

# *Conception de circuits et langage VHDL modélisation et synthèse*

**Patrice NOUEL**

email : [nouel@enseirb.fr](mailto:nouel@enseirb.fr)  
http : <http://www.enseirb.fr/~nouel>

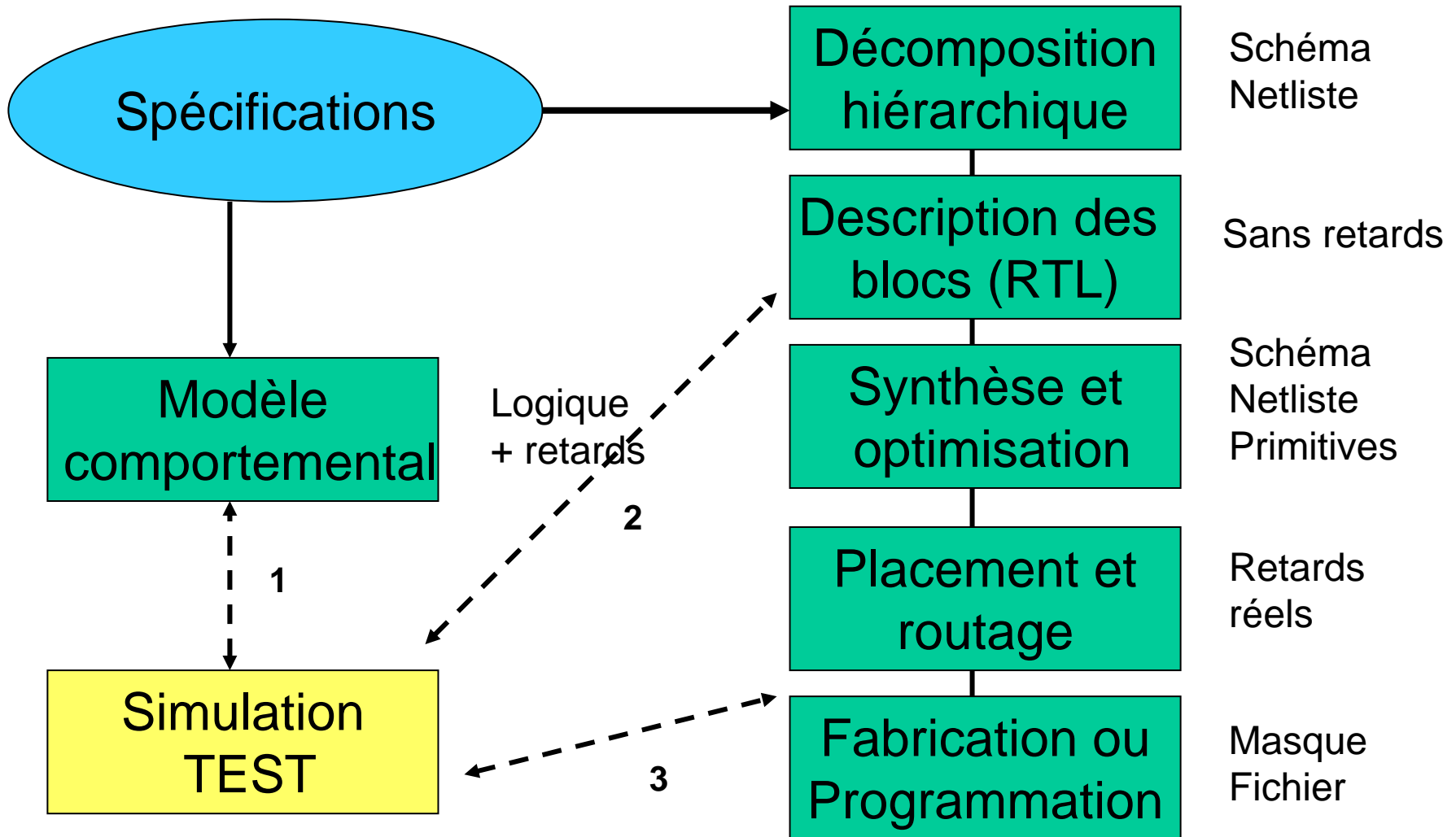
- ↙ Champs d'application du langage VHDL
- ↙ VHDL: langage à instructions concurrentes
- ↙ Les spécificités du langage
- ↙ La synthèse des circuits
- ↙ Modélisation des circuits avec retards

# Champs d'application du langage VHDL

---

- ↙ Flot de conception ASIC/FPGA
- ↙ Qu'est ce que le VHDL ?
- ↙ Modélisation ou Synthèse ?
- ↙ Autres langages
- ↙ Les dates importantes

# Flot de conception Asic/FPGA



# Qu'est ce que le VHDL ?

---

- ↳ Langage **standard** de description de circuits ou de systèmes numériques en vue de
  - ↳ **modélisation** (simulation) des circuits ou systèmes
  - ↳ **synthèse** (génération automatique) niveau RTL
  - ↳ descriptions de programmes de **test** (stimuli)
  - ↳ description de type hiérarchique (**netlist**)

VHSIC Hardware Description Language

# Modélisation ou Synthèse ?

---

## ↳ Modélisation

- ↳ Tout le langage.  
Logique + Temporel
- ↳ Un modèle peut être comportemental, structurel ou de type data-flow.
- ↳ Exemple: créer des programmes de test

## ↳ Synthèse

- ↳ Langage simplifié (pas de retards). Le style d'écriture anticipe une primitive circuit.
- ↳ La synthèse demande une bonne connaissance du circuit et de la technologie.

# Autres langages proches

---

- ↙ Verilog concurrent et plus ancien. La syntaxe est proche de celle du langage C
- ↙ VHDL-AMS Langage de modélisation mixte numérique-analogique IEEE.1076.1-1999. Il est entièrement compatible avec VHDL. Uniquement pour la modélisation.

# Les dates importantes

---

- ↙ 1980 Intermetrics, IBM, Texas Instruments travaillent sur le projet pour le compte du D.O.D
- ↙ IEEE Standard 1076-1987
- ↙ IEEE Standard 1164 rajoute la notion de forces sur les signaux et est souvent appelé MVL9 (multivalued logic, nine values). IEEE 1076-1993
- ↙ IEEE 1076.3 (Numeric Standard) pour la synthèse
- ↙ IEEE 1076.4 : VITAL initiative (VHDL Initiative Toward ASIC Libraries) Pour la génération de modèles de timing

# PLAN

---

- ↙ Champs d'application du langage VHDL
- ↙ VHDL: langage à instructions concurrentes
- ↙ Les spécificités du langage
- ↙ La synthèse des circuits
- ↙ Modélisation des circuits avec retards

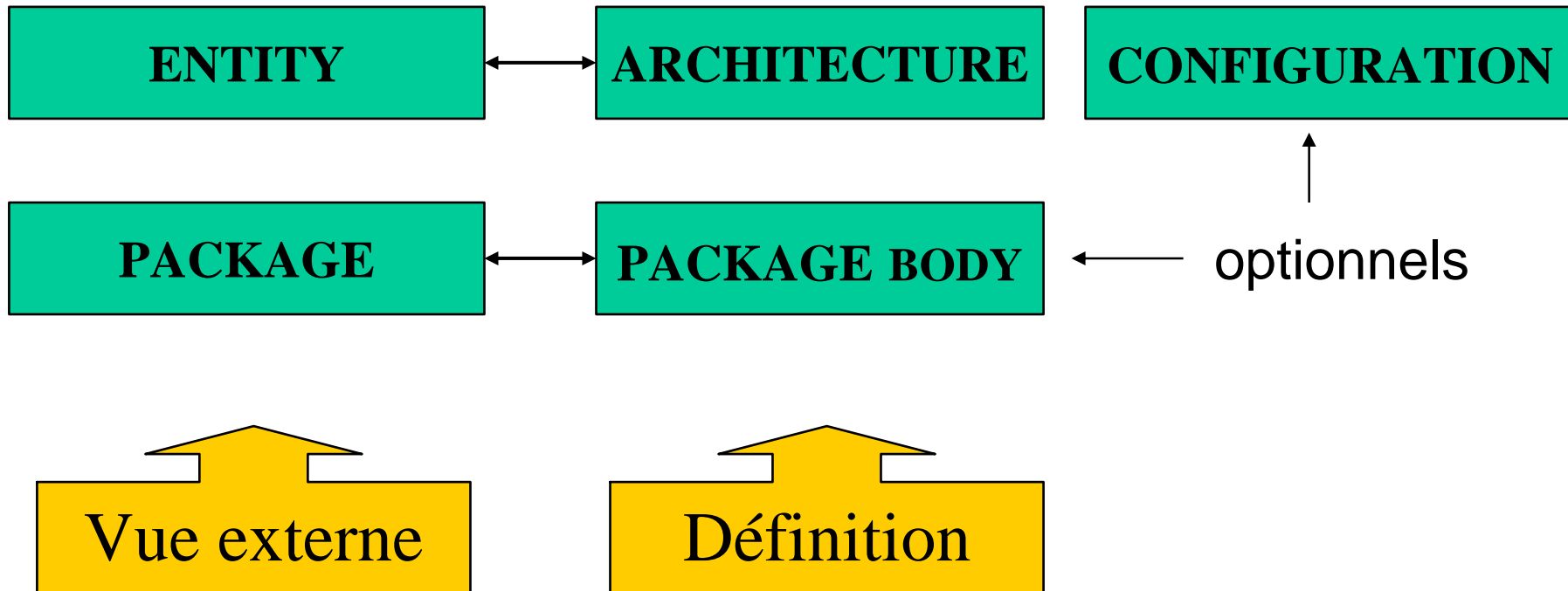
# *VHDL: Un langage à instructions concurrentes*

---

- ↙ Unités de compilation
- ↙ Concurrence, signaux, événements, processus
- ↙ Simulation événementielle
- ↙ Instructions séquentielles
- ↙ Le délai delta
- ↙ Descriptions structurelles

# Unités de compilation

- Peuvent être compilées séparément -  
(Un seul fichier \*.vhd ou plusieurs)



## Entité : Exemple

ENTITY comparateur IS

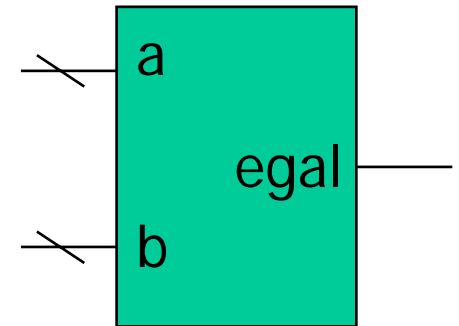
PORT(

SIGNAL a : IN bit\_vector(7 DOWNT0 0);

SIGNAL b : IN bit\_vector(7 DOWNT0 0);

SIGNAL egal : OUT bit);

END comparateur;



Mode  
Protégé en lecture

# Architecture: Exemple

---

ARCHITECTURE simple OF comparateur IS

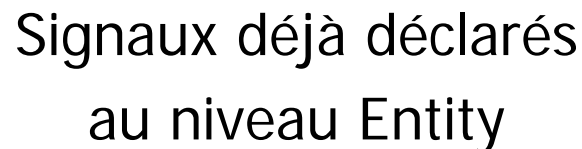
-- zone de déclaration (types, signaux internes,  
fonctions, constantes etc..)

BEGIN

-- instructions concurrentes

```
    egal <= ' 1 ' WHEN a = b ELSE ' 0 ' ;
```

```
END simple;
```



Signaux déjà déclarés  
au niveau Entity

# Instructions concurrentes

---

- ↙ L 'ordre des instructions concurrentes est indifférent
- ↙ Les processus sont explicites ou implicites
- ↙ Les processus communiquent par des signaux
- ↙ Se situent entre BEGIN et END de l 'architecture
- ↙ Le processus correspond souvent à un circuit simple. Les connexions entre circuits sont automatiquement implantées par les signaux d 'échange

# Tout est processus !

ARCHITECTURE portes OF set\_reset IS

SIGNAL s, r, q, qb : bit;

BEGIN

-- premier processus

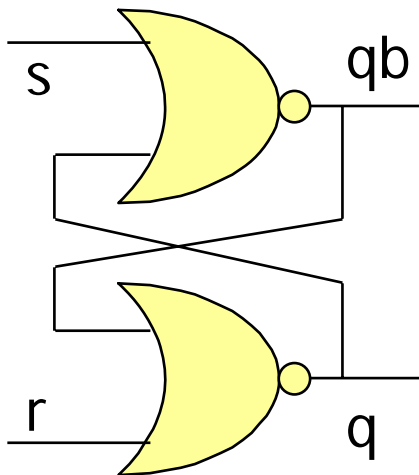
qb <= s NOR q;

-- deuxième processus

q <= qb NOR r;

END ;

-- l'ordre d'écriture est indifférent !!!



# Les signaux

- ↙ Type déclaré
- ↙ État en référence à ce type
- ↙ Un ou plusieurs pilotes (driver) associés

s <= '0' , '1' AFTER 10 ns , '0' AFTER 25 ns; -- un seul pilote

|       |     |     |     |
|-------|-----|-----|-----|
| heure | 0   | 10  | 25  |
| état  | '0' | '1' | '0' |

Pilote de s

# Les événements

---

- ↳ Tout changement d'état d'un signal .Ex: passage de 9 à 10 pour un type entier, passage de '0 ' à '1 ' pour un type bit
- ↳ Un événement est **non mûr** tant que l'heure de simulation est inférieure à l'heure dans le pilote concerné.
- ↳ Une **transaction** est un calcul sur un signal.

