

III. VHDL

Plan global du cours

- I. Introduction

- II. FPGA

- III. VHDL

- Introduction

- Règles d'écriture

- Unités de conception - Objets VHDL - Opérateurs

- Assignations concurrentes/séquentielles

- Assignations conditionnelles/sélectives

- Composant

- Machine à états

- Règles de conception

- Simulation

- Compléments (fonctions, procédures, packages, ...)

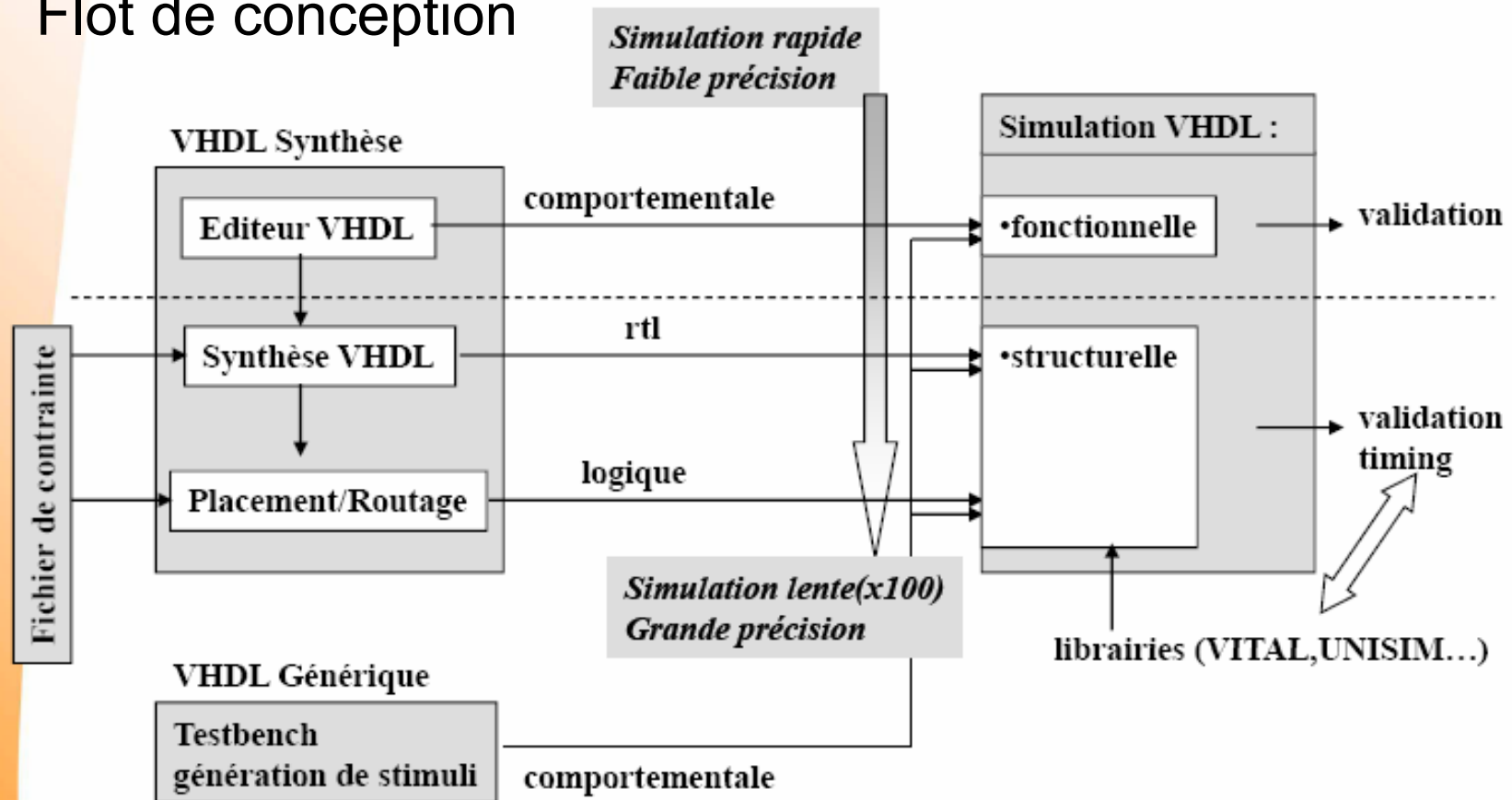
Introduction

Historique

- 1981 : le DoD (*department of defence*) initie le projet VHSIC (*Very High Speed Integrated Circuit*) – IBM, TI, Intermetrix
 - ↳ langage commun pour la description des circuits
- 1987 : normalisation du VHDL (*VHSIC Hardware Description Language*) par l'IEEE (*Institute of Electrical and Electronic Engineers*)
 - La norme est révisée tous les 5 ans (P1076-87, P1076-93, P1076-2000, P1076-2002, P1076-2006)
 - Il existe aussi des normes correspondant à des groupes de travail sur des extensions ou des restrictions pour des besoins spécifiques (P1076.1, P1076.6-2004,...)

Pourquoi et où utilise-t-on le VHDL?

Flot de conception

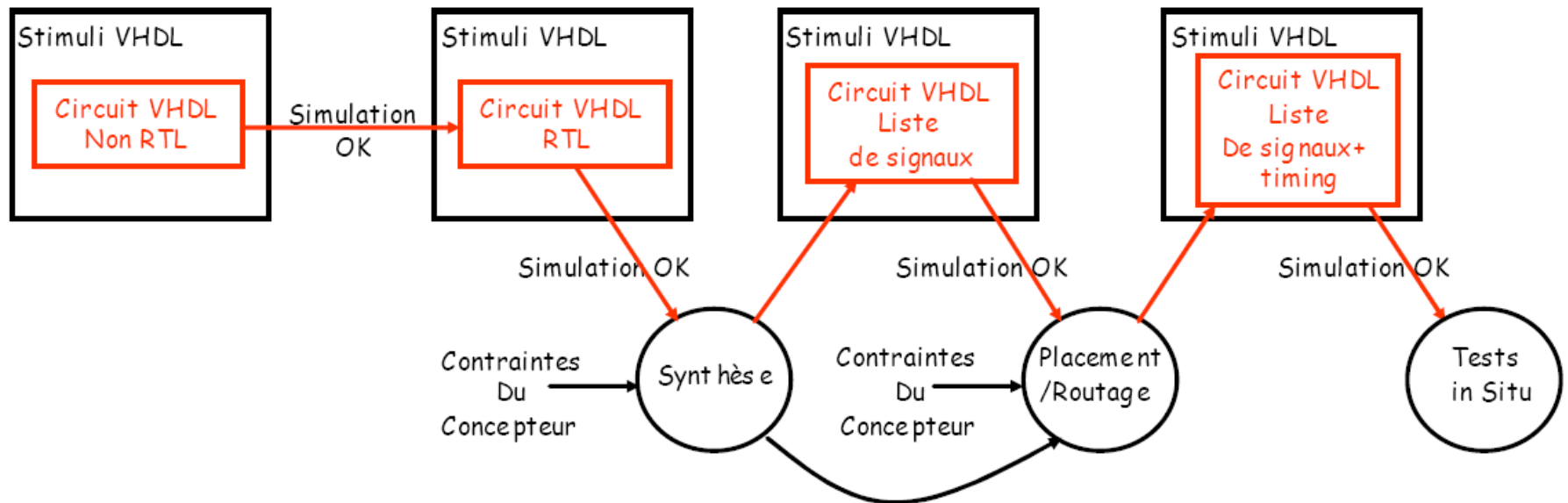


rtl : Register Transfer level : ensemble de registres (synchro.), interconnectés par de la logique combinatoire

Introduction

Pourquoi et où utilise-t-on le VHDL?

Flot de conception



Introduction

Qu'est ce que le VHDL?

- Langage de modélisation et/ou de synthèse de systèmes électroniques
- Langage généraliste ➔ **simulation**
- Langage près des portes logiques ➔ **VHDL de synthèse**
- Système de CAO : Schéma ou VHDL ou mixte

Introduction

Qu'est ce que le VHDL?

- Langage normalisé, sensé être indépendant de tout compilateur ou simulateur
- Cible : CPLD, FPGA, ASIC
- Conception de haut niveau d'abstraction (modification rapide de la conception, réduction cycle, réduction risques, ...)
- Meilleure gestion de projet, développement de grosses conceptions (structure hiérarchisée)
- Adéquation entre le style d'écriture et les compilateurs : résultats en terme de vitesse ou de compacité différents

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

Règles d'écriture

- VHDL 87 (obsolète), VHDL 93, VHDL 2000 (très peu de changements)
- La simulation comportementale est indépendante du style d'écriture, du compilateur utilisé et de la cible ; **pas la synthèse**
- Aucune distinction entre les minuscules et les majuscules
- Commentaires : commencent par 2 tirets et se prolongent jusqu'à la fin de la ligne. Ils sont ignorés par le compilateur

```
--Ceci est un commentaire  
--Et voici la deuxième ligne de commentaire
```

Règles d'écriture

- En général, les instructions se terminent par « ; »

```
A <= B and (not C);  
data <= "0001";
```

- Règles de dénomination :
 - 26 lettres de l'alphabet, les 10 chiffres et '_'
 - Le premier caractère est une lettre
 - Il ne peut y avoir 2 « _ » de suite
 - L'identifieur ne peut se terminer par « _ »

Règles d'écriture

- Les valeurs explicites:
 - Entier : **123** **1_2_3**
 - Entier basé : base#valeur# **2#11#** ($\Leftrightarrow 3$) **16#1F#** ($\Leftrightarrow 31$)
 - Valeurs physiques : toujours laisser un espace entre la valeur et l'unité
100 ps **2 ns** **5 V**
 - Caractère : entre apostrophe '**a**' '@'
 - Bit : entre apostrophe '**0**' '**1**' '**Z**'
 - Chaîne de caractères : entre guillemets (attention là les minuscules et majuscules sont significatives) "**Bonjour** "
 - Bus : entre guillemets "**01111001** "
 - Booléen : **true** **false**

Mots réservés

| | | | | | |
|---------------------|----------------------|----------------|------------------|------------------|-----------------|
| abs | component | generic | next | record | use |
| access | configuration | guarded | nor | register | |
| after | constant | | not | rem | variable |
| alias | | if | null | report | |
| all | disconnect | in | | return | wait |
| and | downto | inout | of | | when |
| architecture | | is | on | select | while |
| array | else | | open | severity | with |
| assert | elsif | label | or | signal | |
| attribute | end | library | others | subtype | xor |
| | entity | linkage | out | | |
| begin | exit | loop | | then | |
| block | | | package | to | |
| body | file | map | port | transport | |
| buffer | for | mod | procedure | type | |
| bus | function | | process | | |
| | | nand | | units | |
| case | generate | new | range | until | |

<http://amouf.chez.com/syntaxe.htm>

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelle/sélective
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

Unités de conception

Structure d'un fichier VHDL

essai.vhd

```
library ieee;
use ieee.std_logic_1164.all;

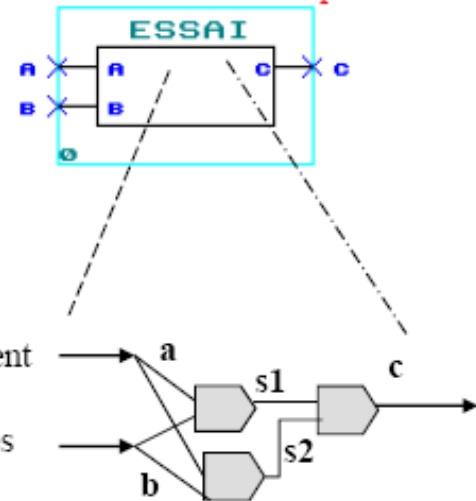
entity essai is
  port (a,b : in std_logic;
        c : out integer);
end essai;
--end entity essai; -- vhd1 93

architecture archi1 of essai is
  Signal s1, s2 : std_logic;
begin
  .
  .
end archi1;
--en architecture archi1; -- vhd1 93
```

Permet d'accéder à d'autres descriptions définies dans des diverses bibliothèques.

Rem. : Plusieurs couples **entity/architecture** sont possibles au sein d'un même fichier. Les clauses **library** et **use** ne sont pas globales. Leur effet s'arrête dès la déclaration d'une nouvelle **entity**. D'office il y a **Standard** et **Work**

Description de l'interface : **mettre le même nom que le fichier !**



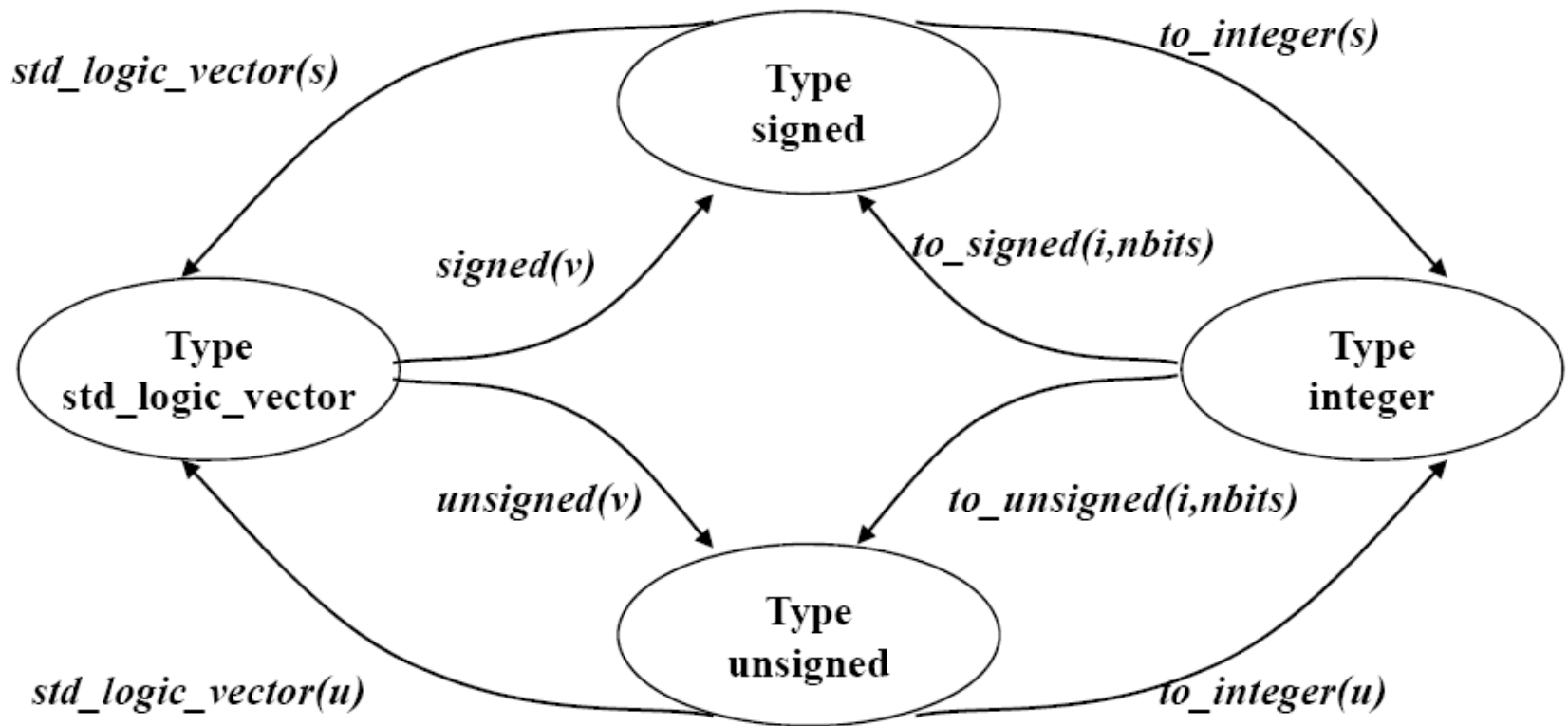
Description du comportement de l'entité. Pour une même entité plusieurs architectures sont possibles.

