';
        int_q(6 DOWNT0 0) <= int_q(7 DOWNT0 1);
      END IF;
    END IF;
    ser_out <= int_q(0);
  END PROCESS;
END rtl;
```

Préchargement du registre

Décalage à droite (vers les bits moins significatifs)



Blocs séquentiels de base : Compteurs

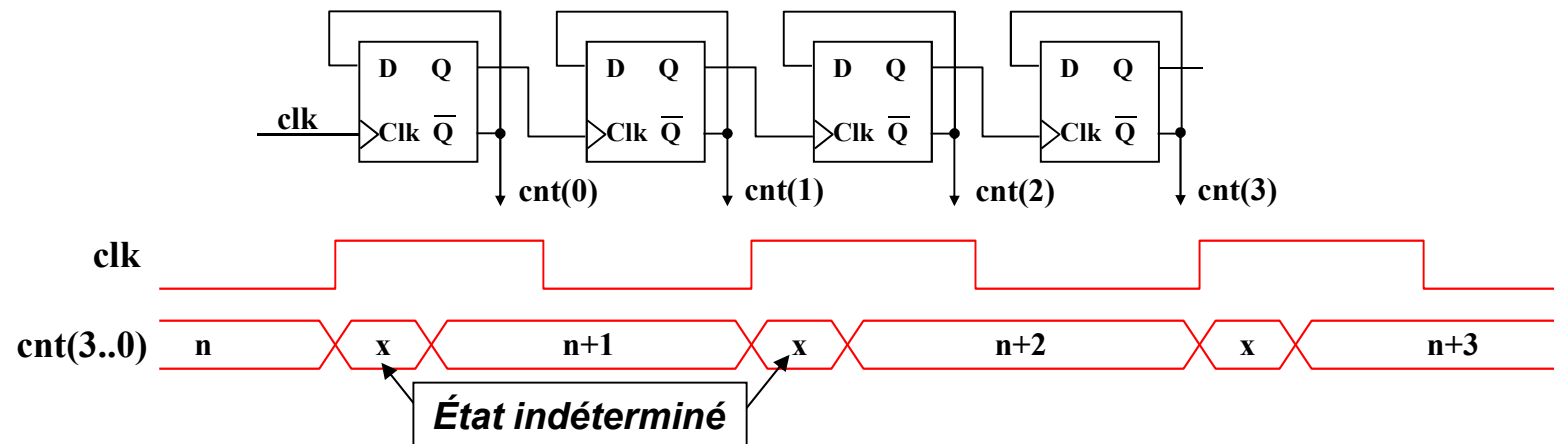
- ⊕ **Compteurs** – blocs logiques permettant de changer leur état (sa valeur binaire sera incrémentée ou décrétementée) suite aux événements externes – fronts du signal d'horloge
- ⊕ **Réalisation d'un compteur**
 - Un jeu de bascules est utilisé pour mémoriser l'état actuel – N bascules doivent être utilisées pour un compteur de 2^N états
 - Les bascules sont précédées par les blocs combinatoires déterminant l'état suivant (incrémentatation, décrémentation, pré-chargement, etc.)
- ⊕ **Types de compteurs**
 - Compteurs asynchrones – les bascules ne basculent pas en même temps
 - Compteurs synchrones – toutes les bascules basculent au même moment

Blocs séquentiels de base : Compteurs asynchrones

⊕ **Caractéristiques**

- *Chaque bascule a un signal d'horloge différent – l'état du compteur est indéterminé pendant leur basculement*
- *Le bloc combinatoire est réduit (dans l'exemple suivant en une simple connexion entre la sortie de la bascule et son entrée)*
- *Compteur localement très rapide, mais son état se stabilise longtemps pour un grand nombre de bits*

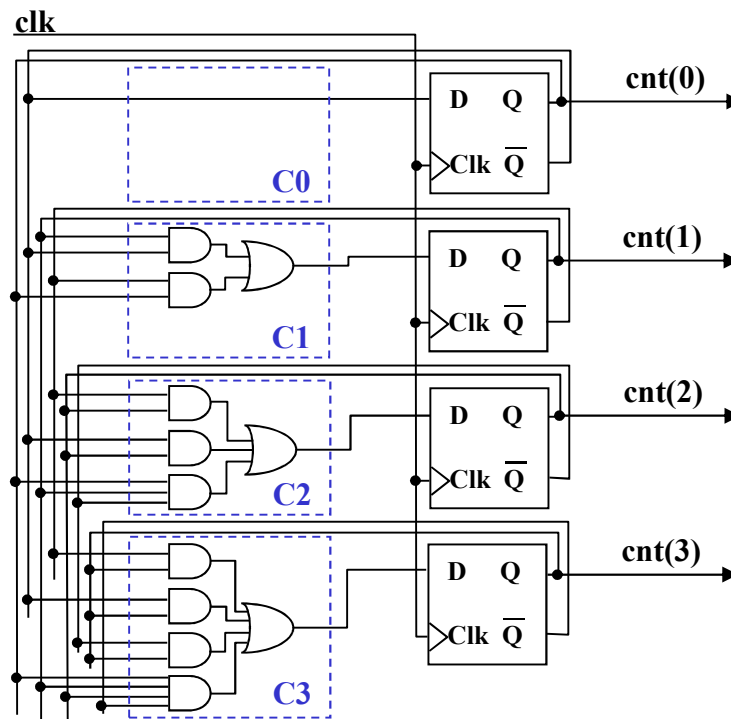
Compteur asynchrone sur 4 bits basé sur les bascules D



Blocs séquentiels de base : Compteurs synchrones

⊕ *Caractéristiques*

- *Les bascules changent d'état en même temps – sur le front montant ou le front descendant du signal d'horloge*
- *Les blocs combinatoires devant les bascules sont plus complexes*



*Exemple d'un compteur
synchrone sur 4 bits
basé sur les bascules D*

Blocs séquentiels de base :

Compteur synchrone de quatre bits avec RAZ

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

ENTITY cmpt IS
PORT (clk : IN std_logic;
      rst : IN std_logic;
      q   : OUT std_logic_vector(3 DOWNTO 0));
END cmpt;

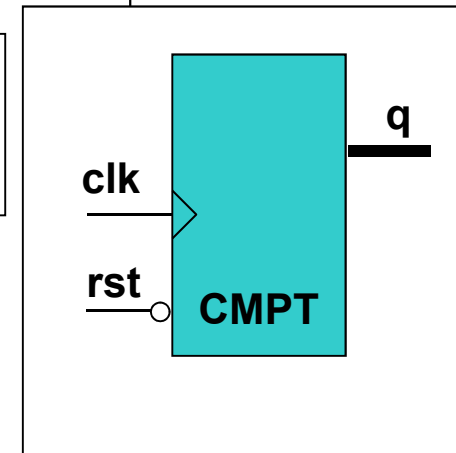
ARCHITECTURE logic OF cmpt IS
  SIGNAL int_q : std_logic_vector(3 DOWNTO 0);
BEGIN
  PROCESS(rst, clk)
  BEGIN
    IF rst = '0' THEN
      int_q <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      int_q <= int_q + 1;
    END IF;
  END PROCESS;
  q <= int_q;
END logic;
```

Rappel :
Paquetage nécessaire
pour les opérations
arithmétiques sur les
vecteurs

**La remise à zéro asynchrone
du compteur est prioritaire par
rapport au signal d'horloge**

**Logique synchrone – l'état du
compteur est changé sur le
front montant de l'horloge**

**Affectation du signal de la sortie
(dans l'architecture, en dehors du
process)**



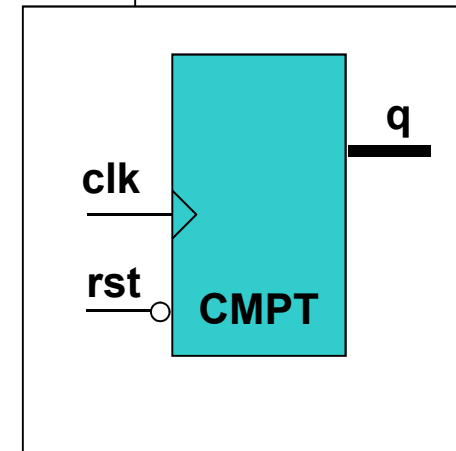
Blocs séquentiels de base :

Compteur synchrone cyclique, avec le cycle réduit

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

ENTITY cmpt IS
PORT (clk, rst : IN std_logic;
      q  : OUT std_logic_vector(3 DOWNTO 0));
END cmpt;

