Architecture des processeurs CE312/CE318

II Les bases du VHDL pour la synthèse

Vincent Beroulle

Grenoble INP ESISAR

Plan global du cours

- I Introduction
- **II Les bases du VHDL pour la synthèse**
- III Les composants programmables

Plan de ce chapitre

- **I. Introduction**
- II. Types
- III. Unités de conception
- IV. Simulations évènementielles
- V. Instructions séquentielles et concurrentes
- VI. Descriptions structurelles et comportementales
- VII. Description de la maquette de test
- VIII. Conclusion

Plan de ce chapitre

- I. Introduction
- **II. Types**
- III. Unités de conception
- IV. Simulations évènementielles
- V. Instructions séquentielles et concurrentes
- VI. Descriptions structurelles et comportementales
- VII. Description de la maquette de test
- VIII. Conclusion

II. Types

MAJUSCULES et minuscules
sont confondues

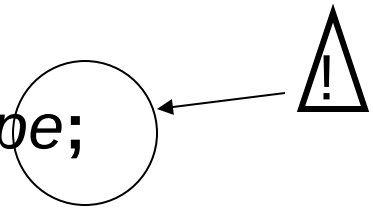
- **Tout objet (variable, signal, constante…) manipulé a un format prédéfini**
  - Seules des valeurs de ce format peuvent être affectées à cet objet
- Il existe plusieurs catégories de type :
  - *Types scalaires* (numériques et énumérés)
  - *Types composés* (tableaux et vecteurs)

II. Types

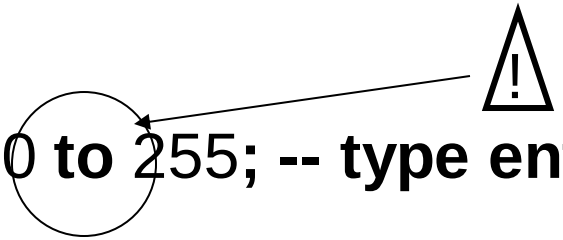
- Possibilité de définir de nouveaux types

Syntaxe :

type *nom-type* **is** *definition-type***;**



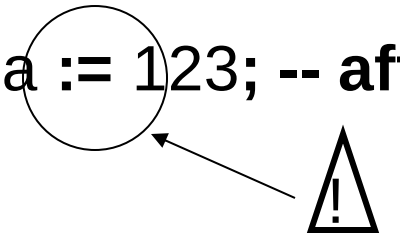
Exemple :



type octet **is integer range** 0 **to** 255**; -- type entier**

variable a **:** octet**; -- déclaration d'une variable de type octet**

a **:=** 123**; -- affectation d'une variable**



II. Types

- Existence de *types prédéfinis :*
  - *types scalaires :*
    - *Énuméré : bit, boolean*
    - *Numérique : integer,…*
  - *type composé : bit_vector…*

II. Types

- Types *scalaires* :
  - Types *énumérés* = *liste de valeurs*

    type bit **is** (‘0’, ‘1’)**;**
    type boolean **is** (false, true)**;**

    
    : ‘

    - *Valeur d'initialisation par défaut* à gauche dans la liste

  - Types *numériques* : domaine de définition
    - Domaine de définition: **range**, **to** ou **downto**

    type i **is integer range** 0 **to** 3**;**

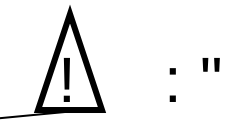
II. Types

- Types composés : collections d'éléments **de même type repérés** par des valeurs d'*indices*.
  - Cas particulier lorsque un seul indice : *vecteur*

Premier exemple :

type word **is array (**0 **to** 8**) øf** bit**;**

constant mot1 **: word :="**000000000**";**



mot1(0)

II. Types

- Exemples :
  - déclaration d'un vecteur *de bit de largeur 32 :*

variable vecteur **: bit_vector**(31 **downto** 0)**;**

Type composé non contraint

Par défaut, il est **impossible** de faire des opérations arithmétiques (+, x) sur des vecteurs de bits

II Types
STD_LOGIC_1164

- Bibliothèque et paquetage utilisés :

  library *IEEE;*

  use *IEEE.STD_LOGIC_1164.**all**;*

- Issus de la norme *IEEE 1164*, ce paquetage permet de définir les types *std_logic* et *std_logic_vector*, les opérateurs sur ces types, et des fonctions de conversion (vers les types *bit* et *bit_vector*)
- Il permet une modélisation détaillée au niveau transistor

II Types
STD_LOGIC_1164

- Définit le type *std_logic* très utile pour la simulation et la synthèse :

```
type std_logic is (
        ‘ U ’, -- non initialisé
        ‘ X ’, -- inconnu
        ‘ 0 ’, ‘ 1 ’,
        ‘ Z ’, -- haute impédance
        ‘ W ’, -- faible inconnu
        ‘ L ’, -- faible 0
        ‘ H ’, -- faible 1
        ‘ - ’ -- quelconque
);
```

II Types
numeric_std

- La norme IEEE 1076.3 définit le package *numeric_std*

  library IEEE;

  use *IEEE.STD_LOGIC_1164.**all**;*

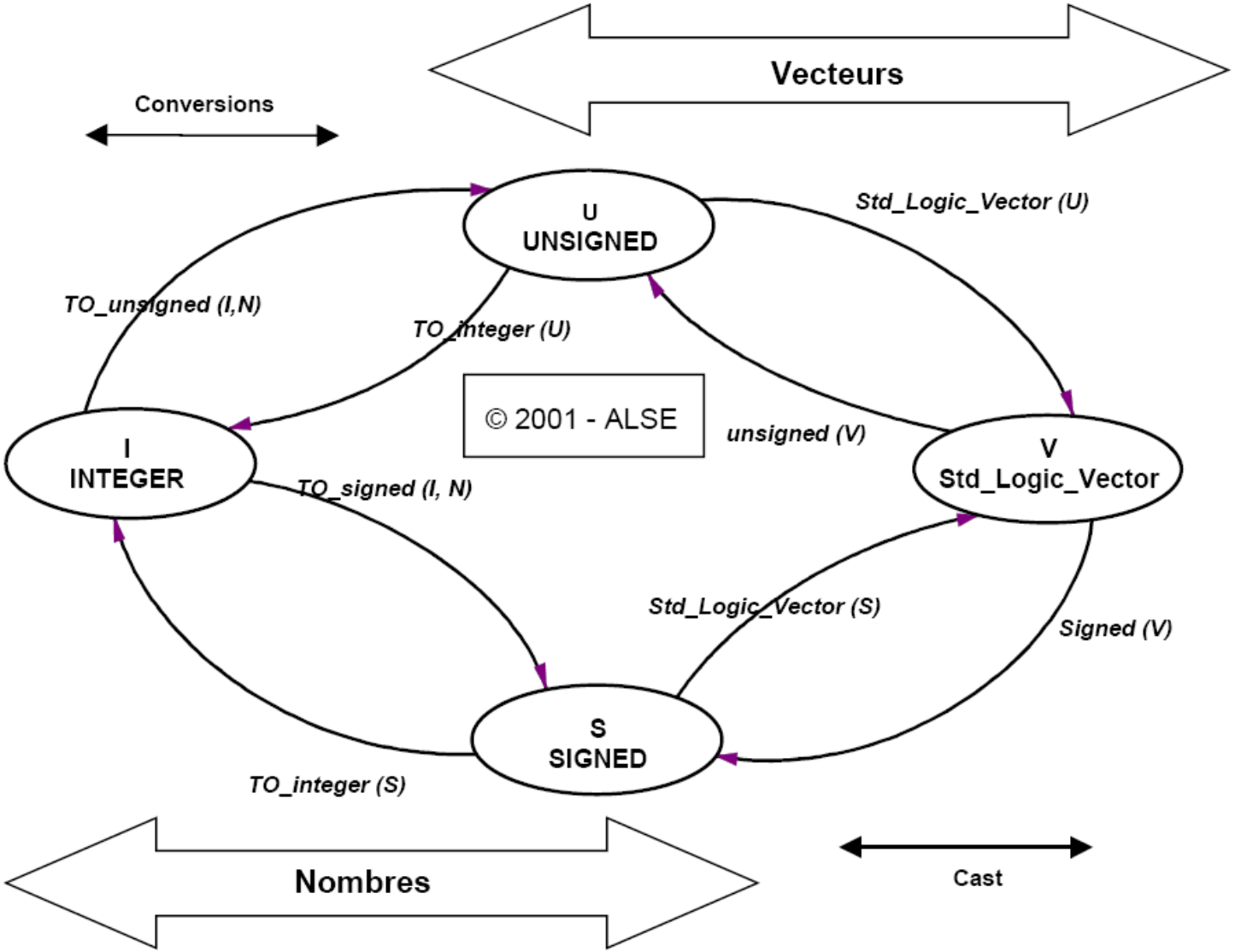
  use IEEE.NUMERIC_STD.**all**;
- Définit les types *signed* et *unsigned* : vecteurs *signés* et *non signés* (de *std_logic)* avec le poids fort à gauche
- Permet d'utiliser les *opérateurs arithmétiques* (+, x) sur des vecteurs de bit
- Fournit des *fonctions de conversion* entre entiers et vecteurs (*to_integer, to_unsigned, to_signed*)

Types
Exercice

- Faire en binaire les multiplications :
  - 6x7
  - -6x7
- Est-ce que les opérations à réaliser pour obtenir 42 et -42 en binaire sont identiques dans les 2 cas non signé et signé?