# Les processus

---


- ↳ **Tout est Processus.** Les processus sont **concurrents**.
- ↳ Un processus est **synchronisé** par une (ou plusieurs) instructions WAIT
  - ↳ WAIT ON <événement>
  - ↳ WAIT UNTIL <condition booléenne>
  - ↳ WAIT FOR <durée>
  - ↳ WAIT
- ↳ Les processus sont **cycliques**
- ↳ Un Processus est explicite (PROCESS) ou implicite (instruction concurrente)

# Déroulement de la simulation

---

↳ Départ : heure de simulation = 0

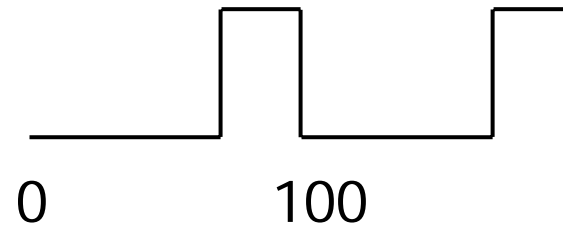
↳ Roue temporelle:

- 
- ↳ Les événements non mûrs sont classés selon les heures futures croissantes
  - ↳ L'heure de simulation courante est avancée au premier événement non mûr de la liste
  - ↳ Cet événement réveille un processus qui s'exécute immédiatement et affecte des pilotes.

# Exemple1: Génération d 'horloge

---

```
SIGNAL h : bit;  
BEGIN  
horloge: PROCESS  
BEGIN  
    h <= ' 0 ', '1 ' AFTER 75 ns; -- affectation du pilote  
    WAIT FOR 100 ns;  
END PROCESS;
```

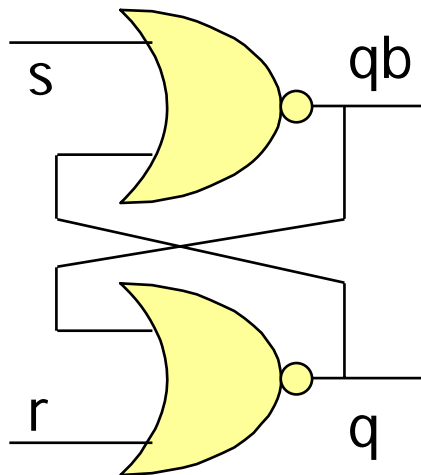


## Exemple2: Mémoire Set-Reset

ENTITY memoire\_sr IS

PORT ( s, r : IN bit;  
q, qb : OUT bit);

END;



ARCHITECTURE simpliste OF memoire\_rs IS

SIGNAL qi : bit ; SIGNAL qbi : bit := '1' ;

CONSTANT tp : time := 2 ns;

BEGIN

qi <= qbi NOR r AFTER tp; -- processus

qbi <= qi NOR s AFTER tp; -- processus

q <= qi; -- processus

qb <= qbi; -- processus

END;

## Exemple2: Simulation

|           |   |   |    |
|-----------|---|---|----|
| heure     | 3 | 8 | 10 |
| événement | s | s | r  |

|           |    |   |    |
|-----------|----|---|----|
| heure     | 5  | 8 | 10 |
| événement | qb | s | r  |

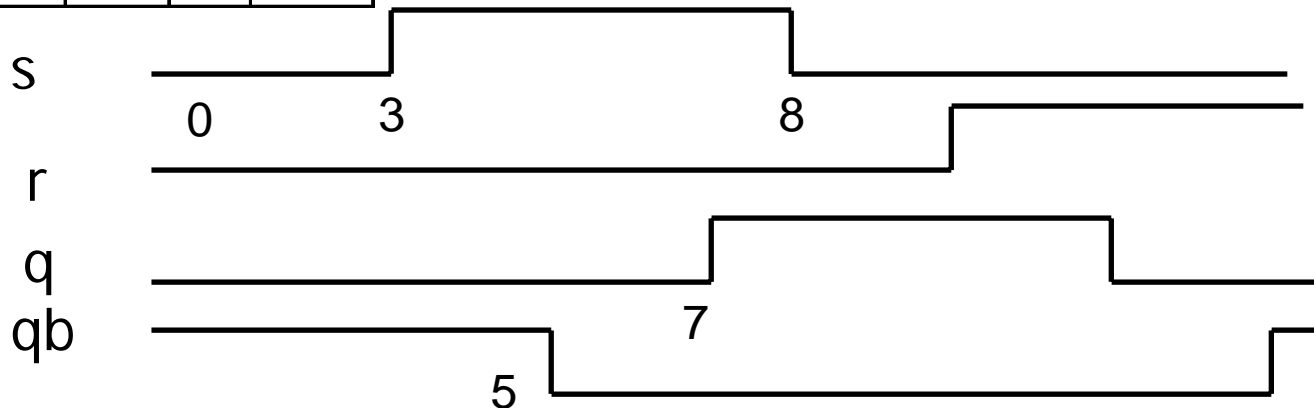
|           |   |   |    |
|-----------|---|---|----|
| heure     | 7 | 8 | 10 |
| événement | q | s | r  |

Initialisation : H=0 s=0 r=0 q=0 qb=1

Première liste pour H=0

deuxième liste pour H= 3

troisième liste pour H=5



# Instructions séquentielles

---

- ↳ Dans quel contexte ?
  - ↳ dans les processus
  - ↳ dans les sous-programmes
- ↳ L'ordre d'écriture est l'ordre d'exécution
- ↳ Les variables n'existent que dans ce contexte
- ↳ Les signaux peuvent être affectés eux aussi séquentiellement

# Signaux et variables (affectation séquentielle)

---

↙  $s \leq 1 + 2;$

↙ C 'est le **pilote** du signal qui est immédiatement affecté

↙ 3 est mémorisé dans le pilote de s

↙ Le signal est affecté plus tard ( Processus endormi par WAIT)

↙  $x := 1 + 2;$

↙ Une **variable** est affectée immédiatement

↙ x prend la valeur 3

↙ une variable n 'est pas visible au dehors du processus

# Différence entre signal et variable

```
-- SIGNAL aa, bb : integer;
```

```
P1: PROCESS
```

```
VARIABLE a : integer := 7;
```

```
VARIABLE b : integer := 6;
```

```
BEGIN
```

```
    WAIT FOR 10 ns;
```

```
    a := 1;
```

```
    b := a + 8;    -- b = 9
```

```
    a := b - 2;    -- a = 7
```

```
    aa <= a;  bb <= b;
```

```
END PROCESS;
```

```
-- aa=7  bb=9  après 10 ns,  
-- -2147483648 entre 0 et 10 ns
```

```
-- SIGNAL aa : integer := 7;
```

```
-- SIGNAL bb : integer := 6;
```

```
P2: PROCESS
```

```
BEGIN
```

```
    WAIT FOR 10 ns;
```

```
    aa <= 1;    -- 1 dans pilote de aa
```

```
    bb <= aa + 8;    -- 15 dans pilote de bb
```

```
    aa <= bb - 2;    -- 4 dans pilote de aa
```

```
END PROCESS;
```

```
-- aa=7  bb=6 pendant 10 ns
```

```
-- aa=4  bb=15 ensuite .....
```

# Équivalence entre processus implicites et explicites

↙ s <= a AND b;  
↙ affectation concurrente

↙ neuf <= '1' WHEN etat = 9 ELSE '0'; -  
- affectation concurrente

```
equivalent:PROCESS  
  BEGIN  
    WAIT ON a,b;  
    s <= a AND b;  
  END PROCESS;
```

```
equivalent:PROCESS(etat)  
  BEGIN  
    IF etat = 9 THEN  
      neuf <= '1';  
    ELSE  
      neuf <= '0';  
    END IF;  
  END PROCESS;
```

# Le délai delta

---

- ↳ Permet de traiter des évènements simultanés.  
Correspond à une itération
- ↳ Vaut 0 en terme d 'heure de simulation
- ↳ Le timestep est la résolution du simulateur ( une femtoseconde)
  - ↳ `s <= e ; -- correspond à un delta time`
  - ↳ `s <= e AFTER 0 ns; -- équivalent`

# Délai delta : exemple simple

```
ENTITY delta IS
PORT ( a : IN bit;
      b : IN bit;
      c : IN bit;
      s : OUT bit);
END delta;
```

Trois exemples de  
description  
d'une fonction très simple

$$s = \overline{(a \cdot b)} + c$$

# Un processus = un délai delta

```
ARCHITECTURE un_process OF delta IS
```

```
BEGIN
```

```
  un:PROCESS (a,b,c)
```

```
    VARIABLE i1, i2 : bit;
```

```
  BEGIN
```

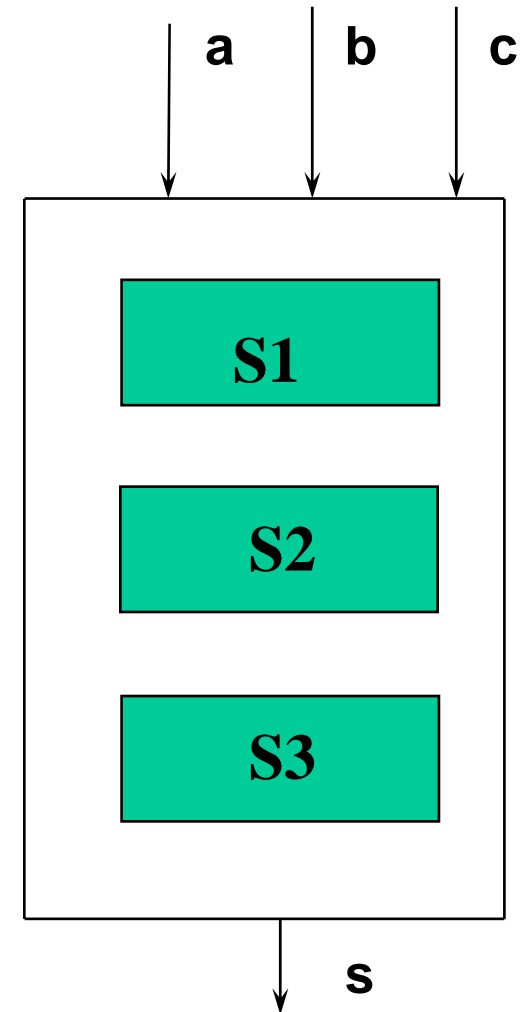
```
    i1:= a AND b;  -- S1
```

```
    i2 := c OR i1;  -- S2
```

```
    s <= NOT i2;   -- S3
```

```
  END PROCESS;
```

```
END un_process;
```



# Une instruction concurrente = un délai delta

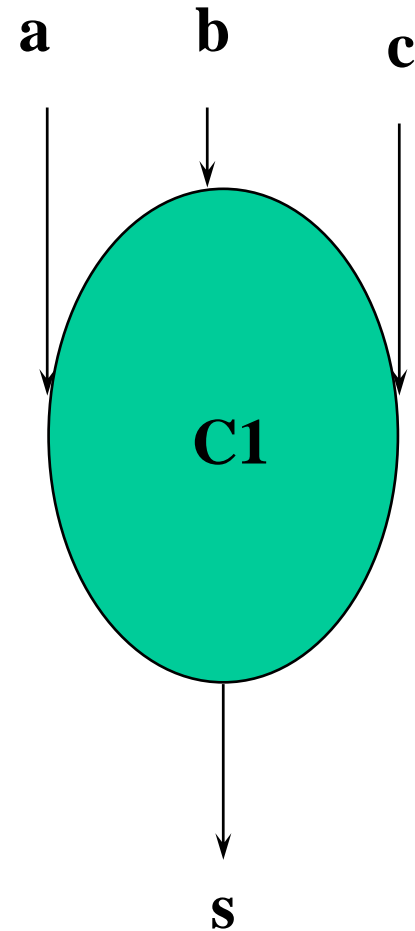
↙ ARCHITECTURE plus\_simple OF  
delta IS

BEGIN

-- une seule instruction concurrente C1

s <= NOT (c OR ( a AND b)) ;

END plus simple;



# Plusieurs instructions concurrentes

ARCHITECTURE concurrent OF delta IS

SIGNAL i1, i2 : bit;

BEGIN

-- l'ordre est indifférent

i2 <= c OR i1; -- C1

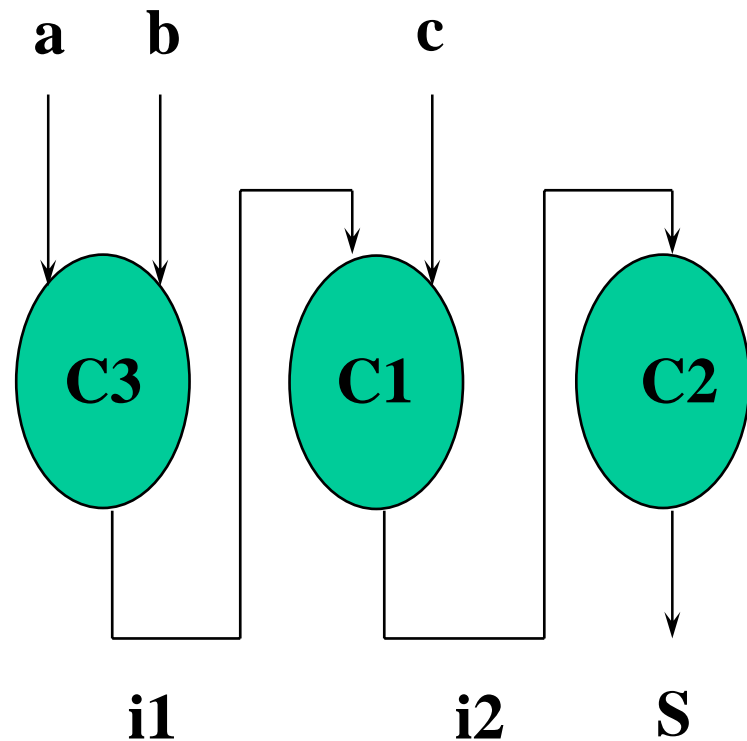
s <= NOT i2; -- C2

i1 <= a AND b; -- C3

END concurrent;