ARCHITECTURE logic OF cmpt IS
  SIGNAL int_q : std_logic_vector(3 DOWNTO 0);
  CONSTANT MAX : integer := 5;
BEGIN
  PROCESS(rst, clk)
  BEGIN
    IF rst = '0' THEN
      int_q <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      IF int_q = MAX THEN
        int_q <= (OTHERS => '0');
      ELSE
        int_q <= int_q + 1;
      END IF;
    END IF;
  END PROCESS;
  q <= int_q;
END logic;
```



Après la valeur MAX le compteur repasse à zéro

Affectation du signal de la sortie (dans l'architecture, en dehors du process)

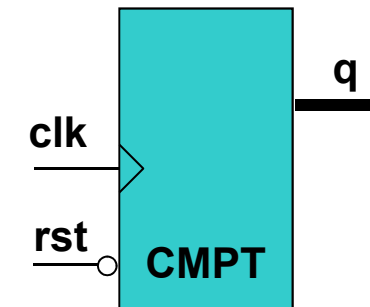
Blocs séquentiels de base :

Compteur synchrone avec l'arrêt à la fin du cycle

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

ENTITY cmpt IS
PORT (clk, rst : IN std_logic;
      q  : OUT std_logic_vector(3 DOWNTO 0));
END cmpt;

ARCHITECTURE logic OF cmpt IS
    SIGNAL int_q : std_logic_vector(3 DOWNTO 0);
    CONSTANT MAX : integer := 5;
BEGIN
    PROCESS(rst, clk)
    BEGIN
        IF rst = '0' THEN
            int_q <= (OTHERS => '0');
        ELSIF rising_edge(clk) THEN
            IF int_q < MAX THEN
                int_q <= int_q + 1;
            END IF;
        END IF;
    END PROCESS;
    q <= int_q;
END logic;
```



*Sur la valeur MAX,
le compteur s'arrête*

*Affectation du signal de la
sortie (dans l'architecture,
en dehors du process)*

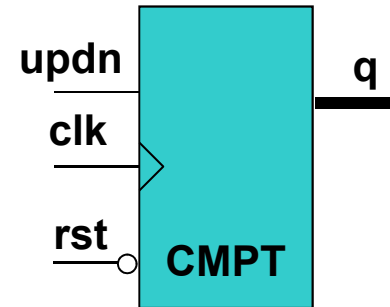
Blocs séquentiels de base :

Compteur synchrone bi-directionnel avec RAZ

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

ENTITY cmpt IS
PORT (clk, rst, updn : IN std_logic;
      q : OUT std_logic_vector(15 DOWNT0 0));
END cmpt;

ARCHITECTURE logic OF cmpt IS
  SIGNAL int_q : std_logic_vector(15 DOWNT0 0);
BEGIN
  PROCESS(rst, clk)
  BEGIN
    IF rst = '0' THEN
      int_q <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      IF updn = '1' THEN
        int_q <= int_q + 1;
      ELSE
        int_q <= int_q - 1;
      END IF;
    END IF;
  END PROCESS;
  q <= int_q;
END logic;
```



**La remise à zéro asynchrone
du compteur est prioritaire par
rapport au signal d'horloge**

**Logique synchrone – le signal
updn est évalué et l'état du
compteur est changé sur le
front montant de l'horloge**

Affectation du signal de la sortie

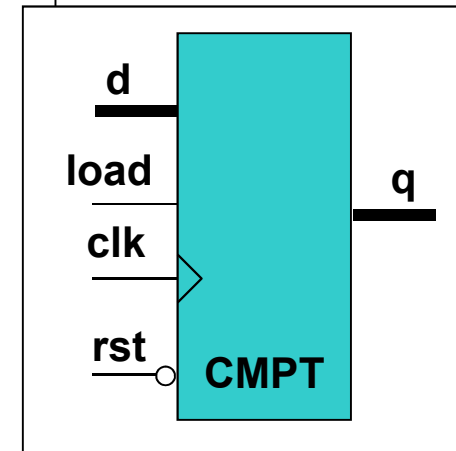
Blocs séquentiels de base :

Compteur unidirectionnel, préchargeable, avec RAZ

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

ENTITY compt_charg IS
PORT (clk, rst, load : IN std_logic;
      d : IN std_logic_vector(15 DOWNTO 0);
      q : OUT std_logic_vector(15 DOWNTO 0));
END compt_charg;

ARCHITECTURE logic OF compt_charg IS
  SIGNAL int_q : std_logic_vector(15 DOWNTO 0);
BEGIN
  PROCESS(rst, clk)
  BEGIN
    IF rst = '0' THEN
      int_q <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      IF load = '1' THEN
        int_q <= d;
      ELSE
        int_q <= int_q + 1;
      END IF;
    END IF;
  END PROCESS;
  q <= int_q;
END logic;
```



*Partie asynchrone (la remise à zéro)
se trouve avant la structure sensible
au signal d'horloge*

*Partie synchrone :
- préchargement
- incrémentation*

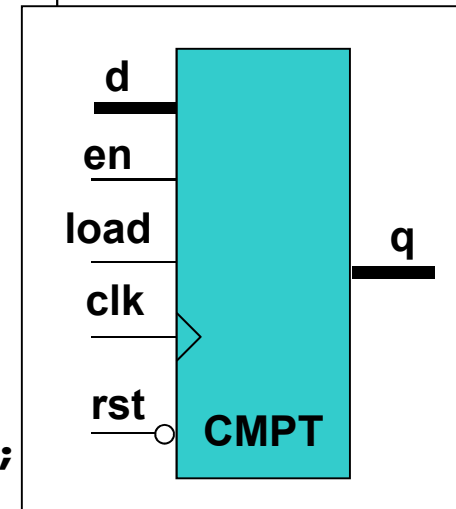
Blocs séquentiels de base :

Compteur préchargeable avec autorisation

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

ENTITY compt_charg IS
PORT (clk, rst, load, en : IN std_logic;
      d : IN std_logic_vector(15 DOWNT0 0);
      q : OUT std_logic_vector(15 DOWNT0 0));
END compt_charg;

ARCHITECTURE logic OF compt_charg IS
  SIGNAL int_q : std_logic_vector(15 DOWNT0 0);
BEGIN
  PROCESS(rst, clk)
  BEGIN
    IF rst = '0' THEN
      int_q <= (OTHERS => '0');
    ELSIF rising_edge(clk) THEN
      IF load = '1' THEN int_q <= d;
      ELSIF en = '1' THEN int_q <= int_q + 1;
      END IF;
    END IF;
  END PROCESS;
  q <= int_q;
END logic;
```



Il manque ELSE, donc il s'agit d'une mémorisation implicite du signal int_q



Chapitre 8

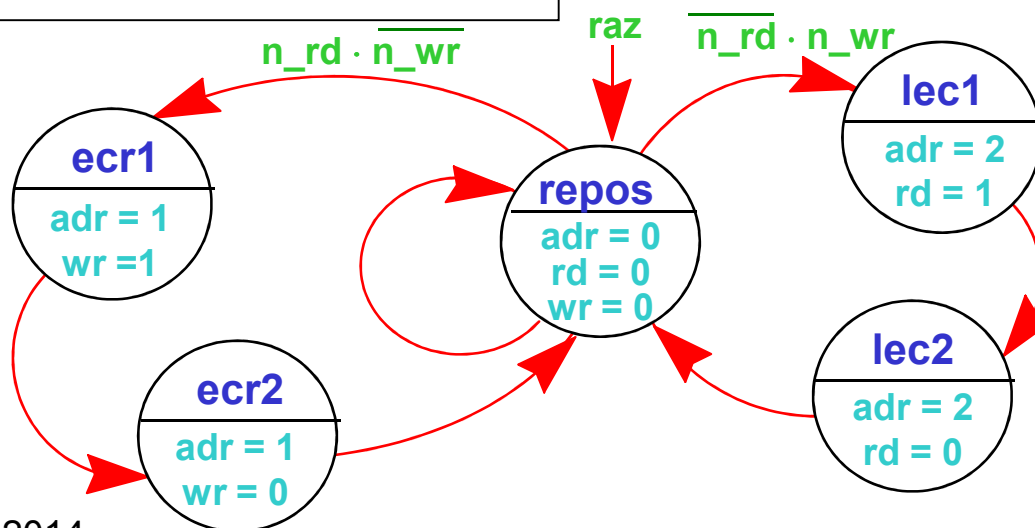
Machines d'états

Machines d'états

⊕ Machine d'état – automate séquentiel cadencé par un signal d'horloge et spécifié par :

- un **jeu d'états** possibles
- un **jeu de transitions** (orientées) entre ces états
- un **jeu de conditions** (expressions logiques basées sur les entrées de la machine) liées à ces transitions
- un **jeu d'équations** spécifiant les valeur à la sortie

Diagramme états - transitions



Entrées :

- clk
- raz
- n_wr
- n_rd

Sorties :

- adr(1..0)
- rd
- wr

Écriture du code VHDL pour les machines d'états (1/5)

- ⊕ **États de la machine** d'états – type de données énuméré :