II Types

numeric_std



II Types

numeric_std

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
...
-- zone de déclaration
signal      a, b:   std_logic_vector(3 downto 0);
signal      sum:    std_logic_vector(3 downto 0));
…
--zone de définition
sum <= std_logic_vector(unsigned(a) + unsigned(b));
```

Plan de ce chapitre

- I. Introduction
- II. Types
- **III. Unités de conception**
- IV. Simulations évènementielles
- V. Instructions séquentielles et concurrentes
- VI. Descriptions structurelles et comportementales
- VII. Description de la maquette de test
- VIII. Conclusion

III. Unités de conception

Primaire et Secondaire

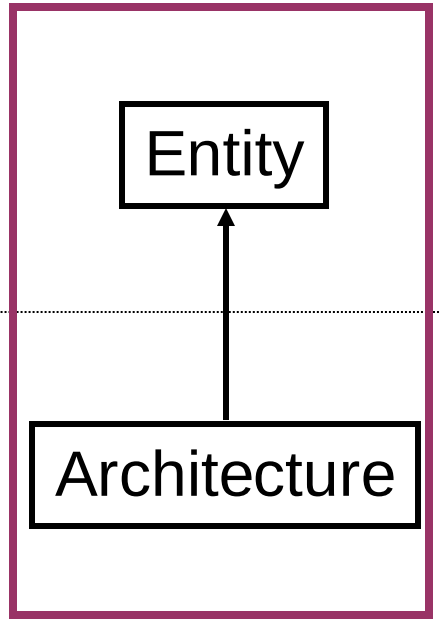
- Définition : *unité de conception*
  - Partie de programme qui peut être compilée séparément

Compilez souvent vos programmes afin d'éliminer les erreurs syntaxiques

III. Unités de conception

Unités primaires

Unités secondaires



Vue externe :
Déclaration des interfaces

Vue interne *:*
Définition fonctionnelles

En pratique, utilisez un fichier texte *nom_entity.vhd* pour chacune de vos entités *nom_entity*

III. Unités de conception

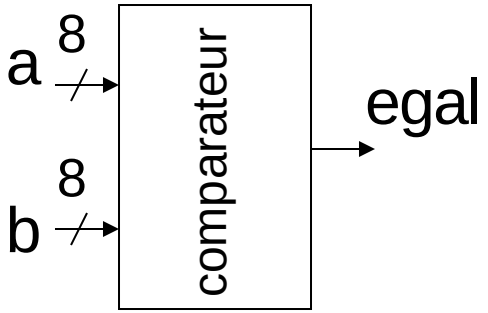
Unités de conception primaires : ENTITE

- Définition de l'entité : **Vue externe d'un composant**
- Spécification d'entité :
  - Ports d'entrées / sorties
  - Type
  - Mode : entrée (**in**), sortie (**out**), entrée/sortie (**inout**)

III. Unités de conception

Exemple d'entité

```
1   entity comparateur is
2   port(
3       a : in bit_vector(7 downto 0);
4       b : in bit_vector(7 downto 0);
5       egal : out bit);
6   end comparateur;
```



- Le mode IN *protège* le signal en écriture.
- Le mode OUT *protège* le signal en lecture.

Exercice : Écrire l'entité d'un additionneur *add4* de deux mots *a* et *b* sur 4 bits en entrée, avec une retenue entrante *ci*, et deux sorties *sum* sur 4 bits et la retenue *co*.

III. Unités de conception

Solution

III. Unités de conception

Unités de conception secondaires : ARCHITECTURE

- Toute architecture est associée à une entité
- Définition de l'architecture : **L'architecture définit les fonctionnalités et les relations temporelles**

```
architecture simple of comparateur is
-- zone de déclaration (ici commentaire uniquement)
begin
        -- zone de définition
        egal <= '1' when a = b else '0';
        ....
end simple ;
```

III. Unités de conception

Unités de conception secondaires : ARCHITECTURE

- Il peut y avoir plusieurs architectures associées à un même composant : sans délai, avec délais…

