VHDL 2 : Instanciation, testbenches et types avancés

Clément Foucher

iut TOULOUSE
Génie Électrique Informatique Industrielle
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE

VHDL

VHSIC Hardware Description Language

COMPOSANTS, TESTBENCHES ET TYPES AVANCÉS

BUT GEII Toulouse S5 ESE

Hiérarchie de composant

INSTANCIATION DE COMPOSANTS

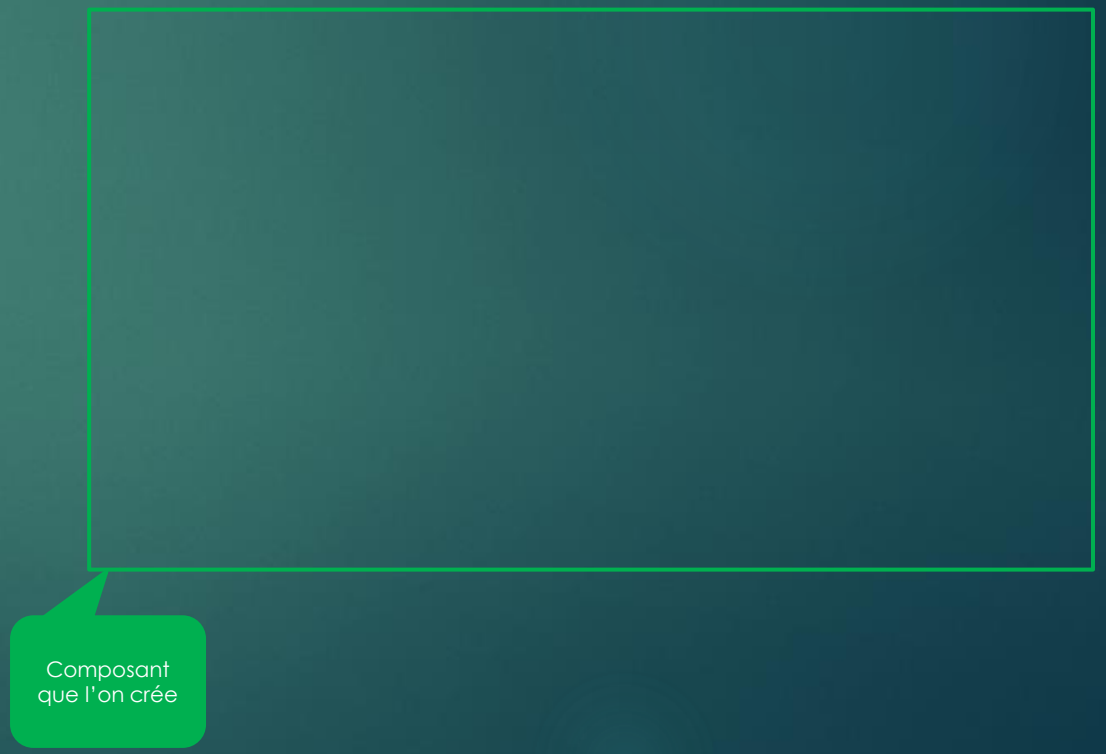
SIMULATION ET TESTBENCHES

TYPES AVANCÉS

Hiérarchie de composant

- Le VHDL (comme le Verilog) raisonne sur la notion de composants
  - On décrit des composants
  - On assemble des composants pour réaliser d'autres composants

