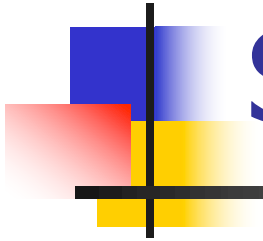
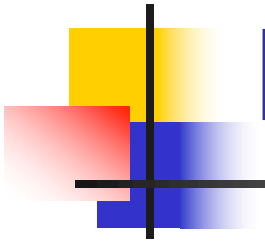


Synthèse Logique

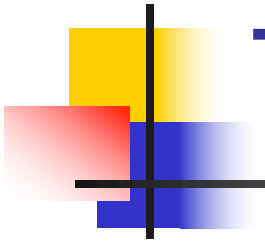
Chapitre 2

Les Bases du VHDL pour la Synthèse



Plan

- **Types**
- Unités de conception
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion

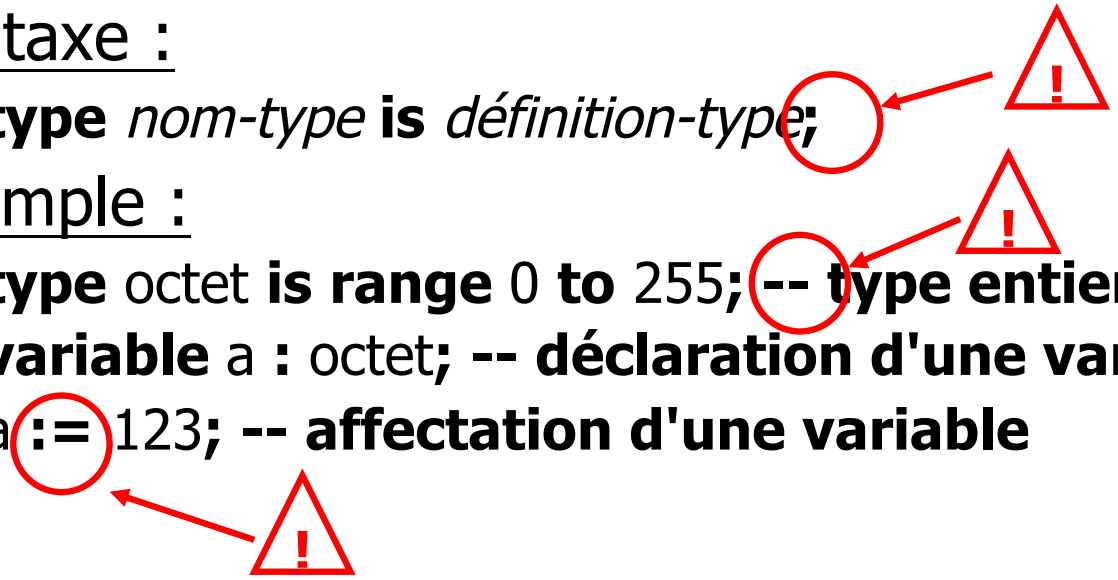


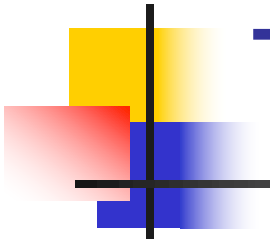
Types

- En VHDL, MAJUSCULES et minuscule sont confondues
- VHDL est un langage fortement typé
  - Tout objet (variable, signal, constante…) manipulé a un format prédéfini. Uniquement des valeurs de ce format peuvent être affectées à cet objet
- Il existe plusieurs catégories de type dont
  - *Scalaires* (numériques et énumérés)
  - *Composés* (tableaux et vecteurs)

Types

- Existence de types prédéfinis
  - type scalaire ⇒ bit, boolean, integer, real…
  - type composé ⇒ bit_vector…
- Type scalaire
  - possibilité de définir de nouveaux types
  - Syntaxe :
    - **type** *nom-type* **is** *définition-type*;
  - Exemple :
    - **type** octet **is range** 0 **to** 255; **-- type entier**
    - **variable** a **:** octet**; -- déclaration d'une variable de type octet**
    - a **:=** 123**; -- affectation d'une variable**





Types

- Types scalaires
  - Types énumérés : liste de valeurs

    type bit **is** (‘0’, ‘1’)**;**

    type boolean **is** (false, true)**;**
    - Valeur d'initialisation à gauche dans la liste
  - Types *numériques* : domaine de définition
    - Domaine de définition : **range**, **to** ou **downto**

    type valim **is range** 0.0 **to** 5.0**;**

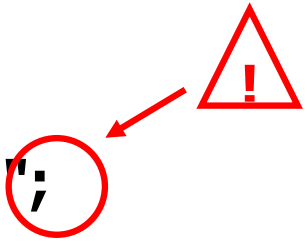
Borne de type flottant

Types

- Types composés
  - collections d'éléments **de même type repérés** par des valeurs d'indices

Exemple 1 :

type word **is array (**0 **to** 7**) of** bit**;**
constant mot1 **: word :="**00000000**";**



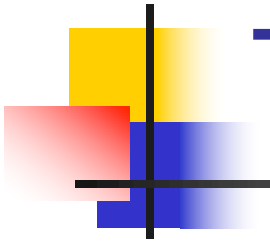
Exemple 2 :

- définition d'un type "vecteur non contraint" :

type bit_vector **is array** (natural **range <>**) **of** bit**;**

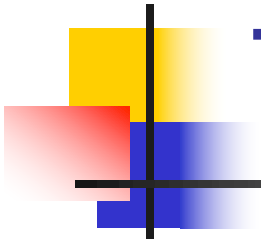
- déclaration d'un vecteur de bit de largeur 32 :

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



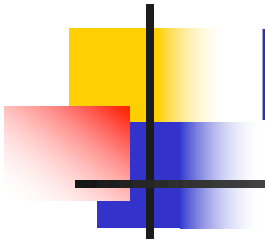
Types

- Exercices
  - Définir un type *etat* composé de la liste des valeurs suivantes : idle, data, jam, nosfb, error
  - Définir le type *table8x4* d'un tableau à 2 dimensions avec des indices allant de 0 à 7 et de 0 à 3, contenant des bits
  - Déclarer et initialiser une variable *exclusive_or* de type *table8x4* contenant des valeurs telles que les trois premiers bits correspondent aux entrées d'un xor et le dernier bit correspond à la valeur retournée par le xor



Types

```
type etat is (‘idle’, ‘preamble’, ‘data’, ‘jam’, ‘nosfd’, ‘error’);

type table8x4 is array (0 to 7, 0 to 3) of bit;

variable exclusive_or : table8x4 := (
                        "0000",
                        "0011",
                        "0101",
                        "0110",
                        "1001",
                        "1010",
                        "1100",
                        "1111");
```