```
architecture simple of comparateur is
begin
        egal <= '1' when a = b else '0';
        ….
end simple ;
architecture complexe of comparateur is
begin
        egal <= '1' after 10 ns when a = b else '0' after 5 ns;
        ….
end complexe ;
```

2 architectures d'un même composant

L'instruction after n'est pas synthétisable!

Plan de ce chapitre

- I. Introduction
- II. Types
- III. Unités de conception
- **IV. Simulations évènementielles**
- V. Instructions séquentielles et concurrentes
- VI. Descriptions structurelles et comportementales
- VII. Description de la maquette de test
- VIII. Conclusion

IV. Simulation évènementielle

Signaux

- Ils existent plusieurs types d'objets :
  - constante : la valeur portée ne change pas
  - variable : affectation immédiate de sa valeur
  - **signal : NOUVEAU TYPE D'OBJET**

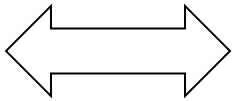
Les signaux permettent de modéliser le parallélisme : en matériel, plusieurs processus peuvent s'exécuter simultanément

IV. Simulation évènementielle

Signaux

- Définition du pilote :
  - **Liste de couples *date-valeur*** (date comptée relativement à *l'heure actuelle* du simulateur)
  - Exemple :

S **<=** '0', '1' **after** 10 ns, '0' **after** 25 ns;



Heure	Valeur
0 (Δ)	'0'
10ns	'1'
25ns	'0'

Pilote de S

Le pilote est un tableau associé à chaque signal dans la mémoire

IV. Simulation évènementielle

Simulation – Générateur d'horloge

signal h : bit;

Zone de déclaration de l'architecture

begin

horloge : **process --** déclaration de processus

-- zone de déclaration du process us

begin -- début de la zone de définition du processus

h <= ‘0’, ‘1’ **after** 75 ns;

wait for 100 ns;

Mise en suspend du *processus* pour une durée déterminée

end process; -- fin du processus

…

IV. Simulation évènementielle

Simulation – Générateur d'horloge

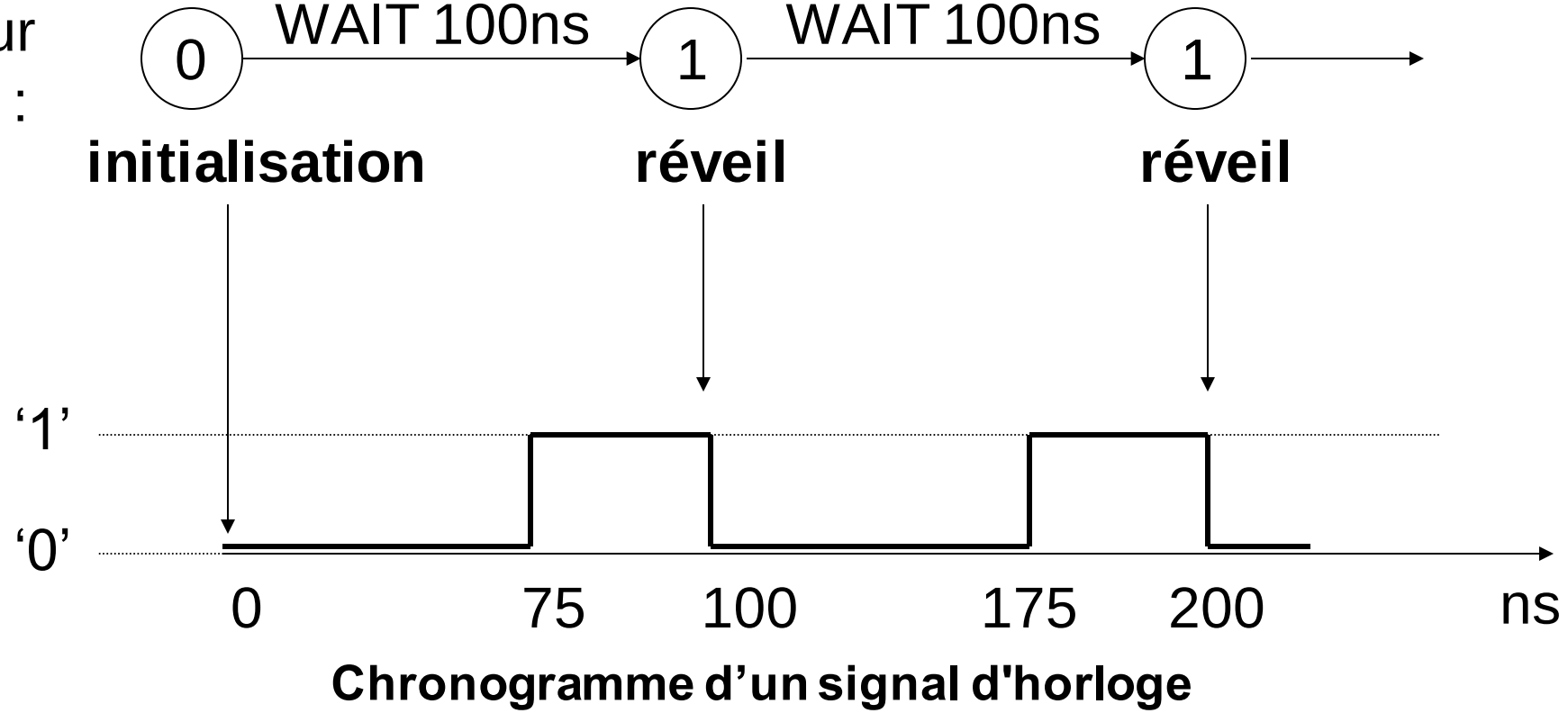
Pilote de h :

T	Etat
0 (Δ)	'0'
75	'1'

T	Etat
0 (Δ)	'0'
75	'1'

T	Etat
0 (Δ)	'0'
75	'1'

Valeur de h :



Chronogramme d'un signal d'horloge

IV. Simulation évènementielle

Signaux

- Contrairement à une variable,

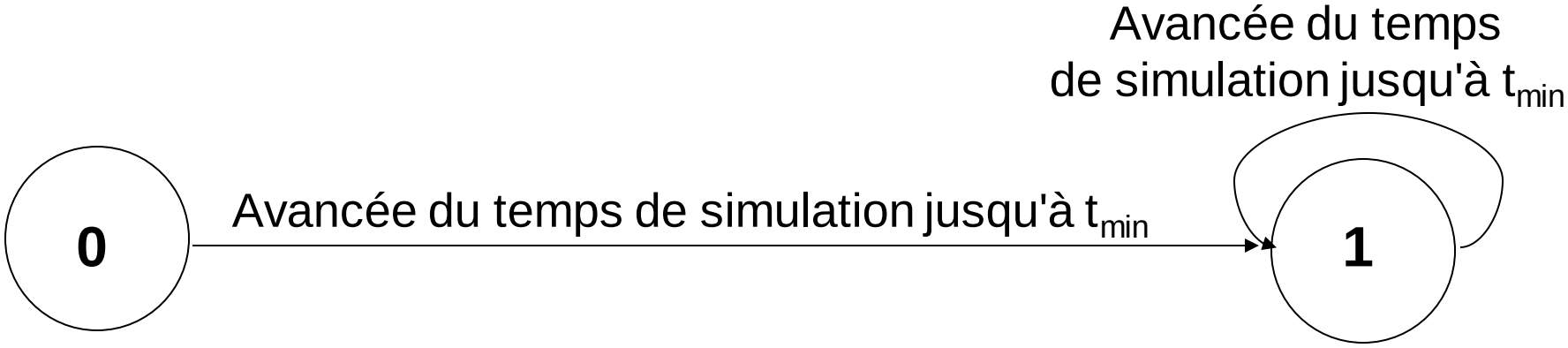
un signal ne prend jamais une valeur directement après son affectation

Mais

uniquement lorsque l'exécution de tous les processus en cours est suspendue (Δ)

IV. Simulation évènementielle

Déroulement d'une simulation



Initialisation de tous les objets à leur valeur initiale, puis **exécution de tous les processus**

Réveil et simulation des processus concernés
Production de **nouveaux événements**
=> nouveau t_{min}

Si t_{min} est égal au temps t_0
alors il y a itération (ou *délai Delta)*

IV. Simulation évènementielle

Processus

- Instruction **WAIT**
  - Synchronise (en les suspendant/réveillant) les processus
  - Plusieurs formes :
    - **WAIT FOR** *durée*;
    - **WAIT ON** *liste de signaux*;
    - **...**

Non synthétisable!

IV. Simulation évènementielle

Processus

- Equivalence entre *liste de sensibilité* et instruction **WAIT ON**

Liste de sensibilité

```
proc1: process (a, b, c)
       begin
         x<=a and b and c;
       end process;
```

⇔

```
proc2: process
       begin
          x<=a and b and c;
          wait on a, b, c;
       end process;
```

IV. Simulation évènementielle

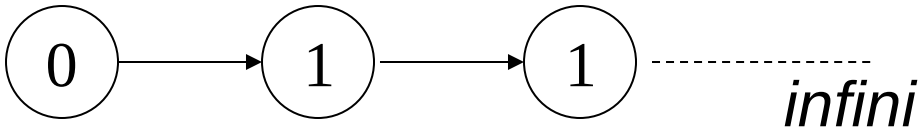
Processus

L’ordre des processus dans un programme n'a pas d'importance

IV. Simulation évènementielle

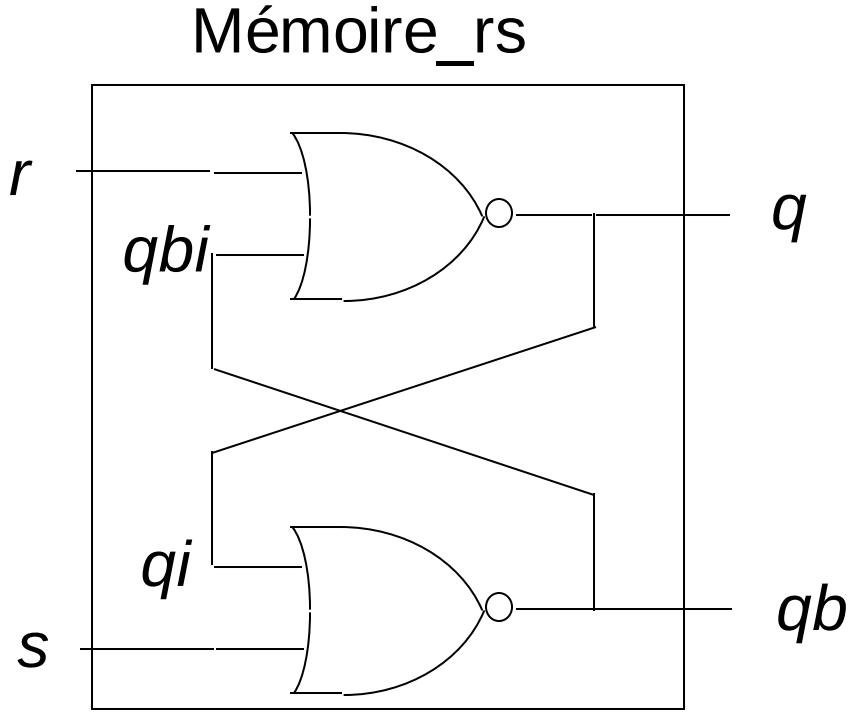
Processus

- Remarque :
  - Il est syntaxiquement possible d'avoir un process sans *liste de sensibilité* ni " **wait** »,
  - Cependant, le temps de simulation n'avance pas car ce process reboucle indéfiniment sur lui même (n'est jamais suspendu)



IV. Simulation évènementielle

Simulation – Bascule RS



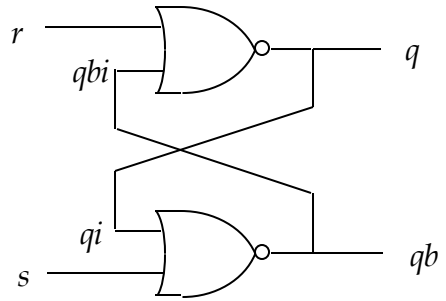
Bascule RS

IV. Simulation évènementielle

Simulation – Bascule RS

Table de vérité de la mémoire RS :

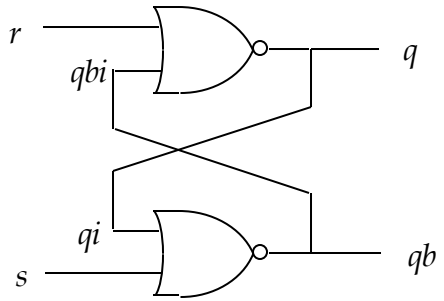
s	**r**	**q**	**qb**	
1	0	1	0	Mise à "1"
0	1	0	1	Mise à "0"
0	0	1	0	Sortie inchangée (après sr = 10)
0	0	0	1	Sortie inchangée (après sr = 01)
1	1	?	?	**À ne pas faire!**



IV. Simulation évènementielle

Simulation – Bascule RS