- IEEE.STD_LOGIC_1164.all
 - Permet de définir les types, opérateurs et fonctions de conversion de base
- IEEE.numeric_std.all
 - Définit les types **signed** et **unsigned**. Un vecteur représente alors un nombre signé ou non signé représenté en C2
 - Permet d'utiliser les opérateurs arithmétiques **+** et **x** sur des vecteurs de bits
 - Fournit des fonctions de conversion entre entiers et vecteurs

Unités de conception

Library

IEEE.numeric_std.all



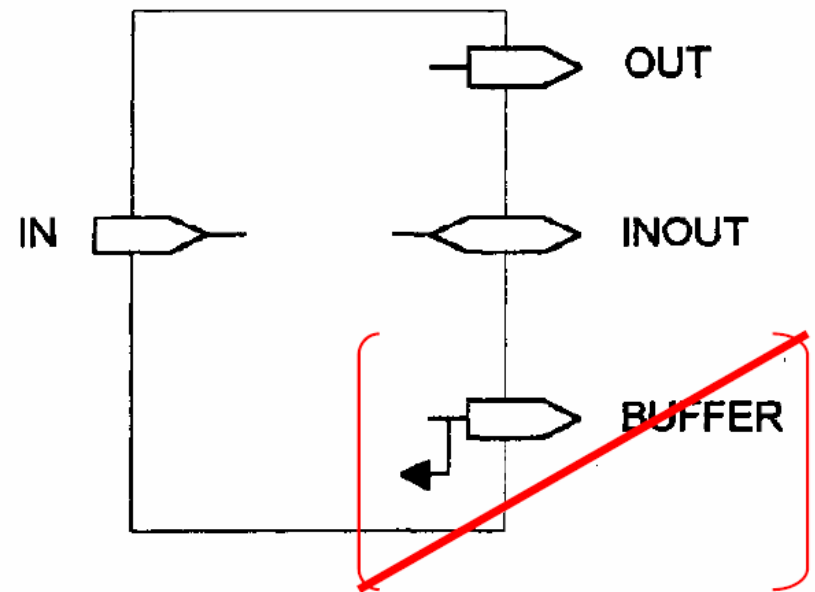
Unités de conception

Entity

- Définition de l'entité : **Vue externe d'un composant**

- Spécifications :

- Ports d'entrée et de sortie
- Type
- Mode : - entrée (**in**)
 - sortie (**out**)
 - entrée/sortie (**inout**)
 - sortie avec retour en interne (**buffer**)



Remarque : le **nom d'un fichier** VHDL doit être celui de l'**entité** qu'il contient

Unités de conception

Entity

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



- Le mode **in** protège le signal en écriture
- Le mode **out** protège le signal en lecture

Unités de conception

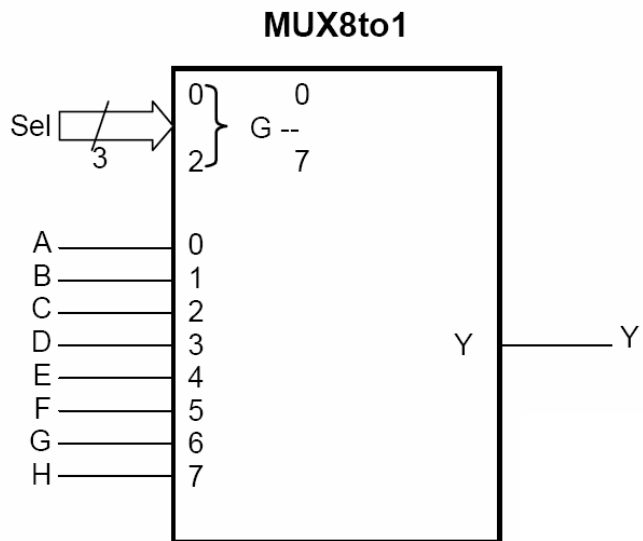
Exercice

Ecrire l'entité d'un additionneur Add4 de deux mots, **a** et **b**, de 4 bits en entrée, avec une retenue entrante **ci**, et une sortie **somme** sur 4 bits avec une retenue sortante **co**.

Unités de conception

Exercice

Ecrire l'entité correspondante au schéma de ce multiplexeur :



Unités de conception

Exercice

Dessiner le schéma du composant correspondant à l'entité suivante :

```
ENTITY parite IS
  PORT (
    b :      in bit ;
    clk :    in bit ;
    raz :    in bit ;
    parite : inout bit
  );
END parite;
```

Unités de conception

Architecture

- Toute architecture est associée à une entité
- Définition : L'architecture définit les fonctionnalités et les relations temporelles. Elle décrit le comportement du composant.

```
ARCHITECTURE arch_comparateur of comparateur is
```

```
begin
```

```
    egal <='1' when a = b else '0';
```

```
end
```

```
arch_comparateur;
```

} Zone de déclaration

} Zone de définition

Unités de conception

Architecture

- Il peut y avoir plusieurs architectures associées à un même composant

```
ARCHITECTURE arch1_comparateur of comparateur is
begin
    egal <='1' when a = b else '0';
end arch1_comparateur ;

ARCHITECTURE arch2_comparateur of comparateur is
begin
    egal <='1' after 10 ns when a = b else '0' after 5 ns;
end arch2_comparateur;
```

2 architectures d'un même composant

Remarque : l'instruction **after** n'est pas synthétisable !

Objets VHDL

- 5 sortes d'objets
 - Les ports d'entrée/sortie (**PORT**)
 - Les signaux (**SIGNAL**)
 - Les constantes (**CONSTANT**)
 - Les variables (**VARIABLE**) - - cf partie assignation séquentielle
 - Les paramètres (**GENERIC**) - - cf partie compléments
- Les objets représentent une valeur
- Ils doivent être typés

Objets VHDL

Type

- Tout objet a un format prédéfini
- Seules des valeurs de ce format peuvent être affectées à cet objet
- Plusieurs catégories de types
 - Types scalaires (numériques et énumérés)
 - Types composés (tableaux et vecteurs)
- Possibilité de définir de nouveaux types

```
type nom_type is definition_type ;
```

Objets VHDL

Type

- Types scalaires :

- Types énumérés : liste de valeurs

```
type bit is ('0','1') ;  
type boolean is (false,true) ;
```

- Types numériques : domaine de définition

range to downto

```
type i is range 0 to 3;
```

- Types composés : collections d'éléments de même type
repérés par des valeur d'indices

```
type word is array (0 to 8) of bit ;  
constant mot : word:="000000001" ;
```

Objets VHDL

Type

- Exemple de types prédéfinis :
 - Bit
 - Boolean
 - Integer
 - Std_logic *dans la bibliothèque std_logic_1164*
 - Bit_vector, Std_logic_vector
 - Signed, Unsigned *dans la bibliothèque numeric_std*
 - Natural : sous type de integer limité aux nombres ≥ 0
 - Positif : sous type de integer limité aux nombres > 0
 - Character, string

Objets VHDL

Type

- Std_logic :

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

Objet VHDL

Exercice Type

Définir un type **pental** composé de chiffres de **0** à **4**

Définir un type énuméré **etat** composé des valeurs **OK**, **HS**,
ERROR

Objets VHDL

Signal

- Un signal représente une équipotentielle
- Il doit être déclaré avant utilisation
- Il peut être déclaré :
 - dans un **package**, il est alors global
 - dans une **entity**, il est alors commun à toutes les architectures de l'entité
 - dans l'**architecture**, il est alors local

```
package essai is  
  signal clk : std_logic ;  
  signal rst : std_logic ;  
end essai;
```

```
entity essai is  
  port (  
    a,b : in std_logic;  
    c : out integer  
  );  
  signal rst : std_logic;  
end essai;
```

```
architecture rtl of essai is  
  signal rst : std_logic;  
begin  
  ...  
end rtl;
```

Objets VHDL

Signal

- L'affectation se fait avec l'opérateur « **<=** »
- Accès à des sous-éléments avec l'opérateur **alias**

alias lsb : **bit_vector**(7 downto 0) **is** add_bus(7 downto 0) ;

- Initialisation rapide

Toto <=(**others** => '0'); est équivalent à Toto <= "000000...0";

Objets VHDL

Constant

- Une constante doit être déclarée avant utilisation
- Elle peut être déclarée :
 - dans un **package**, elle est alors globale
 - dans une **entity**, elle est alors commune à toutes les architectures de l'entité
 - dans l'**architecture**, elle est alors locale
- L'affectation se fait avec l'opérateur « **:=** »

```
ARCHITECTURE arch2_comparateur of comparateur is

    constant VCC : std_logic := '1';
    constant GND : std_logic := '0';

begin

    ...

end arch2_comparateur;
```


Objets VHDL

Variable

- Une variable doit être déclarée avant utilisation
- Elle ne peut être déclarée que dans un « **process** »
- L'affectation se fait avec l'opérateur « **:=** »

```
architecture arch1 of essai is
begin
  P1:process
    variable a : integer;
  begin
    a:=5;
  end process;
end arch1;
```

Opérateurs

- Opérateurs logiques (sur booléens, bits et dérivés)

| Opérateur | Description | Résultat |
|-----------|---------------------------|-----------|
| not | Complément Logique Unaire | même type |
| and | ET logique | même type |
| or | OU logique | même type |
| nand | Non-ET | même type |
| nor | Non-OU | même type |
| xor | OU exclusif | même type |
| nxor | NON-OU exclusif (93) | même type |

- Opérateurs relationnels (sur types scalaires ou dérivés (signed, unsigned))

| Opérateur | Description | Résultat |
|-----------|-----------------------|----------|
| = | Égalité | Booléen |
| /= | Inégalité (différent) | Booléen |
| < | Inférieur | Booléen |
| <= | Inférieur ou Égal | Booléen |
| > | Supérieur | Booléen |
| >= | Supérieur ou Égal | Booléen |

Si numeric_std : signed et integer sont compatibles

unsigned et natural sont compatibles

Opérateurs

- Opérateurs bits (sur vecteurs de bits et types numériques)

| Opérateur | Description | Résultat |
|-----------|---------------|--------------------------|
| & | Concaténation | même type, plus large |

"01" & "00" = "0100"

- Opérateurs arithmétiques (sur types numériques : entiers, signés, non signés, flottant)

| Opérateur | Description | Résultat |
|-----------|---------------------------|-----------|
| + | Addition | dépend |
| - | Soustraction | dépend |
| abs | Valeur absolue | Même type |
| *, ** | Multiplication, puissance | dépend |
| / | Division | dépend |
| mod | Modulo, sur entiers | dépend |
| rem | Reste, sur entiers | dépend |

- Opérateurs de décalage (sur tableaux de bits ou étendus) :
sll, srl, sla, sra, rol, ror

Exercice

Ecrire l'ensemble d'un fichier VHDL (Library, Entity, Architecture) qui décrit une porte **OU** à 2 entrées **a** et **b** de 1 bit (sortie **s**)

Exercice

Ecrire l'ensemble d'un fichier VHDL (Library, Entity, Architecture) qui décrit un additionneur **ADD4** à **2 entrées de 4 bits A et B** signées et **une sortie S** de 4 bits

Exercice

Ecrire l'ensemble d'un fichier VHDL **concat4to8.vhd** (Library, Entity, Architecture) permettant la concaténation de **2 bus de 4 bits A et B en un bus C de 8 bits** (le bus A représente les bits de poids fort).

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

Assignations concurrentes/sequentielles

- Instruction séquentielle : instruction à l'intérieur d'un process
- Instruction concurrente : instruction à l'extérieur des process.

Rappel : *les instructions se placent toujours uniquement entre le **begin** et le **end** de l'architecture*

Assignations concurrentes

- Toutes les déclarations sont exécutées simultanément et en permanence
 - L'ordre des déclarations dans le code source n'a pas d'influence
 - Les déclarations possibles sont :
 - Assignment continue : `<=`
 - Instantiation d'un composant : `port map`
 - Assignment conditionnelle : `when ... else`
 - Assignment sélective : `with ... select ... when ... when`
 - Appel d'un `process`
 - Instruction `generate`
 - Appel d'une fonction
- } cf. compléments

Assignations séquentielles

- Ce mode concerne uniquement les **function**, **procedure** et **process**
- Les **process** manipulent les **variable** et **signal**
- Au sein de ces descriptions, les déclarations sont exécutées de manière séquentielle, l'une après l'autre
- L'ordre des déclarations est donc important
- Les déclarations possibles sont :
 - Assignation continue : `<=` (signal) et `:=` (variable)
 - Assignation conditionnelle : `if ... then ... elsif ... then ... else ... end if;`
 - Assignation sélective : `case ... is ... when ... => ... when ... => ... end case;`
 - Boucles : `for ... in ... loop ... end loop;`
 - Boucles : `while ... loop ... end loop;`
 - Instructions `next` et `exit`

Process

- Dans un process, l'interprétation des instructions est séquentielle mais l'ensemble de leur réalisation est instantanée (le temps extérieur est suspendu)
 - Les **signaux** sont assignés **en sortie du process**
 - Les **variables** sont assignée **immédiatement**
 - Les variables ne sont pas visibles de l'extérieur

```
Label : Process(liste de sensibilité)  
  --Déclaration de constantes et variables  
  Begin  
    --Description séquentielle  
  End process
```

Process

- Lors d'affectations multiples, c'est la dernière qui est prise en compte
- Un process n'est activé que lorsque les signaux de sa liste de sensibilité ont subi un changement
- Absence de liste de sensibilité : le process est réactivé en permanence
- Tous les process d'un design fonctionnent de manière concurrente

```
process(a)
variable b:std_logic;
begin
  b:= a;
  c <= b;
end process;
```

```
process(a)
begin
  b<= a;
  c <= b;
end process;
```

Règle : tout signal qui modifie un autre signal ou variable doit être dans la liste de sensibilité

```
process(a,b)
begin
  b<= a;
  c <= b;
end process;
```

Process

Exemples

```
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;
```

Process

Exemples

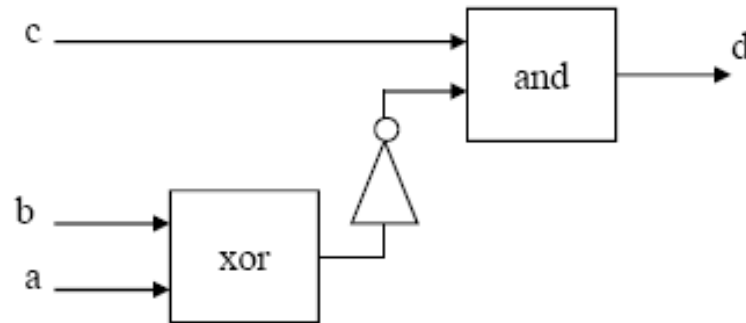
```
p2: process
  begin
    wait for 10 ns;
    aaa <= 1 ;           -- 1 dans pilote de aaa
    bbb <= aaa + 8;      -- 11 dans pilote de bbb
    aaa <= bbb - 2;      -- 0 dans pilote de aaa
  end process;
end;
```

**Seule la dernière
affectation compte!**

Process

Exercice

Décrire cette fonction en utilisant un process :