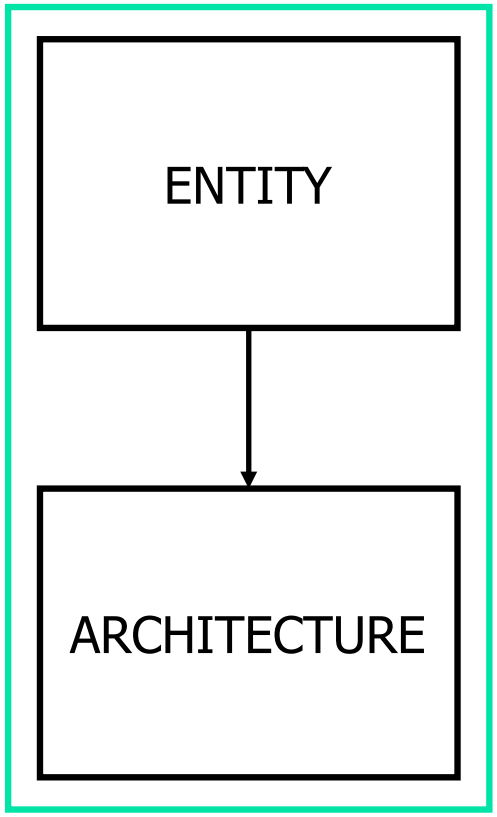
Plan

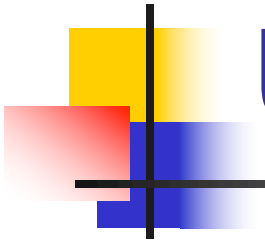
- Types
- **Unités de conception**
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion

Unités de Conception

Unités primaires :
Déclaration des interfaces

Unités secondaires :
Définition fonctionnelles





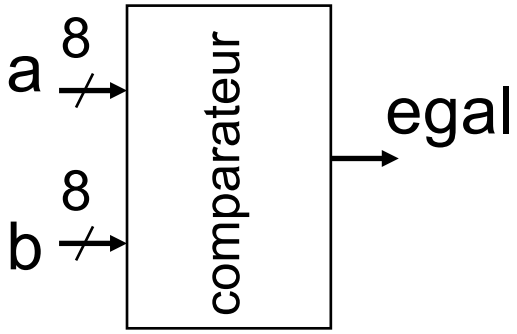
Unités de Conception

- Unités de conception primaires ⇒ ENTITY
- Définition
  - ENTITY (entité)
  - Partie de programme qui peut être compilée séparément
    - Fichier texte contenant une ou plusieurs unités de conception primaires ou secondaires
  - ENTITY ⇒ Vue externe d'un composant
- Spécification d'entité
  - Ports d'entrées / sorties
  - Type
  - Mode : entrée (in), sortie (out), entrée/sortie (inout)

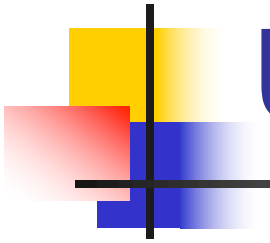
Unités de Conception

- Exemple d'entité

```
entity comparateur is
port (
      a : in bit_vector(7 downto 0);
      b : in bit_vector(7 downto 0);
      egal : out bit);
end ;
```



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



Unités de Conception

- 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*

```
entity add4 is
port (
        a, b    : in    bit_vector(3 downto 0);
        ci      : in    bit;
        sum     : out   bit_vector(3 downto 0);
        co      : out   bit);
end add4;
```

Unités de Conception

- Unités de conception secondaires : ARCHITECTURE
- Toute architecture est relative à une entité
- Définition de l'architecture :
  - L'architecture définit les fonctionnalités et les relations temporelles d'un composant

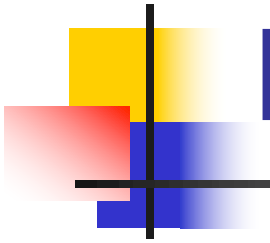
```
architecture simple of comparateur is
-- zone de déclaration (ici commentaire uniquement)
begin
        egal <= '1' when a = b else '0';
        ....
end simple ;
```