```
1 entity memoire_rs is
2       port ( s, r :    in     bit;
3                    q, qb : out bit);
4 end;
5
6 architecture processus of memoire_rs is
7
8       constant tplh : time := 2 ns;
9       constant tphl : time := 1 ns;
10      signal qi : bit := '0';
11      signal qbi : bit := '1';
12 begin
13
```



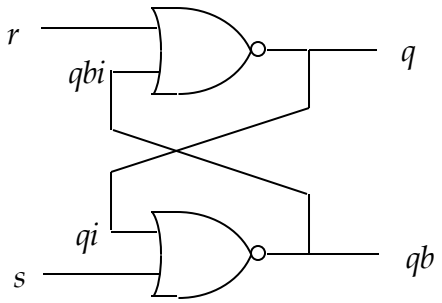
IV. Simulation évènementielle

Simulation – Bascule RS

```
14 n1 : process
15      variable qtmp : bit;
16      begin
17              wait on s, qi ;
18              qtmp := s nor qi; -- la primitive nor
19              if qtmp /= qbi then -- test du changement ?
20                      if qtmp = '0' then
21                              qbi <= qtmp after tphl;
22                      else
23                              qbi <= qtmp after tplh;
24                      end if;
25              end if;
26      end process;
```

déclaration

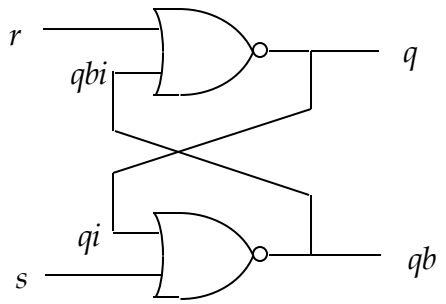
commentaires



IV. Simulation évènementielle

Simulation – Bascule RS

```
28 n2 : process
29      variable qtmp : bit;
30      begin
31              wait on r, qbi ;
32              qtmp := r nor qbi; -- la primitive nor
33              if qtmp /= qi then -- test du changement?
34                      if qtmp = '0' then
35                              qi <= qtmp after tphl;
36                      else
37                              qi <= qtmp after tplh;
38                      end if;
39              end if;
40      end process;
```



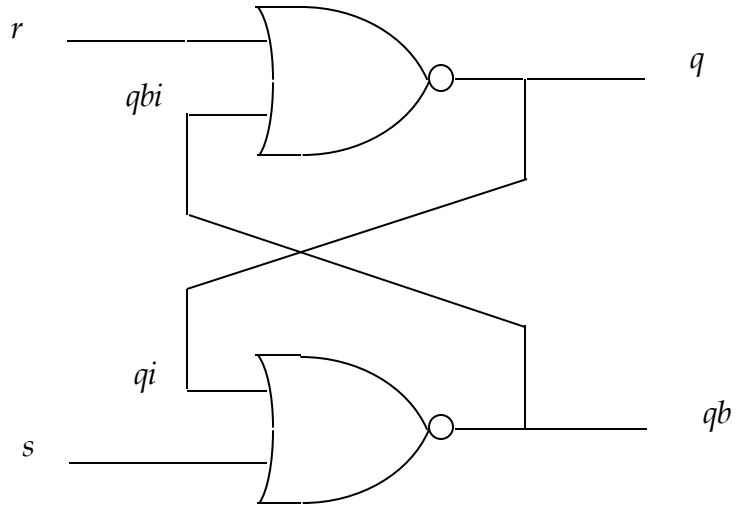
IV. Simulation évènementielle

Simulation – Bascule RS

41 q <= qi;

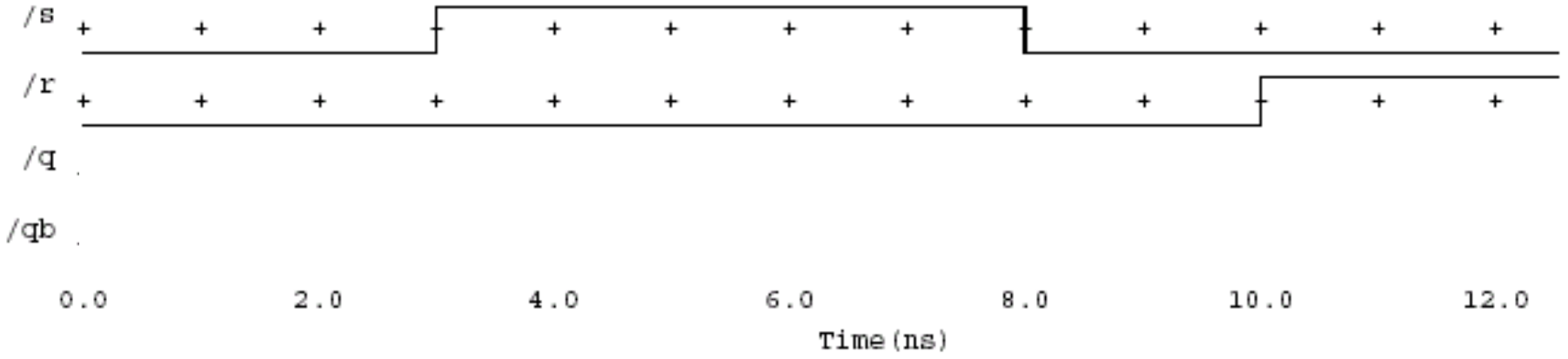
42 qb <= qbi;

43 **end**;



IV. Simulation évènementielle

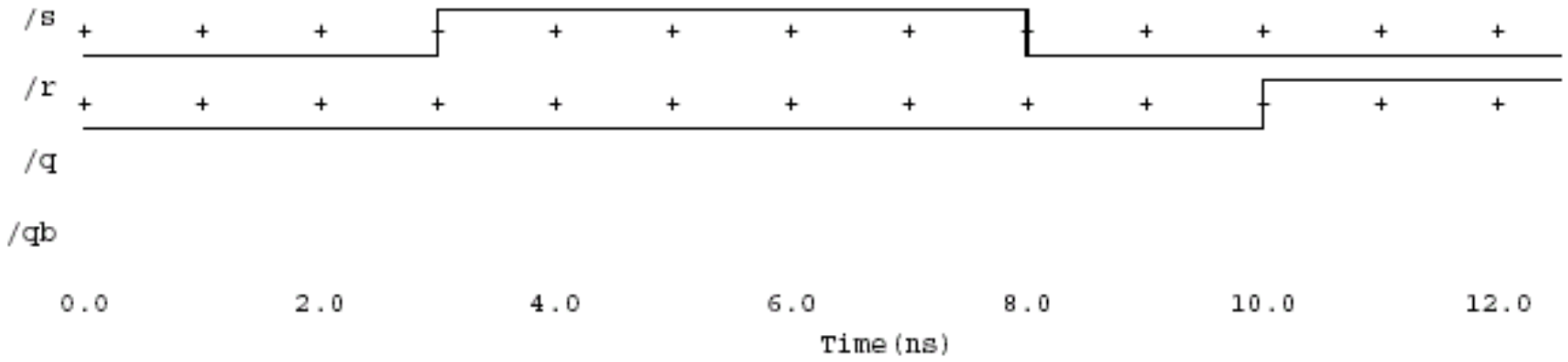
Simulation – Bascule RS



Chronogramme de la bascule RS à compléter

IV. Simulation évènementielle

Simulation – Bascule RS - exercice



A continuer…

V. Instructions concurrentes

Exercice

- Modéliser, avec une unique instruction concurrente, un signal d'horloge *clk* de période *10 ns*.
- Solution :

IV. Simulation évènementielle

Synthèse

- ***AFTER et WAIT FOR sont interdits pour la synthèse***
- ***La plupart des outils de synthèse ne se servent pas de la liste de sensibilité; ils font l'hypothèse que tout " signal opérande " appartient à la liste de sensibilité***

```
PROC3: PROCESS (A, B, C)
    BEGIN
     X<= A AND B AND C;
    END PROCESS;
```

=

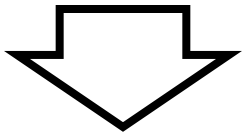
```
PROC4: PROCESS (A, B)
    BEGIN
     X<= A AND B AND C;
    END PROCESS;
```

IV. Simulation évènementielle

Remarque

Dans la majorité des cas,

Plusieurs composants ne pilotent pas le même signal



En VHDL, chaque signal doit être affecté à l'intérieur d'un seul processus

Plan de ce chapitre

- I. Introduction
- II. Types
- III. Unités de conception
- IV. Simulations évènementielles
- **V. Instructions séquentielles et concurrentes**
- VI. Descriptions structurelles et comportementales
- VII. Description de la maquette de test
- VIII. Conclusion

V. Instructions séquentielles

Affection des signaux et variables

- Une variable ne peut exister que dans un contexte séquentiel (= dans un process)
  - **Affectation immédiate**

    X **:=** 1+2**;**
    X prend immédiatement la valeur 3 (sans pilote)

- Un signal peut être affecté hors process ou dans un process
  - **Dans les deux cas, l'affectation a lieu dès que tous les process sont suspendus**

V. Instructions séquentielles

Exemples d'affectations