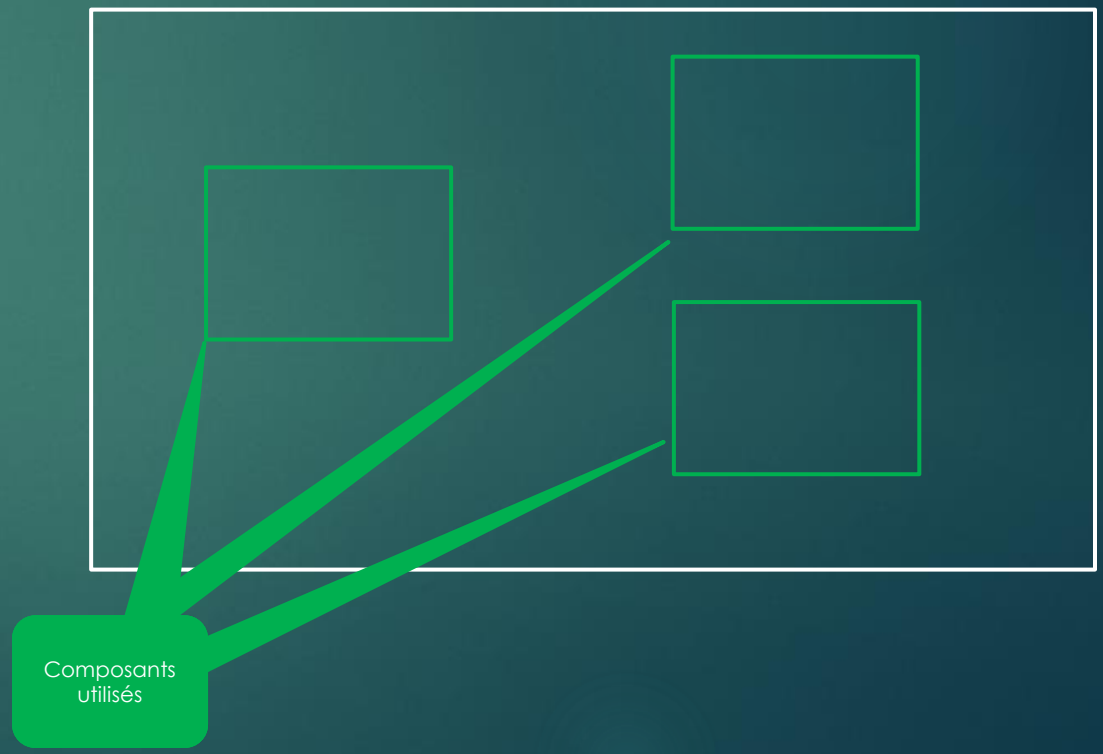
Exemple d'assemblage de composants



Hiérarchie de composant

- Le VHDL (comme le Verilog) raisonne sur la notion de composants
  - On décrit des composants
  - On assemble des composants pour réaliser d'autres composants

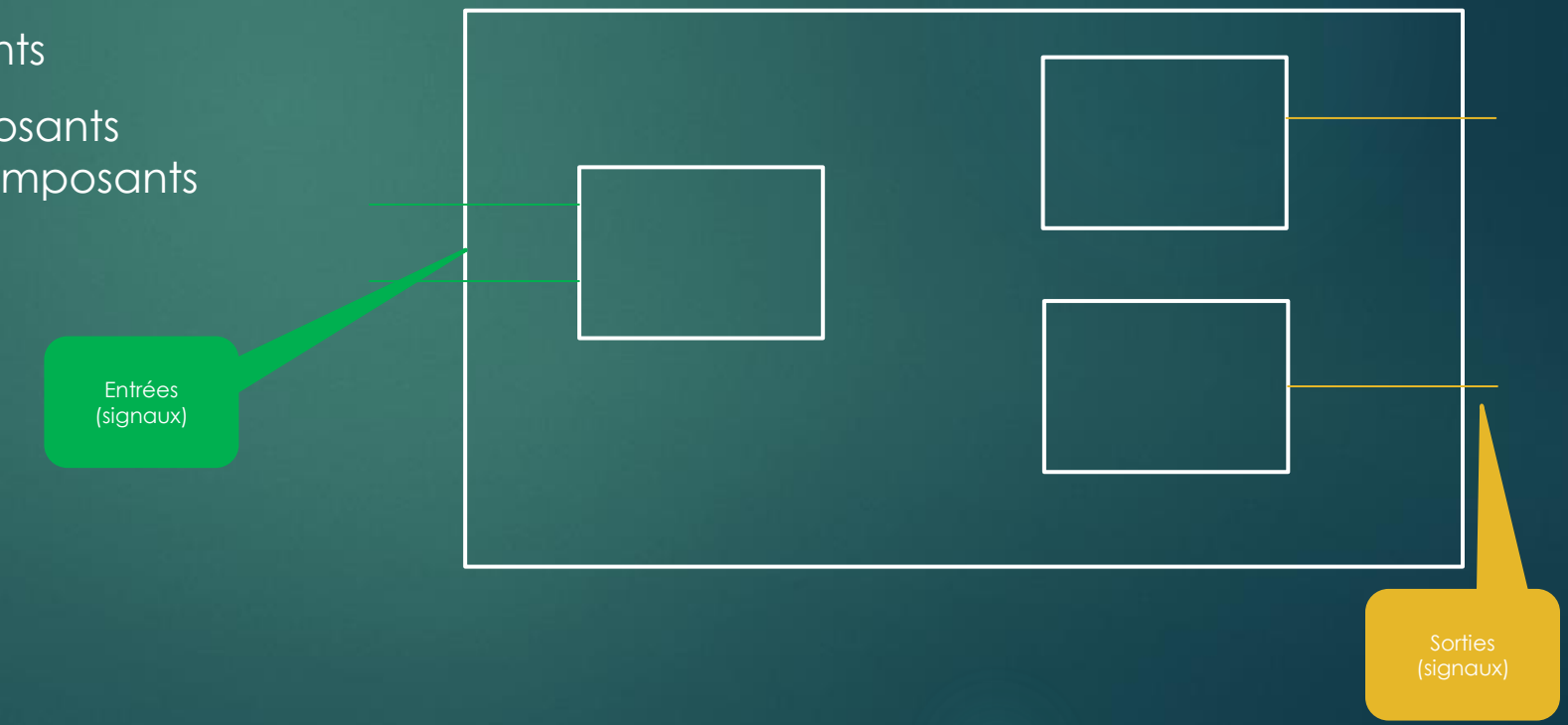
Exemple d'assemblage de composants



Hiérarchie de composant

- Le VHDL (comme le Verilog) raisonne sur la notion de composants
  - On décrit des composants
  - On assemble des composants pour réaliser d'autres composants