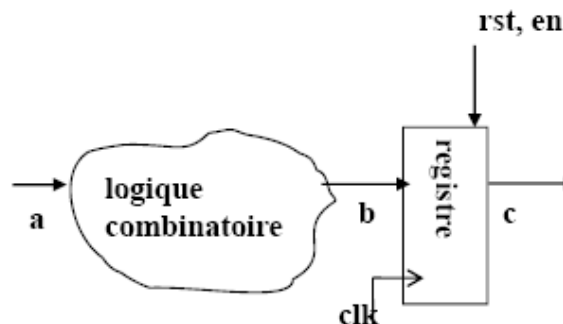


Process

- Mise en œuvre de process synchronisé sur l'horloge
 - **If clk'event and clk='1' then** ou **if rising_edge(clk)**
 - **If clk'event and clk='0' then** ou **if falling_edge(clk)**
 - Absence de liste de sensibilité et **wait until (clk='1')** placé en tête de la partie déclarative du process

Structure d'un process clocké (c'est l'ossature de base d'un design synchrone !)

```
process(clk,rst)
  variables éventuelles
begin
  if rst='1' then
    initialisation asynchrone des signaux
  elsif rising_edge(clk) then
    if en='1' then
      description du bloc combinatoire
    end if
  end if;
end process;
```



Process

Exercice

Tracer les chronogrammes qui correspondent à la description de ce process

```
process (clk)

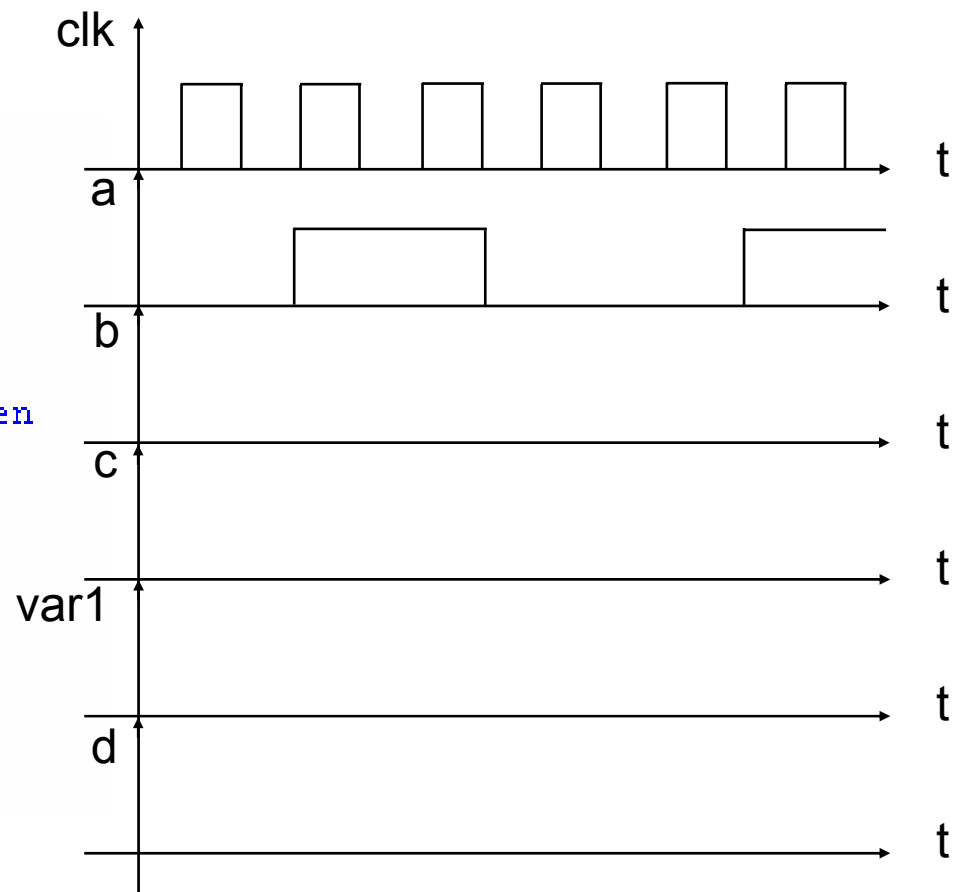
    variable var : bit;

begin

    if clk'event and clk='1' then
        b <= a ;
        c <= b ;

        var := a;
        d <= var;

    end if ;
end process;
```



Exercice

Ecrire l'ensemble d'un fichier VHDL (Library, Entity, Architecture) qui décrit un compteur **COMPT** qui compte sur **3 bits** sur front montant du signal d'horloge **clk**

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

Assignations conditionnelles

Assignation concurrente

- Forme générale :

signal **<=** valeur **when** condition **else** autre valeur **when** autre condition ...;

- Une seule cible peut être assignée

```
S <= "a" when (sel="00") else  
      "b" when (sel="01") else  
      "c";
```

- Les conditions sont sous-entendues exclusives

```
A <= B when c='0' else  
      D when e='1' else      ≈ D when e='1' and c='1' else  
      F;
```

- Mémorisation implicite lorsque toutes les conditions ne sont pas listées

```
S <= D when sel="00" else not D when sel="01" (else S);
```

Assignations conditionnelles

Assignation séquentielle

- Forme générale :

If condition **then** ... **elsif** conditions **then...else** ... **end if**;

```
process(sel)
begin
    if sel='1' then
        op <= a and b ;
    else
        op <= c;
    end if;
end process;
```

- Mémorisation implicite lorsque toutes les conditions ne sont pas listées

```
process(d,en)
begin
    if en='1' then
        q<= not d;
        -- else q<=q - - sous entendu
    end if;
end process;
```

```
process(d,en)
    variable tmp : std_logic;
begin
    tmp:=d;
    if en='1' then
        tmp:=not tmp;
    end if;
    q<= tmp;
end process;
```

Assignations sélectives

Assignment concurrente

- Forme générale :

With sélecteur **select** signal \leq valeur **when** val_sel, valeur **when** val_sel2 ...;

```
with selecteur select  
  x<= a when "00",  
    b when "01",  
    c when "10",  
    d when others;
```

- Clause **when others** qui permet de préciser tous les cas non définis
- Possibilité de regrouper plusieurs valeurs du sélecteur pour une même assignation « | »

```
with selecteur select  
  x<= a when "00",  
    b when "01" | "10",  
    d when others;
```

Assignations sélectives

Assignation séquentielle

- Forme générale :

Case sélecteur **is when** val_selec => instructions; ... **end case**;

- Possibilité de regrouper des valeurs de sélection

| |
|-----------------------------------|
| 'val1_select val2_select ' |
| val1_select to val2_select |

- Attention aux clauses incomplètes pouvant générer des latches

Utilisation de la clause **when others**

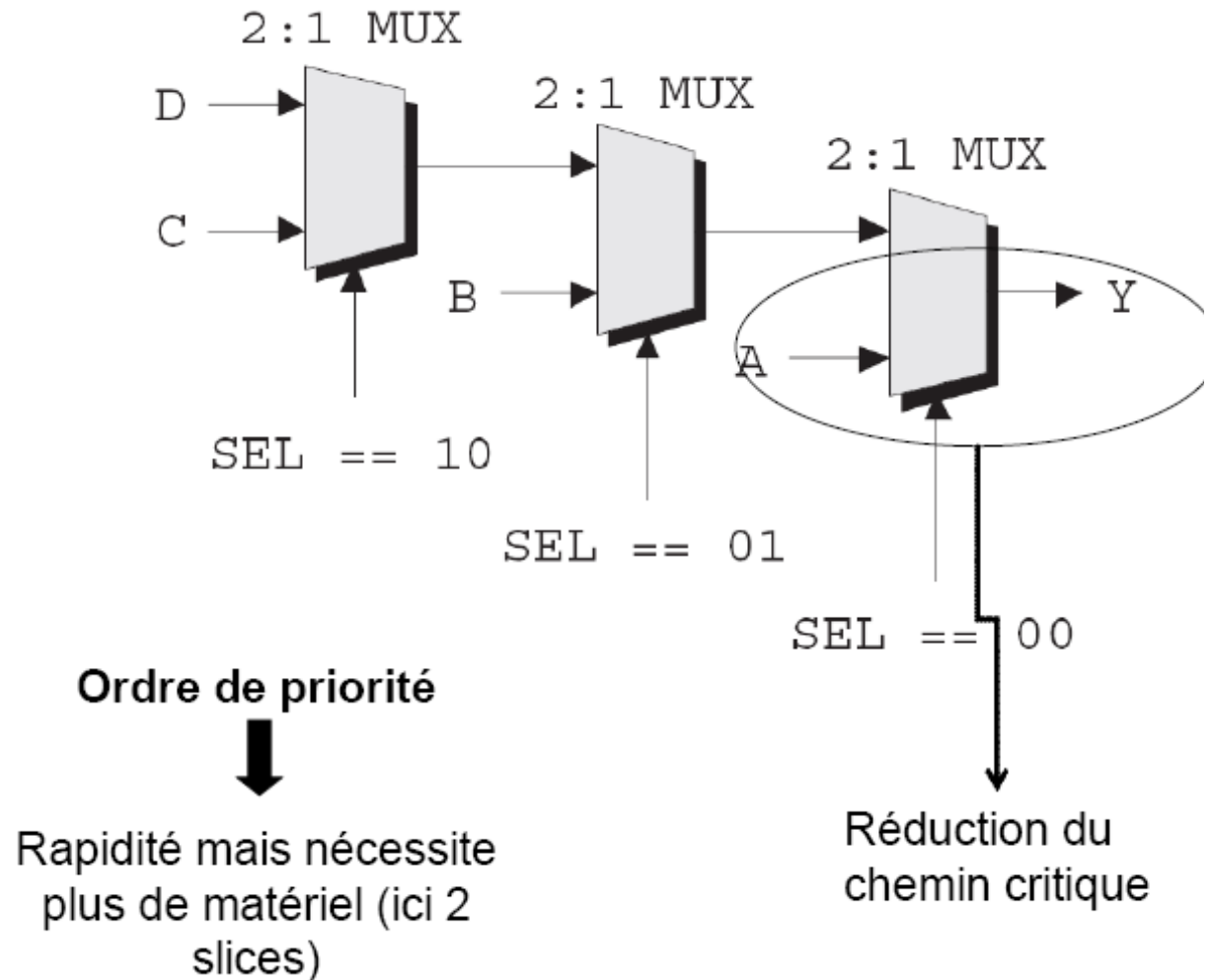
```
process
begin
  if clk'event and clk = '1' then

    case en is
      when "00" | "11" => q <= '1'; c <= '0' ;
      when "10" => q <= '0'; c <= '1' ;
      when others => null;
    end case;
  end process ;
```

Assignations sélectives

If versus case

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



Assignations sélectives

If versus case

case sel is

when « 00 » => Y<= A;

when « 01 » => Y<= B;

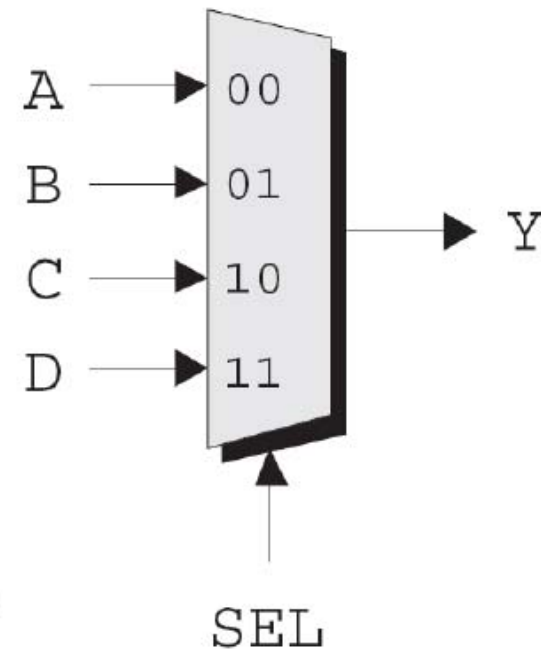
when « 10 » => Y<= C;

when others => Y<= D;

end case;

Sans priorité ➡ compact (1 slice)

4 : 1 MUX



Exercice

Ecrire l'ensemble d'un fichier VHDL (Library, Entity, Architecture) qui décrit un compteur **COMPT5** qui compte sur **3 bits** de **0 à 5** sur front montant du signal d'horloge **clk**

Exercice

Modifier la description de l'exercice précédent pour que lorsque l'entrée **Load** est à '1' la sortie du compteur prenne la valeur de l'entrée **Data** et compte à partir de cette valeur.

Exercice

Ecrire l'ensemble d'un fichier VHDL (Library, Entity, Architecture) qui décrit un comparateur mettant la sortie **EGAL** à '1' lorsque les entrées **A** et **B** (bus de 8 bits) sont égales et à '0' sinon.

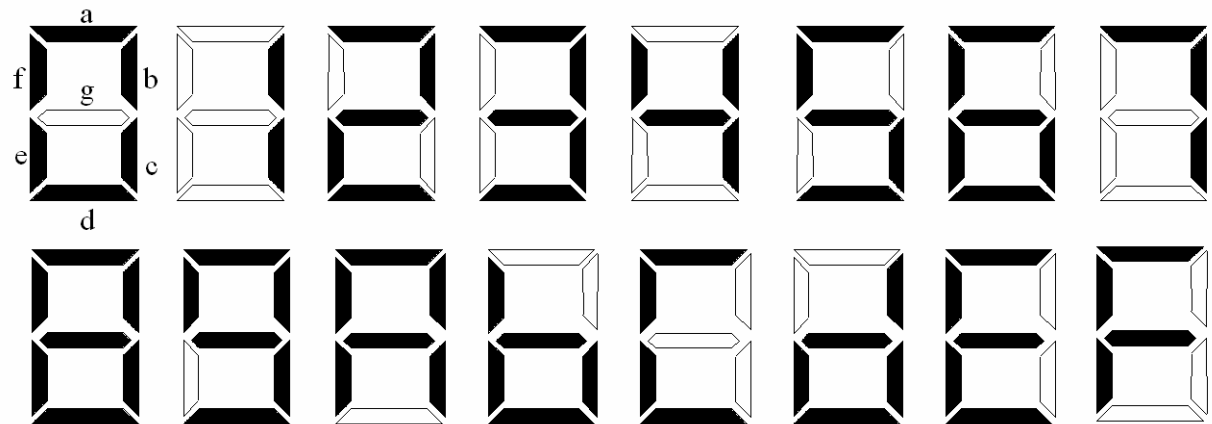
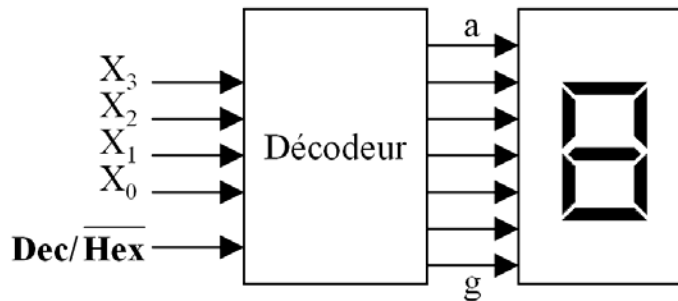
Assignation concurrente

Exercice

Assignation séquentielle

Exercice

Ecrire l'ensemble d'un fichier VHDL (Library, Entity, Architecture) qui décrit un **décodeur binaire / 7 segments**



Exercice

Assignment concurrente

Exercice

Assignation séquentielle

Boucles loop

- Instruction séquentielle
- Forme générale :

For *i* **in** val_deb **to** val_fin **loop** ... **end loop**;

While condition **loop**... **end loop**;

~~**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

Exercice

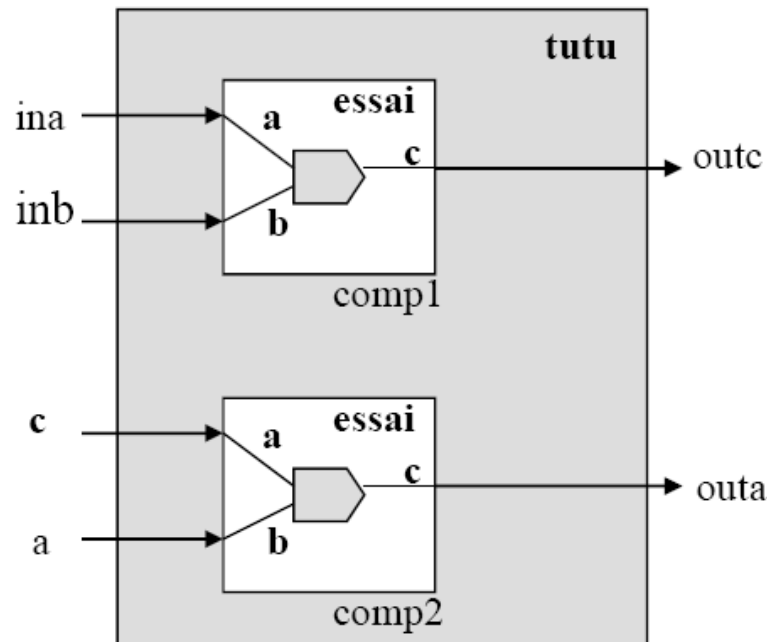
Ecrire l'architecture d'un circuit qui **inverse** bit par bit un bus **data_in** de **8 bits** ($\text{data_out} = \text{not}(\text{data_in})$).