```
TYPE sm_etats IS (repos, ecr1, ecr2, lec1, lec2);
```

- ⊕ L'objet qui va mémoriser l'état courant doit être un SIGNAL d'un type défini par l'utilisateur, le nom de ce signal représentera le **nom de la machine**

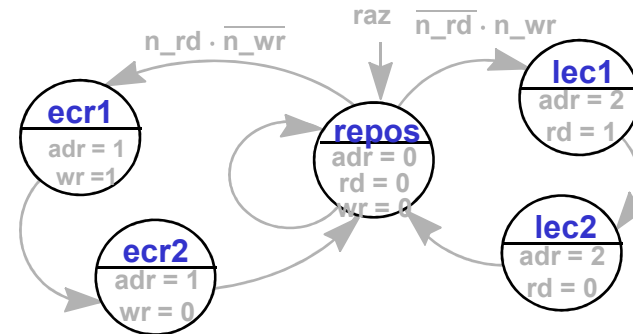
```
SIGNAL sm_machine : sm_etats;
```

- ⊕ Pour déterminer **l'état suivant**, utilisez la structure CASE (rappelez-vous que la machine est une structure séquentielle) à l'intérieur de la structure IF ... THEN sensible au signal d'horloge

- ⊕ Pour déterminer **les sorties** utilisez l'assignation conditionnelle, l'assignation sélectionnée ou la structure CASE

Écriture du code VHDL pour les machines d'états (2/4)

⊕ États de la machine



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

ENTITY machine IS
PORT (clk, raz, n_rd, n_wr : IN std_logic;
      adr : OUT std_logic_vector(1 DOWNTO 0);
      rd, wr : OUT std_logic);
END machine;

ARCHITECTURE comport OF machine IS
    TYPE sm_etats IS (repos, ecr1, ecr2, lec1, lec2);
    SIGNAL sm_machine : sm_etats;
```

Type énuméré

à suivre ...

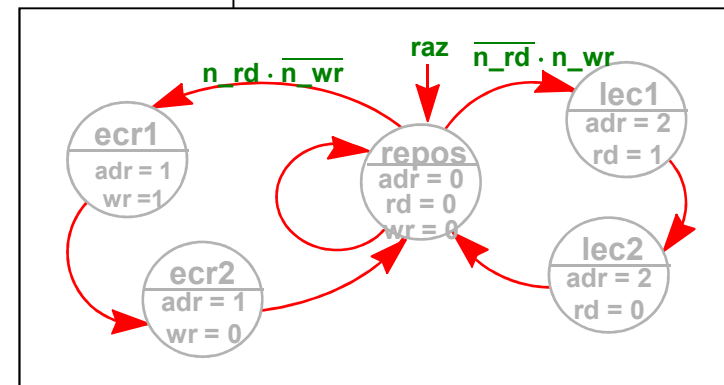
1

Écriture du code VHDL pour les machines d'états (3/4)

```
BEGIN
  PROCESS(raz, clk)
  BEGIN
    IF raz = '1' THEN
      sm_machine <= repos;
    ELSIF rising_edge(clk) THEN
      CASE sm_machine IS
        WHEN repos =>
          IF n_wr = '0' AND n_rd = '1' THEN
            sm_machine <= ecr1;
          END IF;
          IF n_wr = '1' AND n_rd = '0' THEN
            sm_machine <= lec1;
          END IF;
        WHEN ecr1 =>
          sm_machine <= ecr2;
        WHEN ecr2 =>
          sm_machine <= repos;
        WHEN lec1 =>
          sm_machine <= lec2;
        WHEN lec2 =>
          sm_machine <= repos;
        WHEN OTHERS =>
          sm_machine <= repos;
      END CASE;
    END IF;
  END PROCESS;
```

⊕ **Spécification de transitions dans un PROCESS**

Remise à zéro asynchrone



à suivre ...

2

Écriture du code VHDL pour les machines d'états (3/4)

⊕ Spécification de sorties (structures concurrentes)

```
wr <= '1' WHEN sm_machine = ecr1  
      ELSE '0';
```

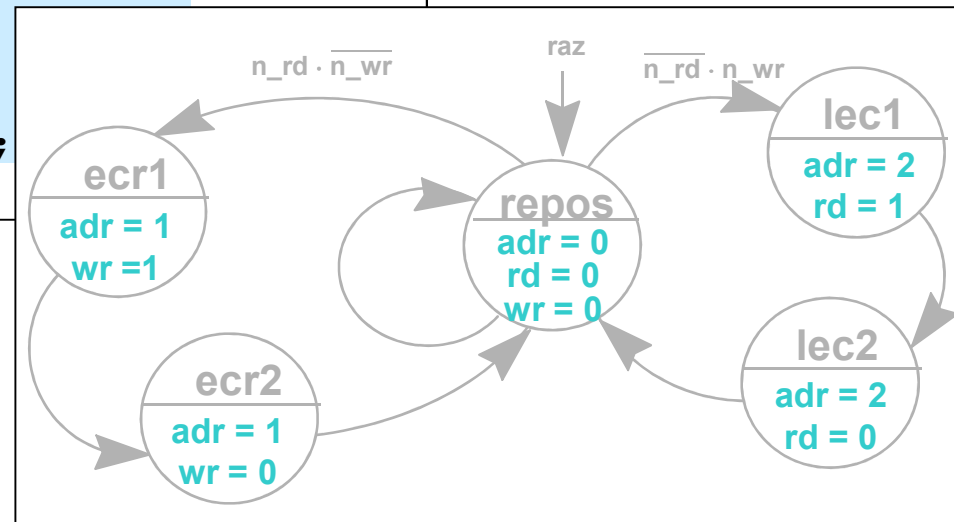
```
rd <= '1' WHEN sm_machine = lec1  
      ELSE '0';
```

```
WITH sm_machine SELECT  
  adr <= "01" WHEN ecr1,  
         "01" WHEN ecr2,  
         "10" WHEN lec1,  
         "10" WHEN lec2,  
         "00" WHEN OTHERS;
```