-- deux délais entre c et s

-- trois délais entre a ou b et s



# Description structurelle

---

- ↙ Description de type hiérarchique par liste de connexions
- ↙ Permet de construire une description à partir de couples entité-architecture
- ↙ Le nombre de niveaux de hiérarchie est quelconque
- ↙ Une description est structurelle au sens VHDL si elle comporte un ou plusieurs composants (**component**)

# Description structurelle: 3 étapes

---

## ↳ Déclarer

- ↳ un composant (COMPONENT) : Support pour le câblage
- ↳ une liste de signaux (SIGNAL) nécessaires au câblage

## ↳ Instancier

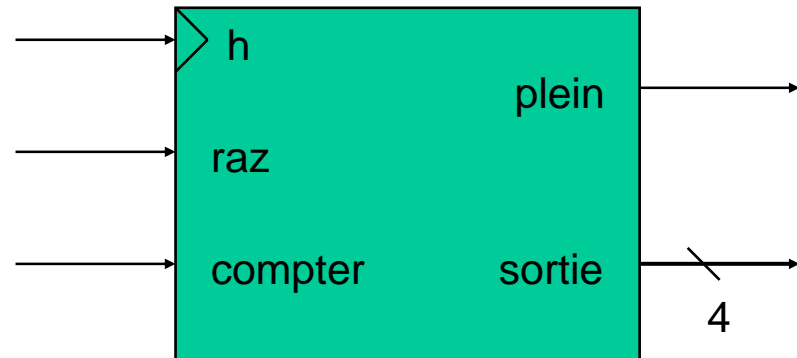
- ↳ chaque composant en fixant les paramètres (GENERIC MAP) et le câblage(PORT MAP)

## ↳ Configurer

- ↳ Choisir pour chaque composant instancié le modèle correct (couple Entité-architecture). (USE)

# Exemple d'école: compteur 4 bits

## Aspect extérieur du compteur 4 bits

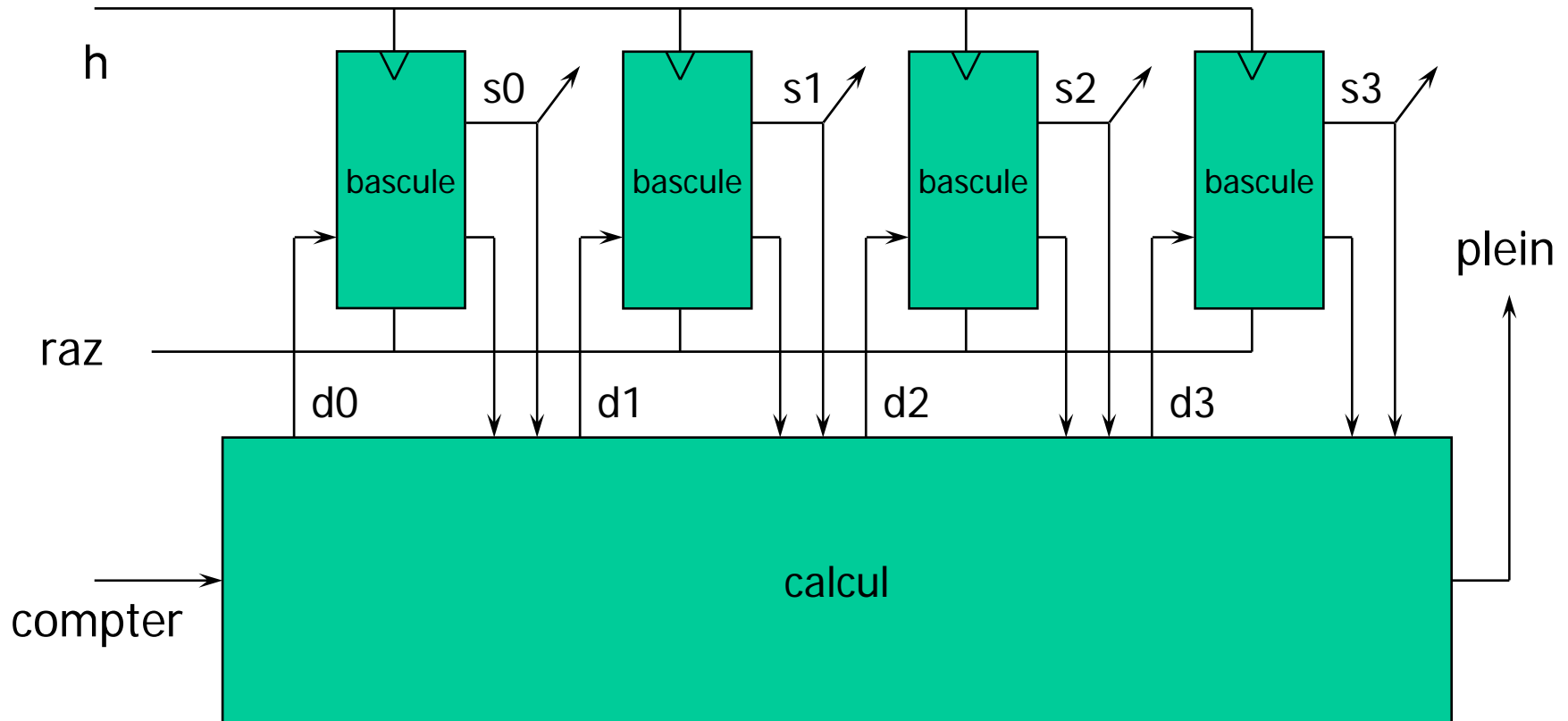


```
ENTITY compteur4 IS
```

```
    PORT (      h : IN bit;  
              raz : IN bit;  
              compter : IN bit;  
              sortie  : OUT bit_vector(3 DOWNT0 0);  
              plein  : OUT bit);
```

```
END compteur4;
```

# Schéma de compteur 4 bits



# Déclarations et configuration:

ARCHITECTURE structurelle OF compteur4 IS

COMPONENT bascule

PORT( h,d,raz : IN bit;  
s,sb : OUT bit);

END COMPONENT;

COMPONENT calcul

PORT( s,sb : IN bit\_vector(3 DOWNT0 0);  
compter : IN bit;  
d : OUT bit\_vector(3 DOWNT0 0);

END COMPONENT;

SIGNAL d, s, sb : bit\_vector(3 DOWNT0 0);

FOR ALL : bascule USE ENTITY WORK.bascule\_d(rtl);

FOR C1 : calcul USE ENTITY WORK.equations(par\_10);

BEGIN

composants

signaux

configuration

# Instanciation et câblage

BEGIN

Par position

B3: bascule

```
PORT MAP (h, d(3), raz, s(3), sb(3));
```

B2: bascule

```
PORT MAP (h => h, d => d(2), raz => raz, s => s(3),  
sb => sb(3));
```

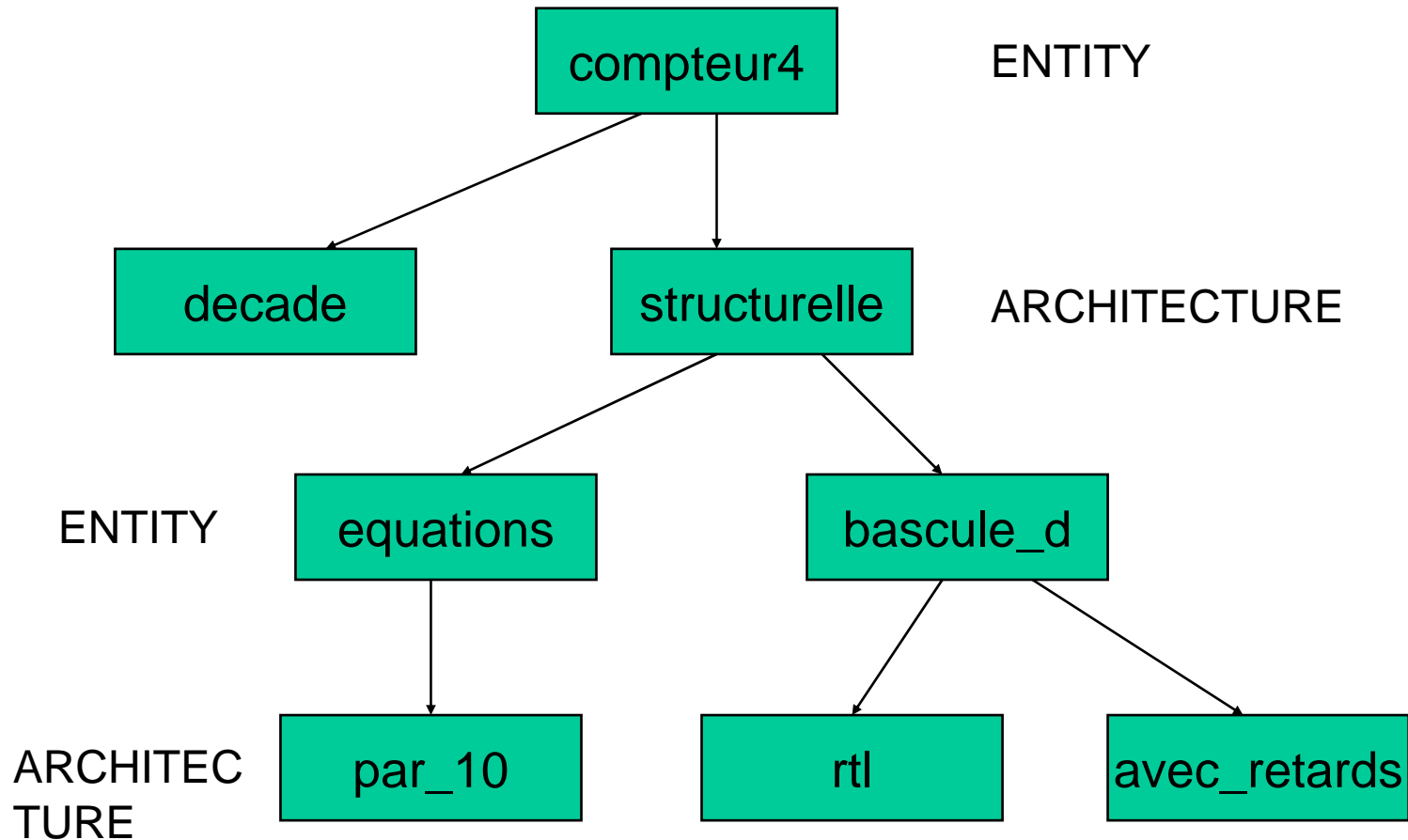
-----  
combi: calcul

```
PORT MAP (s, sb, compter,d, plein);
```

Par dénomination

END structurelle;

# Configurations



# Configuration séparée

---

```
CONFIGURATION compteur4_config OF compteur4 IS
```

```
FOR structurelle
```

```
    FOR ALL: bascule USE ENTITY WORK.bascule_d(rtl);
```

```
    END FOR;
```

```
    FOR C1: calcul USE ENTITY WORK.equations(par_10);
```

```
    END FOR;
```

```
END FOR;
```

```
END compteur4_config;
```

# Simplifications éventuelles

---

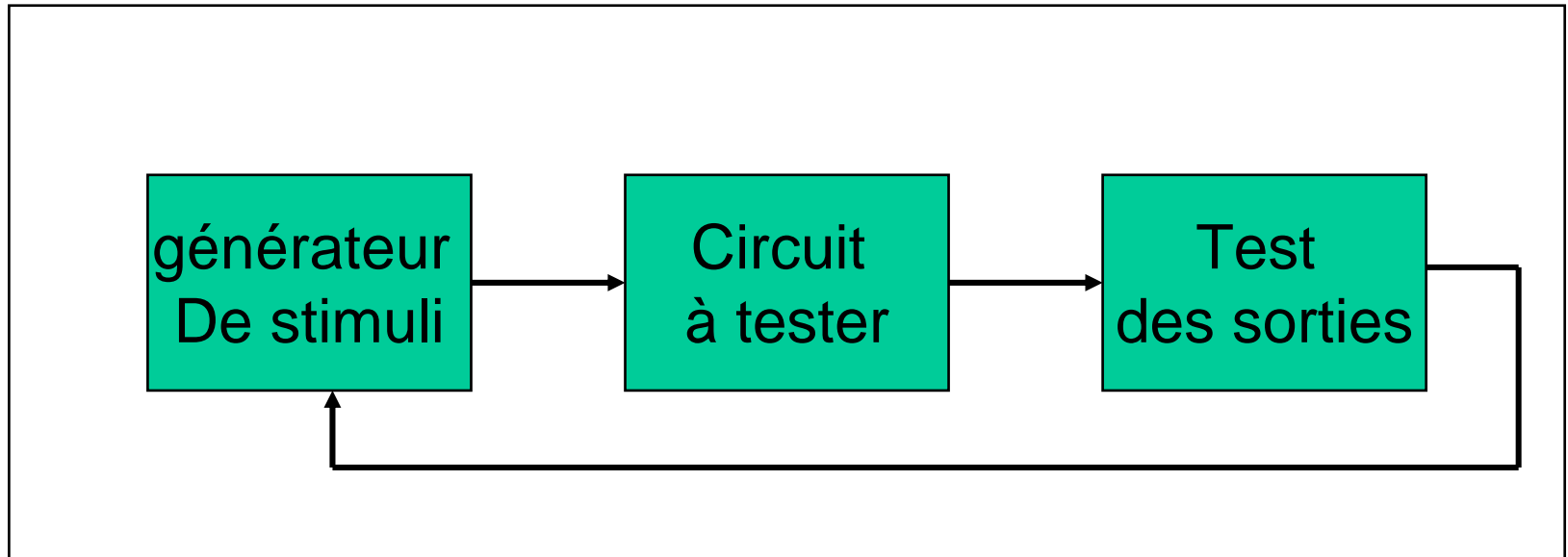
- ↙ Même nom pour COMPONENT et ENTITY: Alors la configuration peut être implicite et donc omise.
- ↙ Instanciation directe : Pas de déclaration de COMPONENT ni de clause de CONFIGURATION  
ex:

ARCHITECTURE

BEGIN

B1: ENTITY WORK.bascule\_d(rtl);

# Maquette de test



```
ENTITY test_circuit IS END;
```

```
ARCHITECTURE tb OF test_circuit IS
```

```
-- instantiation des générateurs , du circuit, des fonctions de test
```

- ↙ Champs d'application du langage VHDL
- ↙ VHDL: langage à instructions concurrentes
- ↙ Les spécificités du langage
- ↙ La synthèse des circuits
- ↙ Modélisation des circuits avec retards

# Les spécificités du langage

---

- ↳ Éléments lexicaux
- ↳ Les structures de contrôle
- ↳ Types et sous-types
- ↳ Sous-programmes (fonctions et procédures)
- ↳ Blocs et généricité
- ↳ Les Attributs
- ↳ Signal avec pilotes multiples
- ↳ Les bibliothèques

# Éléments lexicaux

---

↳ Identificateurs et mots réservés

↳ Littéraux

↳ Agrégats

↳ Opérateurs prédéfinis

# *Identificateurs et mots réservés*

---

↙ Caractère alphabétique (suivi de caractères alphanumériques)

bascule\_d BasculeD ~~bascule\_7474 74LS00~~

↙ Mots réservés du langage ne peuvent être utilisés comme identificateurs

EX: REPORT, CONSTANT, WAIT .....