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

Composant

- Description structurelle : comment est réalisée la fonction ?
- Interconnexions entre des composants (**component**)



- Chaque composant a une entité et une architecture propre

Composant

- 3 façon de déclarer un composant :
 - ~~● Toutes les paires entity/architecture sont déclarées dans le même fichier~~
 - La paire entity/architecture de chaque composant est déclarée dans un fichier qui lui est propre
 - La structure du composant est déclarée dans une bibliothèque via un package

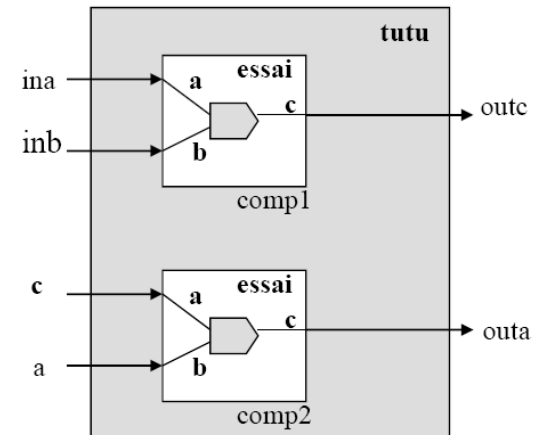
```
package mon_package is  
  component essai ...end component;  
end mon_package;
```

Composant

Mise en oeuvre

- Ecrire l'entité et l'architecture du composant *essai* dans un fichier enregistrer à son nom (*essai.vhd*)
- Dans le fichier du circuit principal *tutu*, déclarer le composant dans l'architecture avant le « begin »

```
architecture arch of tutu is  
  
  component essai  
    port(a,b : in std_logic;  
         c : out std_logic  
        );  
  end component ;  
  
begin  
  ...  
  
end arch ;
```



Composant

Mise en oeuvre

- Après le « begin » de l'architecture, instancier le composant avec **Port Map()** (relier les fils)

```
library ieee ;
use ieee.std_logic_1164.all ;

entity tutu is
port (
    ina, inb, a, c : in std_logic;
    outa, outc      : out std_logic
);
end tutu;

architecture arch of tutu is

component essai
    port(a,b : in std_logic;
         c : out std_logic
    );
end component ;

begin
```

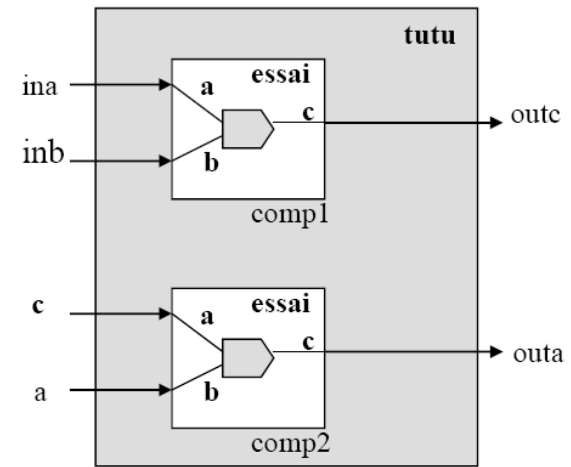
```
    comp1 : essai
```

```
    port map (ina, inb, outc);
```

```
    comp2 : essai
```

```
    port map ( b=>a , c=>outa , a=>c );
```

```
end arch ;
```

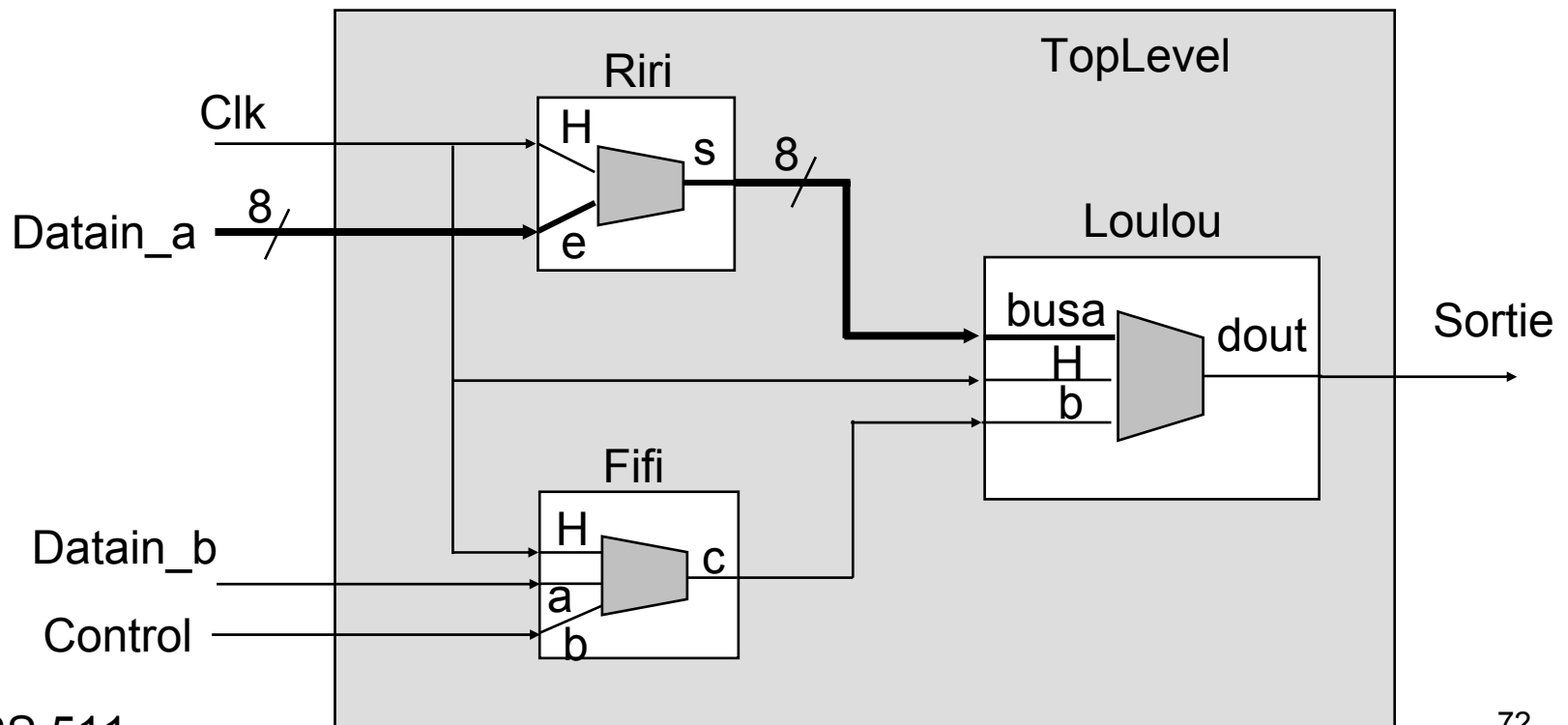


← Instantiation par position

← Instantiation par nomination

Exercice

Ecrire l'architecture du circuit TopLevel, contenant 3 composants Riri, Fifi et Loulou dont les connexions sont schématisées ci-dessous :



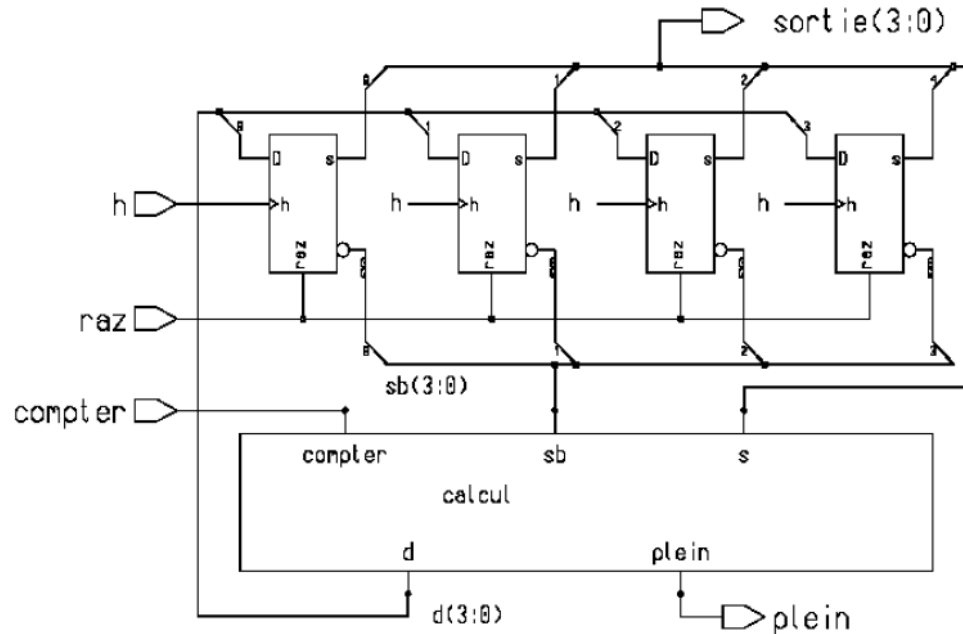
Exercice

Composant

Duplication automatique

- Syntaxe générale :

label : **for** indice **in** val_debut **to** val_fin **generate** ... **end generate** label ;



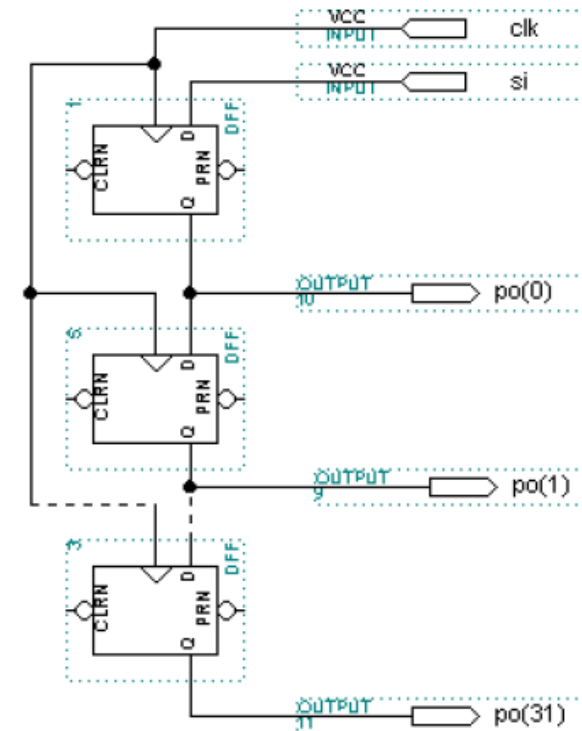
```
implant : for i in 0 to 3 generate
    b : bascule
    port map (h, d(i), raz, s(i), sb(i)) ;
end generate ;
```

Composant

Duplication automatique

- Possibilité d'insérer des conditions dans la boucle
if **condition** then **generate** ... **end generate** ; (pas de else ni de elsif)

```
architecture archi of essai is
  signal temp : std_logic_vector( 0 to 31);
  signal vcc : std_logic ;
begin
  vcc <= '1';
  boucle : for i in 0 to 31 generate
    premier : if i=0 generate
      dffa: DFF port map ( SI,CLK,reset,vcc,temp(0));
    end generate;
    autres : if i>0 generate
      dffb: DFF port map ( temp(i-1),CLK,reset,vcc,temp(i));
    end generate;
  end generate;
  po<=temp;
end archi;
```



Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

Machine à état

- FSM *Finite State Machine*
- Outil pour représenter un **système séquentiel**
- On définit différents **états** dans lesquels peut être le système
- Le passage d'un état à un autre s'effectue si une **condition sur les entrées** est remplie
- Les sorties du système dépendent de **l'état courant** (machine de Moore) ou de **l'état courant et des entrées** (machine de Mealy)
- Dans le cas d'une FSM synchrone, la valeur des entrées est analysée sur front d'horloge

Machine à état

Exemple : détecteur de séquence

La porte ne s'ouvre que si l'on tape la séquence '1' '2' '3'

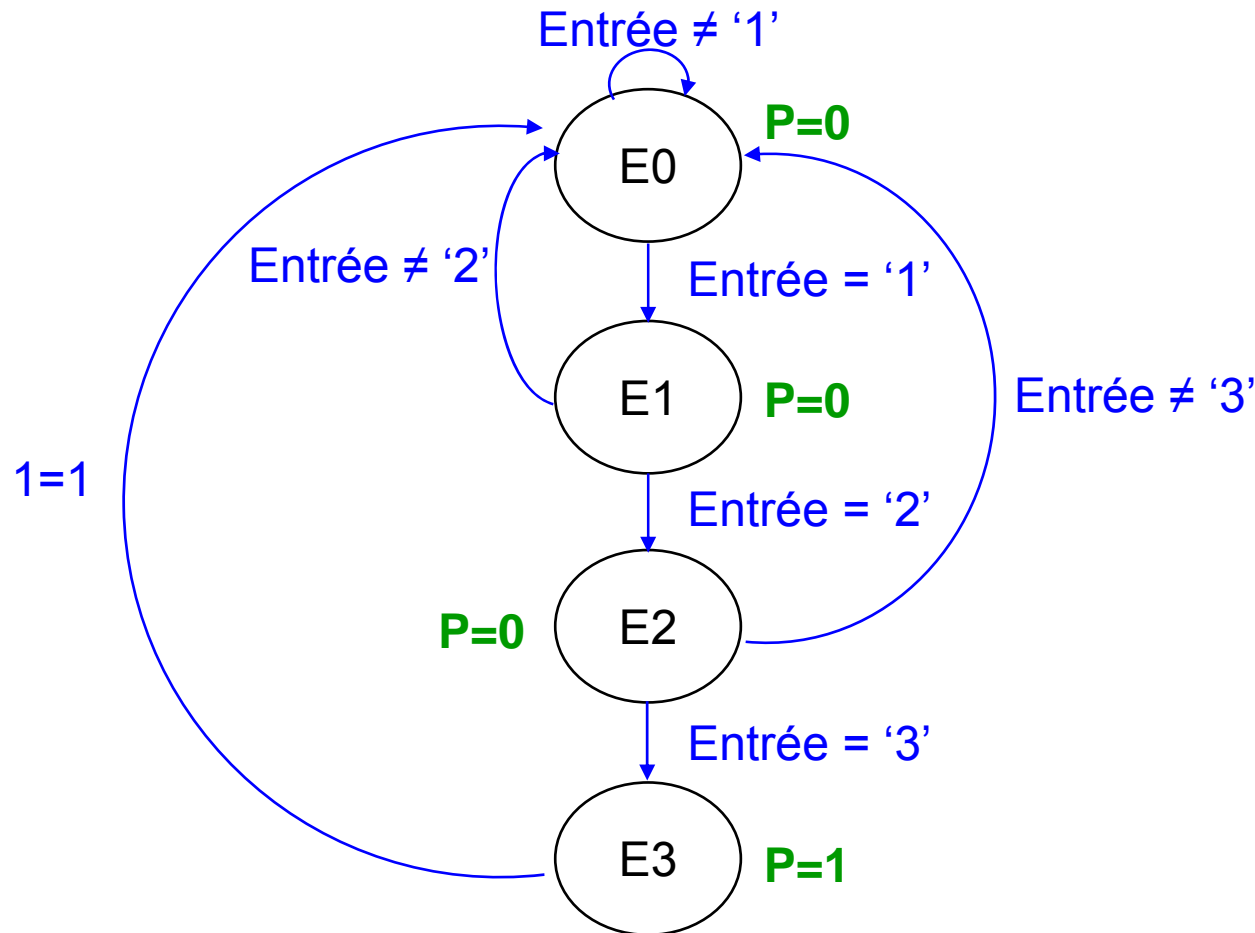
- Etat 0 : le système attend un '1' en entrée, la porte est fermée ($P=0$)
- Etat 1 : le système attend un '2' en entrée, la porte est fermée ($P=0$)
- Etat 2 : le système attend un '3' en entrée, la porte est fermée ($P=0$)
- Etat 3 : la bonne séquence a été entrée, la porte est ouverte ($P=1$)