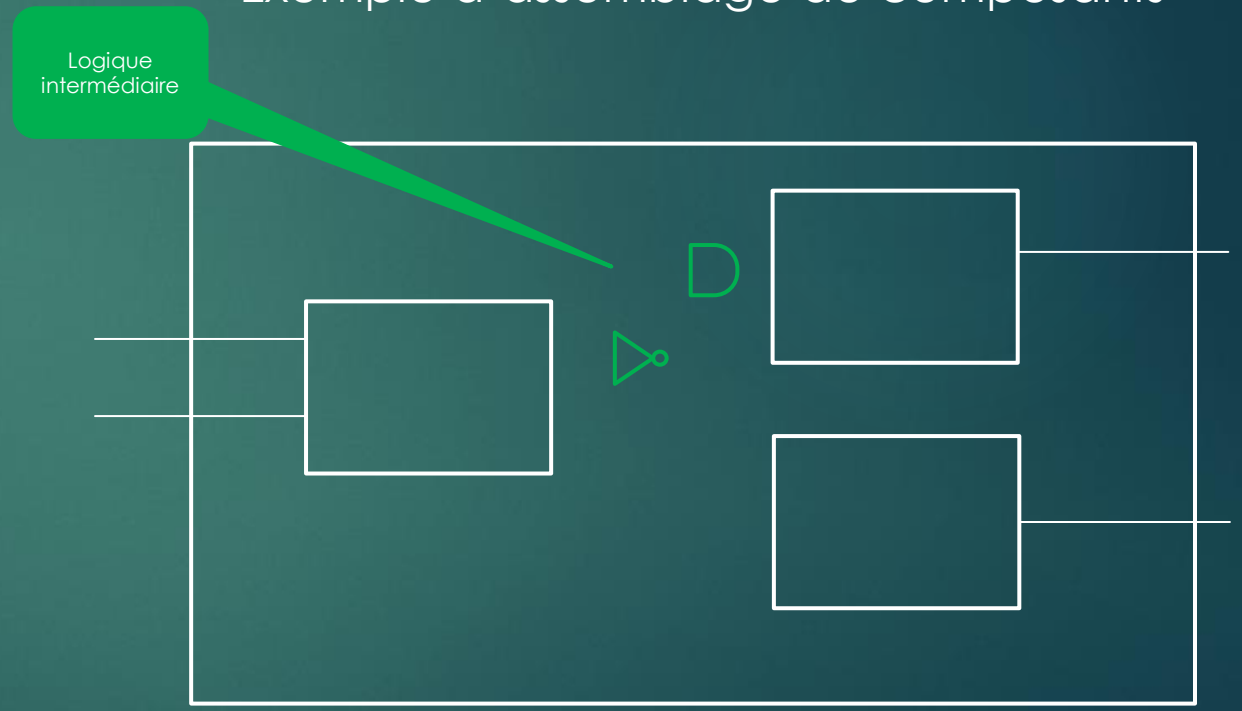
Exemple d'assemblage de composants



Hiérarchie de composant

- Le VHDL (comme le Verilog) raisonne sur la notion de composants
  - On décrit des composants
  - On assemble des composants pour réaliser d'autres composants