Unités de Conception

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

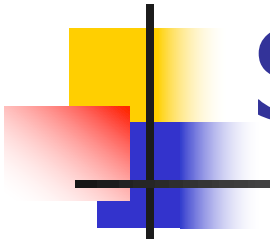
```
architecture simple of comparateur is
begin
        egal <= '1' when a = b else '0';
        ....
end simple ;

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

deux architectures
d'un même composant



Plan

- Types
- Unités de conception
- **Simulations évènementielles**
- Instructions séquentielles et concurrentes
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion



Simulations Evènementielles

- Particularités des systèmes numériques
  - Un système numérique ne "change d'état" (notion d'évènement) qu'à des instants discrets
  - Les sorties de chaque module numérique ne peuvent évoluer que lorsqu'il y a une évolution au moins d'une de leurs entrées
  - Toutes les opérations sont effectués en parallèle

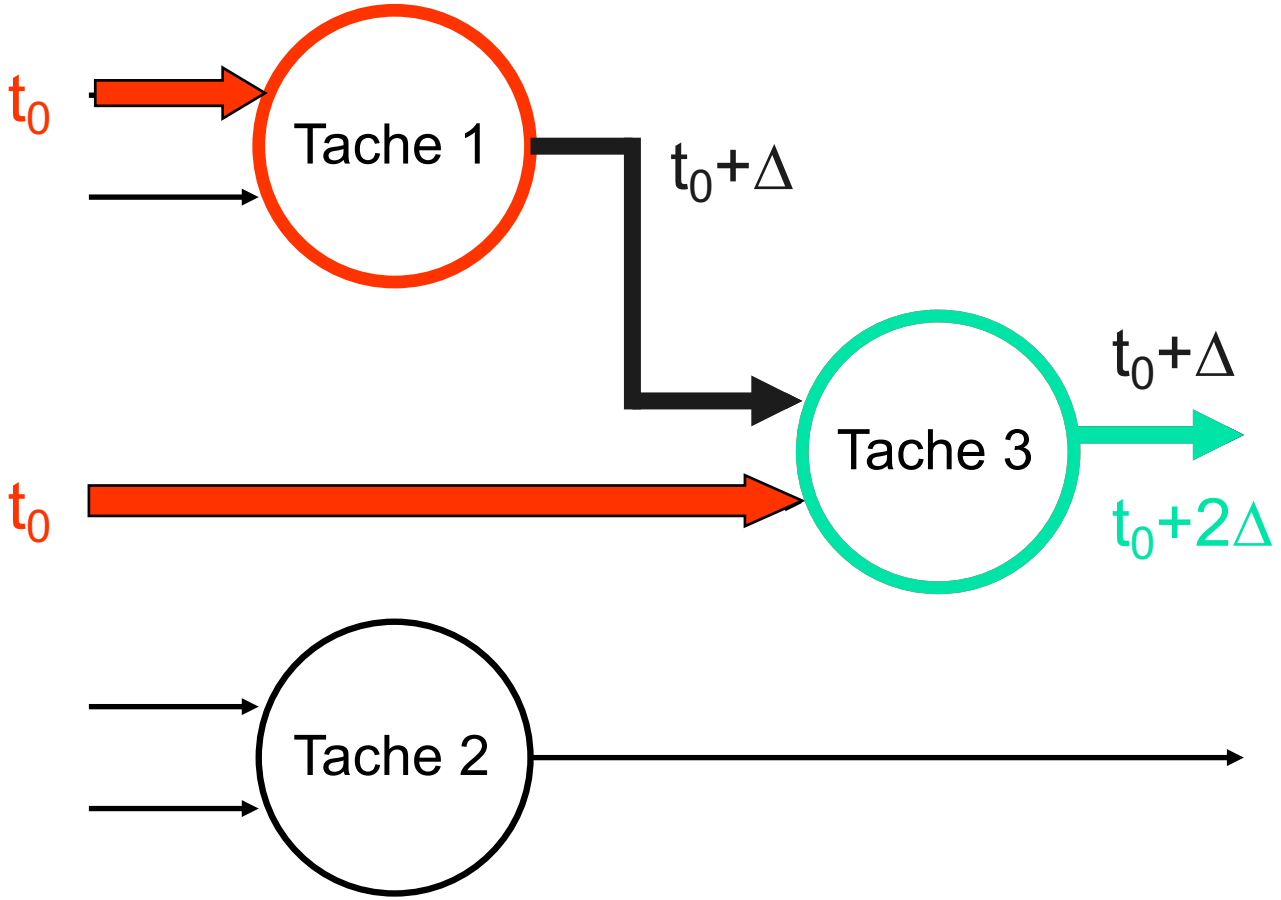
Simulations Evènementielles

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

- Définition des signaux :
  - Objets mémorisant l'information à des instants discrets
  - Les signaux connectent les composants entre eux (équivalent au fil dans un schéma)
  - Ils sont caractérisés par :
    - Un type
    - Une valeur initiale : ' 0 ' par défaut pour le type bit,
    - Un pilote (ou driver)

Simulations Evènementielles

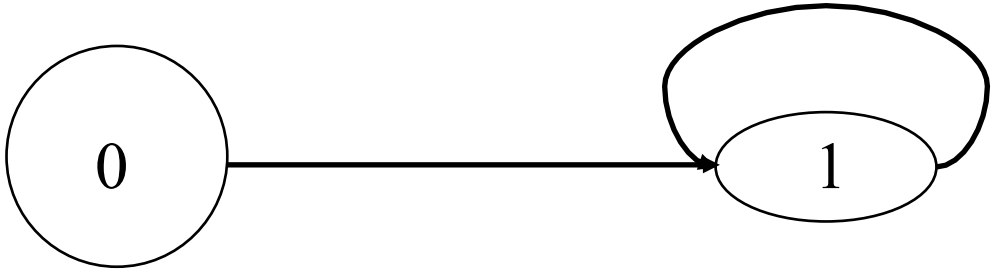
- Intérêt des signaux dans la simulation matérielle



Simulations Evènementielles

- Intérêt des signaux dans la simulation matérielle

Avancée du temps de simulation jusqu'à t_{min}



Initialisation des signaux à leur valeur initiale

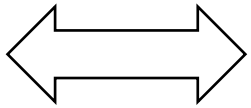
Réveil et simulation des process concernés
Production de **nouveaux événements**
Extraction des signaux avec t_{min}

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

Simulations Evènementielles

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

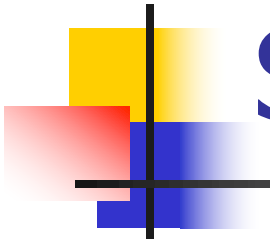
S **<=** '0', '1' **after** 10 ns, '0' **after** 25 ns;



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

Pilote de S

Le pilote est une mémoire associée à chaque signal



Simulations Evènementielles

- Contrairement à une variable, un signal n'est jamais affecté instantanément mais uniquement lorsque l'exécution du *process* est suspendu

- Définition d'un processus :
  - Élément calculatoire élémentaire du simulateur
  - Les processus sont concurrents : leur ordre dans le programme n'a pas d'importance

Simulations Evènementielles

- Exemple : Simulation d'un générateur d'horloge