Machine à état

Exemple : détecteur de séquence

On représente une machine à état par un **graphe d'état**

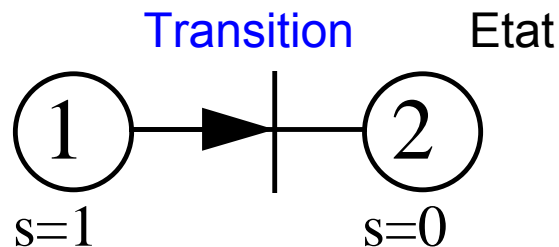


Machine à état

Graphe d'état

Définition :

- **Un diagramme ou graphe d'états** permet d'avoir une représentation graphique d'un système séquentiel.
- Il est constitué par l'énumération de tous les états possible du système.
- Un seul de ces états peut être actif à la fois.
- A chaque état est associé la valeur de la (ou des) grandeur(s) de sortie.

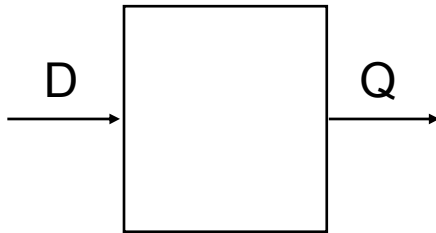


Valeur de la (ou des) sortie(s)

Machine à état

Exercice

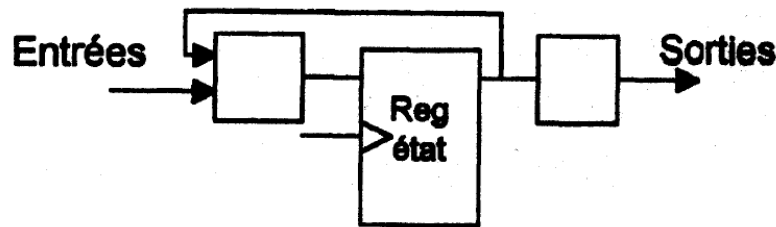
Dessiner le graph d'état d'une bascule D



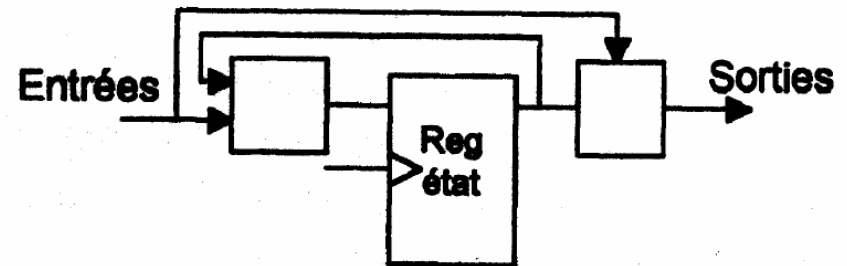
Machine à état

Conception de circuits

- Deux architectures courantes :



Machine de Moore



Machine de Mealy

- Des bascules enregistrent l'état courant
- Des circuits combinatoires sont placés avant et après les bascules pour déterminer l'état suivant et la valeur des sorties

Machine à état

Codage des états

- Codage binaire

Le numéro de l'état est codé en binaire

état 0 = "00", état 1 = "01", état 2 = "10", état 3 = "11"

- One-Hot-One

Chaque état correspond à 1 bit d'un même bus

état 0 \leftrightarrow state = "0001", état 1 \leftrightarrow state = "0010",

état 2 \leftrightarrow state = "0100", état 3 \leftrightarrow state = "1000"

- One-Hot-Zero

Même principe que One-Hot-One mais l'état 0 se code "0000"

Programmer un FPGA par FSM

- Avec le logiciel Quartus, on peut décrire une FSM
 - en utilisant une architecture de Moore ou de Mealy que l'ont fait soit même
 - schématiquement, en rentrant directement le graphe d'état
 - en la décrivant en VHDL

Description de FSM en VHDL

● Entité

```
Entity machine is
port(
  clk : in std_logic;
  bouton : in integer range 0 to 9;
  P : out std_logic);
end machine
```

● Architecture

```
architecture arch of machine is
type StateType is (etat0, etat1, etat2, etat3);
signal etat : StateType;
begin
```

Déclaration d'un nouveau type énuméré
contenant les noms des états

Déclaration d'un signal du
nouveau type juste déclaré

Description de FSM en VHDL

```
begin
```

```
process(clk)
```

```
begin
```

```
if clk'event and clk='1' then
```

```
case etat is
```

```
when etat0 => if bouton = 1 then etat <= etat1;
```

```
else etat <= etat0; end if;
```

```
when etat1 => if bouton = 2 then etat <= etat2;
```

```
else etat <= etat0; end if;
```

```
when etat2 => if bouton = 3 then etat <= etat4;
```

```
else etat <= etat0; end if;
```

```
when etat3 => etat <= etat0;
```

```
end case;
```

```
end if;
```

```
end process;
```

```
P <= '1' when (etat =etat3) else '0';
```

```
end arch;
```

Gestion des états

Gestion de la valeur de la
sortie

Description de FSM en VHDL

Exemples

```
ENTITY machine IS PORT (
  clk, rst : in bit ;
  cond : in bit-vector(2 DOWNTO 0) ;
  sortie: out bit-vector(2 DOWNTO 0)) ;
END machine ;
```

```
ARCHITECTURE arch OF machine IS
  TYPE etats is (etat0, etat1, etat2, etat3, etat4) ;
  SIGNAL etat: etats ;
BEGIN
```

Gestion des états

```
  PROCESS (clk, rst)
  BEGIN
    IF rst='1' THEN
      etat <= etat0 ;
    ELSIF ( clk ' EVENT and clk= ' 1' ) then
      CASE etat IS
        WHEN etat0 => etat <= etat1 ;
        WHEN etat1 => IF cond = "000" THEN etat <= etat0 ;
                        ELSIF cond = "001" THEN etat <= etat4 ;
                        ELSIF cond = "010" THEN etat <= etat3 ;
                        ELSIF cond = "011" THEN etat <= etat2 ;
                        ELSE etat <= etat1 ;
                        END IF ;
        WHEN etat2 => etat <= etat3 ;
        WHEN etat3 => IF cond = "110" THEN etat <= etat0 ;
                        ELSE etat <= etat4 ;
                        END IF ;
        WHEN OTHERS => etat <= etat0 ;
      END CASE;
    END IF;
  END PROCESS;
```

Gestion des sorties

```
  sortie <=  "001" WHEN (etat=etat0) ELSE
              "110" WHEN (etat=etat1) ELSE
              "001" WHEN (etat=etat2) ELSE
              "010" WHEN (etat=etat3) ELSE
              "101" ;
```

```
END ARCH;
```

Description de FSM en VHDL

Exemples

Etats en One-Hot-One

```

ENTITY machine IS PORT (
  clk, rst : in std_logic ;
  cond : in std_logic_vector(2 DOWNTO 0) ;
  sortie: out std_logic_vector(2 DOWNTO 0)) ;
END machine ;
ARCHITECTURE arch OF machine IS
  CONSTANT etat0 : std_logic_vector(4 DOWNTO 0) := "00001";
  CONSTANT etat1 : std_logic_vector(4 DOWNTO 0) := "00010";
  CONSTANT etat2 : std_logic_vector(4 DOWNTO 0) := "00100";
  CONSTANT etat3 : std_logic_vector(4 DOWNTO 0) := "01000";
  CONSTANT etat4 : std_logic_vector(4 DOWNTO 0) := "10000";
  SIGNAL etat: std_logic_vector(4 DOWNTO 0) ;
BEGIN
  -- Gestion des états
  PROCESS(clk, rst)
  BEGIN
    IF rst='1' THEN
      etat <= etat0 ;
    ELSIF ( clk ' EVENT and clk= ' 1' ) then
      CASE etat IS
        WHEN etat0 => etat <= etat1 ;
        WHEN etat1 => IF cond = "000" THEN etat <= etat0 ;
                        ELSIF cond = "001" THEN etat <= etat4 ;
                        ELSIF cond = "010" THEN etat <= etat3 ;
                        ELSIF cond = "011" THEN etat <= etat2 ;
                        ELSE etat <= etat1 ;
        END IF ;
        WHEN etat2 => etat <= etat3 ;
        WHEN etat3 => IF cond = "110" THEN etat <= etat0 ;
                        ELSE etat <= etat4 ;
        END IF ;
        WHEN OTHERS => etat <= "-----" ;
      END CASE;
    END IF;
  END PROCESS;

  -- Gestion des sorties
  sortie <= "001" WHEN (etat=etat0) ELSE
    "110" WHEN (etat=etat1) ELSE
    "001" WHEN (etat=etat2) ELSE
    "010" WHEN (etat=etat3) ELSE
    "101" ;
END ARCH;

```


Description de FSM en VHDL

Exemples : FSM à 2 process

```
Entity machine is
port(
    clk, rst : in std_logic;
    bouton : in integer range 0 to 9;
    P : out std_logic);
end machine

architecture arch of machine is

type StateType is (etat0, etat1, etat2, etat3);
signal etat_courant, etat_suivant : StateType;
```

←
Déclaration de deux
signaux d'état

Description de FSM en VHDL

Exemples : FSM à 2 process

```
begin

    process(etat_courant, bouton)
    begin
        case etat_courant is
            when etat0 => P <= '0';
                        if bouton = 1 then etat_suivant <= etat1;
                        else etat_suivant <= etat0; end if;

            when etat1 => P <= '0';
                        if bouton = 2 then etat_suivant <= etat2;
                        else etat_suivant <= etat0; end if;

            when etat2 => P <= '0';
                        if bouton = 3 then etat_suivant <= etat4;
                        else etat_suivant <= etat0; end if;

            when etat3 => P <= '1';
                        etat_suivant <= etat0;

        end case;
    end process;
```

Premier process pour définir l'état suivant et les valeurs de sortie en fonction de l'état courant

Description de FSM en VHDL

Exemples : FSM à 2 process

Deuxième process qui décrit le passage d'un état à un autre sur les fronts montants d'horloge

```
process (clk)
begin
    if clk'event and clk='1' then
        if rst = 1 then
            etat_courant <= etat0;
        else
            etat_courant <= etat_suivant;
        end if;
    end if;
end process;
end arch;
```

Remarque : Description très proche de la machine de Moore

Description de FSM en VHDL

- Attention au problème de mémorisation implicite en utilisant une syntaxe de type **case**, **when**, ...
- Si la valeur d'un signal n'est spécifiée que dans certains cas, la synthèse produit des bascules non désirées pour mémoriser la valeur du signal dans tous les autres cas.
- Il faut donc affecter **toutes les sorties** dans **toutes les conditions**

Description de FSM en VHDL

Problème de l'état initial

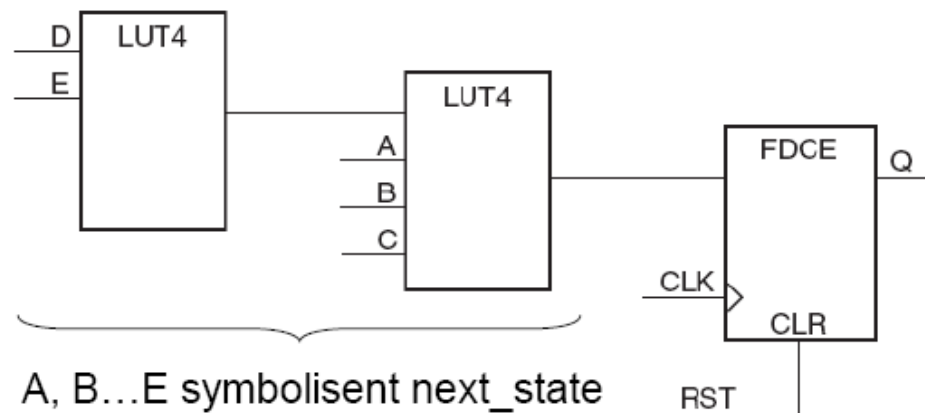
- En simulation, par défaut, la valeur de l'état initiale du système est celle à gauche des états énumérés
`type state is (etat0, etat1, etat2, etat3)`
- Après synthèse, l'état initial peut être n'importe quel état.
- Il faut donc prévoir un **reset** pour forcer la machine à démarrer dans le bon état

Description de FSM en VHDL

Problème de l'état initial

- Reset asynchrone

```
process (CLK, RST)
begin
  if (RST = '1') then
    present_state <= idle;
  elsif (rising_edge(clk)) then
    present_state <= next_state;
  end if;
end process;
```



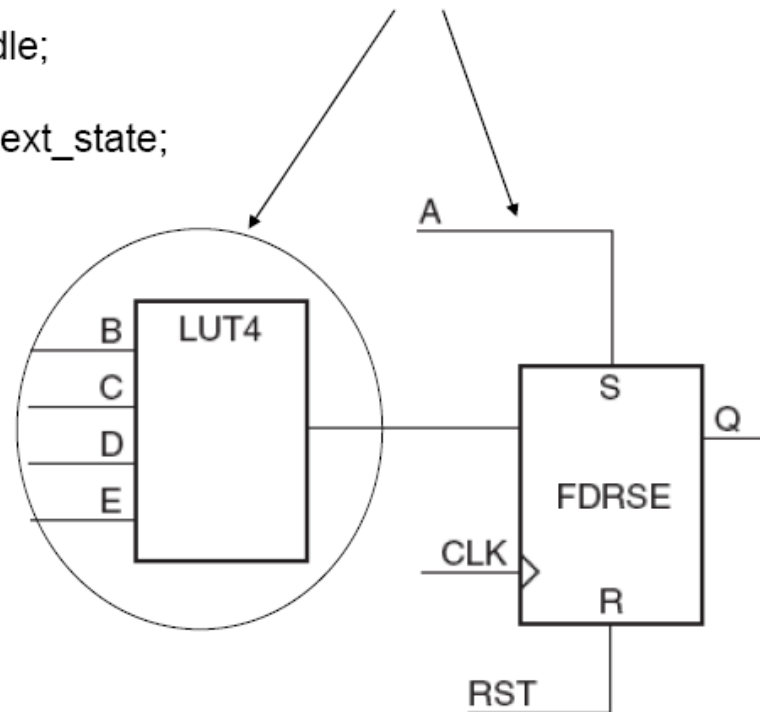
Description de FSM en VHDL

Problème de l'état initial

- Reset synchrone

```
process (CLK)
begin
  if (rising_edge(clk)) then
    if (RST = '1') then
      present_state <= idle;
    else
      present_state <= next_state;
    end if;
  end if;
end process;
```