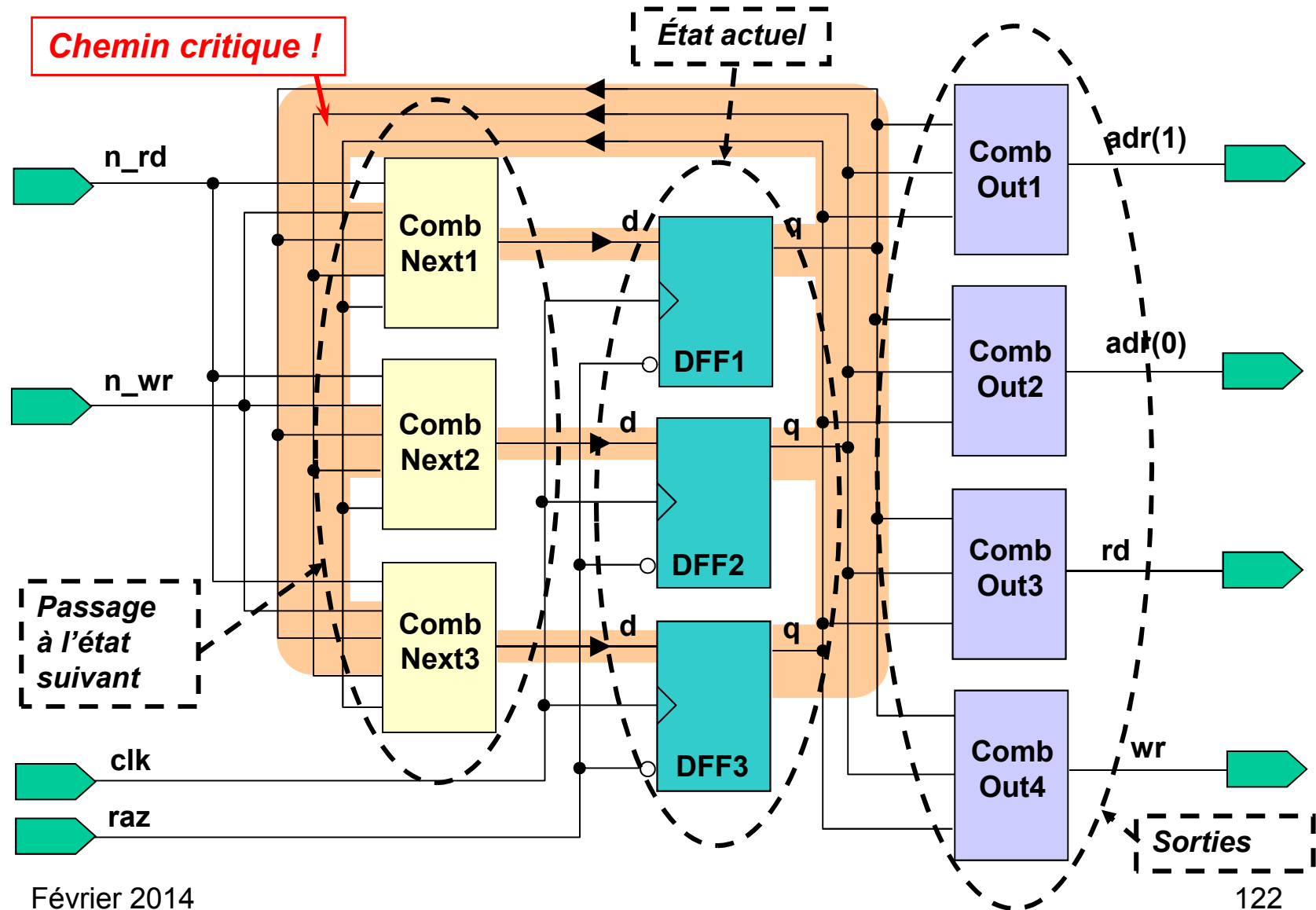
```
END comport;
```

Structure sélective

Structures conditionnelles



Implantation de la machine d'états dans le matériel (1/3)





Implantation de la machine d'états dans le matériel (2/3)

- ⊕ Si le passage par le **chemin critique est plus long que la période du signal d'horloge**, la machine peut entrer dans un état non déterminé où elle restera **bloquée pour toujours !**

Solutions :

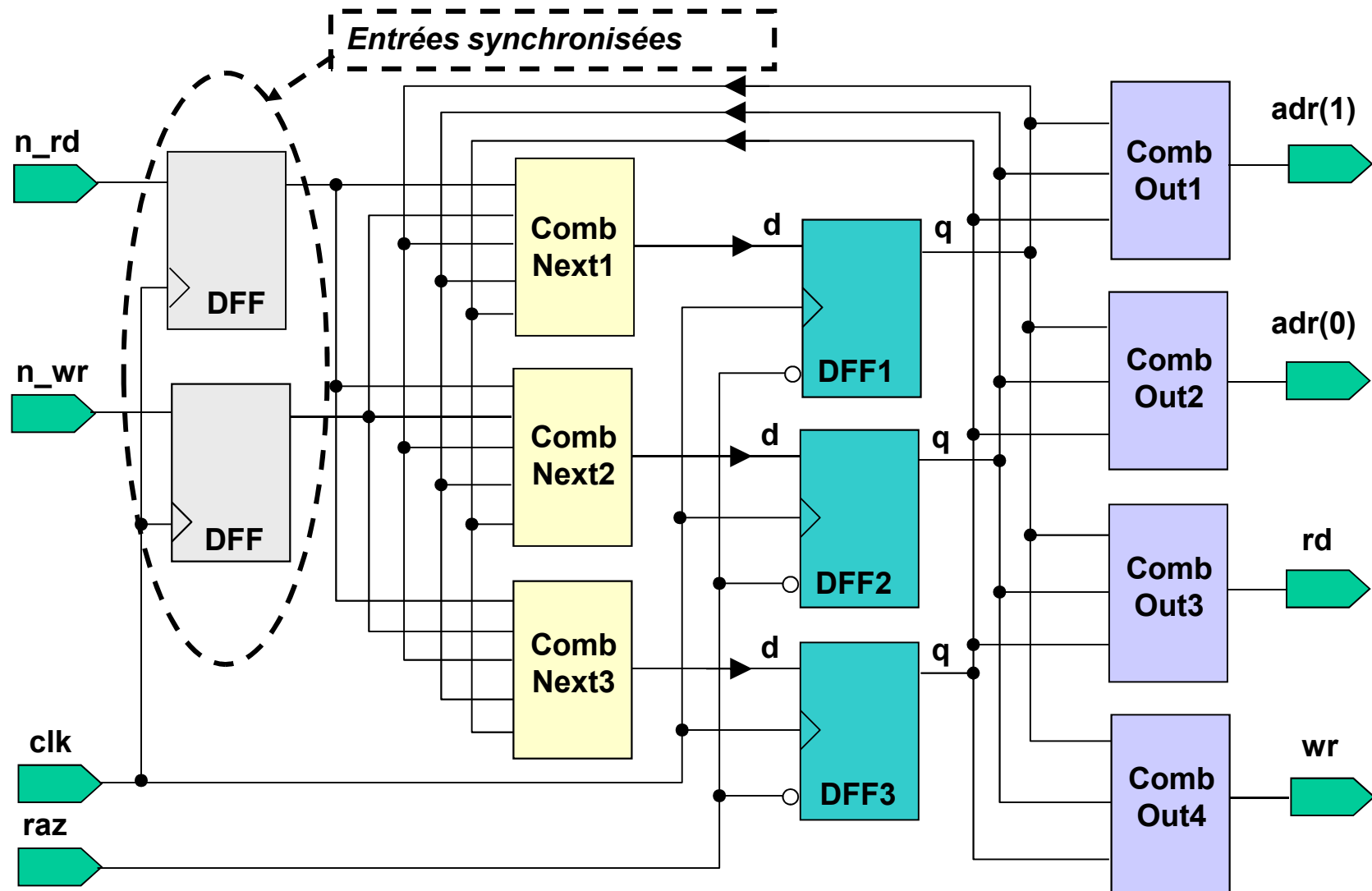
- **Réduire la fréquence** d'horloge
- **Simplifier la logique** combinatoire pour déterminer l'état suivant (voir machines à état décodé)

- ⊕ La machine peut entrer dans un état métastable si les signaux d'entrées changent de valeur **près ou pendant le front montant du signal d'horloge** (violation de paramètres Setup & Hold de la bascule)

Solution :

- **Synchroniser les entrées** avec le signal d'horloge en ajoutant des bascules aux entrées = **indispensable !**

Implantation de la machine d'états dans le matériel (3/3)