```
...
signal h : bit;
begin
        horloge : process -- déclaration de processus
        -- zone de déclaration du process
        begin -- début de la zone de définition du processus
                h <= '0', '1' after 75 ns;
                wait for 100 ns;
        end process; -- fin du processus
...
```

Mise en suspend du process pour une durée déterminée

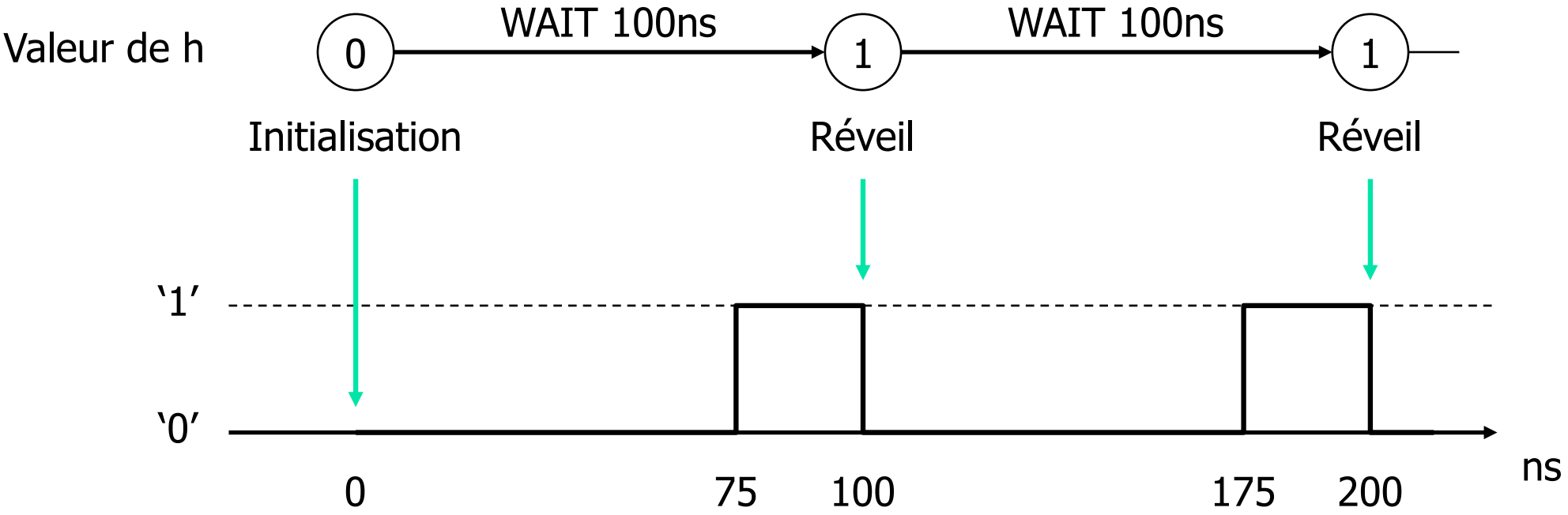
Simulations Evènementielles

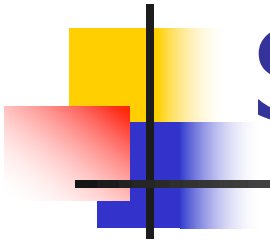
Pilote de h

T	Etat
0 (Δ)	'0'
75	'1'

T	Etat
0 (Δ)	'0'
75	'1'

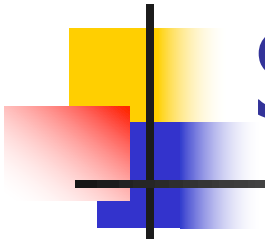
T	Etat
0 (Δ)	'0'
75	'1'





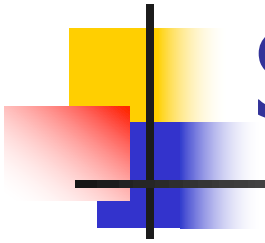
Simulations Evènementielles

- Un processus
  - vit à partir du chargement du code exécutable dans le simulateur
  - est exécuté une fois au début de la simulation
  - meurt avec la fin de la simulation
  - peut être endormi ou suspendu (indéfiniment ou temporairement) avec l'instruction **WAIT**
  - est exécuté chaque fois qu'un évènement intervient dans sa liste de sensibilité (définie ci-après)



Simulations Evènementielles

- Instruction **WAIT**
  - Synchronise (en les suspendant) les processus
  - Quatre possibilités (pouvant être combinées dans un même processus) :
    - **WAIT ON** évènement (sur un ou plusieurs signaux)
    - **WAIT FOR** durée
    - **WAIT UNTIL** condition
    - **WAIT** (stoppe le processus indéfiniment)

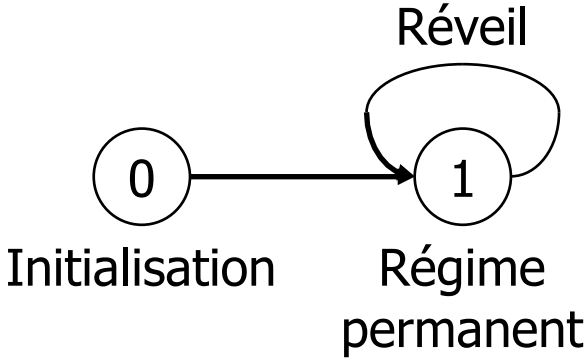


Simulations Evènementielles

- Le **AFTER** et le **WAIT FOR** sont interdits dans les descriptions pour la synthèse
- S'il y en a, ils sont généralement ignorés par les outils de synthèse
  - Comment l'outil de synthèse pourrait-il présager du délai alors que ce délai dépend principalement du routage…?

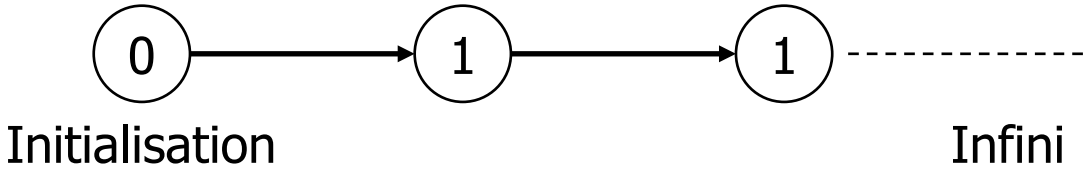
Simulations Evènementielles

- Chaque processus s'exécute une fois à l'initialisation jusqu'à rencontrer un **WAIT**;
- puis, lorsque la condition de réveil est validée,
- l'exécution continue de façon cyclique
- Vie d'un processus



Simulations Evènementielles

- 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 sur lui même infiniment (il n'est jamais suspendu)



Simulations Evènementielles

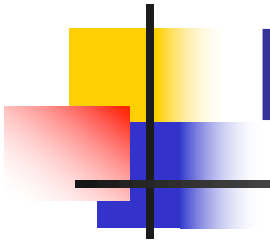
- Illustration de l'équivalence entre la *liste de sensibilité* et l'instruction **WAIT**

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

Liste de sensibilité

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

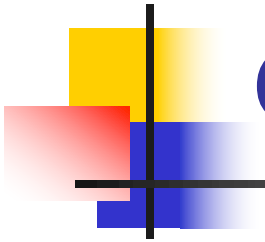
Attente d'évènements



Plan

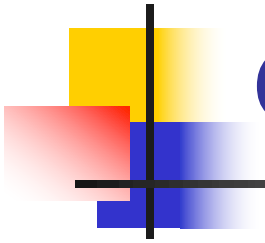
- Types
- Unités de conception
- Simulations évènementielles
- **Instructions séquentielles et concurrentes**
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion

Instructions Séquentielles et Concurrentes



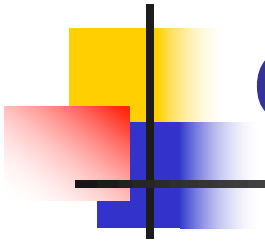
- Définition d'une instruction séquentielle :
  - Instruction **OBLIGATOIREMENT** à l'intérieur d'un process
  - Instruction dont la **POSITION** dans le process à une importance sur le résultat final
- Définition d'une instruction concurrente :
  - Instruction dont la position dans le programme n'a pas d'influence
- Rappel :
  - Les instructions (séquentielles et concurrentes) se placent toujours uniquement entre le **begin** et le **end** de l'architecture

Instructions Séquentielles et Concurrentes



- A l'intérieur d'un processus, il ne peut y avoir que des instructions séquentielles :
  - Exécution classique dans l'ordre d'écriture
  - « ça ressemble à du C »

Instructions Séquentielles et Concurrentes



- 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)

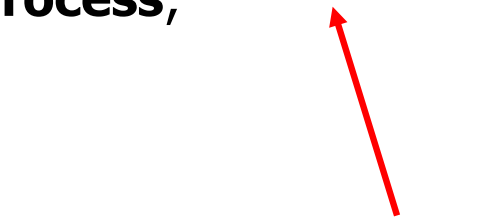
- Seul un signal, suivant le contexte, peut être affecté de façon concurrente ou séquentielle
  - Affectation dès que le process est suspendu

Instructions Séquentielles et Concurrentes

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

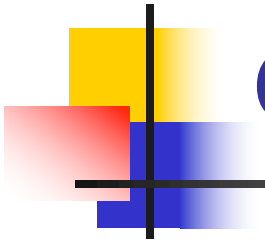
Instructions Séquentielles et Concurrentes

```
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 !!!