↙ Majuscules, minuscules non différenciées

nom, Constante, MOT\_CLEF

# Mots réservés

---

ABS ACCESS AFTER ALIAS ALL AND ARCHITECTURE ARRAY  
ASSERT ATTRIBUTE BEGIN BLOCK BODY BUFFER BUS CASE  
COMPONENT CONFIGURATION CONSTANT DISCONNECT  
DOWNTO ELSE ELSIF END ENTITY EXIT FILE FOR FUNCTION  
GENERATE GENERIC GUARDED IF IN INOUT IS LABEL  
LIBRARY LINKAGE LOOP MAP MOD NAND NEW NEXT NOR  
NOT NULL OF ON OPEN OR OTHERS OUT PACKAGE PORT  
PROCEDURE PROCESS RANGE RECORD REGISTER REM  
REPORT RETURN SELECT SEVERITY SIGNAL SUBTYPE  
THEN TO TRANSPORT TYPE UNITS UNTIL USE VARIABLE  
WAIT WHEN WHILE WITH XOR

# Littéraux

---

↙ Caractères: '0', 'x', 'a ', '% '

↙ Chaînes: "11110101", "xx", "bonjour" , "\$@&"

↙ Chaînes de bits: B"0010\_1101" , X "2D" , O "055"

↙ Décimaux: 27, -5, 4e3, 76\_562, 4.25

↙ Basés: 2#1001#, 8#65\_07, 16#C5#e2

# Agrégats

---

## ↳ Représentation d'une valeur pour un type composite

### ↳ par dénomination

(jour => 28, mois => septembre, année => 1999)

### ↳ par position

( 1, 2, 3, 4, 5) vecteur de 5 entiers

('1','0','1', OTHERS => '0') vecteur de bits

# Opérateurs prédéfinis

---

## ↳ Logiques (boolean, bit, std\_ulogic)

AND, OR, NAND, NOR, XOR, NOT

## ↳ Relationnels ( retournent un boolean)

= /= < <= > >=

## ↳ Arithmétiques

+ - \* / \*\* MOD REM

## ↳ Concaténations d'éléments de tableaux &

"bon" & "jour" => "bonjour"

# Types

---

## ↳ Scalaires: ordonnés et éventuellement restreints

RANGE <bas>TO <haut>

RANGE <haut> DOWNTO <bas>

## ↳ Composites: tableaux et enregistrements

## ↳ Pointeur

## ↳ Fichier

## ↳ Conversion de type

# Scalaire(1)

---

## ↳ Énumérés

- ↳ TYPE boolean IS (FALSE,TRUE);
- ↳ TYPE bit IS ('0','1');
- ↳ TYPE type\_etat IS (debut, etat1, fin);

## ↳ Entiers

- ↳ TYPE integer IS RANGE -2147483648 TO 2147483647 ;  
-- (entier 32 bits)
- ↳ SUBTYPE natural IS integer RANGE 0 TO integer'high;

Les valeurs d 'initialisation sont les limites gauche

# Scalaire(2)

---

## ↙ Flottants

↙ TYPE real IS RANGE -1.0e38 TO 1.0e38;

## ↙ Physiques

↙ TYPE time IS RANGE -2147475648 TO 2147483647

- UNITS

fs;

ps = 1000 fs;

ns = 1000 ps; -- etc...

- END UNITS;

# Composites(1)

---

## ↳ Tableaux:

TYPE bit\_vector IS ARRAY (natural RANGE <> OF bit);  
-- non contraint

TYPE string IS ARRAY ( positive RANGE <> OF character);  
-- non contraint

TYPE matrice IS ARRAY (1 TO 4, 0 TO 7) OF bit;  
-- tableau à 2 dimensions 4 X 8

# Composites(2)

---

## ↳ Enregistrements :

TYPE date IS RECORD

jour : natural RANGE 1 TO 31;

mois : string;

année : integer RANGE 0 TO 4000;

END RECORD;

CONSTANT DateNaissance : date := (29, JUIN, 1963)

# Pointeur

---

↙ Uniquement en modélisation ( n'a pas de sens en synthèse)

TYPE line IS ACCESS string;

-- définie le type line comme étant un pointeur sur une chaîne de caractères;

# Fichier

---

↙ En modélisation, ce type est très utilisé pour stocker des données de mémoires RAM, ROM etc..;

```
TYPE text IS FILE OF string;
```

```
FILE fichier : text IS IN "/net/ecole/dupont/vhdl" ;
```

↙ On dispose de procédures READLINE, READ, WRITE et WRITELINE dans la bibliothèque standard TEXTIO

# Sous-types

---

- ↳ Restent compatibles avec leur type d 'origine
  - ↳ SUBTYPE natural IS integer RANGE 0 TO integer'high
    - natural est compatible avec integer
  - ↳ Type nouvel\_entier IS RANGE 0 TO 99
    - nouvel\_entier n 'est pas compatible avec integer

# Conversion de type

---

↙ Pour des types en relation , on peut forcer le type  
**<type> ( <expression>)**

↙ TYPE donnee\_longue IS RANGE 0 TO 1000

↙ TYPE donnee\_courte IS RANGE 0 TO 15

↙ SIGNAL data : donnee\_longue

↙ SIGNAL donnee : donnee\_courte

↙ data <= donnee\_longue(donnee \* 5); **--correct**

↙ Autrement il faut une fonction de conversion

↙ Donnee\_32bits <= to\_unsigned(donnee\_entiere,32);

# Les structures de contrôle

---

↳ Ce sont des **Instructions séquentielles**

↳ **IF ...THEN ...ELSIF... ELSE ...END IF**

↳ **CASE... IS...WHEN...END CASE**

↳ (FOR / WHILE) **LOOP ... END LOOP**

↳ **ASSERT** : permet d'avoir des messages à l'écran. Existe aussi en instruction concurrente

↳ **ASSERT condition REPORT message <SECURITY ...>**

# IF...THEN...ELSIF...ELSE

```
IF a = '1' THEN
    s := '1';
ELSIF b = '1' THEN
    s := '1';
ELSE
    s := '0';
END IF;
```

Après analyse,  
s := a OR b;



# CASE

---

↳ Touts les choix doivent être finalisés

CASE entrée IS

```
WHEN "11" => s := '1';
```

```
WHEN "01" => s := '0';
```

```
WHEN "10" => s := '0';
```

```
WHEN OTHERS => s := '1';
```

```
END CASE;
```

# LOOP

---

```
boucle1: FOR i IN 0 TO 10 LOOP
```

```
    b := 2**i; -- calcul des puissances de 2
```

```
    WAIT FOR 10 ns; -- toutes les 10 ns
```

```
END LOOP;
```

```
boucle2; WHILE b < 1025 LOOP
```

```
    b := 2**i; -- calcul des puissances de 2
```

```
    WAIT FOR 10 ns; -- toutes les 10 ns
```

```
END LOOP;
```

# ASSERT

---

↳ **Instruction concurrente ou séquentielle**

↳ **Forcer un message**

```
ASSERT FALSE REPORT "Toujours à l'écran "  
SEVERITY note;
```

↳ **Test en contexte séquentiel**

```
WAIT UNTIL h = '1' -- attente front de h
```

```
ASSERT s = '1' REPORT " la sortie vaut '0' et ce  
n'est pas normal " SEVERITY warning;
```

# Les sous-programmes

---

- ↳ Fonctions et Procédures. Eléments de structuration.
- ↳ Les fonctions retournent une valeur typée
  - ↳ fonctions de conversion
  - ↳ fonctions de décodage
- ↳ Les procédures peuvent agir par effet de bord (modification de l'environnement)
- ↳ La définition du sous-programme se place dans un package ou dans une zone réservée aux déclarations ( avant BEGIN)

# Paramètres formels

---

## ↳ Procédures

### ↳ classes d 'objet

- CONSTANT
- SIGNAL
- VARIABLE

### ↳ modes

- IN (CONSTANT par défaut)
- OUT (VARIABLE par défaut)
- INOUT (VARIABLE par défaut)
- par défaut : IN

## ↳ Fonctions

### ↳ classes d 'objet

- CONSTANT
- SIGNAL
- par défaut : CONSTANT

### ↳ mode

- IN
- par défaut : IN

# Exemple de fonction

---

```
FUNCTION convert (b: bit_vector) RETURN natural IS
    VARIABLE temp : bit_vector(b'LENGTH -1 DOWNT0 0) := b;
    VARIABLE valeur : natural := 0;
BEGIN
    FOR i IN temp'REVERSE_RANGE LOOP
        IF temp(i) = '1' THEN
            valeur := valeur + 2**i;
        END IF;
    END LOOP;
    RETURN valeur;
END;
```

# Exemple de procédure

```
PROCEDURE horloge (SIGNAL h : OUT bit;  
                  th, tb : TIME) IS  
  
BEGIN  
    LOOP  
        h <= '0', '1' AFTER tb;  
        WAIT FOR tb + th;  
    END LOOP;  
END;
```

-- UTILISATION DE CETTE PROCEDURE: affectation  
concurrente au signal clk avec période de 30 ns

h1: horloge ( clk , 10 ns, 20 ns);

# Surcharge

---

- ↙ Même nom et profils différents pour plusieurs sous-programmes
- ↙ Profil => nombre de paramètres ,types associés
- ↙ Surcharge des opérateurs est autorisée
  - ↙ FUNCTION convert (n, l : natural) RETURN bit\_vector
  - ↙ FUNCTION convert ( b : bit\_vector) RETURN natural
  - ↙ FUNCTION "+" ( a, b: bit\_vector) RETURN bit\_vector

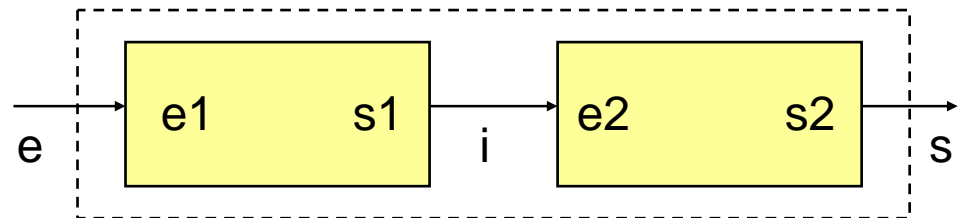
# Blocs

---

- ↙ Élément de base de la structuration
- ↙ Le bloc est une structure concurrente qui regroupe des instructions concurrentes
  - ↙ définir des frontières (PORT)
  - ↙ mettre en commun des conditions logiques (bloc gardé)
  - ↙ bloc générique
- ↙ Le bloc n'est pas exportable
- ↙ Des blocs peuvent être imbriqués

# Blocs avec PORTS

```
ENTITY es IS
  PORT (e : IN bit; s : OUT bit); -- une entrée, une sortie externe
END es ;
ARCHITECTURE deux_blocs OF es IS
  SIGNAL i : bit;                -- signal de liaison entre blocs internes
BEGIN
  B1 : BLOCK PORT (e1: IN bit; s1 : OUT bit); PORT MAP ( e1 => e, s1 => i);
  BEGIN
    s1 <= e1;
  END BLOCK B1;
  B2 : BLOCK PORT (e2: IN bit; s2 : OUT bit); PORT MAP ( e2 => i, s2 => s);
  BEGIN
    s2 <= e2;
  END BLOCK B2;
END deux_blocs;
```



# Bloc avec condition de garde

```
SIGNAL d, q, en : bit;

BEGIN

latch: BLOCK (en = '1')

BEGIN

    q <= GUARDED d ;

-- autres instructions
  concurrentes éventuelles

END BLOCK;
```

```
SIGNAL d, q, en : bit;

BEGIN

latch: PROCESS(en, d)

    BEGIN

        IF en = '1' THEN

            q <= d;

        END IF;

    END PROCESS;

-- autres PROCESS
```

# Bloc registre avec raz

---

Registre: BLOCK (h 'EVENT AND h = '1' )

-- condition de garde front montant de h

PORT (d , raz ; q : OUT bit);

-- entrées-sorties locales

PORT MAP ( d => e1, raz => e2, q => s);

BEGIN

q <= GUARDED '0' WHEN raz = '1' ELSE d ;

END BLOCK;