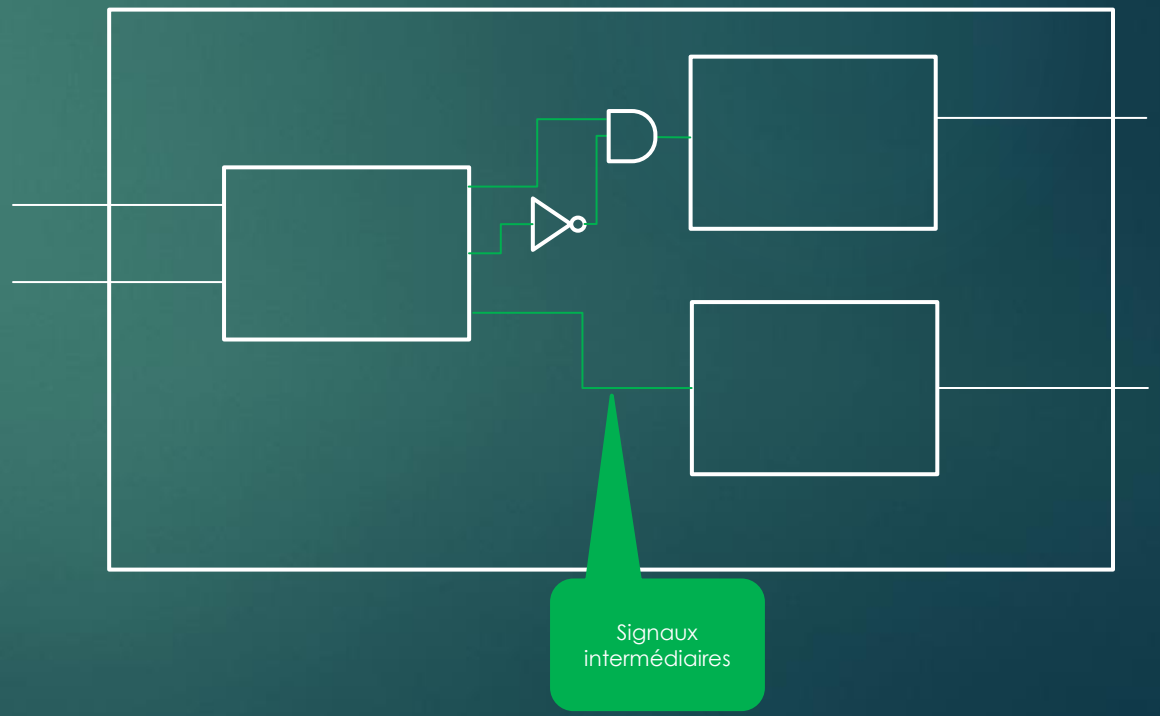
Exemple d'assemblage de composants



Hiérarchie de composant

- Le VHDL (comme le Verilog) raisonne sur la notion de composants
  - On décrit des composants
  - On assemble des composants pour réaliser d'autres composants

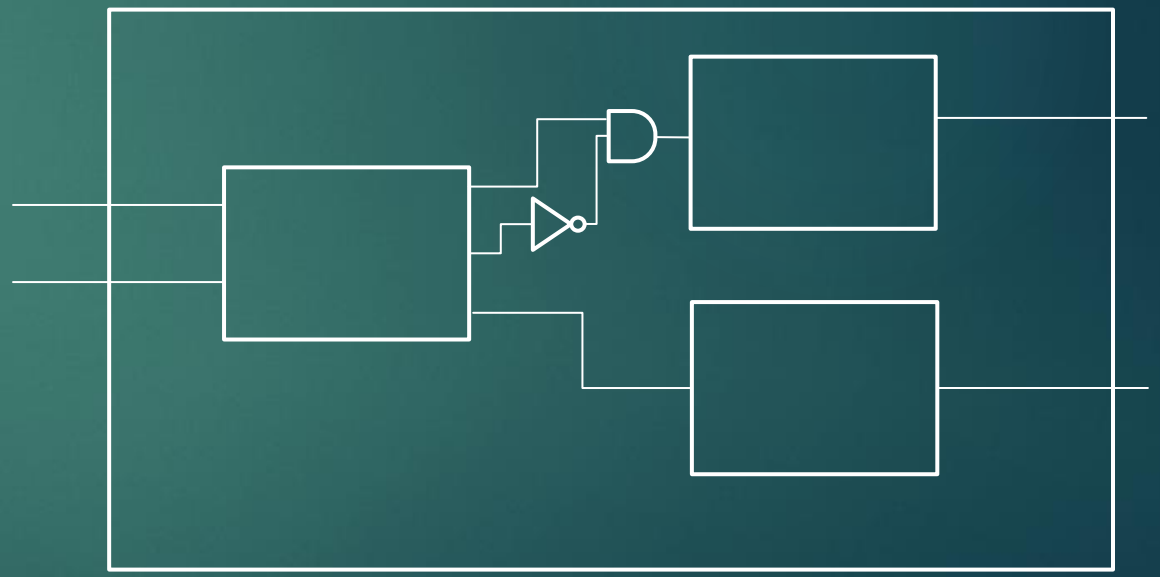
Exemple d'assemblage de composants



Hiérarchie de composant

- Le VHDL (comme le Verilog) raisonne sur la notion de composants
  - On décrit des composants
  - On assemble des composants pour réaliser d'autres composants