Instructions Séquentielles et Concurrentes



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

Instructions Séquentielles et Concurrentes

- Affectation séquentielle simple

```
p1: process
        begin
                wait on a, b ;
                s <= a and b;
end process;
```

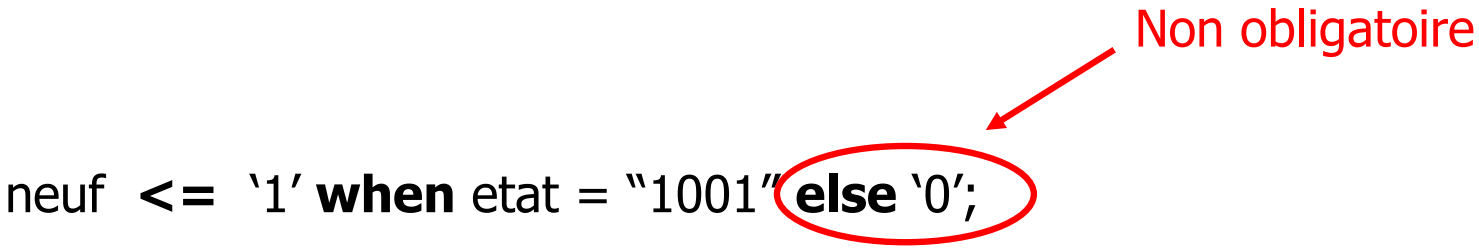
Instructions Séquentielles et Concurrentes

- Affectation séquentielle conditionnelle

```
p2: process
    begin
        wait on etat ;
        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 assignées sinon création de latch)

Instructions Séquentielles et Concurrentes



- Dans les deux cas (**if** et **when**)
  - Non exclusivité des conditions MAIS ordre de priorités!
  - Attention à la logique inutile lors de la synthèse

Instructions Séquentielles et Concurrentes

- Affectation séquentielle sélective

```
p3: process
    begin
        wait on e0, e1, e2, e3, ad;
        case ad is
            when "00" => s <= e0 ;
            when "01" => s <= e1 ;
            when "10" => s <= e2 ;
            when others => s <= e3 ;
        end case;
    end process;
```

Exclusivité des conditions
MAIS
Pas d'ordre de priorités!

Instructions Séquentielles et Concurrentes

- 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;
```

Pas de priorité

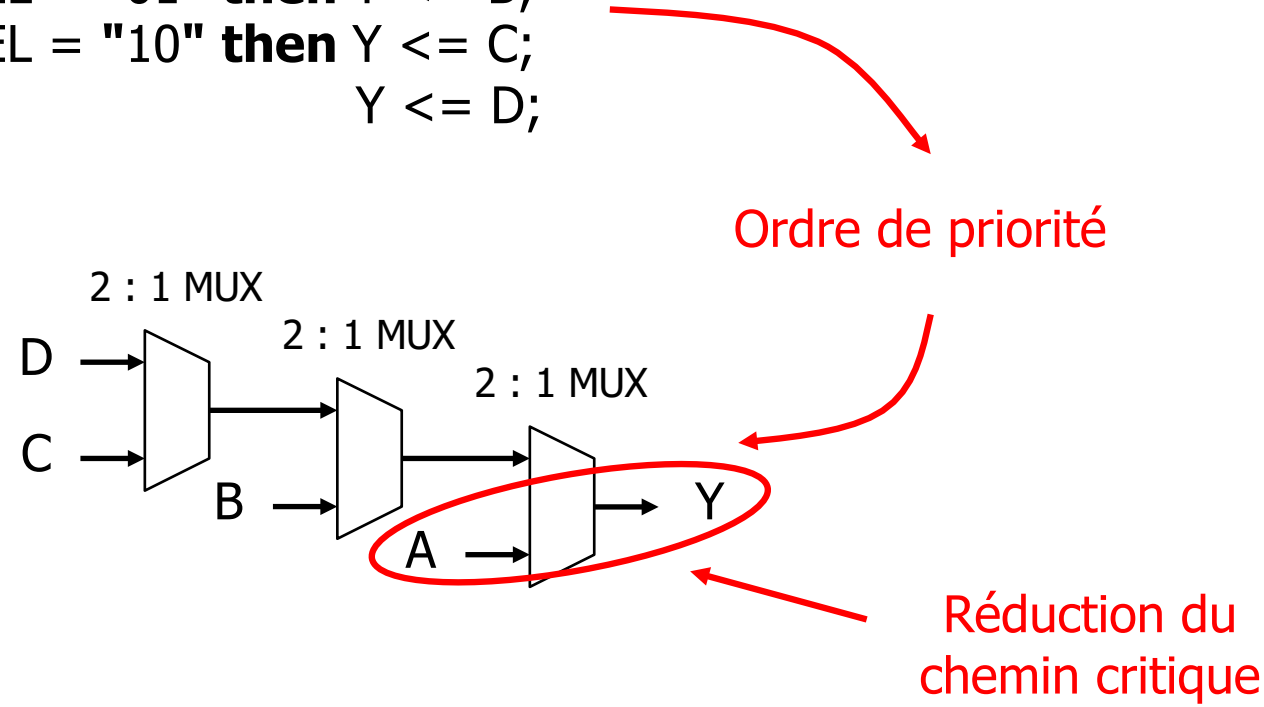
Obligatoire quand toutes les conditions n'ont pas été couvertes

- Structure de type multiplexeur

Instructions Séquentielles et Concurrentes

- Ordre de priorité : **if** versus **case**

```
if      SEL = "00" then Y <= A;
elseif  SEL = "01" then Y <= B;
elseif  SEL = "10" then Y <= C;
else                    Y <= D;
end if;
```

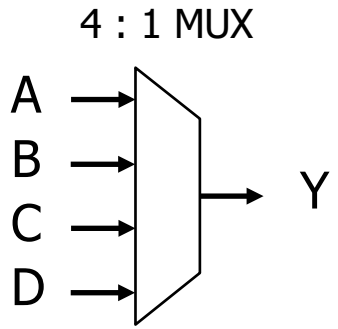


Instructions Séquentielles et Concurrentes

- Ordre de priorité : **if** versus **case**

```
case SEL is
    when "00" => Y <= A ;
    when "01" => Y <= B ;
    when "10" => Y <= C ;
    when others => Y <= D ;
end case;
```

Sans priorité



Instructions Séquentielles et Concurrentes

- 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;
```

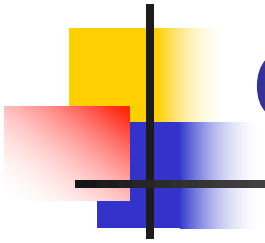
Il ne faut pas déclarer la variable i

Instructions Séquentielles et Concurrentes

- Exemple : and logique de tous les bits de a_bus