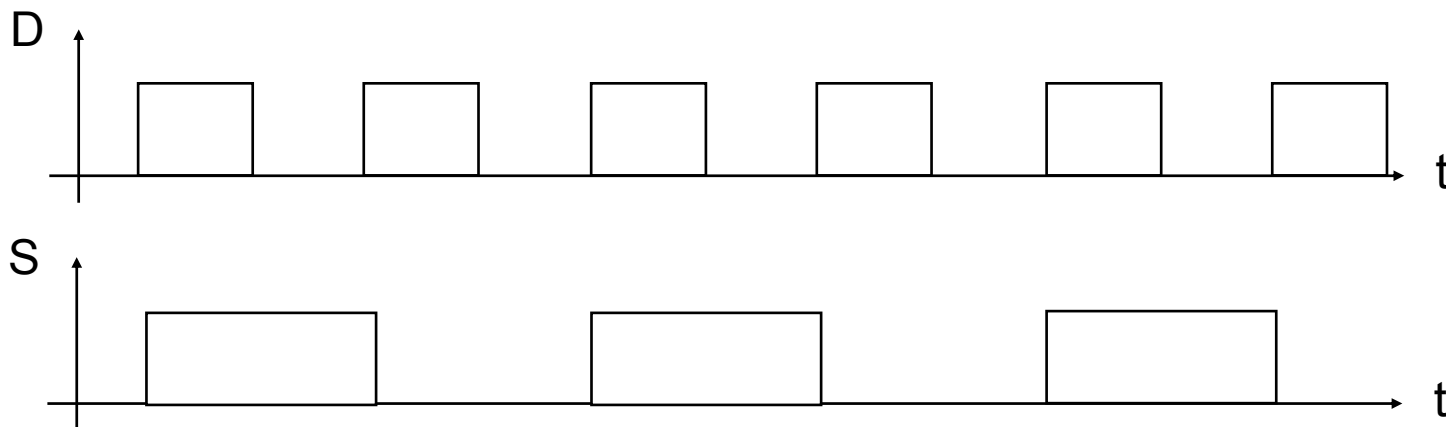
Reset Synchrone
améliore le circuit



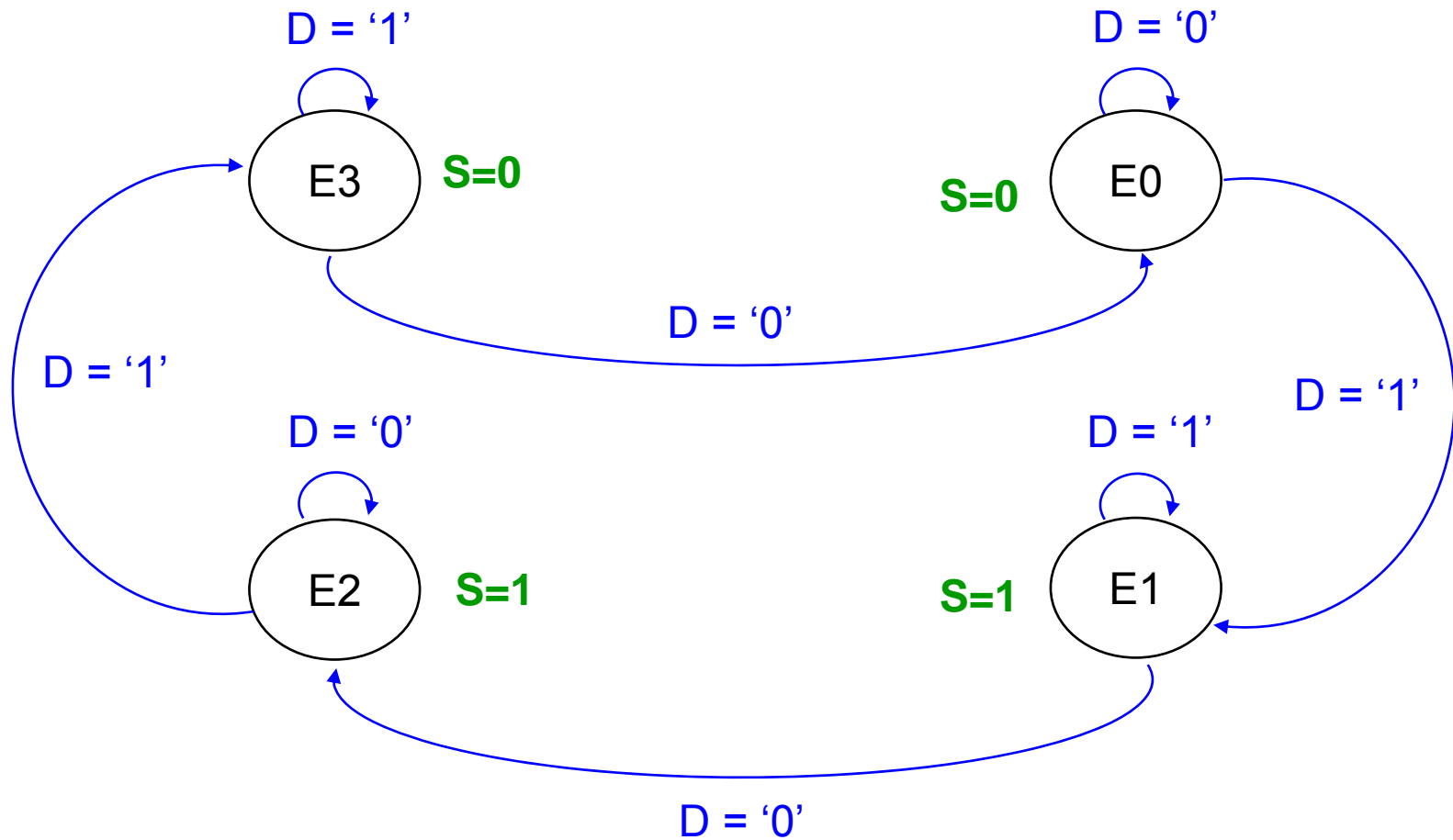
Exercice

Ecrire l'architecture d'une machine à état avec un reset synchrone qui décrit un diviseur de fréquence par 2. On considère que la fréquence d'horloge est beaucoup plus élevée que la fréquence du signal d'entrée D.

- Commencer par représenter le graphe d'état
- Coder la machine en VHDL



Exercise



Exercise

```
architecture A of machine is

    Type TypeEtat is (E0, E1, E2, E3);
    signal etat_courant, etat_suivant : TypeEtat;

begin
    process( etat_courant, D)
    begin
        case etat_courant is
            when E0 => S='0';
                        if D = '1' then etat_suivant <= E1;
                        else etat_suivant <= E0; end if;
            when E1 => S='1';
                        if D = '0' then etat_suivant <= E2;
                        else etat_suivant <= E1; end if;
            when E2 => S='1';
                        if D = '1' then etat_suivant <= E3;
                        else etat_suivant <= E2; end if;
            when E3 => S='0';
                        if D = '0' then etat_suivant <= E0;
                        else etat_suivant <= E3; end if;

        end case;
    end process;

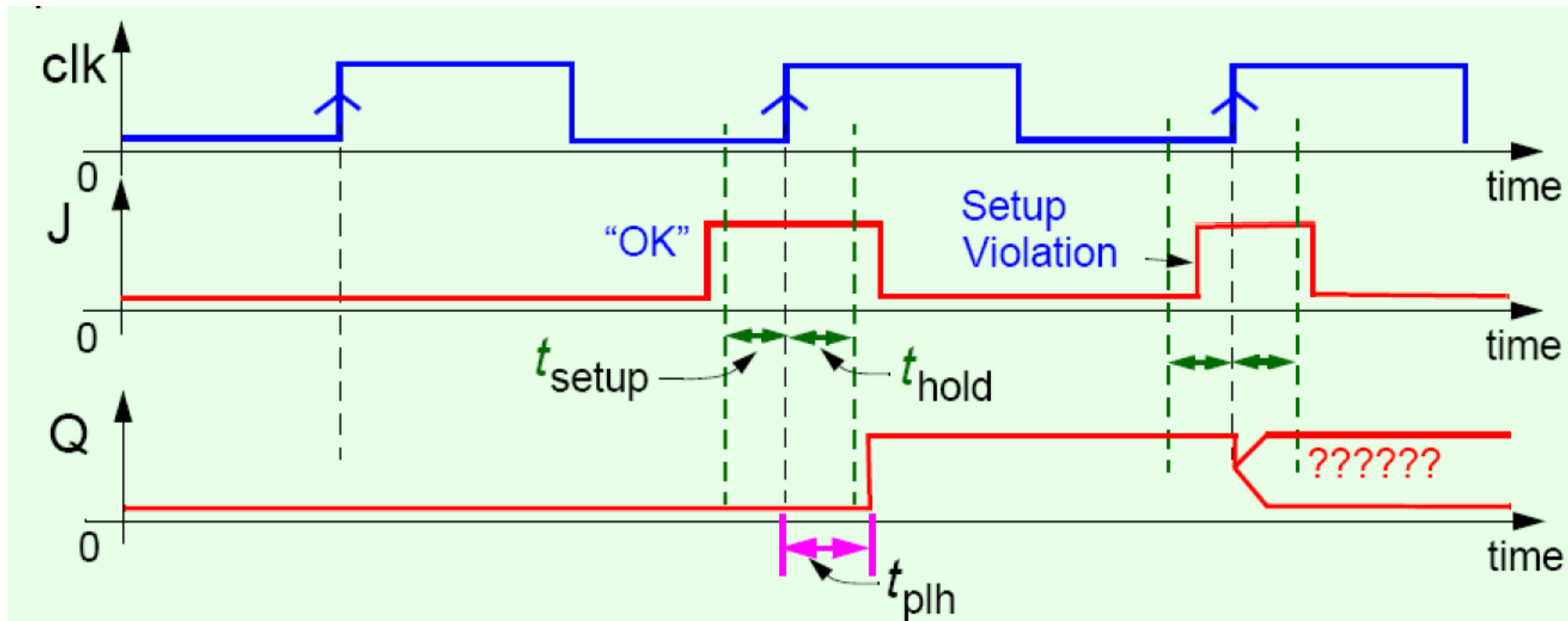
    process (clk)
    begin
        if clk'event and clk='1' then
            if rst = 1 then
                etat_courant <= E0;
            else
                etat_courant <= etat_suivant;
            end if;
        end if;
    end process;
end A;
```

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

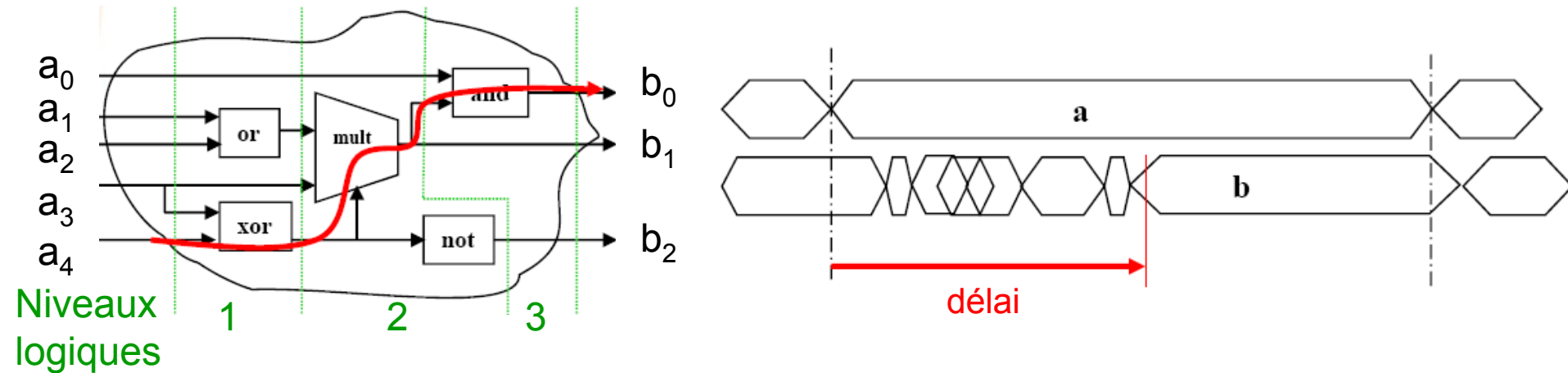
Règles de conception

Timing



Règles de conception

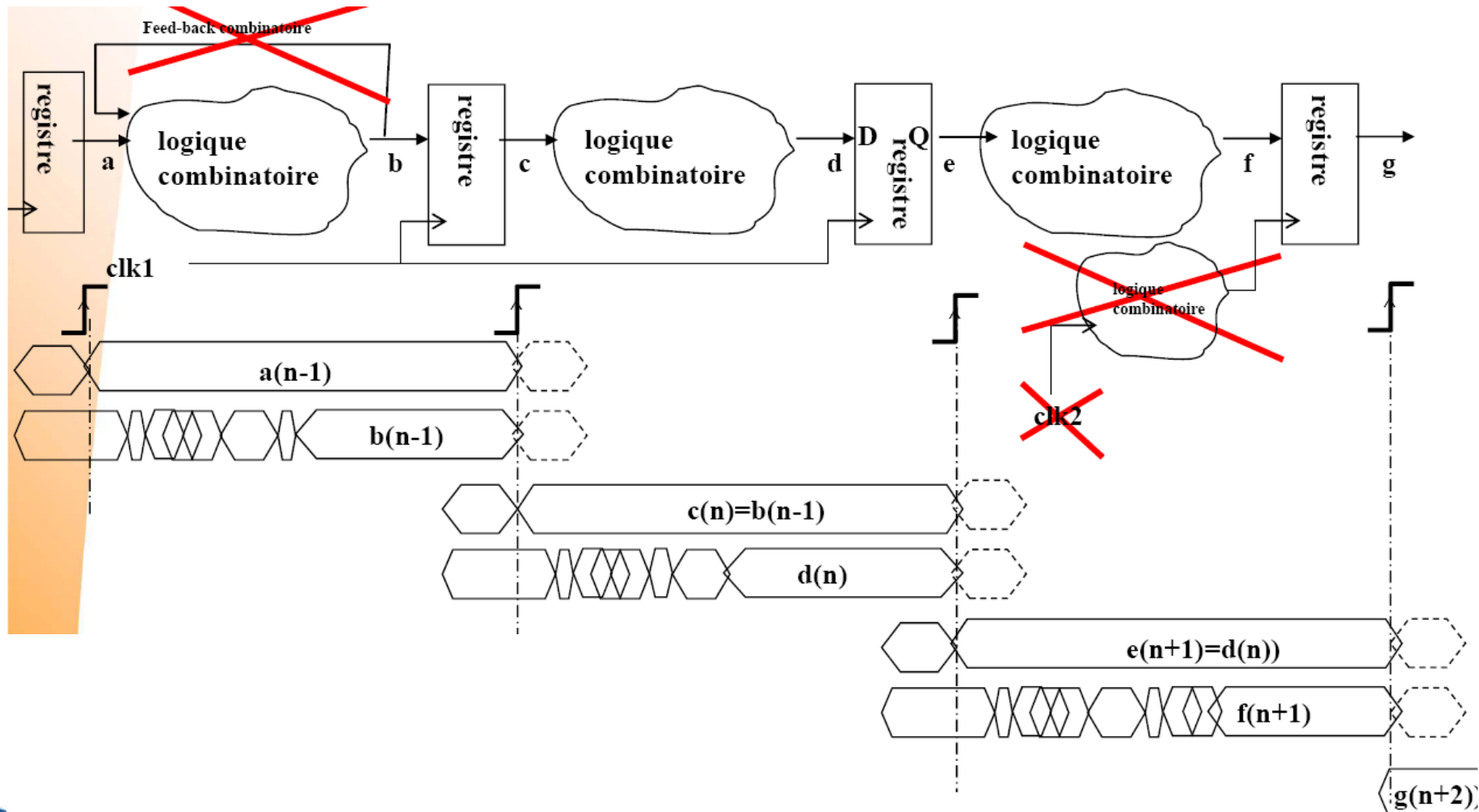
Système combinatoire (asynchrone)