Exemple d'assemblage de composants

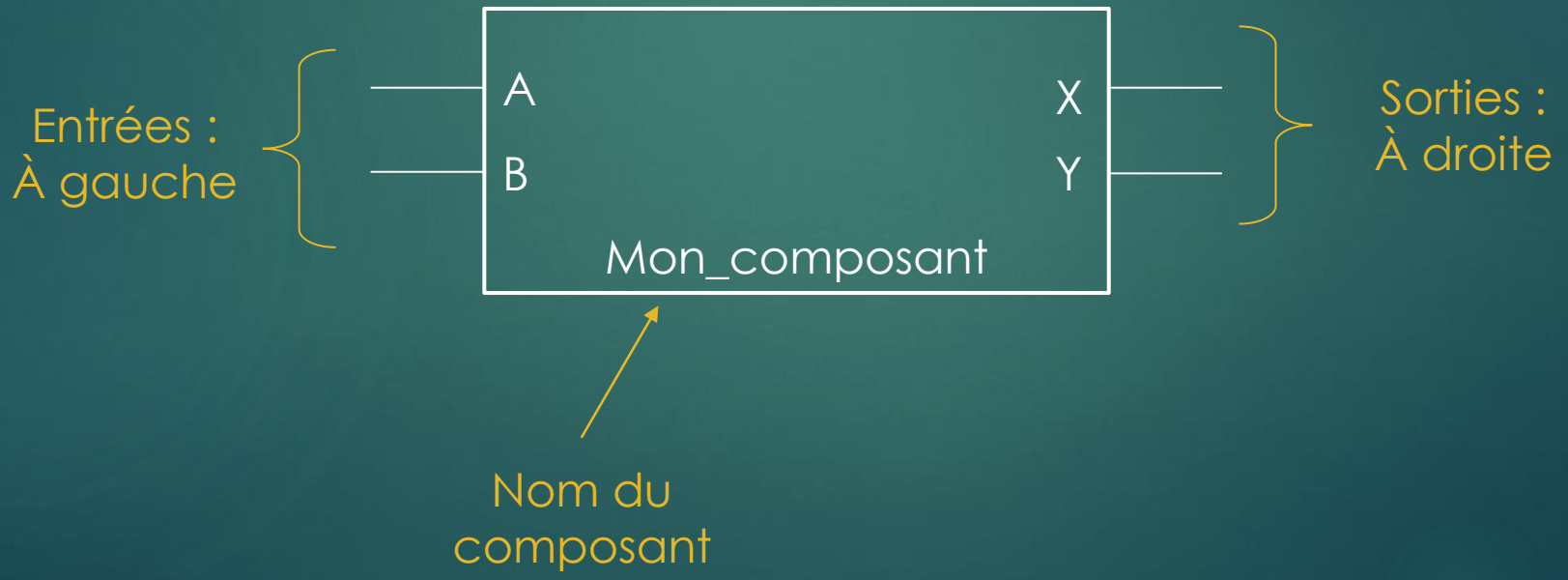


Une fois le composant créé…

- Un composant créé peut être
  - Utilisé directement sur un FPGA
  - Réutilisé dans un autre composant plus complexe
  - Distribué sous forme d'IP

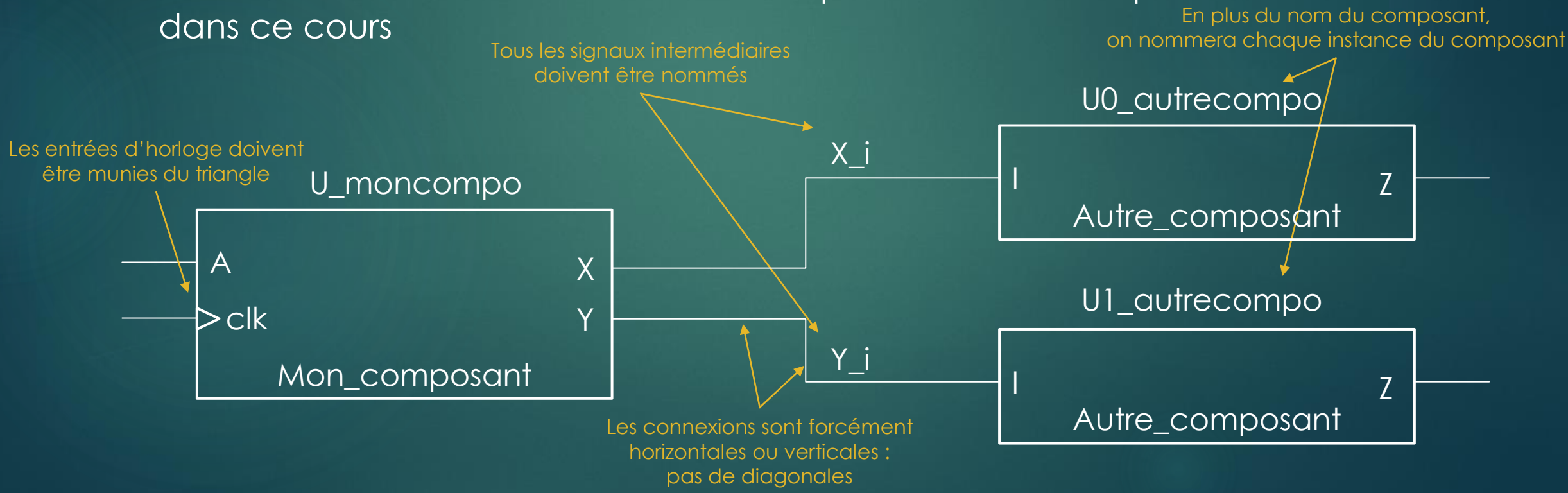
Représentation des composants : la vue externe