# Généricité

---

- ↙ S'applique aux blocs , aux entités
- ↙ Mot clef: GENERIC (CONSTANT ...
- ↙ Permet de paramétrer un modèle
- ↙ Chaque paramètre est une constante dont la valeur peut être différée (GENERIC MAP)
- ↙ La constante est statique, sa valeur sera connue pendant la phase de compilation

# Circuit générique

---

```
ENTITY compteur_gen IS
```

```
    GENERIC( Nb_bits : natural := 4;
```

```
             Modulo : natural := 16;
```

```
             Tpropagation : TIME := 2 ns);
```

```
    PORT(     h : IN bit;
```

```
            sortie : OUT bit_vector(Nb_bits - 1 DOWNTO 0);
```

```
END;
```

-- On est dans l'architecture, un COMPONENT compt a été déclaré, on instancie le compteur générique

```
C1: compt GENERIC MAP ( 5, 17, 10 ns)
```

```
    PORT MAP ( clock, sortie);
```

# Les Attributs

---

- ↙ Définition
- ↙ Attributs de type
- ↙ Attributs de tableau
- ↙ Attributs de signal
- ↙ Attribut de bloc

# Définition d'un attribut

---

↙ Caractéristique associée à un type ou un objet

↙ Déclaration => définit l'attribut

ATTRIBUTE moitie : integer ;

↙ Spécification => associe l'attribut à une classe d'entité

SIGNAL un\_bus : bit\_vector(0 TO 15);

ATTRIBUTE moitie OF un\_bus : SIGNAL IS  
un\_bus'LENGTH/2;

un\_bus'moitie retourne 8

# Attributs prédéfinis

---

↙ Attributs de type et sous-type

↙ Attributs de tableau

↙ Attributs de signal

↙ Attributs de bloc

# Attributs de type et sous-type

↳ BASE, LEFT, RIGHT, HIGH, LOW, POS(x), VAL(x),  
SUCC(x), PRED(x), LEFTOF(x), RIGHTOF(x)

TYPE t1 IS 1 TO 10;

TYPE t2 IS 10 TO 1;

t1'LEFT = 1

t2'LEFT = 10

t1'RIGHT = 10

t2'RIGHT = 1

t1'LOW = 1

t2'LOW = 1

t1'POS(4) = 3

t2'POS(4) = 6

t1 'VAL(0) = 1

t2 'VAL(1) = 9

CHARACTER 'VAL(49) donne le caractère ' 0 '

# Attributs de tableau

↙ LEFT[(N)], RIGHT[(N)], HIGH[(N)], LOW[(N)],  
RANGE[(N)], REVERSE\_RANGE[(N)], LENGTH

↙ Exemple:

```
CONSTANT Nb_bits : natural := 16; -- susceptible de changer
```

```
SIGNAL vecteur : bit_vector(0 TO Nb_bits);
```

```
.....
```

```
FOR i IN vecteur'RANGE LOOP -- écriture constante
```

```
-- donc de 0 TO Nb-bits..... ou alors
```

```
FOR i IN 0 TO vecteur'RIGHT
```

# Attributs de signal

↳ DELAYED[(T)], STABLE[(T)], QUIET[(T)],  
TRANSACTION[(T)], EVENT, ACTIVE, LAST\_EVENT,  
LAST\_ACTIVE, LAST\_VALUE

↳ EXEMPLE:

```
PROCEDURE verif_setup ( SIGNAL h, d : IN bit; Tsetup : IN time) IS  
BEGIN  
    WAIT UNTIL h 'EVENT AND h = '1';  
  
    ASSERT d'LAST_EVENT >= Tsetup REPORT "violation du  
    temps de préconditionnement " SECURITY WARNING ;  
END;
```

## ↙ BEHAVIOR

↙ nom\_architecture 'BEHAVIOR ou label 'BEHAVIOR

Booléen vrai si le bloc ou l'architecture ne contient pas d'instanciation de composant (COMPONENT)

## ↙ STRUCTURE

↙ nom\_architecture 'STRUCTURE ou label 'STRUCTURE

Booléen vrai si le bloc ou l'architecture ne contient aucune affectation de signal

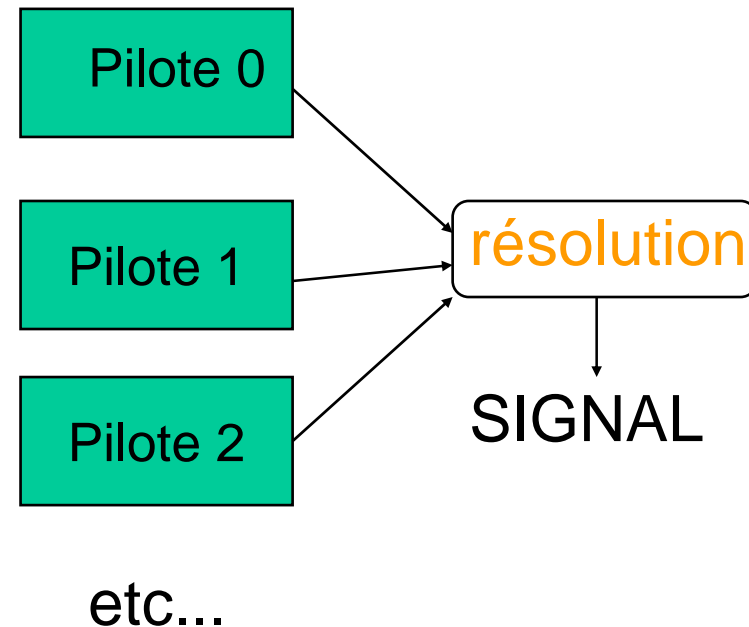
# Fonctions de résolution

---

- ↙ Signal multi-sources (logique câblée, bus)
- ↙ Fonctions de résolution associée à un signal (arbitrage pour la valeur finale)
- ↙ Les entrées de la fonction de résolution sont les valeurs des différents pilotes du même signal
- ↙ Le signal reçoit la valeur retournée par la fonction qui lui est associée
- ↙ La fonction de résolution est automatiquement appelée lors de la simulation

# Fonction de résolution NOR

```
FUNCTION resolution_nor (b: BIT_VECTOR) RETURN bit IS
  VARIABLE valeur : bit := '1';
BEGIN
  FOR i IN b'RANGE LOOP
    IF b(i) = '1' THEN
      valeur := '0';
    END IF;
  END LOOP;
  RETURN valeur;
END;
```



# Des inverseurs câblés par la sortie

SIGNAL a, b, c : bit;

SIGNAL s **resolution\_nor** bit;

-- le signal est défini comme résolu

-- erreur à la compilation si le signal était défini comme simple

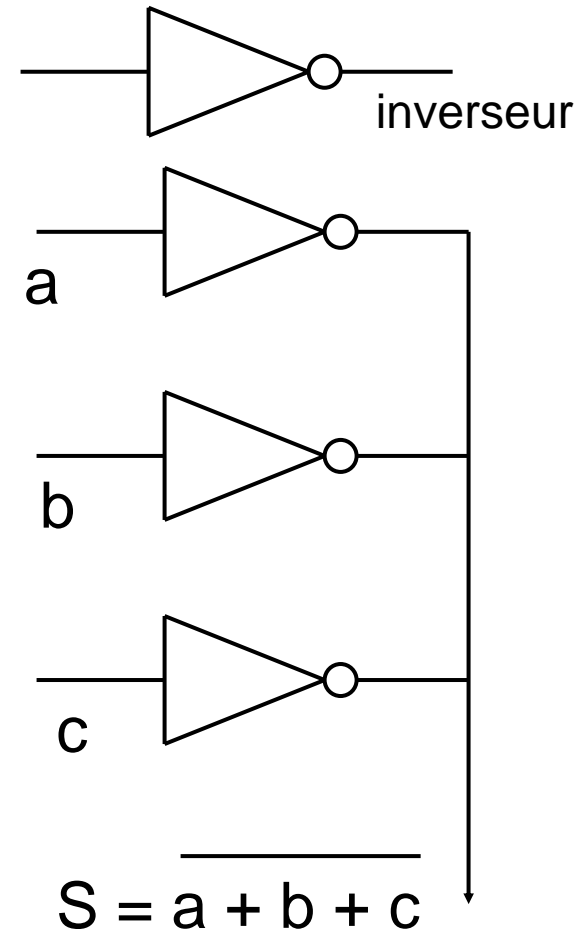
BEGIN

s <= a;

s <= b;

s <= c;

END;



Logique câblée

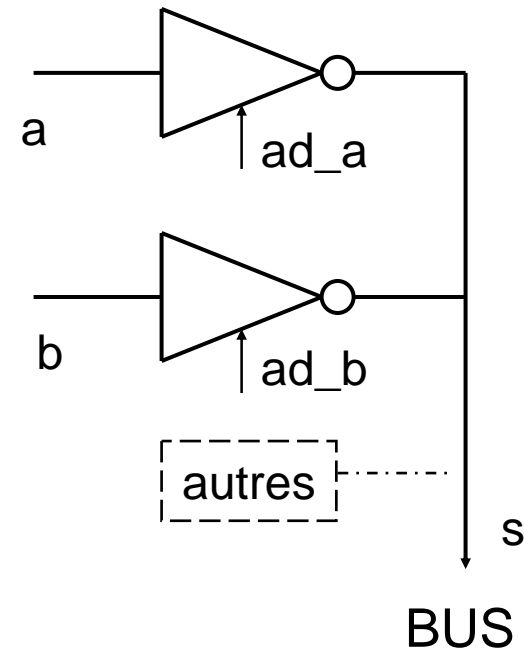
# Signal gardé

---

- ↙ Signal avec type BUS ou REGISTER dans sa déclaration
- ↙ Affectation gardée (dans bloc avec GUARDED)
  - ↙ Si condition de garde = TRUE , Alors affectation normale du signal
  - ↙ Si condition de garde = FALSE, Alors signal déconnecté ( transaction NULL)
- ↙ Déconnexion
  - ↙ Le délai de déconnexion est défini par défaut après 0 ns
  - ↙ L 'instruction DISCONNECT permet de fixer un délai autre
  - ↙ Pour un type REGISTER la dernière valeur est conservée

# Bus trois-états

```
SIGNAL a, b, ad_a, ad_b : std_ulogic; -- ('u','x','0','1','z','w','l','h','-')
SIGNAL s : std_logic BUS; -- std_ulogic avec fonction de résolution
BEGIN
trois_etats_a: BLOCK ( ad_a = '1')
    BEGIN          -- condition de garde
        s <= GUARDED a;
    END BLOCK;
trois_etats_b: BLOCK ( ad_b = '1')
    BEGIN          -- condition de garde
        s <= GUARDED b;
    END BLOCK;
-- autres
```



# Paquetages

---

- ↙ PACKAGE, unité de compilation permettant de regrouper (famille) des définitions de **types** , **constantes** ou des déclarations de **sous-programmes**
- ↙ PACKAGE BODY (optionnel) , unité de compilation permettant d'instancier des constantes, de décrire les sous-programmes
- ↙ Les bibliothèques de ressource regroupent les paquetages par famille.(LIBRARY)
- ↙ USE <Library.package.nom> pour rendre visible nom

# *Bibliothèques standards*

---

↙ LIBRARY WORK est la bibliothèque de travail

↙ Bibliothèque standard 1987 (LIBRARY STD)

↙ PACKAGE standard

↙ PACKAGE textio

↙ Bibliothèque MVL9 (LIBRARY IEEE)

↙ PACKAGE std\_logic\_1164

↙ PACKAGE numeric\_std

# Bibliothèque IEEE-package std\_logic\_1164

---

```
TYPE std_ulogic IS (
    ' U ',    -- non initialisé
    ' X ',    -- inconnu (conflit)
    ' 0 ',    -- 0 fort
    ' 1 ',    -- 1 fort
    ' Z ',    -- haute impédance
    ' W ',    -- faible indéterminé
    ' L ',    -- 0 faible
    ' H ',    -- 1 faible
    ' - ' );  -- cas indifférent (synthèse)

SUBTYPE std_logic IS resolved std_ulogic;
```

# Package `ieee.std_logic_1164`

---

## ↳ Type à 9 états et dérivés

↳ `std_ulogic`, `std_ulogic_vector`, `std_logic`, `std_logic_vector`,  
`X01`, `X01Z`, `UX01`, `UX01Z`

## ↳ Fonctions (en surcharge)

↳ **de résolution:** `resolved`

↳ **logiques:** `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `not`

↳ **de conversion:** `to_bit`, `to_bitvector`, `to_stdulogic`,  
`to_stdulogicvector`, `to_x01`, `to_X01Z`, `to_UX01`

↳ **de détection de front:** `rising_edge`, `falling_edge`

↳ **booléenne:** `is_X`

# Package *ieee.numeric\_std*

---

## ↙ Type à 9 états et dérivés

↙ signed, unsigned

## ↙ Fonctions

↙ **de résolution:** std\_ulogic\_wired\_or, std\_ulogic\_wired\_and

↙ **de conversion:** to\_integer, to\_unsigned, to\_signed, conv\_signed, zero\_extend, sign\_extend

↙ **surcharge:** +, -, \*, abs, /, mod, rem, \*\*, sla, sra, sll, srl, rol, ror, eq, =, ne, /=, lt, <, gt, >, le, <=, ge, >=, and, or, nor, xor, not, xnorand\_reduce, nand\_reduce, or\_reduce, nor\_reduce, xor\_reduce, xnor\_reduce, maximum, minimum

# Résumé

---

## ↳ Instructions Séquentielles

(dans **processus** ou **sous-programmes**)

- ↳ Contrôle :IF, CASE, LOOP
- ↳ Synchronisation: WAIT
- ↳ Affectation de **variable**
- ↳ Affectation de **signal**
- ↳ ASSERT...REPORT
- ↳ RETURN, NEXT, EXIT
- ↳ NULL
- ↳ Appel de sous-programme