```
entity var_sig is
end;

architecture exercice of var_sig is
        signal aa, aaa : integer := 3;
        signal bb, bbb : integer := 2;
```

V. Instructions séquentielles

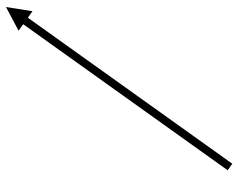
Exemples d'affectations

```
begin
        p1: process
                variable a: integer := 7;
                variable b: integer := 6;
                begin
                        wait for 10 ns;
                        a := 1;          --- a est égal à 1
                        b := a + 8 ;     --- b est égal à 9
                        a := b - 2 ;     --- a est égal à 7
                        aa <= a;         -- 7 dans pilote de aa
                        bb <= b ;        -- 9 dans pilote de bb
                end process;
```

V. Instructions séquentielles

Exemples d'affectations

```
p2: process
        begin
                wait for 10 ns;
                aaa <= 1 ;              -- 1 dans pilote de aaa
                bbb <= aaa + 8;         -- 11 dans pilote de bbb
                aaa <= bbb - 2;         -- 0 dans pilote de aaa
        end process;
end;
```



Seule la dernière affectation compte!

V. Instructions séquentielles

Synthèse

- **Eviter d'employer des variables, on leur préfèrera les signaux**
- Les variables sont principalement utilisées pour contenir les index des boucles

V. Instructions …

Définitions

- Définition d'une instruction séquentielle :
  - **Instruction à l'intérieur d'un process**
- Définition d'une instruction concurrente :
  - **Instruction à l'extérieur des process**
  - **la position des instructions concurrentes dans le programme n'a pas d'influence**
- Rappel : Les instructions se placent toujours uniquement entre le **begin** et le **end** de l'architecture

V. Instructions séquentielles

Équivalence séquentiel/concurrent

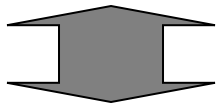
- Toute instruction concurrente peut être décrite grâce à une instruction séquentielle équivalente "utilisant un processus"

V. Instructions séquentielles

Affectation séquentielle simple

```
p1: process(a,b)
begin
        s <= a and b after 10 ns;
end process;
```

(Ou possibilité d'utiliser la liste de sensibilité du processus…)



```
s  <= a and b after 10 ns;
```

Liste de sensibilité implicite

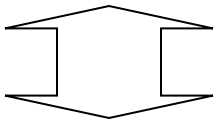
V. Instructions séquentielles

Affectation séquentielle conditionnelle

```
p2: process(etat)
begin
        if etat = "1001" then
                neuf <= ´1´ ;
        else
                neuf <= ´0´ ;
        end if;
end process;
```

Permet de réaliser la synthèse de portes logiques combinatoires

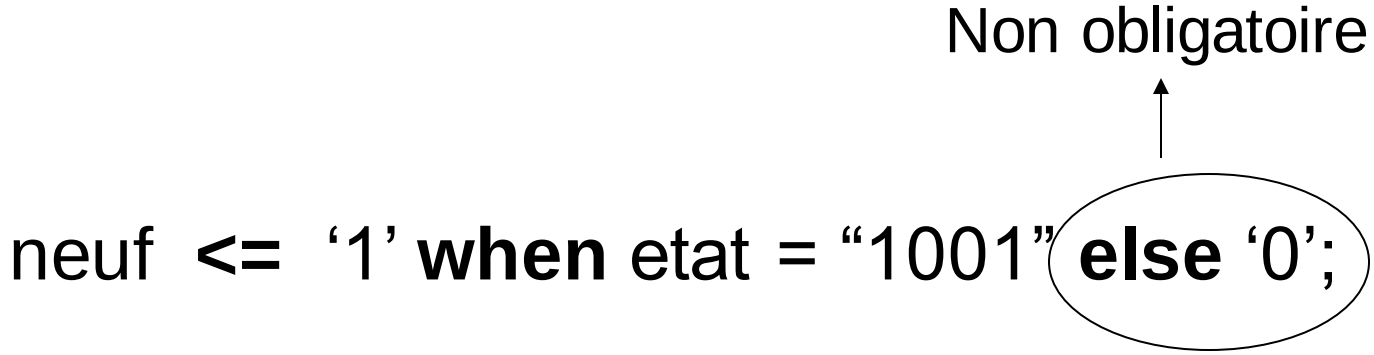
(si toutes les sorties sont toujours affectées sinon création de mémoire cf. mémorisation implicite)



…

V. Instructions séquentielles

Équivalence séquentiel/concurrent



Dans les deux cas (if et when):

- ***Non exclusivité des conditions MAIS ordre de priorités!***
- ***Attention à la logique inutile lors de la synthèse***

V. Instructions séquentielles

Affectation sélective séquentielle

Exclusivité des conditions

MAIS

Pas d'ordre de priorités!

```
1 p3: process
2       begin
3             wait on e0, e1, e2, e3, ad;
4             case ad is
5                   when "00" => s <= e0 ;
6                   when "01" => s <= e1 ;
7                   when "10" => s <= e2 ;
8                   when others => s <= e3 ;
9             end case;
10      end process;
```

Si pas d'instruction alors pas de ;

…

V. Instructions séquentielles

Équivalence séquentiel/concurrent

```
-- Affectation concurrente sélective
   signal e0, e1, e2, e3, s : bit;
   signal ad : bit_vector( 1 downto 0);
begin
   with ad select
        s <=  e0 when "00",
              e1 when "01",
              e2 when "10",
              e3 when others;
end;
```

Toutes les valeurs doivent être listées (de manière exclusive)

Pas de priorité

Structure de type Multiplexeurs

Obligatoire quand toutes les conditions n'ont pas été couvertes

V. Instructions séquentielles

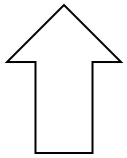
Boucles

```
while i<10 loop
   i := i + 1;
   …
end loop;
```

Il faut déclarer la variable *i*

```
for i in 10 downto 1 loop
   -- instructions utilisant i
   …
end loop;
```

Il ne faut pas déclarer la variable *i*



Seule boucle utilisée en synthèse

V. Instructions séquentielles

Exemples

- Exemple d ’erreur archi-classique :

- Que veut-on faire ici?
- Pourquoi cela ne fonctionne pas?
- Que faut-il faire pour corriger cette erreur (sans variable!)?

```
entity mon_xor is
port(    a_bus : in bit_vector(7 downto 0);
         x : out bit);
end mon_xor;
architecture fonctionne_pas of mon_xor is
begin
p1:process (a_bus)
         begin
                  x <= a_bus(0) xor a_bus(1);
                  for i in 1 to 6 loop
                           x <= a_bus(i+1) xor x;
                  end loop;
         end process;
end fonctionne_pas;
```

V. Instructions séquentielles

Exemples

Solution :

```
architecture fonctionne of mon_xor is
signal temp : bit_vector(7 downto 1);
begin
p1:process (a_bus, temp)
begin
        temp(1) <= a_bus(0) xor a_bus(1);
        for i in 1 to 6 loop
                temp(i+1) <= a_bus(i+1) xor temp(i);
        end loop;
end process;
x <= temp(7);
end fonctionne;
```

V. Instructions séquentielles

Précédence

- Remarque : il n'y a pas d'ordre de ***précédence*** entre les opérateurs logiques élémentaires :
  - il faut obligatoirement des parenthèses s'il y a un doute… sinon cela conduit à une erreur de compilation

~~***X <= A or B and C***~~

X <= (A or B) and C

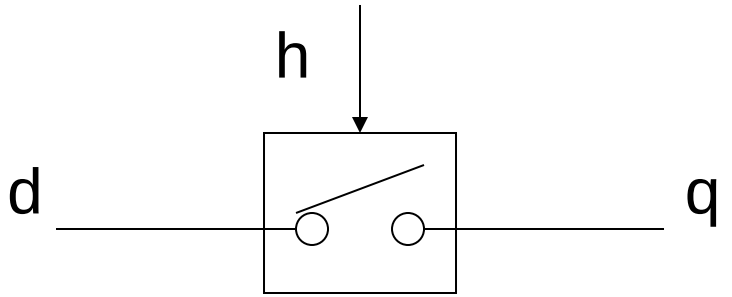
V. Instructions séquentielles

Modélisation d'éléments de logique synchrone