- Nous allons avoir besoin de dessiner des schéma d'assemblage de composants
- Pour cela, il faut déjà savoir représenter un composant : c'est ce que l'on appelle la vue externe



Représentation des composants : le schéma-bloc

- Le schéma-bloc est un dessin permettant de représenter la manière dont plusieurs composants sont reliés entre eux
- Nous allons fixer certaines conventions que vous devrez respecter dans ce cours



HIÉRARCHIE DE COMPOSANT

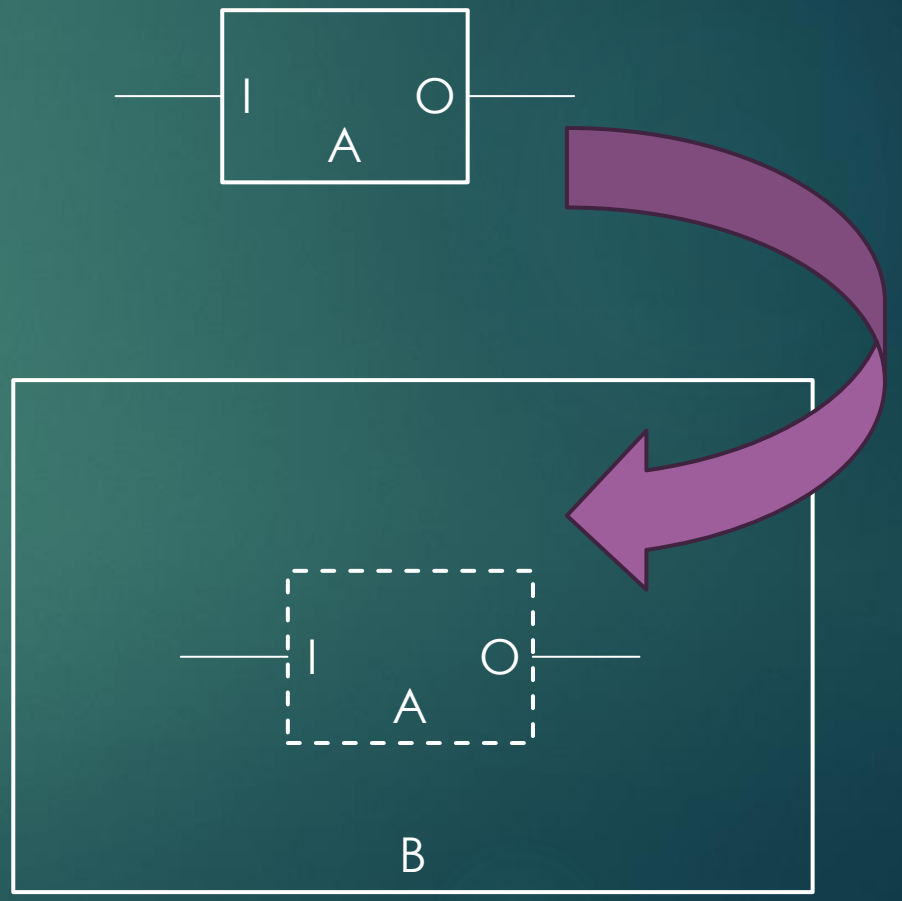
Instanciation de composants

SIMULATION ET TESTBENCHES

TYPES AVANCÉS

Instancier un composant

- On peut réutiliser (instancier) un composant A existant dans un composant B que l'on crée



Instancier un composant

- On peut réutiliser (instancier) un composant A existant dans un composant B que l'on crée
- On utilise l'instruction `port map` pour relier les entrées/sorties du composant A à des signaux du composant B
  - Il est également nécessaire de donner un nom d'instanciation au composant instancié
  - Cela permet notamment de différencier les composants si on en instancie plusieurs

```
architecture ar of B is

    signal entree  : std_logic;
    signal sortie1 : std_logic;
    signal sortie2 : std_logic;

begin

    uA1:entity work.A
      port map(I => entree,
               O => sortie1);

    uA2:entity work.A
      port map(I => entree,
               O => sortie2);

end architecture;
```

Syntaxe de l'instanciation


```
<nom d'instanciation>:entity work.<nom du composant utilisé>
  port map(<liens>);
```

- Les liens doivent avoir la forme suivante :
  - `<nom du port du composant instancié> => <nom du signal à relier>`
  - Notez que :
    - Le port du composant est *toujours* à gauche, que le port soit une entrée ou une sortie !
    - La flèche est *toujours* vers la droite, que le port soit une entrée ou une sortie !
    - Les liens sont séparés par des virgules (et non des « ; » contrairement à l'entité)