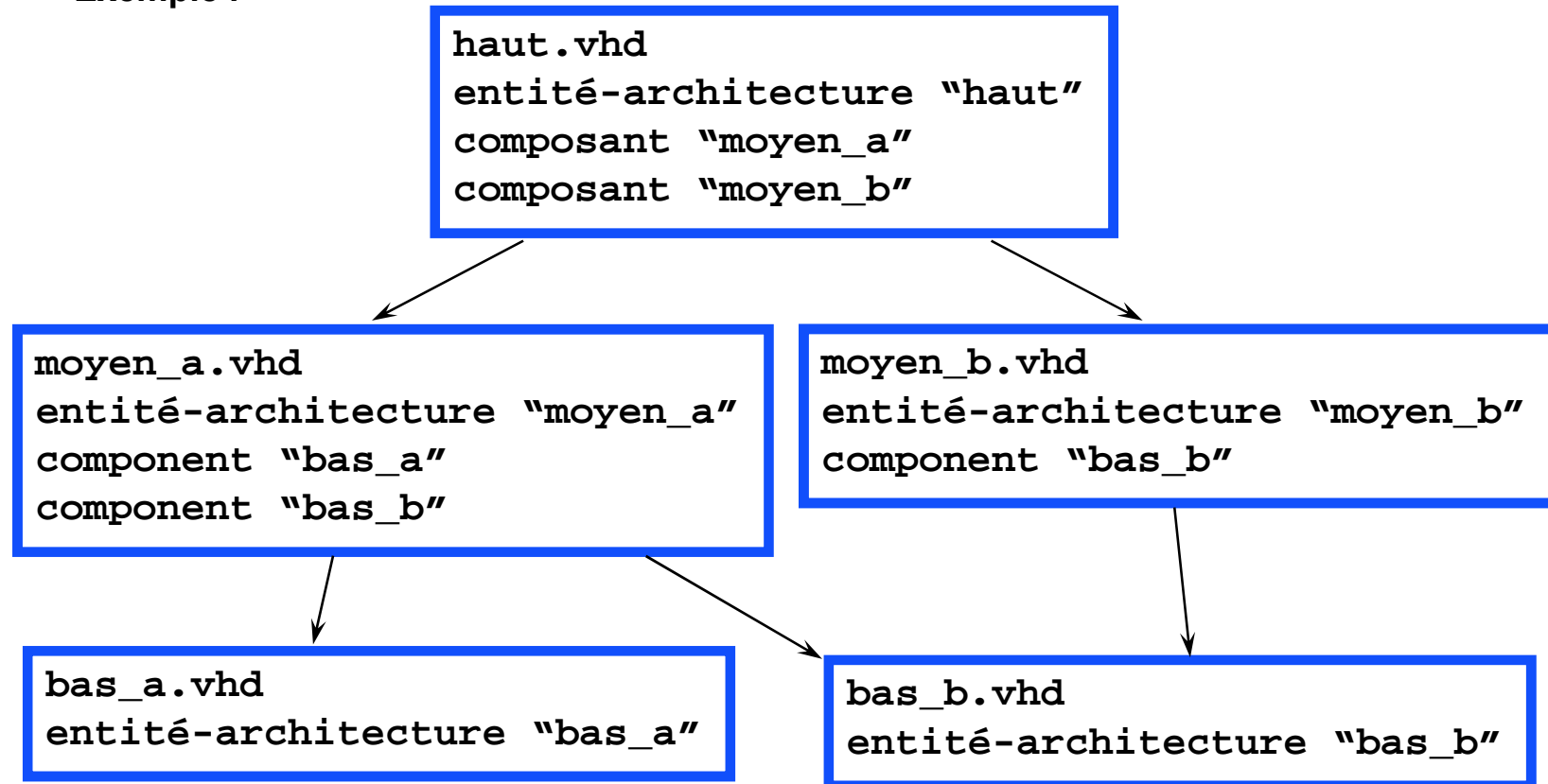
Chapitre 9

Conception hiérarchique de systèmes logiques

Conception hiérarchique en VHDL

- ⊕ Nécessite la déclaration et l'instanciation des composants

Exemple :



Déclaration et instantiation du composant

⊕ Déclaration du composant

- Utilisée pour déclarer les types de ports et de données d'un élément de conception de niveau plus bas

```
COMPONENT <nom_du_composant_niveau_plus_bas> IS
PORT  (<nom_port>  : <type_port> <type_donnée>; ← Point-virgule
      :
      <nom_port>  : <type_port> <type_donnée> ← Sans point-virgule
      );
END COMPONENT;
```

⊕ Instantiation du composant

- Utilisée pour associer les ports du composant du niveau plus bas d'hérarchie aux signaux du niveau actuel

```
<nom_instance> : <nom_du_composant_niveau_plus_bas>
PORT MAP (<nom_port_niveau_bas> => <nom_signal_niveau_actuel>, ← Virgule
      :
      <nom_port_niveau_bas> => <nom_signal_niveau_actuel>
      ); ← Sans virgule
```

Obligatoire



Avantages d'une conception hiérarchique

- ⊕ Chaque membre de l'équipe peut créer **les modules** (composants) **dans des fichiers séparés**
- ⊕ Ces composants peuvent être **partagés** par d'autres collaborateurs ou **réutilisés ultérieurement**
- ⊕ Conception hiérarchique améliore **la modularité et la portabilité** des projet
- ⊕ Conception hiérarchique facilite la possibilité **d'implanter et de tester plusieurs versions** d'un module
- ⊕ Les options de compilation (pour améliorer la performance) peuvent être **appliquées seulement par les modules !**
- ⊕ **Plus de niveaux de hiérarchie signifie plus de souplesse !**

Paramétrage de modules

- ⊕ Permet d'**augmenter la portabilité** de différents modules
- ⊕ Apporte une **grande flexibilité** aux description de composants
- ⊕ Constitue le principe d'utilisation de la **bibliothèque LPM** (Library of Parameterized Modules) utilisée pour proposer les fonctions adaptées au matériel par les fabricants d'outils CAO
- ⊕ Réalisé par la **structure GENERIC**, exemple :

```
ENTITY multiplexeur IS
  GENERIC (LARGEUR : integer := 4);
  PORT(sel : IN bit;
        a, b : IN bit_vector(LARGEUR-1 DOWNT0 0);
        c : OUT bit_vector(LARGEUR-1 DOWNT0 0));
END multiplexeur;
ARCHITECTURE a_mux OF multiplexeur IS
BEGIN
  c <= a WHEN sel = '0' ELSE b;
END a_mux;
```

Valeur par défaut

Instanciation du composant :

```
mux_inst: multiplexeur GENERIC MAP (LARGEUR => 8)
  PORT MAP(sel => selh, a => ah, b => bh, c => ch);
```

Valeur actuelle



Chapitre 10

Familles technologiques de circuits intégrés logiques



Familles technologiques de circuits logiques

⊕ **Technologie TTL** (*Transistor-Transistor-Logic*)

- Basée sur les **transistors bipolaires** du type NPN et PNP en mode commutation
- Technologie **la plus utilisée il y a 20 ans** environ (circuits logiques standards de la série Intel 74 xxx)
- Alimentée en 5 V (TTL classique)

⊕ **Technologie CMOS** (*Complementary Metal Oxyd Semiconductor*)

- Basée sur les **transistors unipolaires** du type NMOS et PMOS
- Technologie **la plus utilisée actuellement** (circuits logiques standards série Motorola 4000, puis 74xx, processeurs, circuits logiques spécifiques à l'application, circuits logiques configurables, ...)
- Alimentée en 5 V (CMOS de la série 74) ou en 3,3 V (LV CMOS)

⊕ **Technologie ECL** (*Emitter-Coupled Logic*)

- Basée sur les transistor bipolaires travaillant dans un régime actif
- Technologie la plus rapide, mais très gourmande...
- Alimentée en -5,2 V

Famille TTL (1/4)

⊕ Tension d'alimentation

- $V_{CC} = 5\text{ V} \pm 10\%$

⊕ Niveaux logiques

- Niveau "1" = 5 V
- Niveau "0" = 0 V

⊕ Niveaux des tensions d'E/S

- $V_{OLMax} = 0,4\text{ V}$
- $V_{OHMin} = 2,7\text{ V}$
- $V_{ILMax} = 0,8\text{ V}$
- $V_{IHMin} = 2,0\text{ V}$
- $V_{NL} = 0,4\text{ V}$
- $V_{NH} = 0,7\text{ V}$

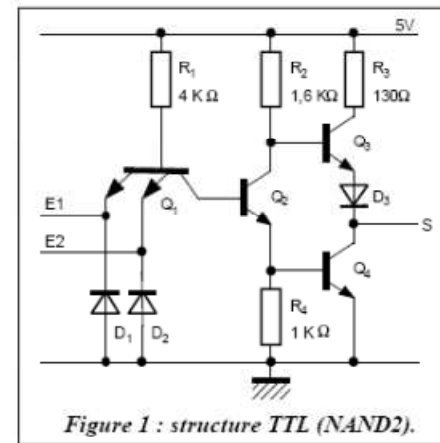


Figure 1 : structure TTL (NAND2).

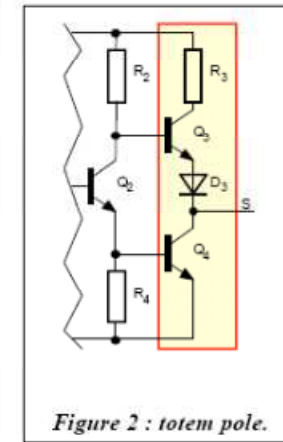
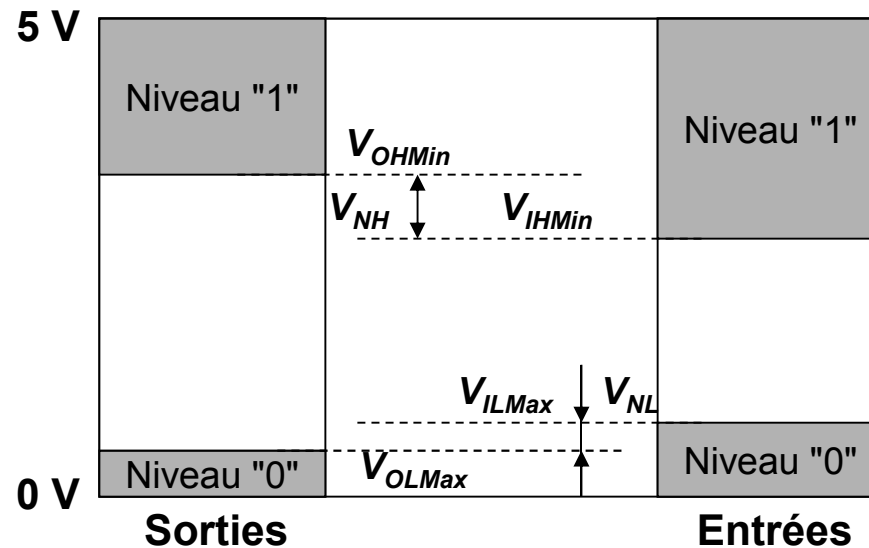


Figure 2 : totem pole.