```
architecture fonctionne of mon_and is
begin
p1:process (a_bus)
    variable x_temp : bit;
    begin
        x_temp := ' 1 ';
        for i in 7 downto 0 loop
                x_temp := a_bus(i) and x_temp;
        end loop;
        x <= x_temp;
    end process;
end fonctionne;
```

Instructions Séquentielles et Concurrentes

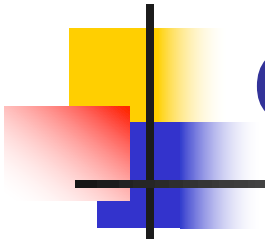


- Précédence
  - Il n'y a pas d'ordre de précédence entre les opérateurs logiques élémentaires
  - Il faut obligatoirement des parenthèses s'il y a un doute… sinon cela conduit à une erreur de compilation

X <= A or B and C

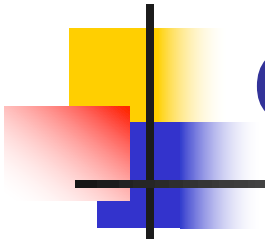
X <= A or (B and C)

Instructions Séquentielles et Concurrentes



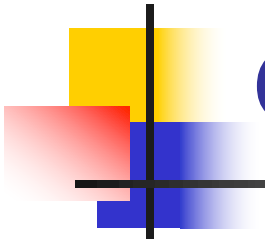
- Exercice d'application : Modélisation d'éléments de logique combinatoire
  - Décrire (en utilisant uniquement des instructions concurrentes) le module *simple_alu* possédant deux entrées a et b de type vecteur de bit de largeur 4, une entrée ctrl de type booléen et une sortie z de type vecteur de bit de largeur 4 également
  - *simple_alu* est sensible à a, b et ctrl.
  - de plus :
    - Si ctrl est vrai alors
      - z = a et b
    - Sinon
      - z = a ou b

Instructions Séquentielles et Concurrentes



```
entity simple_alu is
port (    a, b       : in       bit_vector(3 downto 0);
    ctrl    : in       boolean;
    z       : out      bit_vector(3 downto 0)
       );
end entity;

architecture bhv of simple_alu is
begin
    z <= a and b when ctrl = true else a or b;
end;
```

Instructions Séquentielles et Concurrentes



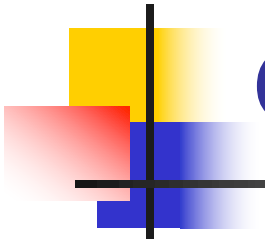
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d’une bascule RS

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule RS

```
entity rs is
port (      r, s          : in        bit ;
            q             : out       bit );
end entity;
architecture bhv of rs is
signal qtemp : bit;
begin
     rs: process (r, s)
     begin
          if s = ´0´ and r = ´0´ then qtemp <= qtemp ;
          elseif s = ´0´ and r = ´1´ then qtemp <= ´0´ ;
          elseif s = ´1´ and r = ´0´ then qtemp <= ´1´ ;
          end if;
     end process;
     q <= qtemp ;
end bhv;
```

Instructions Séquentielles et Concurrentes



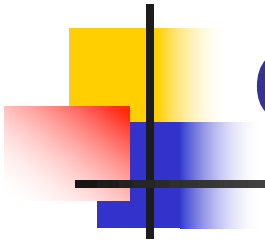
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d’une bascule Latch

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule Latch

```
entity latch is
port (      d, h        : in        bit;
            q, qb       : out       bit
      );
end entity;
architecture bhv of latch is
signal qtemp : bit;
begin
     latch: process (h, d)
     begin
            if h = ´1´ then qtemp <= d ;
            end if;
     -- mémorisation implicite lorsque la condition est fausse
end process;
q <= qtemp ;
qb <= not (qtemp) ;
end bhv;
```

Instructions Séquentielles et Concurrentes



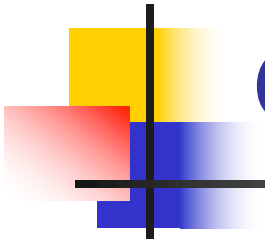
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d’une bascule D avec RAZ synchrone

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule D avec RAZ synchrone

```
entity dff  is
port (      d, h, raz    : in        bit;
            q            : out       bit
      );
end entity;
architecture bhv of dff is
begin
    dff: process
    begin
            wait until h'event and h = '1';
            if raz = '1' then q <= '0'; -- raz prioritaire si h'event
            else q <= d;
            end if;
    end process;
end bhv;
```

Instructions Séquentielles et Concurrentes



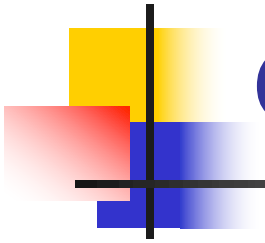
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d’une bascule D avec RAZ asynchrone

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule D avec RAZ asynchrone

```
entity dff  is
port (      d, h, raz    : in        bit;
            q            : out       bit
      );
end entity;
architecture bhv of dff is
begin
    dff: process
    begin
            wait on h, raz;
            if raz = ´1´ then q <= ´0´;      -- raz prioritaire hors h'event
            elsif h´event and h = ´1´ then q <= d;
            end if;
    -- mémorisation implicite lorsque la condition est fausse
    end process;
end bhv;
```

Instructions Séquentielles et Concurrentes



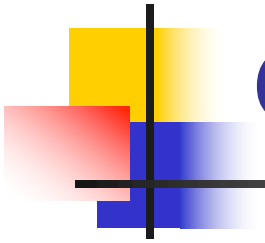
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d’une bascule JKFF

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule JKFF

```
entity jkff  is
port (      h, j, k      : in       bit;
            q, qn        : out      bit );
end entity;
architecture bhv of jkff is
signal qtemp : bit;
begin
    jkff: process
    begin
            wait until h'event and h = '1';
            if j = ´0´ and k = ´0´ then qtemp <= qtemp ;
            elseif j = ´0´ and k = ´1´ then qtemp <= ´0´ ;
            elseif j = ´1´ and k = ´0´ then qtemp <= ´1´ ;
            else qtemp <= not (qtemp);
            end if;
    end process;
    q <= qtemp ;
    qn <= not (qtemp) ;
end bhv;
```

Instructions Séquentielles et Concurrentes



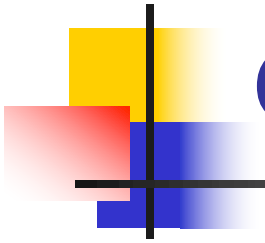
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d’un registre

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'un registre

```
entity registre is
port (    h, raz      : in      bit;
          e           : in      bit_vector(3 downto 0);
          q           : out     bit_vector(3 downto 0));
end entity;
architecture bhv of registre is
begin
    reg: process
    begin
          wait until h'event and h = '1';
          if raz = ´1´ then q <= "0000" ;
          else q <= e ;
          end if;
    end process;
end bhv;
```

Instructions Séquentielles et Concurrentes



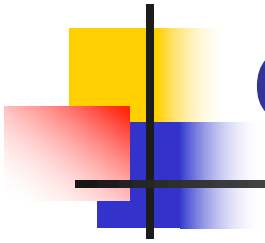
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d’un compteur 4 bits

Instructions Séquentielles et Concurrentes

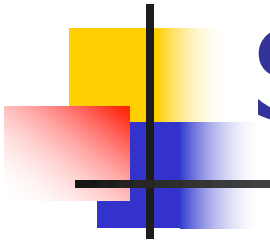
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'un compteur 4 bits