- Ne pas confondre :
  - La flèche vers la gauche (`<=`) sert aux *affectations*
    - Elle n'apparait jamais dans un port map
    - Oralement, on la lit « reçoit » ou « prend la valeur »
  - La flèche vers la droite (`=>`) réalise un lien avec un port
    - Elle n'apparait que dans les port map
    - Oralement, on la lit « est connecté à »

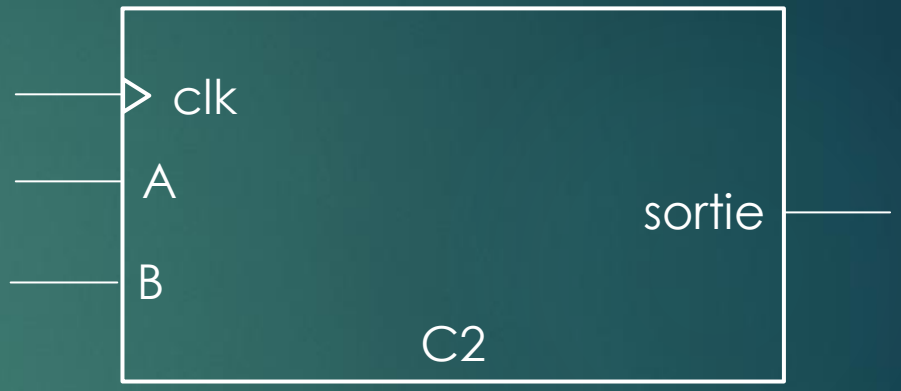
Exercice

- Décrire l'entité du composant C1 en VHDL


```
entity C1 is
    port(clk : in  std_logic;
         en  : in  std_logic;
         S   : out std_logic_vector(1 downto 0)
    );
end entity;
```

Exercice

- Décrire l'entité du composant C2 en VHDL




```
entity C2 is
    port(clk    : in  std_logic;
         A      : in  std_logic;
         B      : in  std_logic;
         sortie : out std_logic
        );
end entity;
```

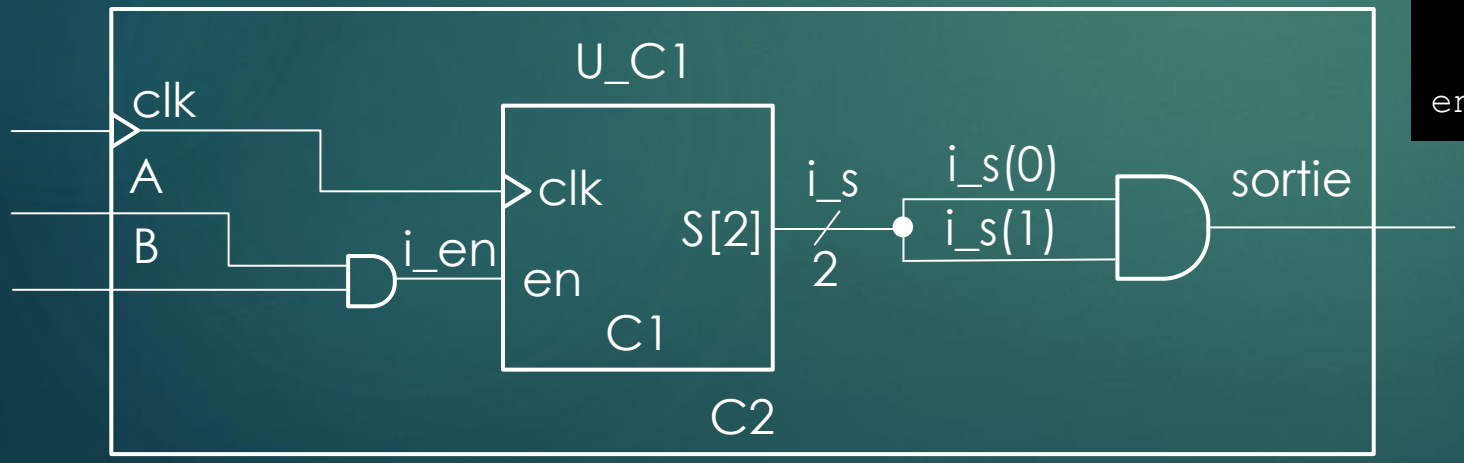
Exercice

```
<nom d'instanciation>:entity work.<nom du composant utilisé>
  port map(<liens>);
```

```
<nom du port du composant instancié> => <nom du signal à relier>
```


- Décrire l'architecture de C2
  - 1) Faire les déclarations de signaux nécessaires
  - 2) Décrire l'instanciation de C1
  - 3) Ajouter le code supplémentaire nécessaire

```
architecture ar of C2 is

    signal i_en : std_logic;
    signal i_s  : std_logic_vector(1 downto 0);

begin

    u_C1:entity work.C1
      port map(clk => clk,
               en  => i_en,
               S   => i_s);

    i_en <= A and B;
    sortie <= i_s(0) and i_s(1);

end architecture;
```