- Plus le nombre de niveaux logiques est grand, plus le délai augmente
- Mauvaise performance en temps
- Difficile à mettre au point
- Difficile de tester tous les cas possibles

Règles de conception

Conception synchrone



Règles de conception

Conception synchrone

- Améliore des performances en vitesse
- Simplifie de la vérification fonctionnelle
- Autorise des analyses statiques du timing
- Assure une parfaite testabilité
- Correspond à l'architecture des composants

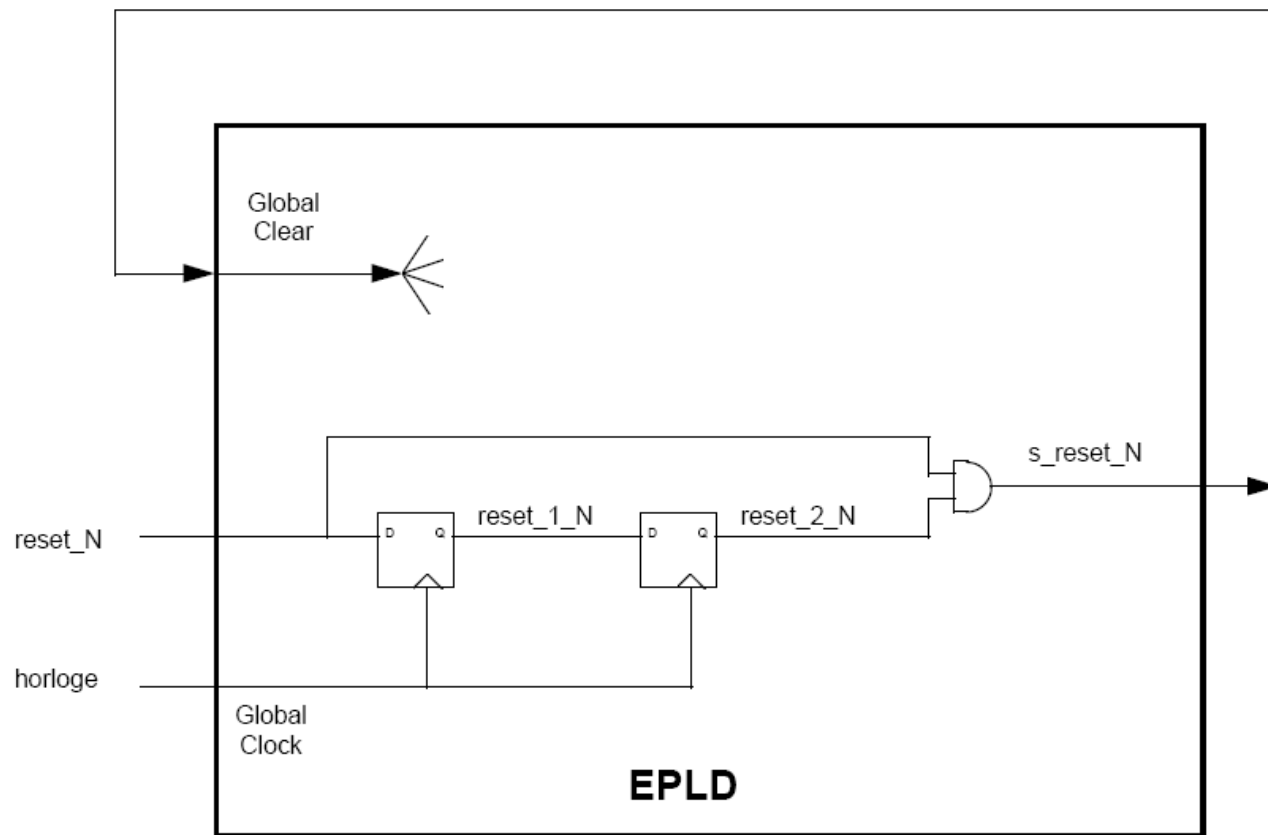
Règles de conception

Règles à suivre :

- 1 seule horloge
- Jamais de logique sur un signal d'horloge (utiliser enable)
- Si plusieurs domaines d'horloge, prévoir des FIFOs tampons
- Resynchroniser tous les signaux asynchrones pour éviter la métastabilité (2 bascules D à la suite)

Règles de conception

Exemple de synchronisation d'un signal reset (actif asynchrone mais inactif de manière synchrone)



Règles de conception

- Adopter une démarche qualité (dénomination, hiérarchisation)
- Attention aux assignations incomplètes (mémorisation implicite)
- Attention à l'utilisation des variables
- Penser à l'implantation (orienter le compilateur)
- Trouver le bon compromis entre ressources et vitesse en choisissant le bon degré de parallélisme

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

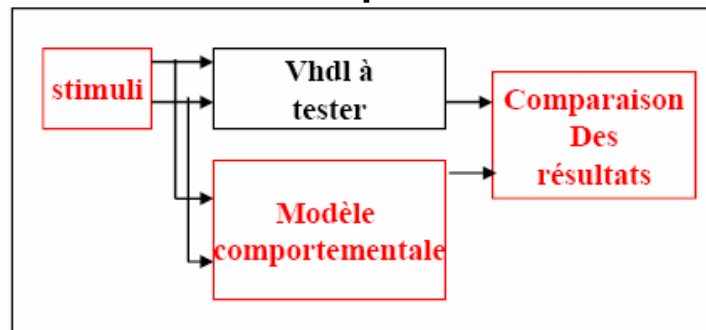
Simulation

Simulation pour valider tout ou une partie du design

- Vérification manuelle sur chronogrammes
 - Fastidieux voir impossible si design complexe
 - Taux de couverture?

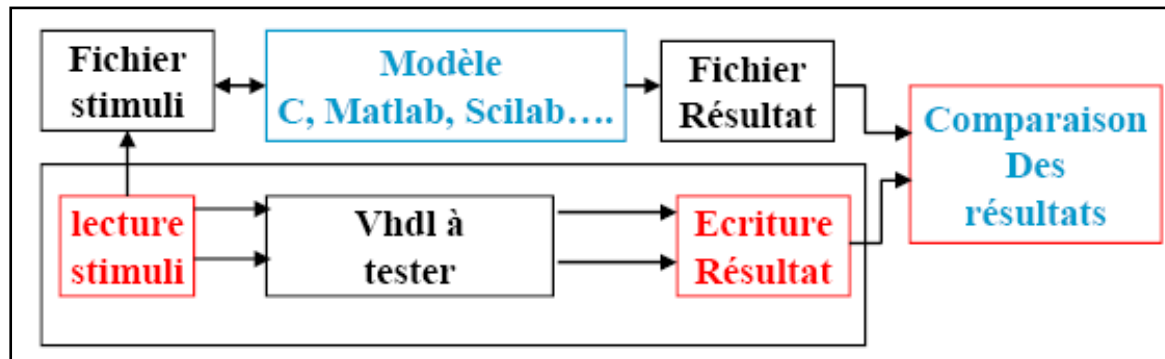


- Vérification automatique :
 - Efficace
 - Mais, validité du modèle comportemental?
 - Vitesse?



Simulation

- Vérification automatique
 - Très efficace




Simulation

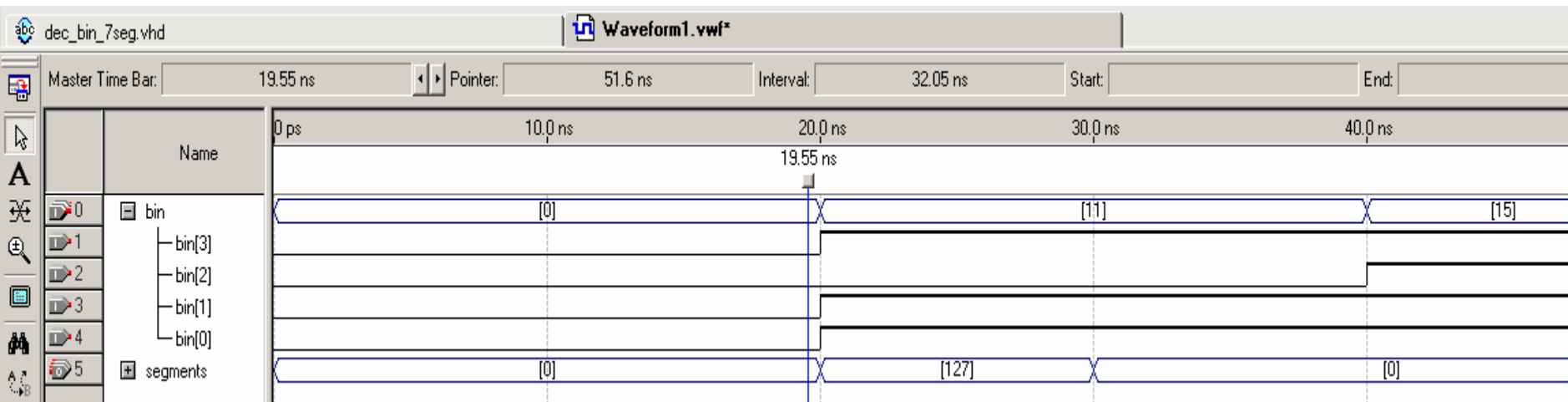
Type de simulation

- Simulation fonctionnelle :
 - Le programme simule un circuit idéal sans temps de propagation
 - Permet de vérifier que les fonctions sont réalisées correctement
- Simulation temporelle :
 - Prise en compte des délais de propagation
 - Permet de vérifier que contraintes temporelles sont respectées (vitesse d'horloge)

Simulation

Ecriture d'un testbench

- Création graphique de chronogrammes
 - Pour Quartus : fichier vector waveform .vwf
 - Generate fonctionnal simulation netlist
 - Lancement de la simulation 
 - Fastidieux, taux de couverture ?



Simulation

Exercice

Tracer les chronogrammes de deux signaux en entrée d'une porte ET pour tester son bon fonctionnement

Simulation

Ecriture d'un testbench en vhdl

- Possibilité d'utiliser des ressources vhdl non synthétisables
- Composition du fichier
 - Library
 - Entity (vide)

```
entity atester_tb is  
end atester_tb;
```

- Architecture
 - Déclaration du composant à tester
 - Définition des signaux et constantes
 - Description du chronogramme des signaux
 - Mapping du composant

Simulation

Ecriture d'un testbench en vhdl

```
library ieee;
use ieee.std_logic_1164.all;

entity atester_tb is
end atester_tb;

architecture vhd of atester_tb is

    component atester
    port (    rst : in std_logic;
            clk : in std_logic;
            a  : in std_logic;
            b  : in std_logic;
            c  : out std_logic);
    end component ;

    signal rst : std_logic := '1';
    signal clk : std_logic := '0';
    signal a : std_logic := '0';
    signal b : std_logic := '0';
    signal c : std_logic;
    constant periode : time := 20 ns;

    Begin

        clk <= not clk after periode/2;
        rst <= '1', '0' after 45 ns;

        A <= .....
        B <= .....

        -- instantiation du composant à tester
        test : atester
        port map (rst => rst, a => a,
                  b => b, c => c );

    end vhd;
```

Simulation

Ecriture d'un testbench en vhdl

La durée s'exprime avec un type physique : fs, ps, ns, us,...

- Affectation d'un signal

Signal <= valeur **after** durée, valeur **after** durée, ... ;

```
S<= "01" after 10 ns, "11" after 30 us ;
```

- Conditions temporelles (dans un process)

wait on liste_signal **until** condition **for** durée;

```
wait for 100 us ;
```

- wait on : attente sur événement
- wait until : attente de conditions
- wait for : attente pour un certain temps

Simulation

Ecriture d'un testbench en vhdl

- Utilisation d'attribut
 - Signal'stable : vrai s'il n'y a pas d'événement pendant la durée
 - Signal'last_event : durée depuis le dernier événement
 - Signal'last_value : avant dernière valeur

Simulation

Génération d'horloge

signal h : bit;

Zone de déclaration de
l'architecture

begin

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

-- zone de déclaration du processus

begin -- début de la zone de définition du processus

h <= '0', '1' **after** 75 ns;

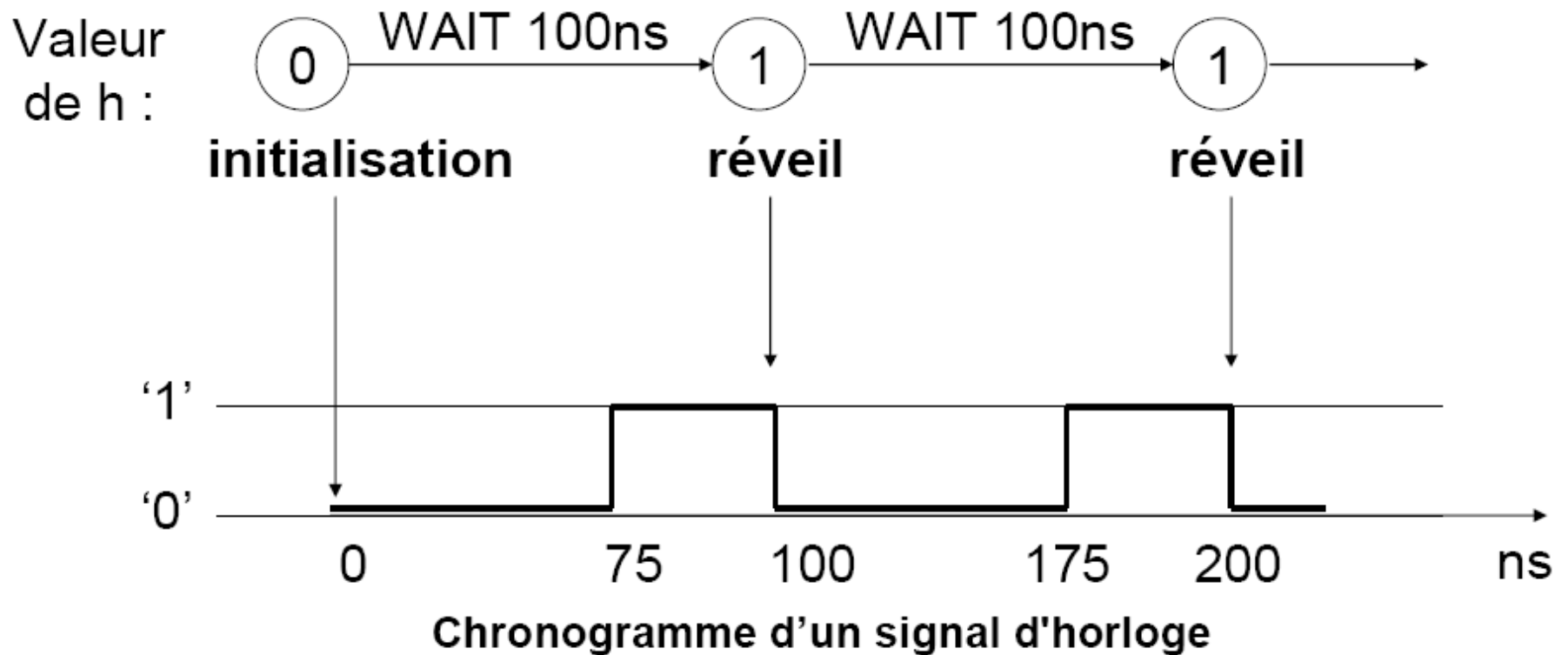
wait for 100 ns; ←

Mise en suspend du
processus pour une
durée déterminée

end process; -- fin du processus

Simulation

Génération d'horloge



Simulation

Génération d'horloge

Autres possibilités :