```
entity cpt is
port (     h, raz      : in       bit;
           q           : out      bit_vector(3 downto 0));
end entity;
architecture bhv of cpt is
signal qtemp : bit_vector(3 downto 0);
begin
    cpt : process
    begin
          wait until h'event and h = '1';
          if raz = ´1´ then qtemp <= "0000" ;
          else qtemp <= qtemp + 1 ;
          end if ;
    end process;
    q <= qtemp ;
end bhv;
```

Instructions Séquentielles et Concurrentes



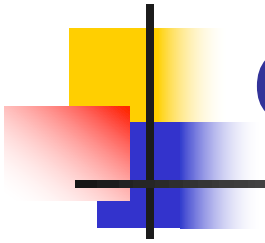
- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'un compteur permettant de compter de 3 (0011) à 11 (1011) par pas de 2
    3 : 0011
    5 : 0101
    7 : 0111
    9 : 1001
    11 : 1011
  - Ce compteur doit aussi posséder une entrée d'initialisation (init) asynchrone permettant de placer le compteur dans son état initial 3 (0011).



Solution

```
entity compteur is
   port ( clk, int     : in  bit;
          q            : out  bit_vector(3 downto 0);
end compteur;
architecture behavioral of compteur is
signal cpt : bit_vector(3 downto 0);
begin
    process (clk, init)
    begin
          if init = '1' then cpt <= "0011";
          elsif clk'event and clk = '1' then
                    if cpt = "1011" then cpt <= "0011";
                    else cpt <= cpt + 2;
                    end if;
          end if;
    end process;
    q <= cpt;
end behavioral;
```

Instructions Séquentielles et Concurrentes



- Conseils pour la synthèse
  - Ne pas donner de valeur d'initialisation pour les signaux
  - Spécifier explicitement le nombre de bits pour chaque signal
  - Utiliser les valeurs '–' lors de l'affectation de signaux permet à l'outil de synthèse d'optimiser les fonctions booléennes
  - Utiliser les parenthèses permet de contrôler le traitement concurrent de fonctions combinatoires (donc les délais)
  - Utiliser l'instruction "select" permet de générer moins de logique car sans notions de priorité

Instructions Séquentielles et Concurrentes

- Quelques exemples

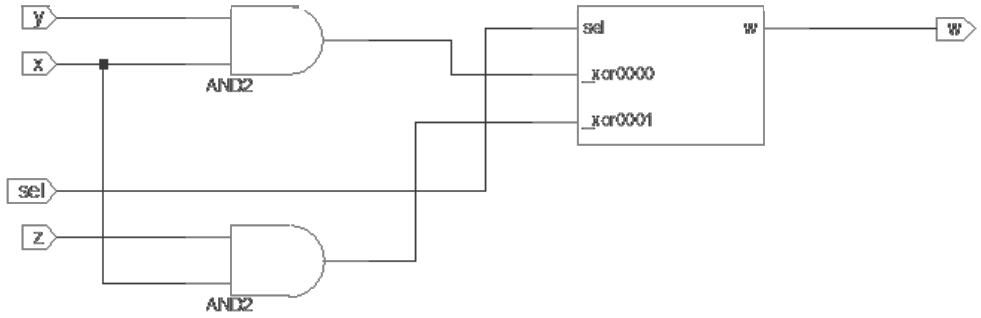
```
library IEEE;
use IEEE.std_logic_1164.all;
entity inference is
port(       sel : in std_logic;
            x, y, z : in std_logic;
            w : out std_logic);
end entity inference;

architecture behavioral1 of inference is
begin
 process(x,y,z,sel) is
 begin
  if (sel='1') then
   w <= x and y;
  else
   w <= x and z;
  end if;
  end process;
end architecture behavioral1;
```

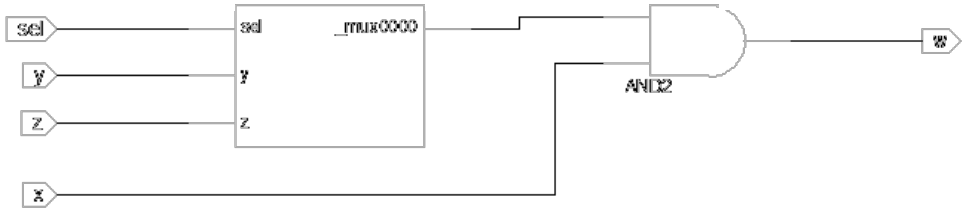
Est-ce que behavioral 1 et 2 produisent le même résultat de synthèse ?

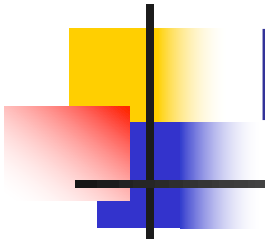
```
architecture behavioral2 of inference is
begin
process(x,y,z,sel) is
            variable right : std_logic;
            begin
            if (sel='1') then
                        right := y;
            else
                        right := z;
            end if;
            w <= x and right;
end process;
end architecture behavioral2;
```

Instructions Séquentielles et Concurrentes

- La synthèse de l'architecture behavioral1 génère deux opérateurs AND alors que behavioral2 un seul AND précédé d'une sélection des opérandes





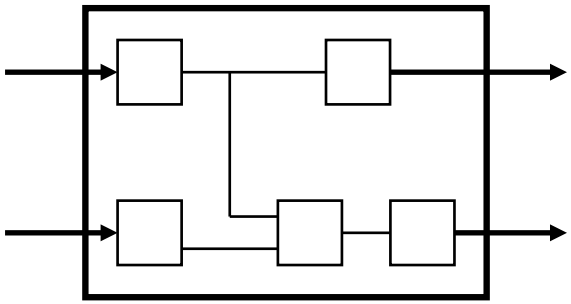


Plan

- Types
- Unités de conception
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- **Descriptions structurelles**
- Description de la maquette de test
- Conclusion

Descriptions structurelles

Circuit



Code VHDL

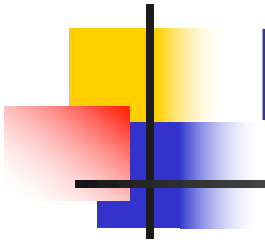
```
entity comparateur is
port (
        signal a : in bit_vector(7 DOWNTO 0);
        signal b : in bit_vector(7 DOWNTO 0);
        signal egal : out bit);
end entity;
…..
```

Description Comportementale

⇒ *Décrit ce que la fonction doit réaliser*

Description Structurelle

⇒ *Décrit comment la fonction est réalisée*



Descriptions structurelles

- Description de type hiérarchique des interconnexions entre composants
  - Contient un ou plusieurs composants
    - COMPONENT
  - Structures imbriquées à un ou plusieurs niveaux hiérarchiques

Descriptions structurelles

- Exemple : Compteur 4 bits synchrone avec autorisation de comptage