Famille TTL (2/4)

⊕ Immunité aux bruits

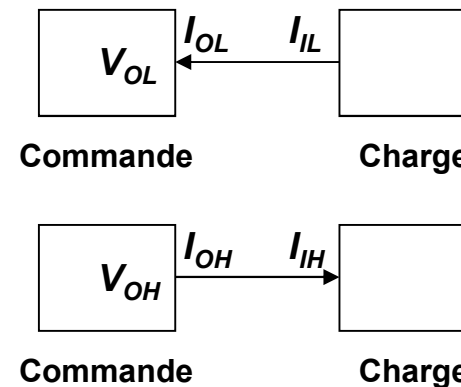
- A l'état bas : $V_{NL} = V_{ILMax} - V_{OLMax} = 0,4 \text{ V}$
- A l'état haut : $V_{NH} = V_{OHMin} - V_{IHMin} = 0,7 \text{ V}$

⊕ Courants d'entrée et de sortie

- $I_{OLMax} = + 8 \text{ mA}$ (consommation)
- $I_{OHMin} = - 0,4 \text{ mA}$ (production)
- $I_{ILMax} = -0,4 \text{ mA}$
- $I_{IHMin} = + 20 \text{ uA}$

- Si une sortie commande plusieurs entrées :

$$I_{OL} = \sum I_{IL} \qquad I_{OH} = \sum I_{IH}$$



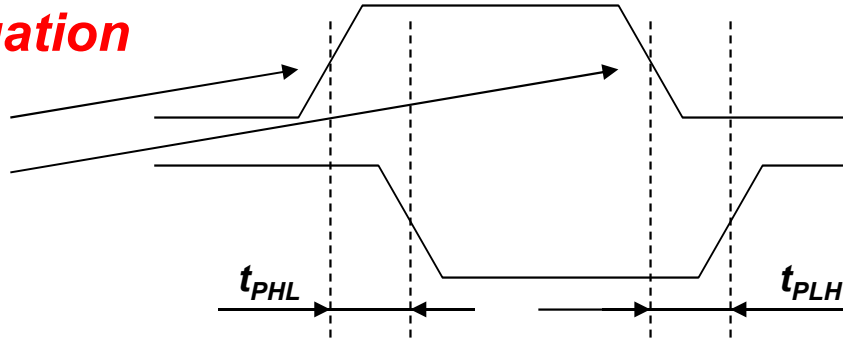
⊕ Sortance (Fan out)

- Nombre maximal d'entrées qui peuvent être connectées à une sortie - 20

Famille TTL (3/4)

⊕ Temps de propagation

- t_r – rising time
- t_f – falling time



⊕ Consommation

- Courant consommé pour les sorties au niveau haut : I_{CCH}
- Courant consommé pour les sorties au niveau bas : I_{CCL}
- Puissance moyenne : $P_{MOY} = V_{CC} (I_{CCH} + I_{CCL})/2$

Séries	74 N XXX Standard	74 S XXX Schottky	74 LS XXX Low Power Schottky	74 ALS XXX Advanced LS	74 F XXX Fast
Retard de propagation t_p (ns)	10	3	10	4	3
Consommation (mW)	10	20	2	2	4
Fréquence maximale (MHz)	35	80	40	70	150



Famille TTL (4/4)

- ⊕ **Phénomènes transitoires à la commutation des sorties**
 - *« Totem pôle » à la sortie cause des impulsions de courant*
 - *Il faut filtrer l'alimentation de chaque circuit*

- ⊕ **Câblage des entrées inutilisées**
 - *Il est fortement déconseillé de laisser « en l'air » une entrée non utilisée*
 - *Solution : la connecter à la masse, VCC ou une autre entrée*

Famille CMOS (1/4)

⊕ Tension d'alimentation

- **CMOS** : $V_{CC} = 5\text{ V}$ (2-6 V)
- **LVC MOS** : $V_{CC} = 3,3\text{ V}$ (2-3,6 V)

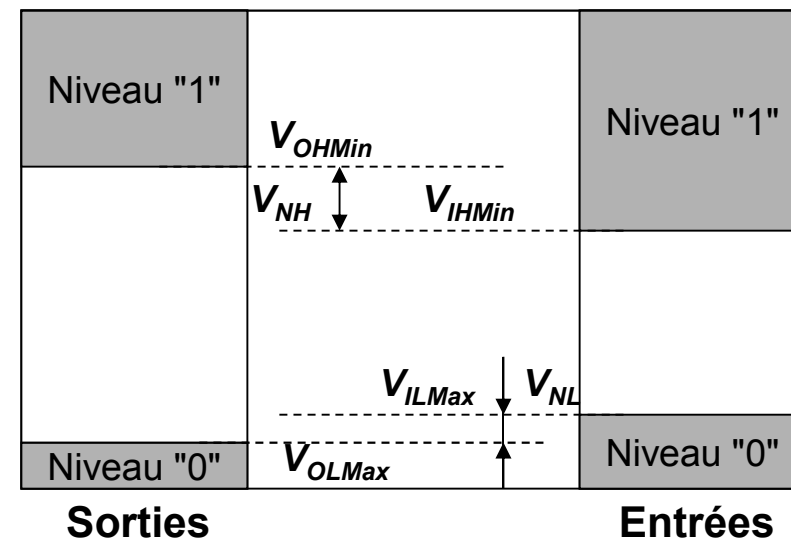
⊕ Niveaux logiques

- Niveau "1" = 5 V (3,3 V)
- Niveau "0" = 0 V

⊕ Niveaux des tensions d'E/S

- $V_{OLMax} = 0,33\text{ V}$ (0,4 V)
- $V_{OHMin} = 4,4\text{ V}$ (2,4 V)
- $V_{ILMax} = 1,5\text{ V}$ (0,8 V)
- $V_{IHMin} = 3,5\text{ V}$ (2,0 V)

- $V_{NL} = 1,17\text{ V}$ (0,4 V)
- $V_{NH} = 0,9\text{ V}$ (0,4 V)





Famille CMOS (2/4)

⊕ **Courants d'entrée et de sortie**

- *Courant d'entrée très faible (μA)*
- *Courants de sortie relativement grands (mA)*

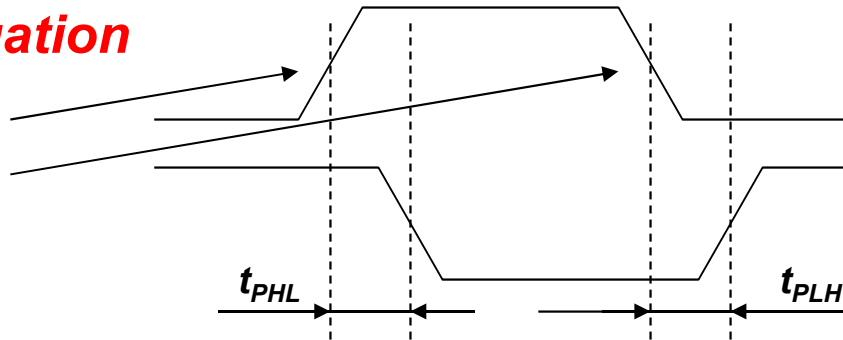
⊕ **Sortance (Fan out)**

- *N'est pas limitée par le courant, mais par la vitesse (liée au déchargement de la somme de capacités parasites d'entrées connectées à la sortie)*

Famille CMOS (3/4)

⊕ Temps de propagation

- t_r – rising time
- t_f – falling time



Séries	4000	74 HC XXX	74 AC XXX	74 AHC XXX	74 LVC XXX	74 ALVC XXX
Retard de propagation t_p (ns)	100	7	5	4	5	3
Fréquence maximale (MHz)	12	50	160	170	100	150

⊕ Consommation

- Puissance : $P_D = (C_L + C_{PD}) * f * V_{CC}$
- C_L – capacité de charge en sortie
- C_{PD} – capacité interne de dissipation de l'ordre de 25 pF



Famille CMOS (4/4)

- ⊕ **Phénomènes transitoires à la commutation des sorties**
 - *Déchargement de la capacité cause des impulsions de courant*
 - *Il faut filtrer l'alimentation de chaque circuit*

- ⊕ **Précautions d'emplois des circuits CMOS**
 - *Les circuits CMOS sont très sensibles aux décharges électrostatiques – attention à la manipulation !*
 - *Il est **interdite** de laisser « en l'air » une entrée non utilisée (destruction du circuit) !*