## ↳ Instructions Concurrentes

(entre BEGIN et END d'un bloc ou d'une architecture)

- ↳ Affectation de **signal**, simple, conditionnelle, avec sélection
- ↳ PROCESS
- ↳ BLOCK
- ↳ Instanciation de COMPONENT
- ↳ Instruction GENERATE
- ↳ Appel de procédures

# PLAN

---

- ↙ Champs d'application du langage VHDL
- ↙ VHDL: langage à instructions concurrentes
- ↙ Les spécificités du langage
- ↙ La synthèse des circuits
- ↙ Modélisation des circuits avec retards

# *La synthèse des circuits*

---

- ↙ Restrictions du langage
- ↙ Les étapes de la synthèse
- ↙ Description de circuits combinatoires
- ↙ Description de circuits séquentiels
- ↙ Structuration des circuits synchrones: description des séquenceurs
- ↙ Description des mémoires
- ↙ Description des bus

# Restrictions du langage

---

- ↙ Un seul WAIT par processus
- ↙ Les retards sont ignorés
- ↙ Initialisations des variables/signaux ignorées
- ↙ Pas d'équation logique sur une horloge ( conseillé)
- ↙ Restriction sur les boucles (LOOP)
- ↙ Restriction sur les attributs de détection de fronts (EVENT, STABLE)
- ↙ Pas de fichier ou de pointeur
- ↙ Pas de type REAL
- ↙ Pas d'attributs BEHAVIOR, STRUCTURE, LAST\_EVENT, LAST\_ACTIVE, TRANSACTION
- ↙ Pas de mode REGISTER et BUS

# Les étapes de la synthèse

---

- ↳ Description en VHDL synthétisable. **Le style d'écriture** va déterminer l'implantation
- ↳ Le synthétiseur génère une description structurelle au niveau RTL basé sur des primitives qui lui sont propres (portes, bascules, RAM)
- ↳ La description RTL est optimisée et ciblée sur des primitives technologiques (propres à un fabricant particulier). C'est le niveau PORTE / GATE .Les critères d'optimisation sont :
  - ↳ Vitesse
  - ↳ Surface / Nombre de blocs

# Description de circuits combinatoires

---

## ↳ Ce que l'on doit implanter

- ↳ équation logique simplifiée ou non
- ↳ table de vérité, fonctions complètes ou non

## ↳ Méthodes possibles

- ↳ Instructions concurrentes
- ↳ Processus avec les entrées en liste de sensibilité
- ↳ Fonctions appliquées au signal de sortie
- ↳ Tableau de constantes

# Affectation concurrente des signaux

## ↳ Affectation simple

↳ `S <= a AND b AFTER 10 ns;`

## ↳ Affectation conditionnelle

↳ `neuf <= ' 1' WHEN etat = " 1001 " ELSE ' 0' ;`

## ↳ Affectation avec sélection

↳ `WITH ad SELECT`

`s <= e0 WHEN 0,`

`e1 WHEN 1,`

`e2 WHEN 2,`

`e3 WHEN OTHERS;`

Bonne Lisibilité  
Bien adapté  
aux circuits  
combinatoires

# Equations

```
S1 <= ( NOT a AND b) OR (NOT b  
AND a);
```

```
S2 <= a XOR b;
```

```
-- Multiplexeur
```

```
WITH ad SELECT
```

```
    s <= e0 WHEN 0,
```

```
        e1 WHEN 1,
```

```
        e2 WHEN 2,
```

```
        e3 WHEN OTHERS;
```

```
-- Multiplexeur
```

```
Multi: PROCESS(ad,e0,e1,e2,e3)
```

```
BEGIN
```

```
    CASE ad IS
```

```
        WHEN 0 => s <= e0;
```

```
        WHEN 1 => s <= e1;
```

```
        WHEN 2 => s <= e2;
```

```
        WHEN OTHERS => s <= e3;
```

```
    END CASE;
```

```
END PROCESS;
```

# Table de vérité

```
TYPE table_7seg IS ARRAY (natural RANGE <>) OF unsigned(6  
    DOWNTO 0);
```

```
CONSTANT Hex_7seg : table_7seg (0 TO 15) :=
```

```
("0111111", "0011000", "1101101", "1111100", "1011010", "1110110",  
"1110111", "0011100", "1111111", "1111110", "1011111", "1110011",  
"0100111", "1111001", "1100111", "1000111");
```

-- suite des valeurs d'affichage de 0 à F

-- implantation dans ARCHITECTURE. Cas d'un décodeur placé  
en sortie d'un compteur

```
afficheur <= Hex_7seg(compteur);
```

# Description de circuits arithmétiques

---

- ↳ Les opérateurs de base arithmétiques ou logiques sont reconnus. +, -, NOT, \* à condition d'utiliser les bonnes bibliothèques (ieee.numeric\_std).
- ↳ D'autres fonctions particulières pourraient enrichir le catalogue de primitives
- ↳ Attention au problème de conversion de type automatique

Exemple:  $S \leq a + b$  ; -- génère un circuit additionneur

*S, a, b comportent le même nombre de bits*

# Circuits arithmétiques : exemples

---

- ↙  $S \leq '0' \& e(7 \text{ downto } 1);$  -- ne génère rien car il s'agit d'une multiplication par 2. Décalage de fils.
- ↙  $S \leq a + 2*a + 4*a;$  -- multiplication par 7 impliquant 2 additionneurs.
- ↙  $S \leq 8*a - a;$  -- multiplication par 7 impliquant 1 seul soustracteur.
- ↙  $S \leq a * b;$  -- La multiplication n'est pas toujours connue des synthétiseurs

# Description de circuits séquentiels

---

## ↳ Ce que l'on doit implanter

- ↳ Mémoires sur niveau et tampons (latch)
- ↳ Bascules sur front ou registres parallèles
- ↳ Registres a décalage
- ↳ Compteurs

## ↳ Méthodes possibles

- ↳ Donner la préférence aux processus explicites
- ↳ Attention à la mémorisation implicite sur les parties combinatoires. Utiliser systématiquement des variables.

# Conception synchrone

---

- ↙ Un système est synchrone si
  - ↙ Tous les éléments de mémorisation sont sensibles à un front (bascules) et non sensibles à un niveau (tampon ou latch).
  - ↙ L'horloge d'entrée de chaque bascule est construite à partir d'une même horloge primaire.
- ↙ La conception de circuits doit être synchrone afin de pouvoir supporter les aléas de continuité inhérent aux réseaux combinatoires.

# Mémoire sur niveau (Latch)

Latch : PROCESS (en, entree)

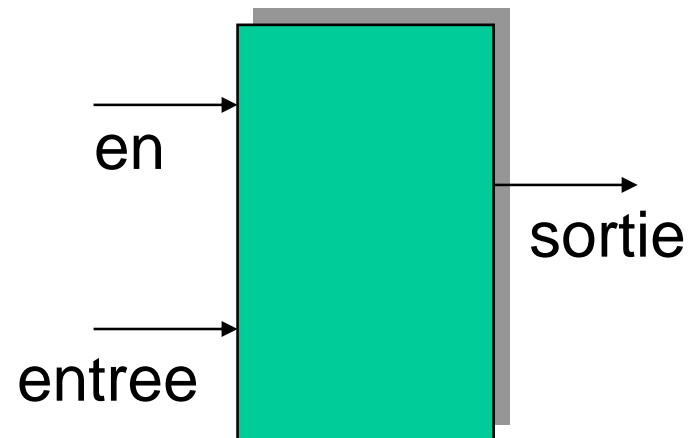
BEGIN

IF en = '1' THEN

Sortie <= entree ;

END IF;

END PROCESS latch ;



-- Ce genre de circuit est déconseillé.

# Description du front d'horloge

```
⚡ WAIT UNTIL h 'EVENT AND h = '1' ;  
    x := a;  
    WAIT UNTIL h 'EVENT AND h = '1' ;
```

-- Programmation incorrecte. Il faut comptabiliser les fronts. Un seul WAIT

```
WAIT UNTIL h 'EVENT AND h = '1' ;  
I := i+1;  
IF i = 1 THEN x := a; ELSE ..
```

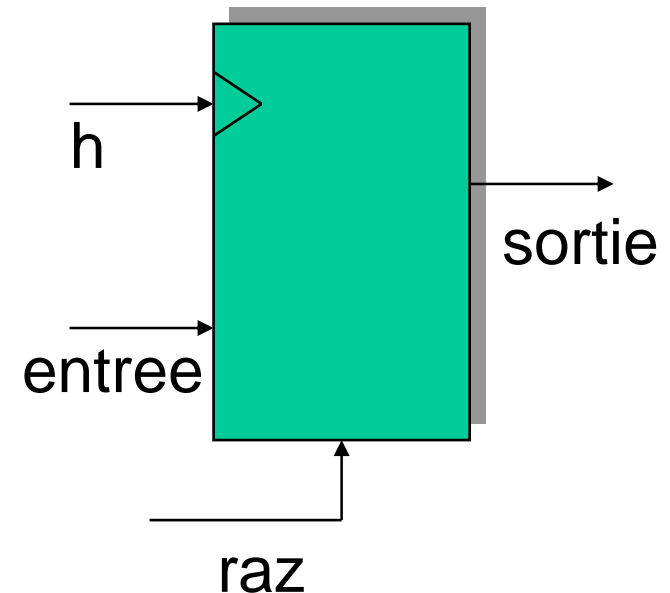
```
⚡ WAIT UNTIL h 'EVENT AND h = '1' AND raz = '0' ;
```

-- Incorrect :pas de combinaison logique sur une horloge

```
WAIT UNTIL h 'EVENT AND h = '1' ;  
IF raz = '0' THEN
```

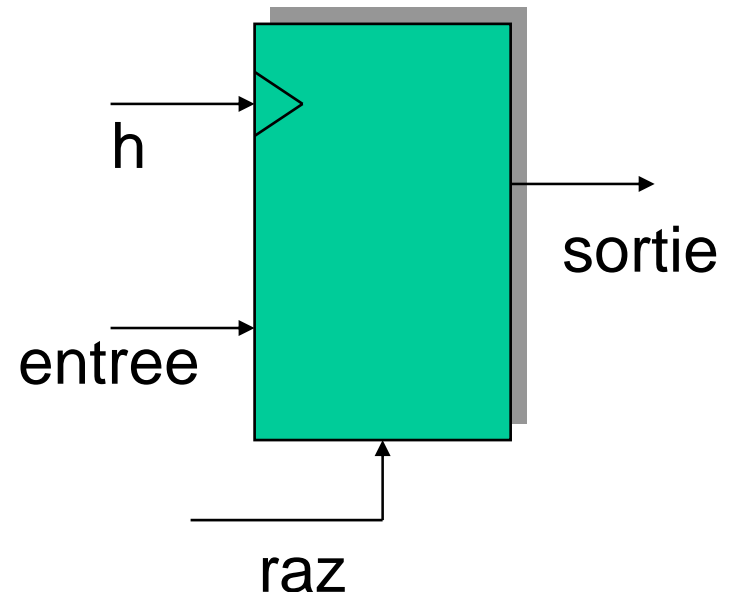
# Bascule et registre avec raz synchrone

```
Reg_sync : PROCESS
BEGIN
    WAIT UNTIL rising_edge(h);
    IF raz = '1' THEN
        Sortie <= (OTHERS => '0');
    ELSE
        Sortie <= entree;
    END IF;
END PROCESS Reg_sync ;
```



# Bascule et registre avec raz asynchrone

```
Reg_async : PROCESS
BEGIN
    WAIT ON h, raz ;
    IF raz = '1' THEN
        Sortie <= (OTHERS => '0');
    ELSIF rising_edge(h) THEN
        Sortie <= entree;
    END IF;
END PROCESS reg_async ;
```



# Bascule ou registre E

```
Reg_e : PROCESS
```

```
BEGIN
```

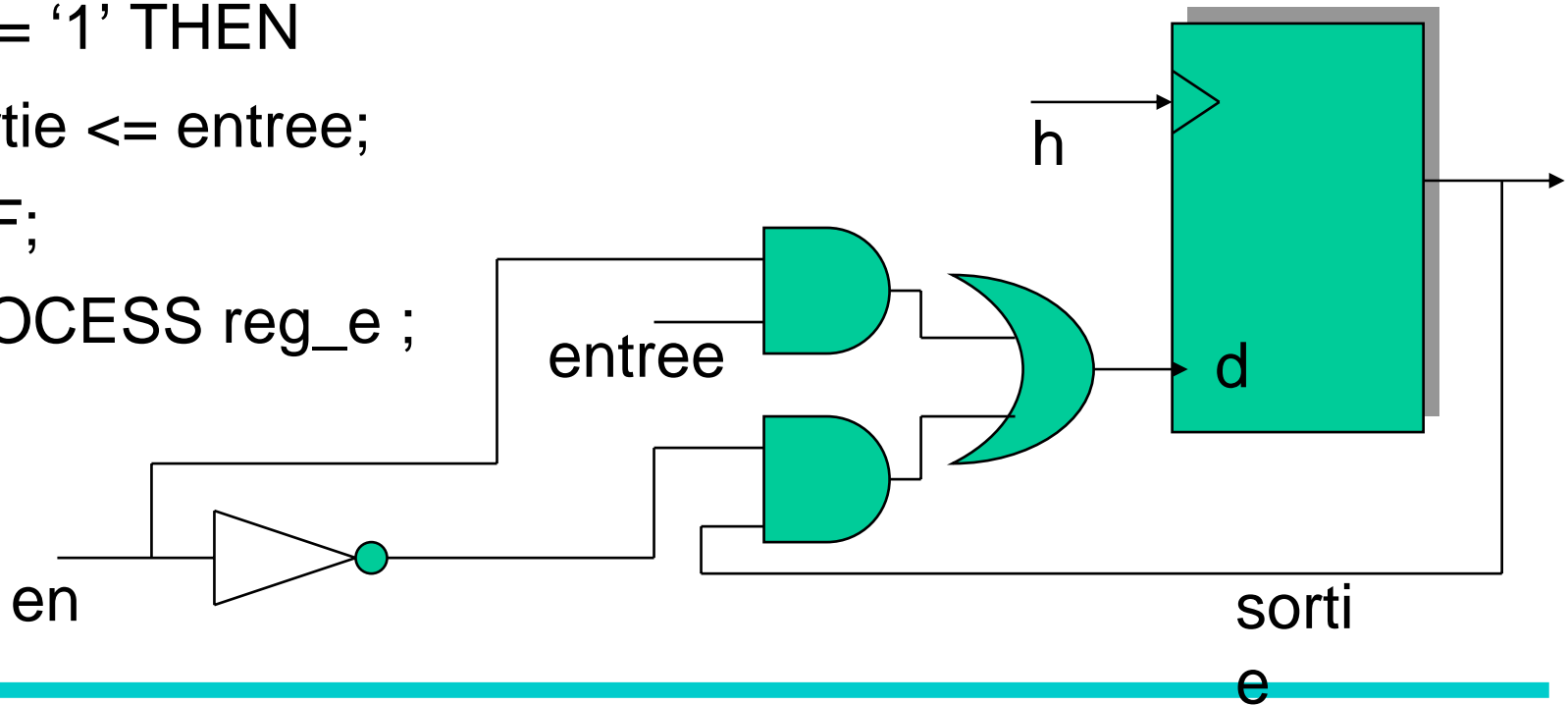
```
  WAIT UNTIL rising_edge(h);
```

```
  IF en = '1' THEN
```

```
    Sortie <= entree;
```

```
  END IF;
```

```
END PROCESS reg_e ;
```