```
process(h,d)
        begin
                if h= ´1´ then
                        q <= d ;
                end if;
-- mémorisation implicite lorsque la condition est fausse
        end process;
```

De quel composant s'agit-il?

Modélisation d'une *Latch* (*bascule verrou*)



V. Instructions séquentielles

Modélisation d'éléments de logique synchrone

```
p2: process(h)
begin
    if h´event and h = ´1´ then
        if raz = ´1´ then
                q <= ´0´;
        else
                q <= d;
        end if;
end if;
-- raz prioritaire
end process;
```

De quel composant s'agit-il?

Modélisation d'une Bascule D FF avec RAZ synchrone

V. Instructions séquentielles

Modélisation d'éléments de logique synchrone

```
P3: process(h, raz)
begin
        if raz = ´1´ then -- raz prioritaire
                q <= ´0´;
        elsif h´event and h = ´1´ then
                q <= d;
        end if;
-- mémorisation implicite lorsque
-- la condition est fausse
end process
```

De quel composant s'agit-il?

Modélisation d'une Bascule D FF
avec **RAZ asynchrone**

V. Applications

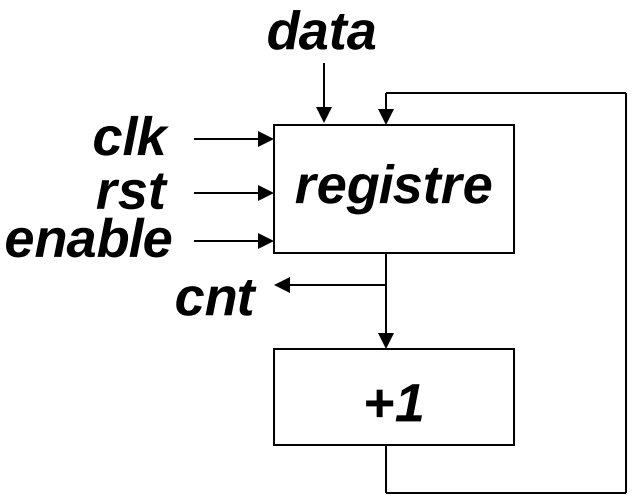
Synthèse de compteurs – à chargement parallèle

Compléter avec les fonctions de conversion adéquates

```
entity cnt8 is port(
      clk, rst:          in std_logic;
      enable, load:      in std_logic;
      data:              in std_logic_vector(7 downto 0);
      cnt:               inout std_logic_vector(7 downto 0));
end cnt8;
```

```
architecture archcnt8 of cnt8 is
begin
count: process (rst, clk)
  begin
      if rst = '1' then
            cnt <= "00000000";
      elsif (clk'event and clk='1') then
            if load = '1' then
                  cnt <= data;
            elsif enable = '1' then
                  cnt <= cnt + 1;
            end if;
      end if;
  end process count;
end archcnt8;
```

8 bascules



V. Applications

Synthèse de compteurs – Solution

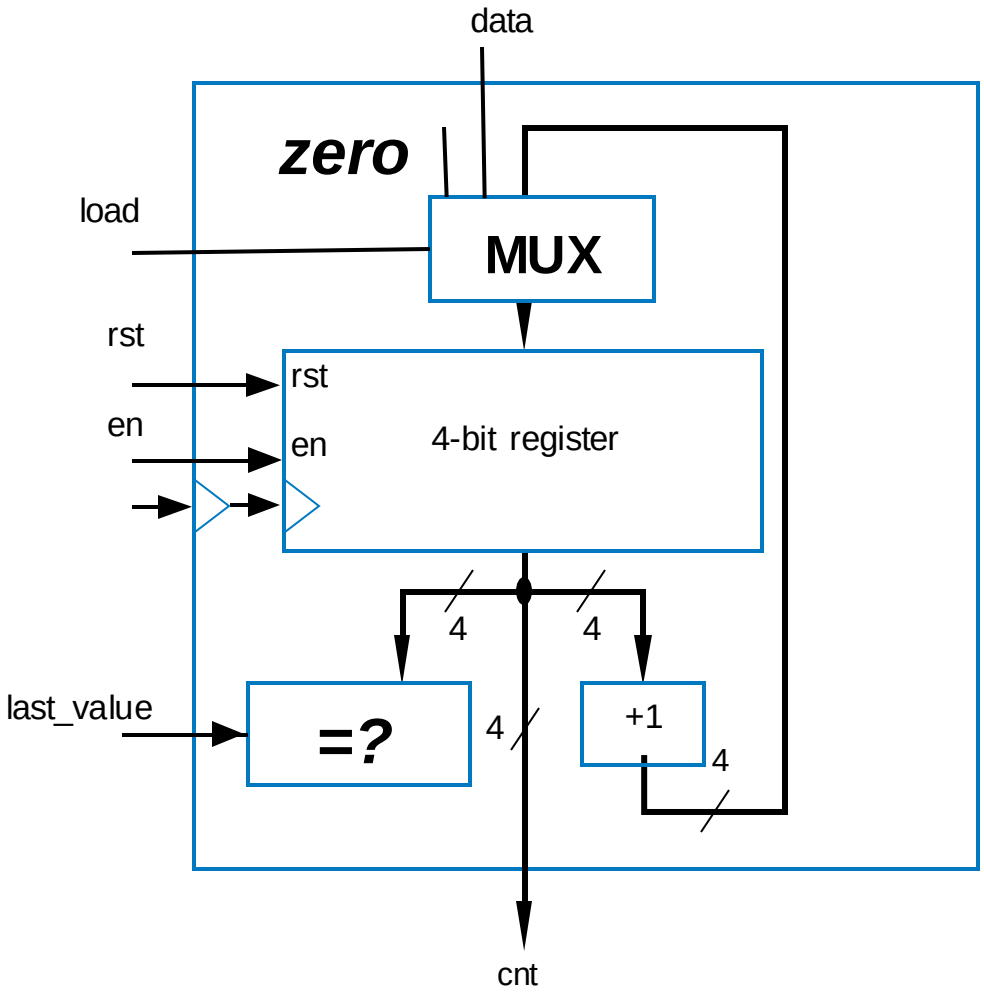
V. Applications

Synthèse de compteurs – "Modulo bizarre"

```
if rst = '1' then
        cnt <= zero;
elsif (clk'event and clk='1') then
        if load = '1' then
                cnt <= data;
        elsif enable = '1' then
          if cnt = last_value then
            cnt <= zero;
          else
            cnt <= cnt + 1;
          end if;
        end if;
 end if;
```

Faire un schéma avec des macro-composants de cette fonction

4-bit modulo counter



V. Applications

Synthèse de compteurs

Pour les compteurs de petites tailles : Compteur de Johnson