Chapitre 11

Familles fonctionnelles de circuits intégrés logiques



Familles fonctionnelles de circuits logiques

- ⊕ **Circuits numériques standards** (*Standard Logic Devices*)
 - Circuits logiques génériques pour réaliser des fonctions logiques de base (porte logiques, compteurs, multiplexeurs, pilotes du bus, ...)
 - Processeurs génériques (8-bit, 16-bit, 32-bit, 64-bit)
 - Mémoires semi-conducteur (SRAM, DRAM, ROM, EEPROM, FLASH)
- ⊕ **ASSPs** (*Application Specific Standard Products*)
 - Circuits standards produits pour une application spécifique et pour un groupe quasi-illimité de clients (ex. : circuit interface USB)
- ⊕ **ASICs** (*Application Specific Integrated Circuits*)
 - Circuits logiques spécifiques à l'application - produits pour une application spécifique et habituellement pour un seul client ou un group restreint de clients (ex. : processeur d'une carte à puce)
- ⊕ **Circuits logiques configurables** (*Programmable Logic Devices*)
 - Circuits logiques de grande capacité avec la structure logique interne configurable et donc le fonctionnement modifiable et adaptable aux besoins (ex. : FPGA)



Évolution de circuits logiques

⊕ **Années 80**

- *Circuits logiques génériques de base, processeurs simples, mémoires semiconducteur de faible capacité*
- *Niveau d'intégration faible (SSI, MSI, LSI)*

⊕ **Années 90**

- *Circuits logiques génériques et processeurs plus performants, ASICs et Circuits Logiques Programmables de la première génération (PLA, PLD)*
- *Circuits LSI et VLSI*

⊕ **Aujourd'hui**

- *Circuits logiques configurables (FPGAs)*
- *ASSPs*
- *ASICs (Full Custom, Pré-caractérisés, Pré-diffusés, Structurels)*
- *Circuits VLSI*

SSI, MSI, LSI, VLSI - Small-, Middle-, Large-, Very Large-Scale Integration



Circuits logiques standards

⊕ **Avantages**

- *Faible prix par unité*
- *Bonne disponibilité*
- *Fiabilité éprouvée (circuit complètement testé)*

⊕ **Inconvénients**

- *Fonction logique du circuit relativement simple – un grand nombre de circuits est nécessaire pour concevoir les système complexes*
- *Fiabilité relativement faible du système basé sur des circuits standards (circuit imprimé – soudures)*
- *Évolution (modification) du système difficile*
- *Mauvaise utilisation de la surface de circuit imprimé*
- *Vitesse relativement faible*



Circuits (logiques) spécifiques à l'application - ASICs

⊕ **Avantages**

- *Potentiellement très faible prix par l'unité (pour une grande production)*
- *Fiabilité élevée (circuit complètement testé, souvent avec les tests internes effectués en temps réel)*
- *Vitesse très élevée*
- *Faible surface utilisée*
- *Structure logique interne peut être très complexe*

⊕ **Inconvénients**

- *Évolution (modification) pratiquement impossible*
- *Conception très chère*



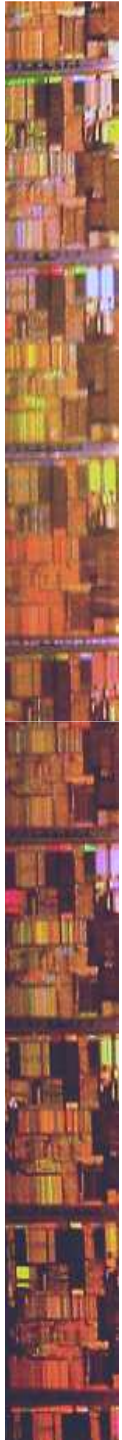
Circuits logiques configurables (programmables)

⊕ **Avantages**

- *Fiabilité élevée (circuit complètement testé dans la dernière phase de fabrication)*
- *Faible surface utilisée*
- *Structure logique interne peut être très complexe*
- *Flexibles – évolution facile*
- *Conception rapide et peu chère*

⊕ **Inconvénients**

- *Piratage plus facile*
- *Vitesse élevée, mais plus basse que pour les ASICs*
- *Plus chers que les ASICs pour une grande quantité, mais moins chers pour faibles quantités*



Chapitre 12

Circuits logiques configurables



Circuits logiques configurables – CLC (1/2)

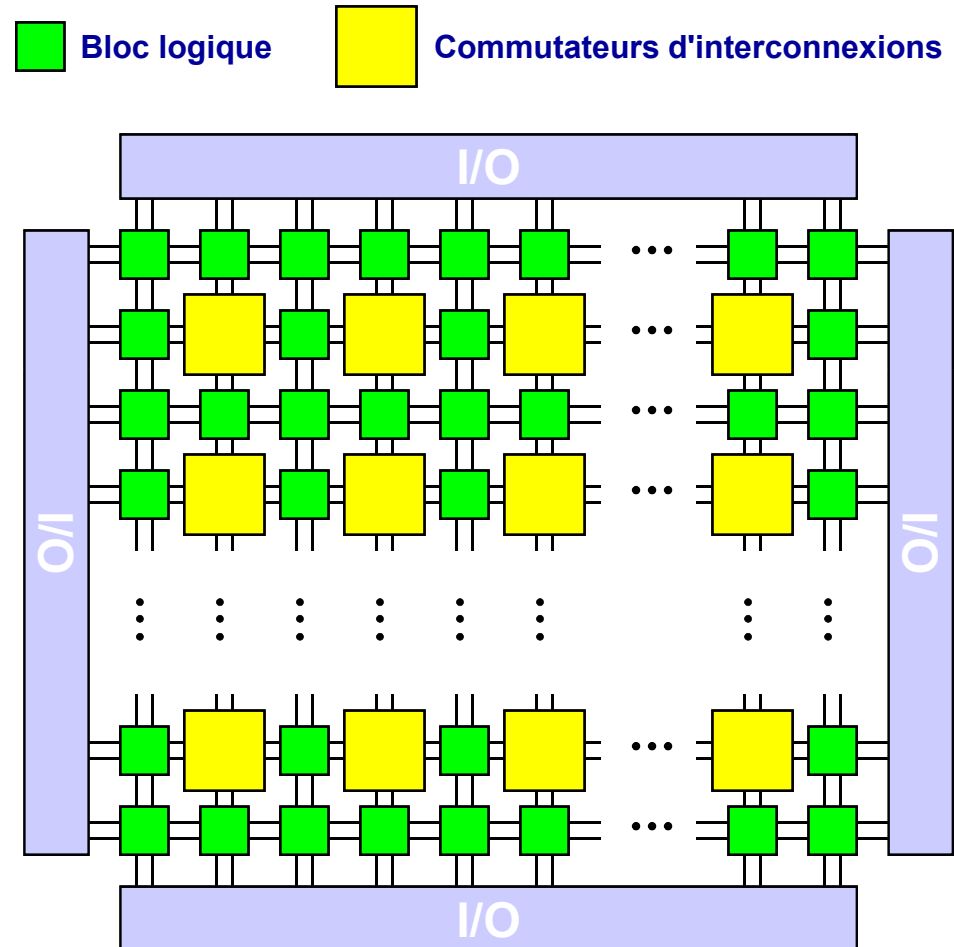
⊕ **CLC - structure à deux niveaux :**

- **Structure logique configurable** - un ensemble
 - d'éléments logiques (cellules logiques)
 - d'interconnexions
 - d'entrées/sorties
 - de modules embarqués additionnels
- **Procédé de configuration** représentant l'ensemble des ressources utilisées pour configurer le circuit

⊕ Chaque élément configurable est **associé à un bit mémoire** (mémoire morte ou mémoire vive) dans le circuit

- **Conception** : validation des différents éléments de configuration du circuit
- « **Programmation** » : chargement de la configuration dans la mémoire rassemblant l'ensemble des bits de configuration

Circuits logiques configurables – CLC (2/2)



Cellules logiques configurables constituant les blocs logiques

- Pour implémenter la logique séquentielle et combinatoire

Interconnexions segmentées configurables

- Pour interconnecter la logique et les E/S

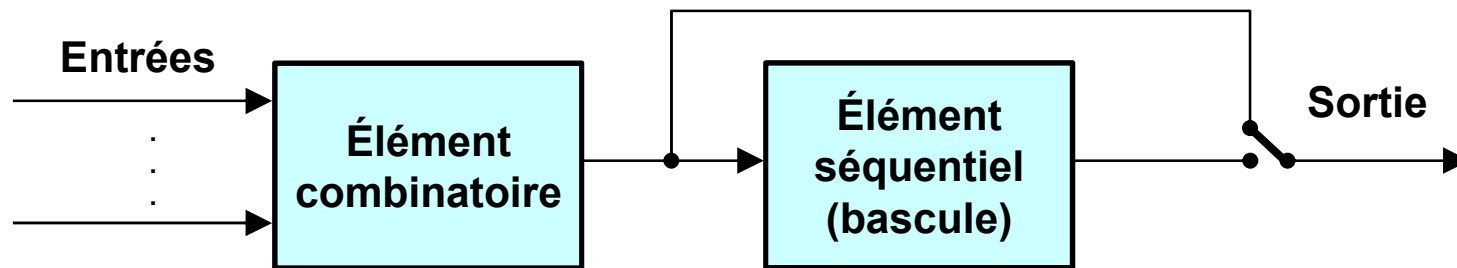
Blocs d'entrées sorties configurables

- Pour les connexions externes

Cellules logiques configurables

⊕ *Cellule logique :*

- Ensemble figé de **signaux d'entrée**
- Élément **combinatoire configurable**
- Élément **séquentiel** (qui peut être éventuellement évité)