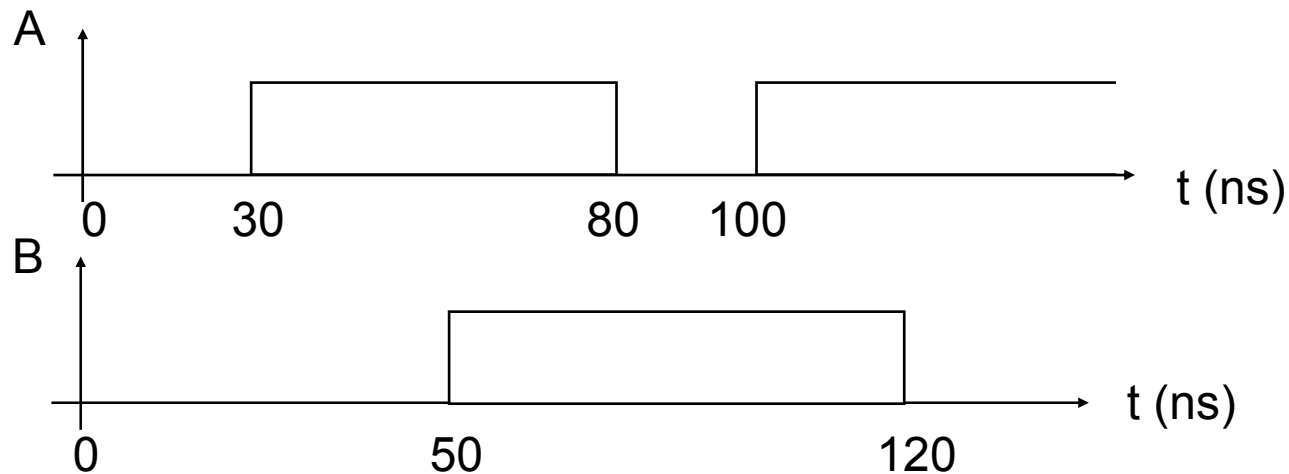
```
constant CLK_PERIOD: time:=40 ns;  
signal clk: std_logic;  
...  
clk_process: process  
begin  
    clk<='0';  
    wait for CLK_PERIOD/2;  
    clk<='1';  
    wait for CLK_PERIOD/2;  
end process;
```

```
process (clk)  
begin  
    wait for clk_periode/2;  
    clk<=not (clk);  
end process;
```

Simulation

Exercice

Ecrire le fichier vhdl permettant de simuler une porte ET ayant deux entrées A et B et une sortie S. Les chronogrammes des signaux A et B doivent suivre les chronogrammes suivants :



Simulation

Exercice

Simulation

Assertions

- Permettent d'avoir des informations dynamiques sur la simulation

assert *test* **report** *message* **severity** *action*

- Si le test est négatif, on affiche *message* avec arrêt ou non de la simulation en fonction de *action*

| Action | Effets | |
|---------|-----------|---------------------------|
| NOTE | poursuite | néant |
| WARNING | poursuite | écriture sur fichier .log |
| ERROR | poursuite | écriture sur écran |
| FAILURE | arrêt | écriture sur fichier |

now = temps de
simulation

Simulation

Assertions

```
architecture comp of dut_tb is
    ...
begin

    -- L'assertion est vérifiée à chaque fois que BCD_obs change
    assert (BCD_obs>9) report "BCD a dépassé sa capacité"
        severity WARNING;

    process test
    begin
        ...
        -- L'assertion n'est vérifiée que lorsque l'exécution du
        -- processus passe à cette instruction
        assert (A_obs=A_ref) report "résultat incorrect pour A"
            severity ERROR;

        ...
        assert (A_obs=15) report "La sortie A ne vaut pas 15"
            severity ERROR;
    end test;
end comp;
```

Simulation

Assertions

- Il est possible d'afficher la valeur d'un signal ou d'une constante *type*'**image**(*signal*)

```
signal entier : integer;  
signal unbit  : std_logic;  
...  
process  
begin  
    ...  
    report "La valeur de entier est" & integer'image(entier);  
  
    report "Le signal unbit vaut: " & std_logic'image(unbit);  
end process;
```

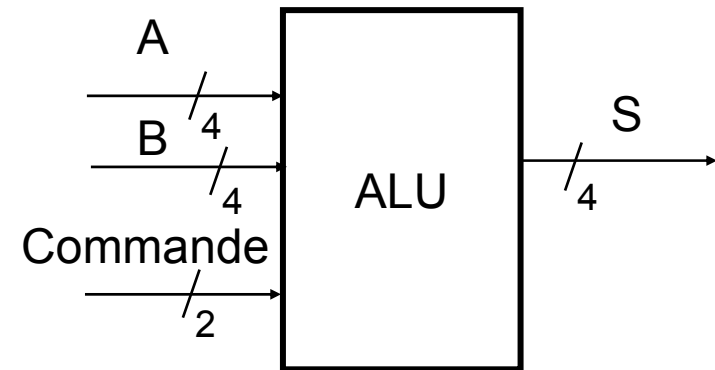
- Attention, pas de fonction prévue pour le type `std_logic_vector`

Simulation

Exercice

Ecrire le fichier vhdl permettant de simuler une ALU qui peut réaliser les opérations suivantes :

- $S = A$ si commande = 00
- $S = B$ si commande = 01
- $S = A + B$ si commande = 10
- $S = A - B$ si commande = 11



Lorsqu'une opération est réalisée, utiliser les assertions pour faire apparaître un message d'erreur s'il y a une erreur de calcul.

A la fin de la simulation, faire apparaître «Fin de simulation».

Simulation

Exercice

```
library ieee ;
use ieee.std_logic_1164.all ;

entity alu_tb is
end alu_tb ;

architecture A of alu_tb is

component ALU
port(a : in integer range 0 to 15;
     b: in integer range 0 to 15;
     s : out integer range 0 to 15;
     commande : in std_logic_vector (1 downto 0));
end component;

signal sig_A : integer := 9;
signal sig_B : integer := 3;
signal sig_S : integer;
signal sig_com : std_logic_vector := "00";
constant add : integer := 12;
constant sous : integer := 6;
```

Simulation

Exercice

```
begin

    report "A=" & integer'image(sig_A);
    report "B=" & integer'image(sig_B);

    process (sig_com)
    begin
        sig_com <= "00";
        assert (sig_S=sig_A) report "résultat incorrect pour S=A. S= " & integer'image(sig_S) severity ERROR;
        wait for 20 ns;

        sig_com <= "01";
        assert (sig_S=sig_B) report "résultat incorrect pour S=B. S= " & integer'image(sig_S) severity ERROR;
        wait for 20 ns;

        sig_com <= "10";
        assert (sig_S=add) report "résultat incorrect pour S=A+B. S= " & integer'image(sig_S) severity ERROR;
        wait for 20 ns;

        sig_com <= "11";
        assert (sig_S=sous) report "résultat incorrect pour S=A-B. S= " & integer'image(sig_S) severity ERROR;
        wait for 20 ns;

        assert (now<80 ns) report "Fin de simulation" severity FAILURE;

    end process;

    ALUtest : ALU port map (sig_A, sig_B, sig_S, sig_com);

end A;
```

Plan global du cours

- I. Introduction
- II. FPGA
- III. VHDL
 - Introduction
 - Règles d'écriture
 - Unités de conception - Objets VHDL - Opérateurs
 - Assignations concurrentes/séquentielles
 - Assignations conditionnelles/sélectives
 - Composant
 - Machine à états
 - Règles de conception
 - Simulation
 - Compléments (fonctions, procédures, packages, ...)

Paramètres génériques

- Permettent de paramétrer des composants
- Se déclarent dans l'**entity**
- On leur donner une valeur par défaut (**:=**)
- L'instanciation se fait avec **generic map (...)**

Paramètres génériques

```
entity mux is
  generic (width : integer :=8);
  port (sel :in std_logic;
        a,b : in std_logic_vector (width-1 downto 0);
        c : out std_logic_vector (width-1 downto 0));
end mux;
```

```
architecture behav of mux is
begin
  c<=a when sel='0' else b;
end behav;
```

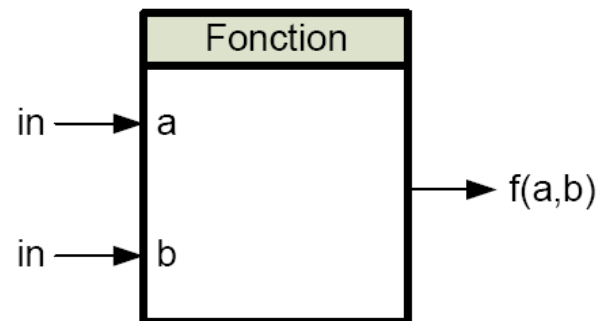
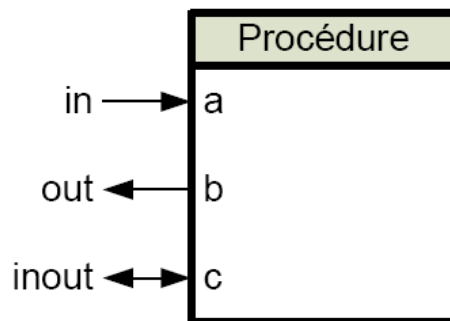
```
comp1 : mux generic map ( 4)
port map(sel=>seldata, a=> dataa, b=> datab, c=>data);
comp2 : mux port map (sel=>seladr, a=>adra, b=> adrb, c=>adr);
```

a, b et c ont 4 bits
de largeur

a, b et c ont 8 bits de largeur (valeur par défaut)

Sous-programmes

- Permettent de modulariser le code
- Permettent de regrouper des actions répétitives
- Permettent de réutiliser des fonctions
- Rendent les fichiers plus lisibles
- Permettent d'automatiser des actions (simulation)
- Fonction et procédure



Fonction

- Les paramètres d'une fonction sont uniquement en entrée
- Une fonction ne retourne qu'une seule valeur
- Elle a un type, celui de la valeur retournée
- Elle doit être déclarée dans un package, une entity ou une architecture

```
architecture ... is
    function ...
begin
    ...
end architecture;
```

```
process ...
    function ...
begin
    ...
end process;
```

- Exemple : conversion de types

Fonction

- Syntaxe :

function *nom* (*paramètre : type*) **return** *type* **is**

Zone de déclaration

Begin

Zone d'instructions

return *valeur*;

end *nom*;

- Exemple :

```
function min(a,b: integer) return integer is  
begin  
    if a<b then  
        return a;  
    else  
        return b;  
    end if;  
end min;
```

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'RIGHT TO temp'LEFT
        LOOP
            IF temp(i) = '1' THEN
                valeur := valeur + (2**i);
            END IF;
        END LOOP;
        RETURN valeur;
    END;
```

Procédure

- Les paramètres d'une procédure peuvent être en entrée, en sortie ou les deux
- Une procédure peut retourner plusieurs valeurs
- Elle doit être déclarée dans un package, une entity ou une architecture

```
architecture ... is
    procedure ...
begin
    ...
end architecture;
```

```
process ...
    procedure ...
begin
    ...
end process;
```

- Elle permet de réduire les lignes de code

Procédure

- Syntaxe :

procedure *nom* (*class paramètre : mode type*) **is**
Zone de déclaration
Begin
Zone d'instructions
end *nom*;

Signal, variable ou constant

in, out ou inout

- Exemple :

```
procedure min(a,b: in integer; c: out integer) is
begin
    if a<b then
        c:=a;
    else
        c:=b;
    end if;
end min;
```


Procédure

Déclaration de la procédure cycle

```
-- Procédure permettant d'attendre plusieurs cycles d'horloge
-- Premier appel termine le cycle précédent si pas complet
procedure cycle (nombre_de_cycles : Integer := 1) is
begin
    for i in 1 to nombre_de_cycles loop
        wait until Falling_Edge(Horloge_s);
        wait for 2 ns; --assigne stimuli 2ns
                        --apres flanc descendant
    end loop;
end cycle;
```

Procédure

Déclaration de la procédure verif

```
procedure Verif(  
    constant Message      : in   String;  
    constant Signal_ref   : in   Std_Logic;  
    constant Signal_Obs   : in   Std_Logic;  
    variable Nbr_Err      : inout Natural  ) is  
begin  
    if Signal_Obs /= Signal_Ref then  
        Nbr_Err := Nbr_Err + 1;  
        report "Erreur verif Signal " & Message  
            severity ERROR;  
    end if;  
end Verif;
```

Procédure

Utilisation de la procédure `verif`

```
process
begin
  ...
  -- Simulation avec une boucle for
  for I in 0 to (2**Nbr_E)-1 loop
    Entrees_Sti <= Std_logic_Vector(To_Unsigned(I,Nbr_E));

    --Calcul de l'etat de la sortie avec un algorithme
    Sortie_Ref <= . . . .

    wait for Pas_Sim/2;
    verif("Test avec boucle for",
          Sortie_ref, Sortie_Obs, Nbr_Err);
    wait for Pas_Sim/2;
  end loop;
  ...
end process;
```

Package

- Un package est une unité de compilation permettant de regrouper constant, type, component, function et procedure

```
package nom_du_package is  
-- description du package  
-- vue externe  
end package nom_du_package;
```

```
package body nom_du_package is  
-- description interne  
end package body nom_du_package;
```

- Un package est compilé à part soit dans une bibliothèque spécifique, soit dans la bibliothèque courante work
- Il doit être compilé avant les composants qui l'utilisent

```
use work.nom_package.all;
```

```
library lib;  
use lib. nom_package.all;
```

Package

```
PACKAGE math IS -- déclaration de l'entête du package
FUNCTION minval (CONSTANT a,b : IN integer)
RETURN integer;
FUNCTION maxval (CONSTANT a,b : IN integer)
RETURN integer;
CONSTANT maxint: integer:=16#FFFF#;
END math;
```

**Liste de ce qu'il y a
dans le package**

```
PACKAGE BODY math IS -- définition du corp du package
FUNCTION minval (CONSTANT a,b : IN integer) RETURN integer IS
BEGIN
    IF a < b THEN
        RETURN a;
    ELSE
        RETURN b;
    END IF;
END minval;
FUNCTION maxval (CONSTANT a,b : IN integer) RETURN integer IS
BEGIN
    IF a > b THEN
        RETURN a;
    ELSE
        RETURN b;
    END IF;
END maxval;
```

**Description de ce
que font les
fonctions ou les
procédures**