# Compteur synchrone

---

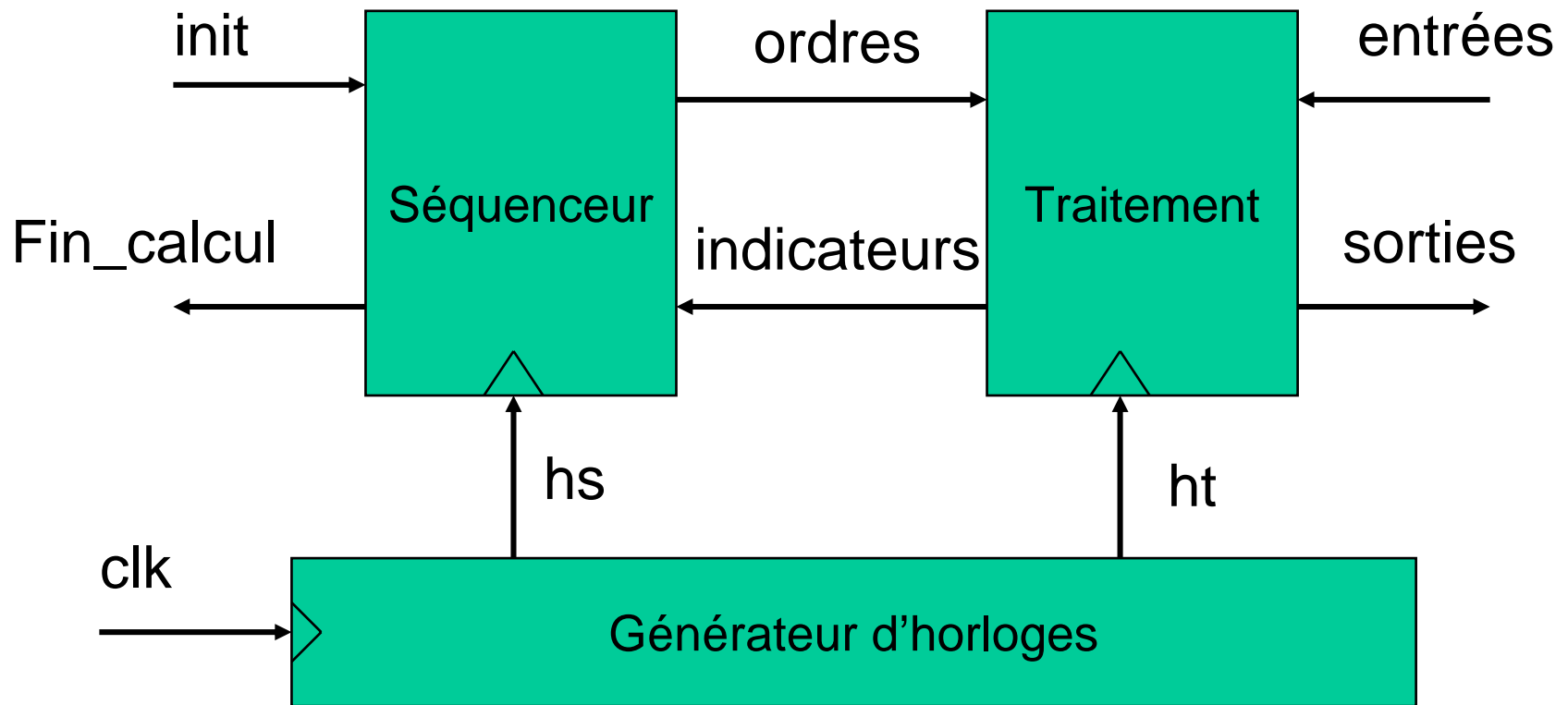
```
Compteur : PROCESS
VARIABLE c : natural RANGE 0 TO Modulo - 1;
BEGIN
    WAIT UNTIL rising_edge(h) ;
    IF compteur THEN
        IF c < Modulo - 1 THEN
            c := c + 1 ;
        ELSE
            c := 0 ;
        END IF;
        sortie <= to_unsigned( c , sortie'length);
    END IF;
END PROCESS Compteur;
```

# Registre à décalage

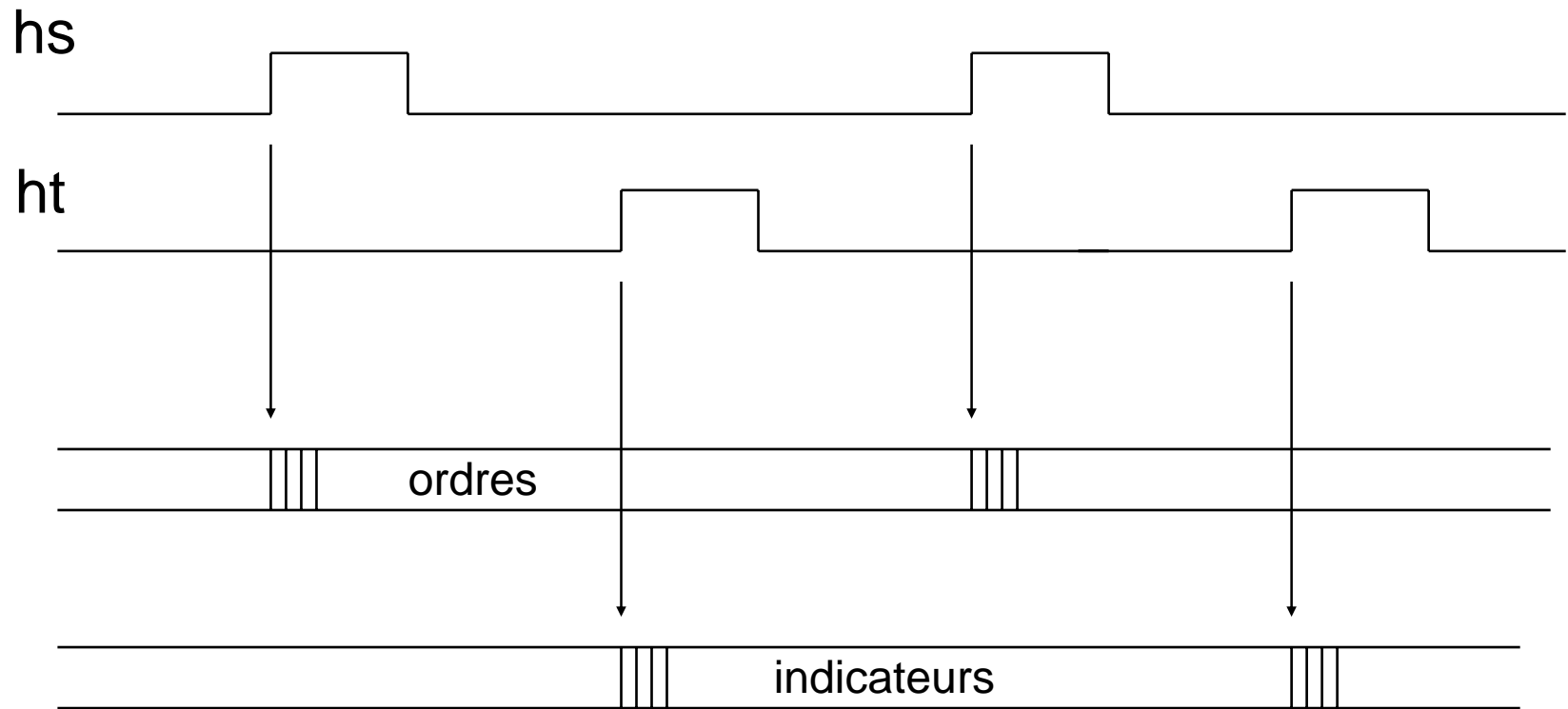
---

```
Reg_dec : PROCESS
VARIABLE stmp : std_ulogic_vector(3 DOWNT0 0);
BEGIN
    WAIT UNTIL rising_edge(h) ; -- entièrement synchrone
    CASE selection IS
        WHEN "11" => stmp := d_entree ; -- chargement parallele
        WHEN "10" => stmp := stmp(2 DOWNT0 0) & edg; -- gauche
        WHEN "01" => stmp := edd & stmp(3 DOWNT0 1); --droite
        WHEN OTHERS => NULL ; -- mémorisation
    sortie <= stmp ;
    END CASE;
END PROCESS Reg_dec ;
```

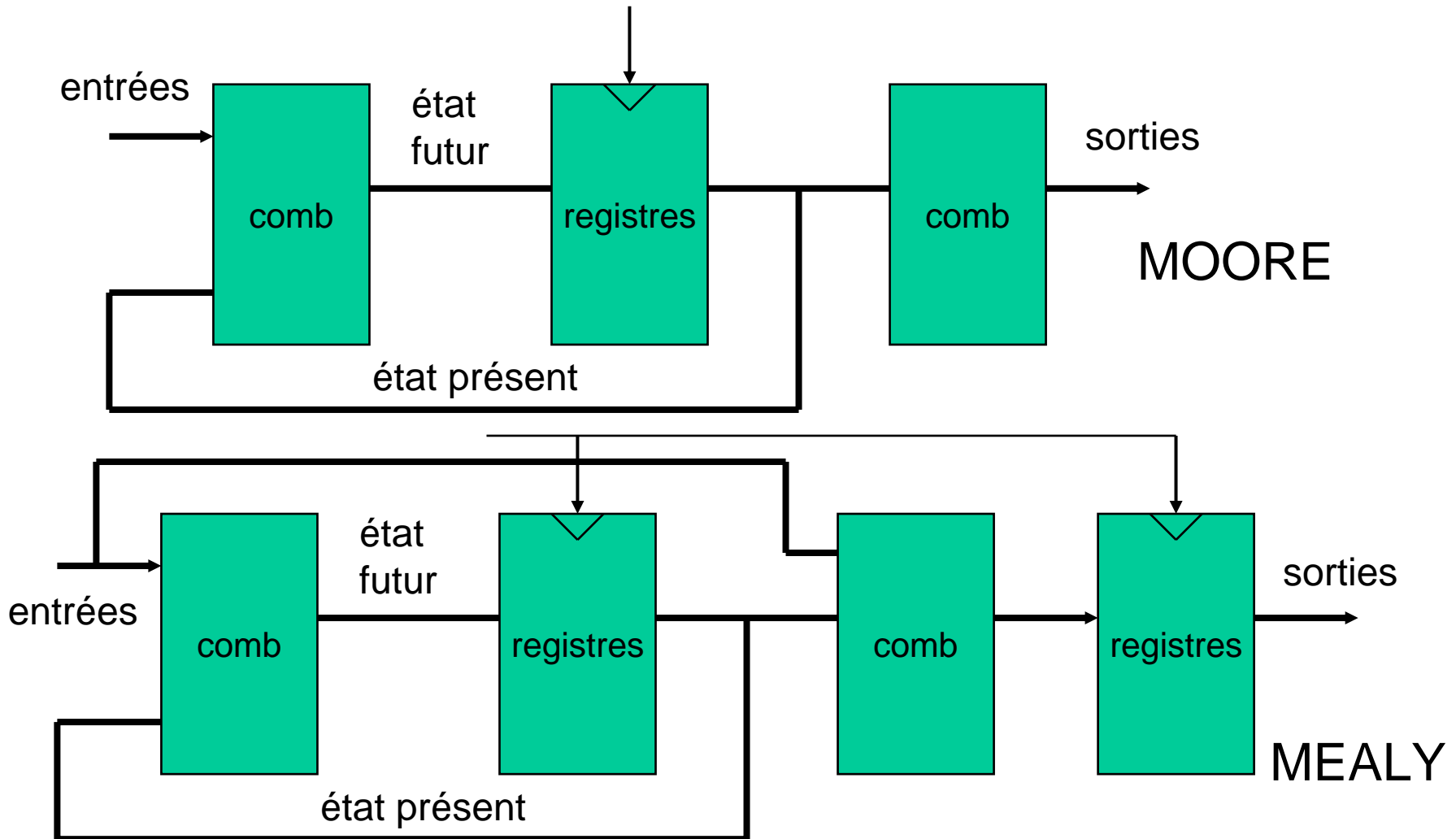
# Structuration d'un circuit synchrone



# Séquencement

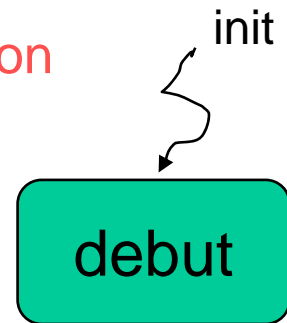


# Séquenceurs synchrones



# Description de machines d'état

```
TYPE etat IS (debut, etat1, etat2,fin);  
SIGNAL h, init : bit;  
SIGNAL etat_present, etat_futur : etat;  
SIGNAL c1, c2, c3 : boolean; -- conditions de transition  
  
BEGIN  
  coeur:PROCESS  
    BEGIN  
      WAIT UNTIL h 'EVENT AND h = ' 1 ' ; -- synchrone  
      IF init = ' 1 ' THEN etat_present <= debut; ELSE  
        etat_present <= etat_futur ;  
      END IF;  
    END PROCESS;
```