⊕ *Cellules logiques (8 à 16) sont groupées dans des **blocs logiques***

⊕ *Il y a des dizaines de milliers de cellules logiques dans un circuit*



Interconnexions configurables

⊕ *Trois niveaux hiérarchiques d'interconnexions*

- *Interconnexions globales*
 - *Signaux accessibles partout dans le circuit*
 - *Nombre très limité*
 - *Vitesse relativement faible (grande longueur)*
- *Interconnexions régionales*
 - *Signaux accessible localement (colonnes et lignes, quadrants)*
 - *Nombre important*
 - *Vitesse moyenne*
- *Interconnexions locales*
 - *Signaux accessibles sur un voisinage (~ 8 cellules)*
 - *Nombre globalement très important (mais localement limité)*
 - *Grande vitesse*



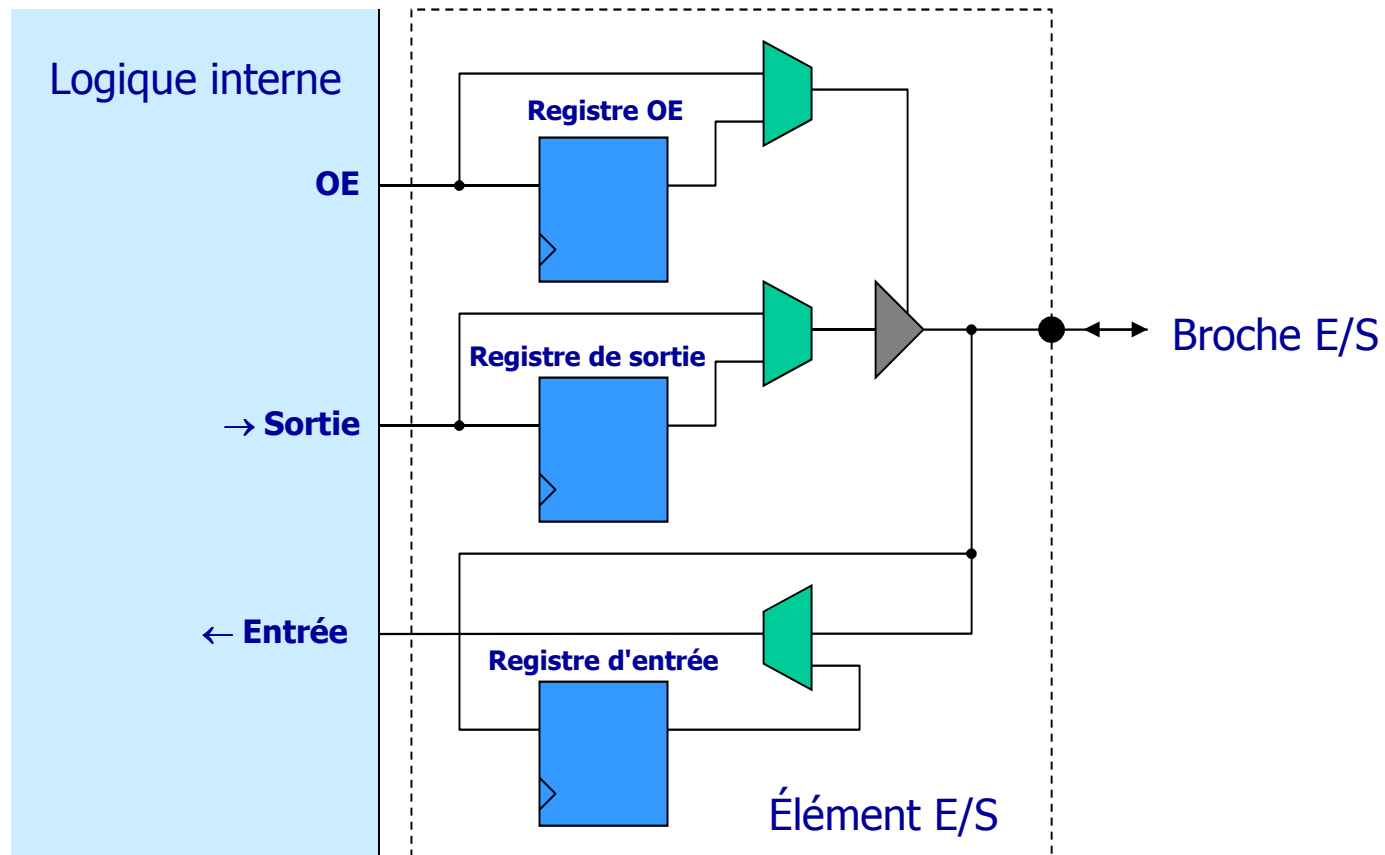
Entrées/sorties configurables

✦ **Caractéristiques de type**

- Support de **multiples normes d'E/S** (simples ou différentielles)
 - LVTTL, LVCMOS, LVDS, ...
- Support du **BST (Boundary-Scan Test)** de l'interface **JTAG (Joined Test Action Group)**
- **Réglage de la force (du courant) de sortie**
- **Réglage de la pente/descente (slew-rate) du signal à la sortie**
- **Implantation de buffers trois-états**
- **Fonction "Bus-hold" (maintien du niveau du bus)**
- **Résistances "Pull-up", "Pull-down" programmables**
- **Délais d'entrées et sorties programmables**
- **Sorties collecteur ouvert**
- **Support pour DDRAMs**

Élément d'entrée/sortie configurable

- ✚ *Un élément par broche E/S (parfois des centaines de broches)*
- ✚ **Structure logique** (simplifiée)





Normes actuelles pour les E/S

- ⊕ **LVTTTL** (*Low Voltage TTL*) - 1.8, 2.5, 3.3 V (*single ended*)
- ⊕ **LVC MOS** (*Low Voltage CMOS*) - 1.5, 1.8, 2.5, 3.3 V (*single ended*)
- ⊕ **3.3 V PCI** - équivalente à 3.3 V LVC MOS avec une diode de clamping
- ⊕ **LVDS** (*Low Voltage Differential Signaling*)
- ⊕ **SSTL-2** (*Stub-Series Terminated Logic for 2.5 V*) - nécessite une tension de référence (1.25 V), norme utilisée pour SDRAMs
- ⊕ **SSTL-3** (*Stub-Series Terminated Logic for 3.3 V*) - nécessite une tension de référence (1.5 V), norme utilisée pour SDRAMs



Outils et méthodes de conception de CLCs

⊕ **Outils de base**

- *Simulation (fonctionnelle, temporelle)*
- *Synthèse/compilation (traduction, optimisation)*
- *Placement/routage (fitter)*
- *Analyse temporelle*
- *Outils de configuration*

⊕ **Outils (méthodes) complémentaires :**

- *Scripting - automatisation du travail avec le langage TCL (Tool Command Language)*
- *Vérification formelle (techniques mathématiques de vérification)*
- *Estimation de consommation (ex. Power calculator)*
- *Floorplanner*
- *Chip editor*
- *Analyseur logique enfoui*



Technologies de configuration de CLCs

⊕ **Configurations non volatiles**

- **PROM (fusibles), antifusibles - configurables une fois**
 - *Élément de configuration métallique – petit, faible résistance, donc rapide*
- **EPROM, EEPROM, FLASH - reconfigurables**
 - *Élément de configuration semi-conducteur – grande surface, relativement grande résistance, donc lent*

⊕ **Configurations volatiles**

- **CMOS RAM - reconfigurables (aussi dynamiquement et localement)**
 - *Élément de configuration semi-conducteur – grande surface, relativement grande résistance, relativement rapide*



Processus de configuration de CLCs à technologie volatile

⊕ **Décomposé en trois phases**

- **Transfert de la suite binaire de configuration vers le circuit**
 - Transfert **série**
 - Plus lent
 - Moins de broches utilisées
 - Transfert **parallèle**
 - Plus rapide
 - Plus de broches utilisées
- **Configuration**
 - **Déchiffrement** de données de configuration (optionnel)
 - **Décompression** de données de configuration (optionnelle)
 - **Calcul** et vérification de la valeur condensée (Checksum - CRC)
 - **Enregistrement** de données de configuration dans la mémoire
- **Initialisation**
 - **Mise à l'état initial** de toutes le bascules et de toutes les entrées sorties