```
signal div_32: std_logic_vector (15 downto 0) := (others => '0');
begin
  process (CLK)
  begin
    if CLK'event and CLK='1' then
      div_32 <= div_32(14 downto 0) & not div_32(15);
    end if;
  end process;
```

Dessiner un schéma équivalent au niveau porte et expliquer son fonctionnement

Plan de ce chapitre

- I. Introduction
- II. Types
- III. Unités de conception
- IV. Simulations évènementielles
- V. Instructions séquentielles et concurrentes
- **VI. Descriptions structurelles et comportementales**
- VII. Description de la maquette de test
- VIII. Conclusion

VI. Description structurelle

Différence entre comportemental et structurel

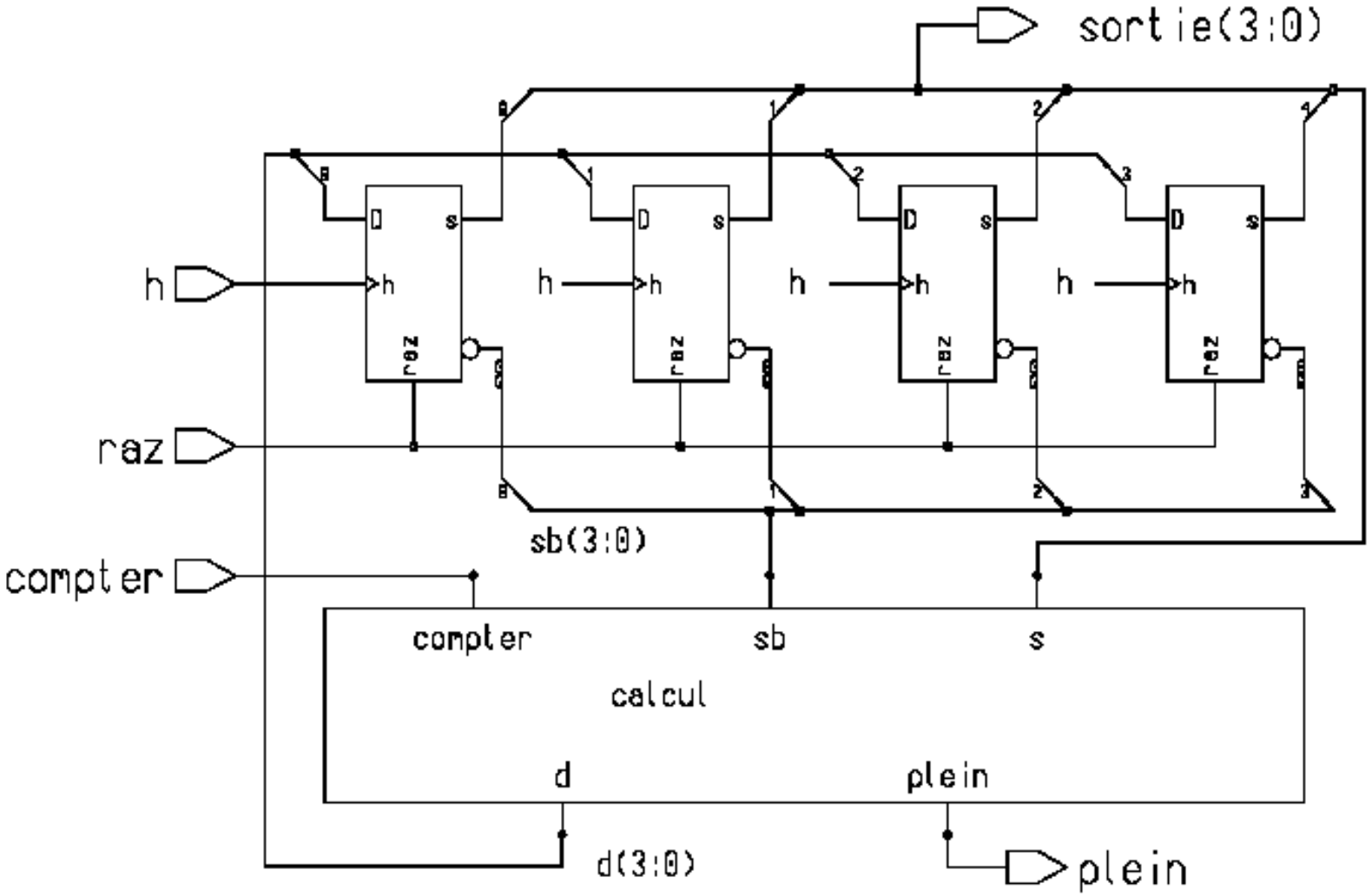
Description structurelle
Comment est-ce que
la fonction est réalisée

$\not\equiv$

Description comportementale
Décrit ce que
la fonction réalise

VI. Description structurelle

Exemple d'un compteur sur 4 bits modulo 10

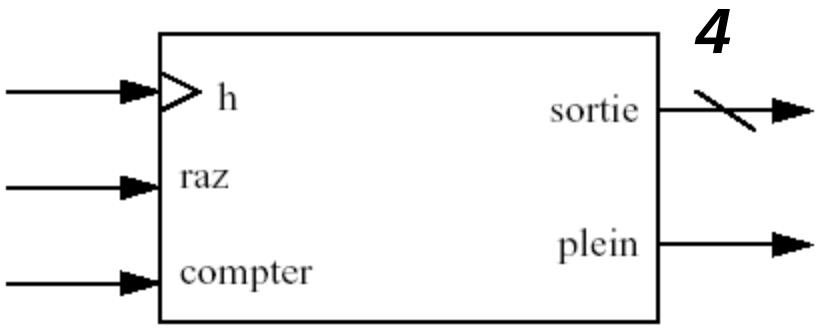


VI. Description structurelle

Exemple

- **Compteur 4 bits synchrone avec autorisation de comptage**

```
1 entity compteur4 is
2       port ( h, raz, compter : in bit;
3                   sortie : out bit_vector( 3 downto 0);
4                   plein : out bit);
5 end;
```



VI. Description structurelle

Exemple – déclaration 1 & 2

```
6   architecture structure1 of compteur4 is
7   signal d, s , sb : bit_vector( 3 downto 0);

10  component bascule
11      port (h, d, raz : in bit;
12            s, sb : out bit);
13  end component;

15  component calcul
16      port ( s, sb : in bit_vector( 3 downto 0);
17            compter : in bit;
18            d : out bit_vector( 3 downto 0);
19            plein : out bit);
20  end component;
```

Déclaration des composants

VI. Description structurelle

Mise en œuvre: déclaration

- **Déclarer tous les composants nécessaires**
  - Dans l'exemple suivant, ligne 10 à 20
- **Déclarer les signaux internes**
  - Ligne 7

VI. Description structurelle

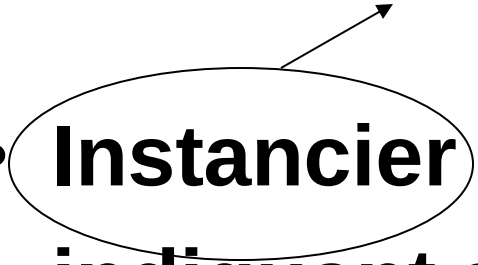
Exemple – instanciation

```
24 begin
25      ba : bascule -- instanciation par position
26      port map ( h, d(3), raz, s(3), sb(3));
27      bb : bascule -- instanciation par dénomination
28      port map ( h => h, d => d(2), raz => raz, s => s(2), sb => sb(2));
29      bc : bascule -- instanciation par position et dénomination
30      port map ( h, d(1), sb => sb(1), s => s(1), raz => raz);
31      bd : bascule -- instanciation par dénomination
32      port map ( sb => sb(0), s => s(0), h => h, d => d(0), raz => raz);

33      combi : calcul
34      port map ( s, sb, compter, d, plein);

35      sortie <= s;
36 end structure1;
```

VI. Description structurelle

Mise en œuvre : instanciation

Créer une instance reposant sur un modèle de composant



- **Instancier chaque composant en indiquant sa liste de connexions**
  - Ligne 26 et suivantes

VI. Description structurelle

- **Description des inter-connexions entre composants**
  - Contient un ou plusieurs composants (**COMPONENT**)

Annexe :

Description du composant ‘Calcul’

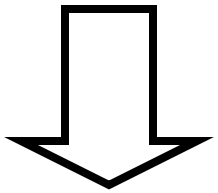
```
1 entity calcul is
2           port (      s, sb       : in bit_vector( 3 downto 0);
3                       compter     : in bit;
4                       d           : out bit_vector( 3 downto 0);
5                       plein       : out bit);
6 end;
7
14 architecture par_10 of calcul is
15          signal pas_compter : bit;
17 begin
18          pas_compter <= not compter;
19          d(3) <= (compter and s(2) and s(1) and s(0))
20          or (s(3) and (sb(0) or pas_compter)) ;
21          d(2) <= (compter and sb(2) and s(1) and s(0))
22          or ( s(2) and (pas_compter or sb(1) or sb(0)));
23          d(1) <= (compter and sb(3) and sb(1) and s(0))
24           or ( s(1) and (pas_compter or sb(0)));
25          d(0) <= compter xor s(0);
26
27          plein <= s(3) and s(0);
28 end ;
```

Description comportementale
(sans instanciation)

VI. Description structurelle

Configuration d'un projet

- **Lorsque plusieurs architectures d'une même entité existent, que se passe t-il?**



- Sans spécification explicite de la configuration, c'est la dernière architecture analysée dans la bibliothèque WORK qui est utilisée.

⇨ **Comportement aléatoire à éviter!**

VI. Description structurelle

Configuration d'un projet

L'*instanciation directe* permet de :
- préciser l'architecture utilisée
- éviter de déclarer préalablement le composant

=> on remplace les lignes 33-34 par :

```
33      combi : entity calcul(par_10)
34      port map ( s, sb, compter, d, plein);
```

" par_10 »

nom de l'architecture à utiliser

VI. Description structurelle

Generate

Dans l'exemple précédent, on peut remplacer les lignes de 25 à 32 par:

```
28        implant : for i in 0 to 3 generate
29                  b : bascule
30                  port map (h, d(i), raz, s(i), sb(i));
31        end generate;
```

generate permet de réaliser des instanciations multiples

Plan de ce chapitre

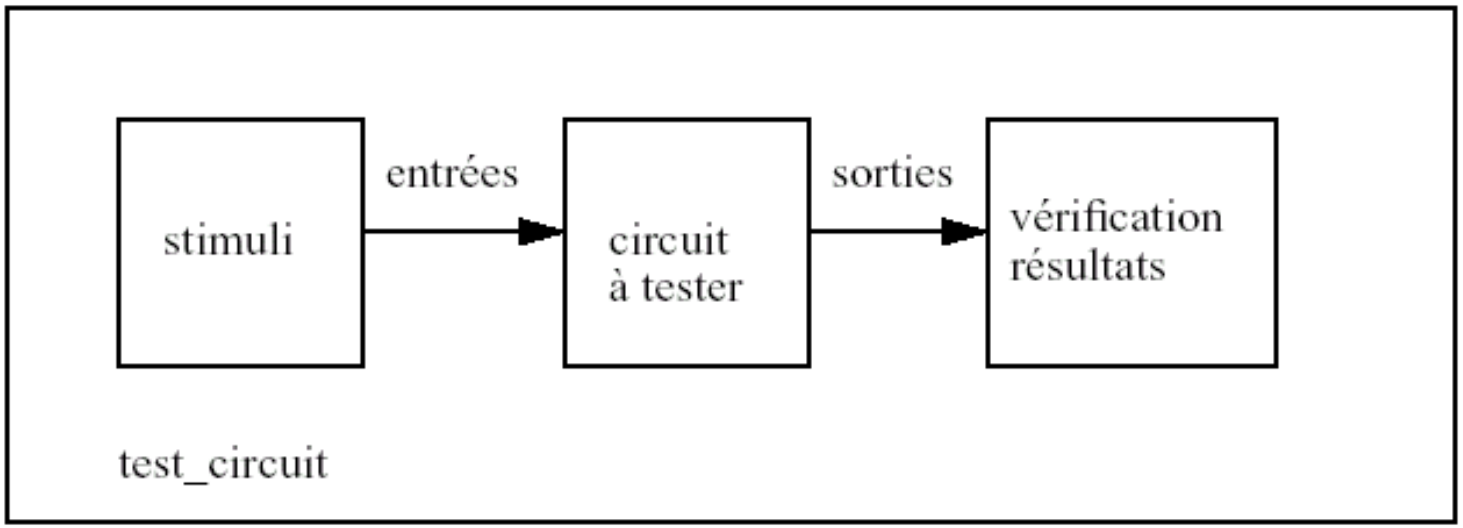
- I. Introduction
- II. Types
- III. Unités de conception
- IV. Simulations évènementielles
- V. Instructions séquentielles et concurrentes
- VI. Descriptions structurelles et comportementales
- **VII. Description de la maquette de test**
- VIII. Conclusion

VII. Description du test

Testbench : " banc de test »

Comment simuler les programmes réalisés?

⇨ En utilisant un **testbench**



Testbench = entité dénuée d'entrées/sorties

VII. Description du test

Testbench

```
entity test_compteur4 is
end;
```

→ Aucun port d'entrée/sortie

```
architecture tb of test_compteur4 is
-- déclaration des signaux utiles pour les connexions
signal h, raz, compter , plein : bit;
signal sortie : bit_vector(3 downto 0);
begin
-- instanciation directe
        c1: entity work.compteur4(structure1)
                port map(h, raz, compter, sortie, plein);

        h <= not(h) after 10 ns;

        compter <= '0', '1' after 100 ns;

        raz <= '1', '0' after 200 ns, '1' after 400 ns, '0' after 500 ns;
end;
```

VII. Description du test

Paramètres génériques

- Dans la *spécification de l'entité,* outre les ports de connexion, on peut également **préciser des paramètres "à passer" lors de l'instanciation du composant** : ce sont les paramètres génériques

Ces paramètres sont très utiles car ils permettent de préciser le comportement d'un composant sans modifier son architecture

VII. Description du test

Paramètres génériques

-- spécification de l'entité

entity source **is**

generic (k : **integer :=**3**);** → k est un *paramètre générique*

port (a : **in integer**; b **out integer);**

end;

-- instanciation d'un composant *source*

l1: source

generic map (k=>2)

port map (entrée, sortie);

⚠ ***Pas de ;***

VII. Description du test

Paramètres génériques - exemple

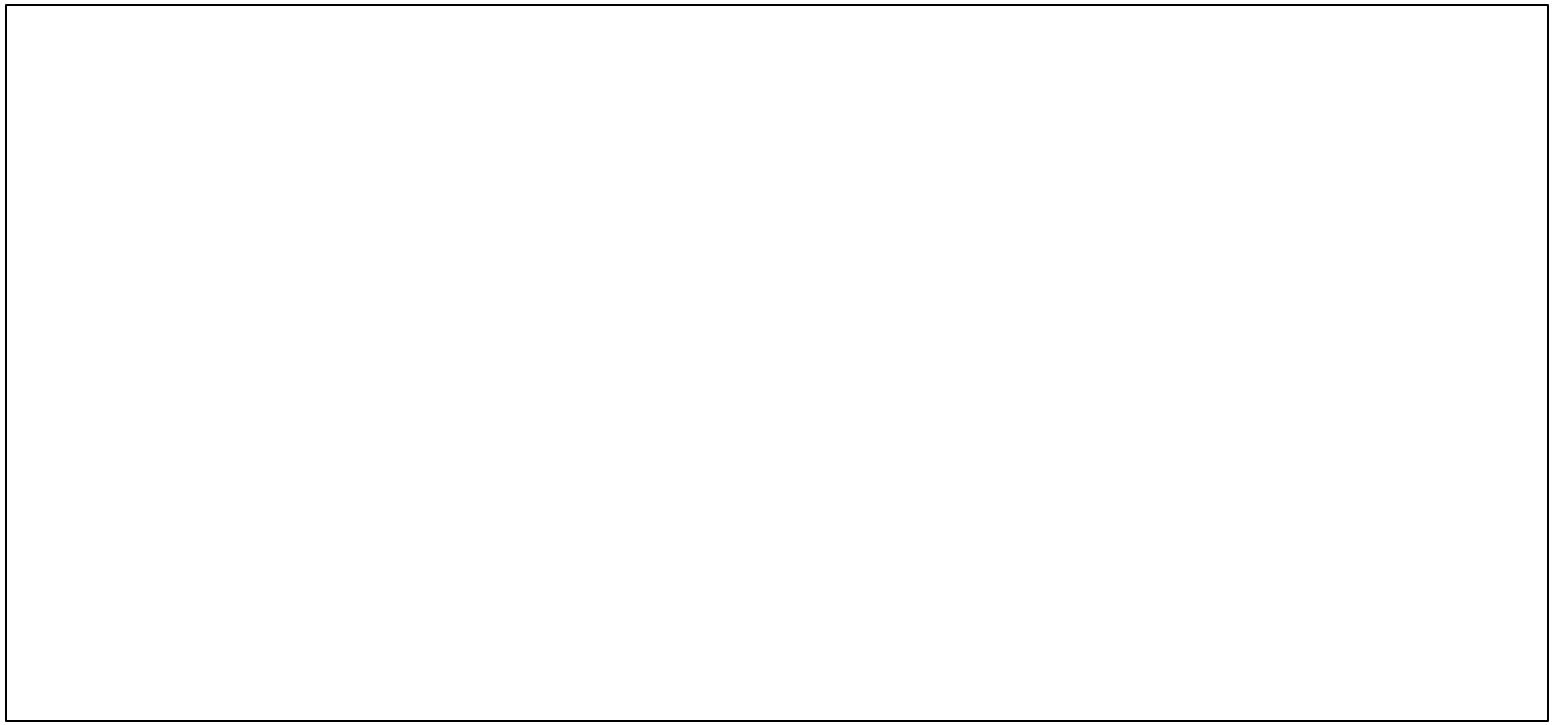
```
library IEEE;
use IEEE.std_logic_1164.all;

entity Nbit_adder is
generic(N : integer := 4);
port (
   a,b : in std_logic_vector(N-1 downto 0);
   ci : in std_logic;
   sum : out std_logic_vector(N-1 downto 0);
   co : out std_logic);
end entity;

architecture arc of Nbit_adder is

component FA
port(  A,B,Cin : in std_logic;
       S, Cout : out std_logic);
end component;

signal c :std_logic_vector(N downto 0);
```

Décrire un additionneur binaire générique utilisant N

VI. Description structurelle

Exercice

Développer une description structurelle d'un registre N bits avec une entrée série DIN et une sortie parallèle DOUT basé sur des composants FLIPFLOP



- Sortie : Contenu du registre **DOUT** (N bits).

Plan de ce chapitre

- I. Introduction
- II. Types
- III. Unités de conception
- IV. Simulations évènementielles
- V. Instructions séquentielles et concurrentes
- VI. Descriptions structurelles et comportementales
- VII. Description de la maquette de test
- **VIII. Conclusion**

VIII. Conclusion

- Une bonne description est une **description lisible**
  - **Structurer**
  - **Indenter et commenter vos programmes**
  - **Séparer les parties synchrones des parties combinatoires**
    - **Utiliser des process explicites pour décrire les parties synchrones**
    - **Utiliser des instructions concurrentes pour décrire les parties combinatoires**

VIII. Conclusion

- ***Une manière de savoir si votre code est synthétisable est d'essayer de trouver les équations logiques manuellement. Si vous n'y parvenez pas, alors c'est que votre description est trop abstraite…***