```
entity compteur4 is
port ( h, raz, compter : in bit;
       sortie : out bit_vector( 3 downto 0);
       plein : out bit);
end;
architecture structure1 of compteur4 is
signal d, s , sb : bit_vector( 3 downto 0);
component bascule
port (h, d, raz : in bit;
      s, sb : out bit);
end component;
component calcul
port ( s, sb : in bit_vector( 3 downto 0);
       compter : in bit;
       d : out bit_vector( 3 downto 0);
       plein : out bit);
end component;
```

Déclarer les signaux internes nécessaires

Déclarer tous les composants nécessaires

Descriptions structurelles

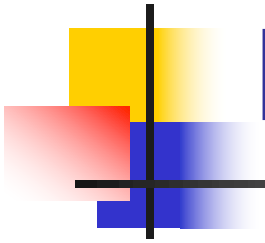
```
begin
    ba : bascule -- instanciation par position
    port map ( h, d(3), raz, s(3), sb(3));
    bb : bascule -- instanciation par dénomination
    port map ( h => h, d => d(2), raz => raz, s => s(2), sb => sb(2));
    bc : bascule -- instanciation par position et dénomination
    port map ( h, d(1), sb => sb(1), s => s(1), raz => raz);
    bd : bascule -- instanciation par dénomination
    port map ( sb => sb(0), s => s(0), h => h, d => d(0), raz => raz);

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

    sortie <= s;
end structure1;
```

Instancier chaque composant en indiquant sa liste de connexions

Descriptions structurelles

- Description du composant 'Calcul'

```
entity calcul is
port (      s, sb       : in bit_vector( 3 downto 0);
            compter     : in bit;
            d           : out bit_vector( 3 downto 0);
            plein       : out bit);
end;
architecture par_10 of calcul is
signal pas_compter : bit;
begin
    pas_compter <= not compter;
    d(3) <= (compter and s(2) and s(1) and s(0))
    or (s(3) and (sb(0) or pas_compter)) ;
    d(2) <= (compter and sb(2) and s(1) g s(0))
    or ( s(2) and (pas_compter or sb(1) or sb(0)));
    d(1) <= (compter and sb(3) and sb(1) and s(0))
     or ( s(1) and (pas_compter or sb(0)));
    d(0) <= compter xor s(0);
    plein <= s(3) and s(0);
end ;
```



Descriptions structurelles

- 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

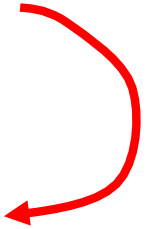
- Comportement aléatoire à éviter!

Descriptions structurelles

- Instanciation directe

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

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



- L'instanciation directe permet
  - de préciser l'architecture utilisée
  - d'éviter la déclaration préalable du composant

Descriptions structurelles

- Une spécification de configuration permet de modifier ces associations par défaut en précisant quelle unité de conception doit être employée pour un composant

for *liste_labels* : *composant* **use entity** *librairie*.*entité*[(*architecture*)]

Nom des labels des composants

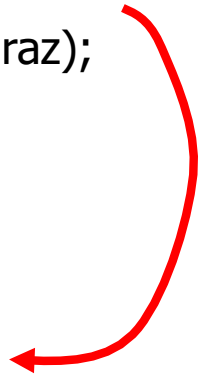
Nom du composant à configurer

Unité de conception à utiliser

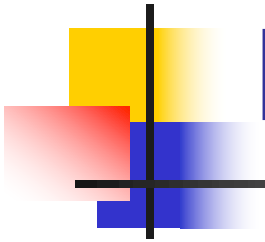
Descriptions structurelles

- GENERATE permet de réaliser des instanciations multiples

```
ba : bascule -- instanciation par position
port map ( h, d(3), raz, s(3), sb(3));
bb : bascule -- instanciation par dénomination
port map ( h => h, d => d(2), raz => raz, s => s(2), sb => sb(2));
bc : bascule -- instanciation par position et dénomination
port map ( h, d(1), sb => sb(1), s => s(1), raz => raz);
bd : bascule -- instanciation par dénomination
port map ( sb => sb(0), s => s(0), h => h, d => d(0), raz => raz);
```



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

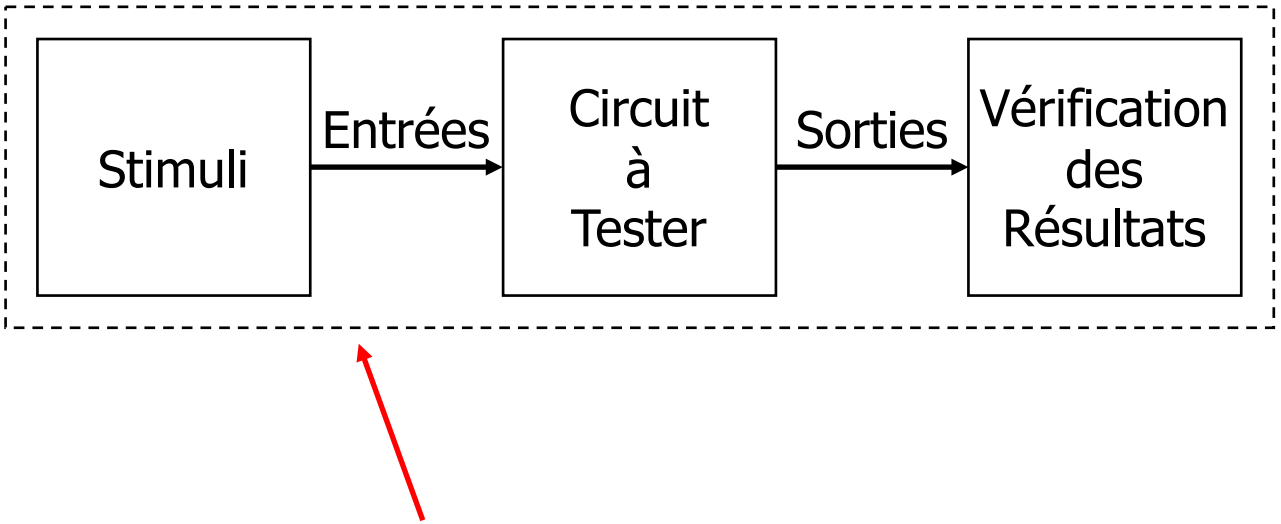


Plan

- Types
- Unités de conception
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- Descriptions structurelles
- **Description de la maquette de test**
- Conclusion

Description de la maquette de test

- Comment simuler les description VHDL
  - Utiliser un Testbench $\Rightarrow$ Maquette de Test



Description de la maquette de test

```
entity test_compteur4 is
end;

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(decade)
                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;
```

Aucun port d'entrée/sortie

Instancier le composant à tester

Description des stimuli