# Transitions

graphe: PROCESS (etat\_present, c1, c2, c3)

BEGIN

CASE etat\_present IS

WHEN debut => IF c1 THEN

etat\_futur <= etat1;

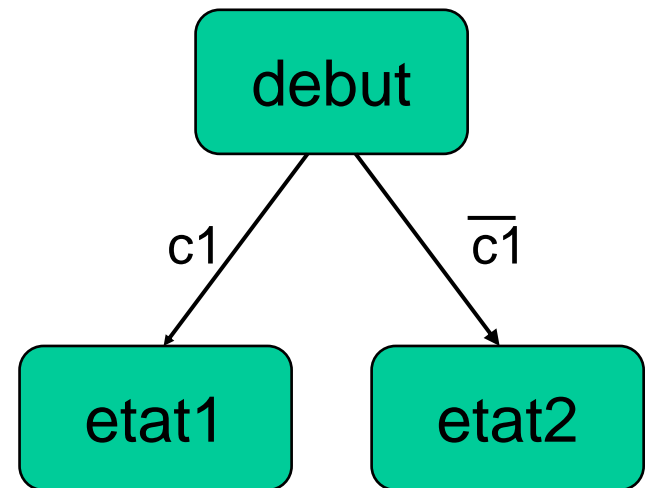
ELSE etat\_futur <= etat 2;

END IF;

WHEN etat1 => -- etc .....

END CASE

END PROCESS



# Sorties Moore ou Mealy

---

↙ Moore : la sortie ne dépend que de l'état:

```
sortie <= '1' WHEN etat_present = fin ELSE '0';
```

↙ Mealy: la sortie dépend de l'état et des entrées

-- partiellement asynchrone, doit être re-synchronisée

```
sortie <= '1' WHEN (etat_present = fin AND c3) ELSE '0';
```

# Mémoire RAM

---

⚡ Le **style d'écriture** permet au synthétiseur de reconnaître une catégorie de RAM (Leonardo)

```
TYPE matrice IS ARRAY(0 to 1023) of std_logic_vector(7 downto 0);  
SIGNAL mem : matrice
```

```
BEGIN
```

```
PROCESS(inclok, outclock, we, address)
```

```
BEGIN
```

```
    IF inclok = '1' and inclok'EVENT THEN
```

```
        IF we = '1' THEN
```

```
            Mem (address) <= data;
```

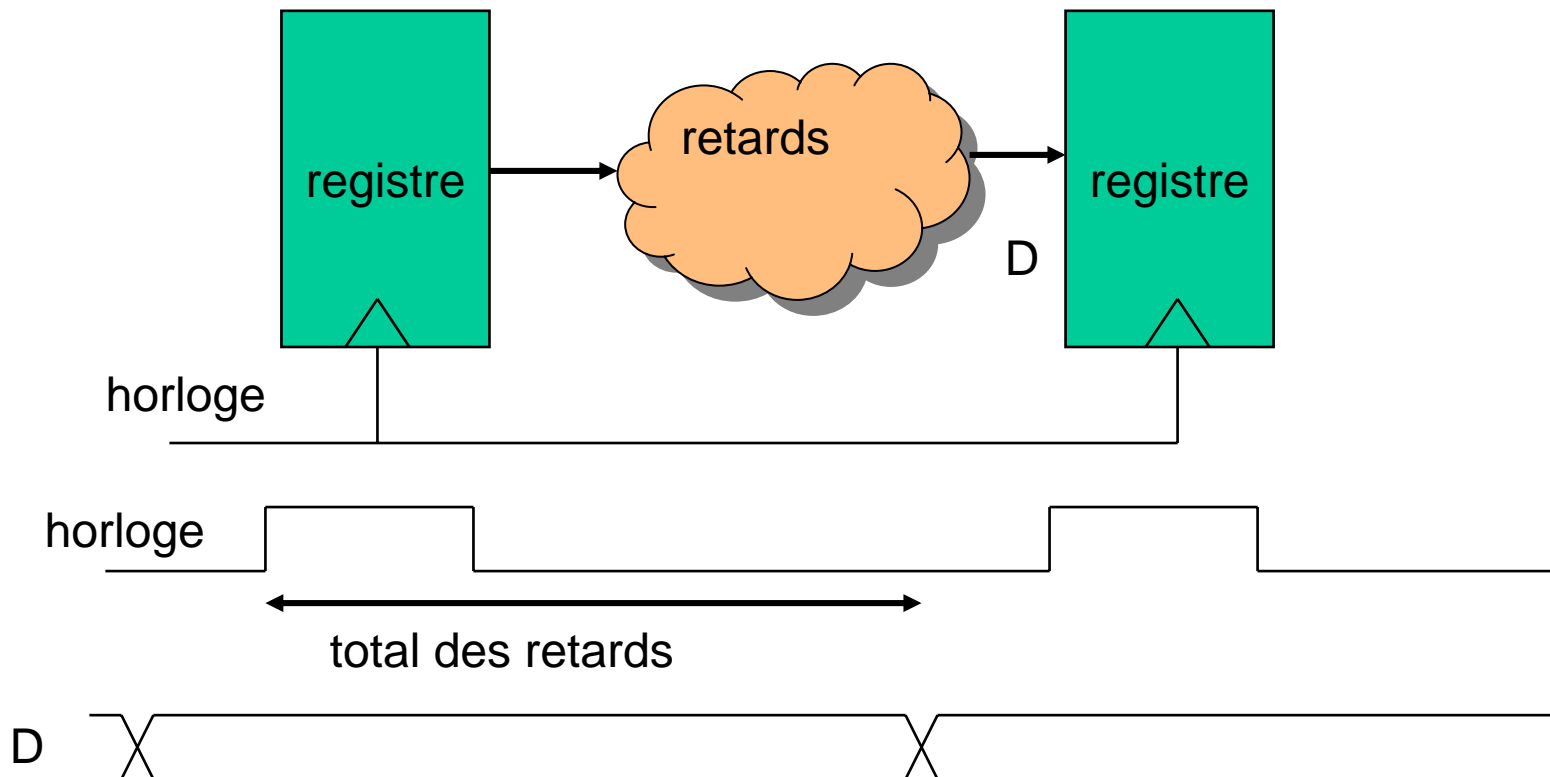
```
        END IF;
```

```
    END IF;
```

```
END PROCESS;
```

# Performances temporelles

Période d'horloge > temps de propagation bascule + retard combinatoire max + Tsetup bascule



# *Bus unidirectionnel*

---

```
ENTITY bus_trois_etat IS
PORT (periph_a, periph_b : IN std_logic_vector (7 DOWNT0 0);
      cs_a, cs_b : IN std_logic;
      sortie_commune : OUT std_logic_vector );
END ;
ARCHITECTURE synthetisable OF bus_trois_etat IS
BEGIN
    sortie_commune <= periph_a WHEN cs_a = '1'
                    ELSE "ZZZZZZZZ";
    sortie_commune <= periph_b WHEN cs_b = '1'
                    ELSE "ZZZZZZZZ";
END;
```

# *Bus bidirectionnel*

---

```
ENTITY bus_bidirectionnel IS
PORT( liaison : INOUT std_logic ; -- un seul fil
     ...);
END;
ARCHITECTURE synthetisable OF bus_bidirectionnel IS
    SIGNAL signal_interne, entree_interne, direction : std_logic ;
BEGIN
    Liaison <= signal_interne WHEN direction = '1' ELSE 'Z'
    entree_interne <= liaison ;
    ...
END ;
```

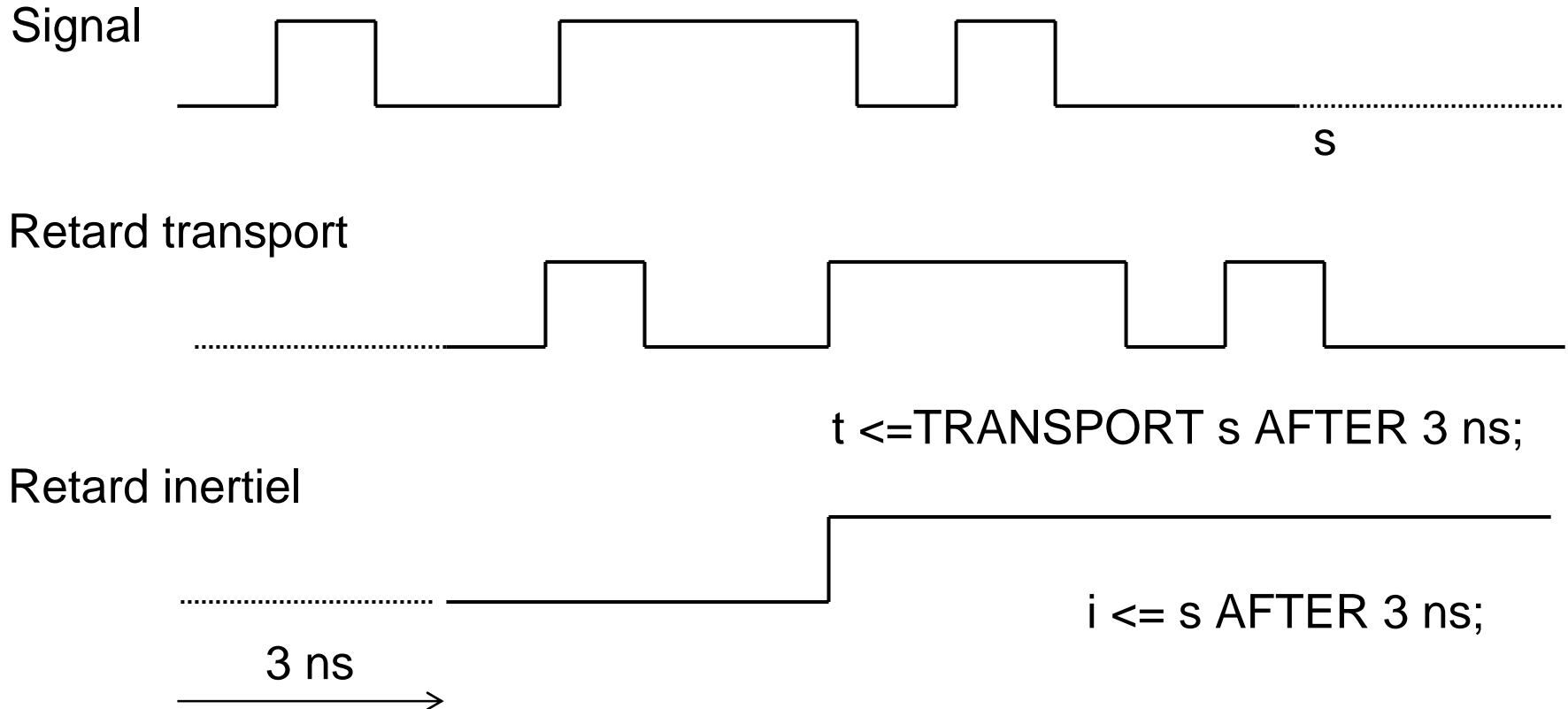
- ↙ Champs d'application du langage VHDL
- ↙ VHDL: langage à instructions concurrentes
- ↙ Les spécificités du langage
- ↙ La synthèse des circuits
- ↙ Modélisation des circuits avec retards

# Les moyens

---

- ↙ Affectation de signal: `s <= a AFTER Tp;`
- ↙ Instruction FOR: `WAIT FOR delay1;` -- affecte tous les signaux du Processus
- ↙ Fonction NOW: retourne l'heure actuelle de simulation. Limitée au contexte séquentiel
- ↙ Attributs définis fonctions du temps: `S'delayed[(T)]`, `S'Stable [(T)]`, `S'Quiet [(T)]`.
- ↙ Bibliothèques VITAL

# Modélisation des retards



# Bibliothèques VITAL

---

- ↙ VHDL Initiative Towards ASIC Libraries : organisme ayant spécifié un standard de modélisation des retards et des fonctions associées;
- ↙ Packages standards
  - ↙ VITAL\_Timing: Type de données et sous-programmes de modélisation des relations temporelles ( ex: Setup, Hold)
  - ↙ Vital\_Primitives : Jeu de primitives combinatoires et tables de vérité
  - ↙ Vital\_Memory: Spécifique pour modélisation des mémoires

# Bibliographie

---

- ↙ VHDL: Du langage à la modélisation. R.Airiau, J.M.Bergé, V.Olive et J.Rouillard - CNET
- ↙ Reuse Methodology Manual For System On Chip Designs, Michael Keating and Pierre Bricaud – Kluwer Academic Publishers 1999
- ↙ VHDL Du langage au circuit, du circuit au langage. J.Weber et M.meaudre - Masson
- ↙ <http://www.eda.org>
- ↙ Newsgroup : comp.lang.vhdl