HIÉRARCHIE DE COMPOSANT

INSTANCIATION DE COMPOSANTS

Simulation et Testbenches

TYPES AVANCÉS

Simulation

- Pour simuler, on crée un composant spécial (le testbench) qui va stimuler les entrées du composant à tester
- Dans un testbench il est possible d'utiliser des instructions non synthétisables (càd qui ne pourraient pas être utilisées pour décrire un composant réel)
  - On utilisera notamment des notions de temps qui ne sont pas synthétisables
  - *Rappel : d'où vient le temps normalement dans un composant numérique ?*
    - *L'horloge (clock) est la **seule** notion de temps dans un composant réel, et n'est présente que dans un composant séquentiel*

Structure du testbench

- Un composant testbench n'est pas « réel »
  - Il représente un environnement (banc de test) virtuel dans lequel on plonge le composant à simuler
  - Il n'a donc pas d'entrées/sorties
- Il faut instancier le composant à simuler dans le testbench
- On utilise ensuite un process pour indiquer les actions à réaliser sur les entrées du composant

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


entity testbench is
end entity;

architecture ar of testbench is
begin

    uut:entity work.XXX
      port map(…);

    process
    begin
        (…)
    end process;

end architecture;
```

Le process

- Un process est une suite d'instructions qui permet de décrire séquentiellement ce que l'on souhaite réaliser
  - Attention, dans un composant réel, les instructions ne sont pas *exécutées* séquentiellement (on n'est pas sur un processeur), mais permettent de décrire une certaine logique qui sera convertie en un circuit logique
  - Nous reviendrons plus en détail sur le process dans le prochain cours
- Pour un testbench, nous allons utiliser un process de manière « spéciale » en introduisant l'instruction non synthétisable `wait for`
  - Rappel : il n'est *pas possible* d'utiliser cette instruction dans un composant réel : la notion de temps n'existe pas en dehors de l'horloge

Syntaxe du process

- Un process débute par

  `process`

  - Dans le cas des testbenches, nous n'aurons rien de plus sur la première ligne
- Et se termine par

  `end process;`
- Notez le « ; » qui termine l'instruction `process`
- Le mot-clé `begin` doit être placé au début du process
  - Il est possible de placer certaines déclarations avant le `begin`, mais nous n'en aurons pas besoin dans ce module

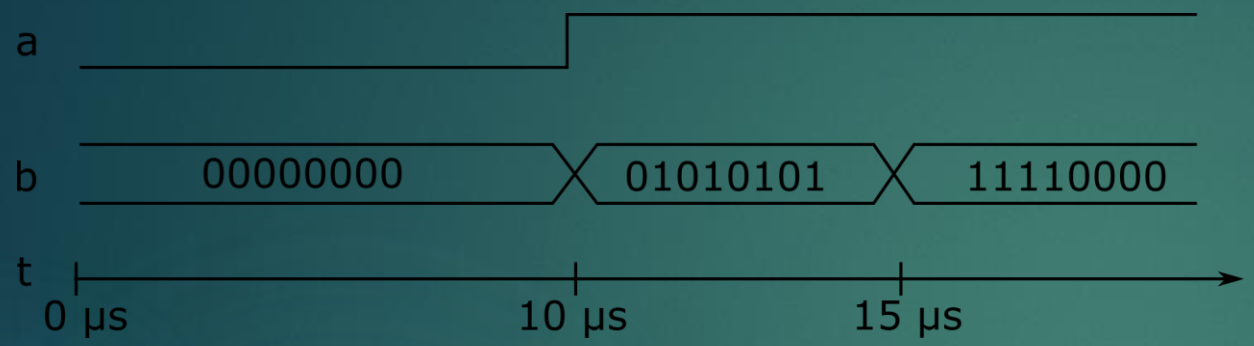
```
process
begin
      (…)
end process;
```

Contenu du process

- Dans le cas d'un testbench, le contenu d'un process est composé d'affectations séparées par des `wait for` et terminées par un `wait`
- Les instuctions placées avant le 1er `wait for` sont les valeurs initiales : *tous* les signaux doivent avoir une valeur initiale
- Par la suite, il n'est pas nécessaire de donner une valeur à tous les signaux à chaque fois, seulement à ceux qui doivent changer de valeur

```
architecture ar of testbench is

    signal a : std_logic;
    signal b : std_logic_vector(7 downto 0);

begin

    uut:entity work.XXX port map(…);

    process
    begin
        a <= '0';
        b <= "00000000";
        wait for 10 us;

        a <= '1';
        b <= "01010101";
        wait for 5 us;

        b <= "11110000";
        wait;

    end process;

end architecture;
```

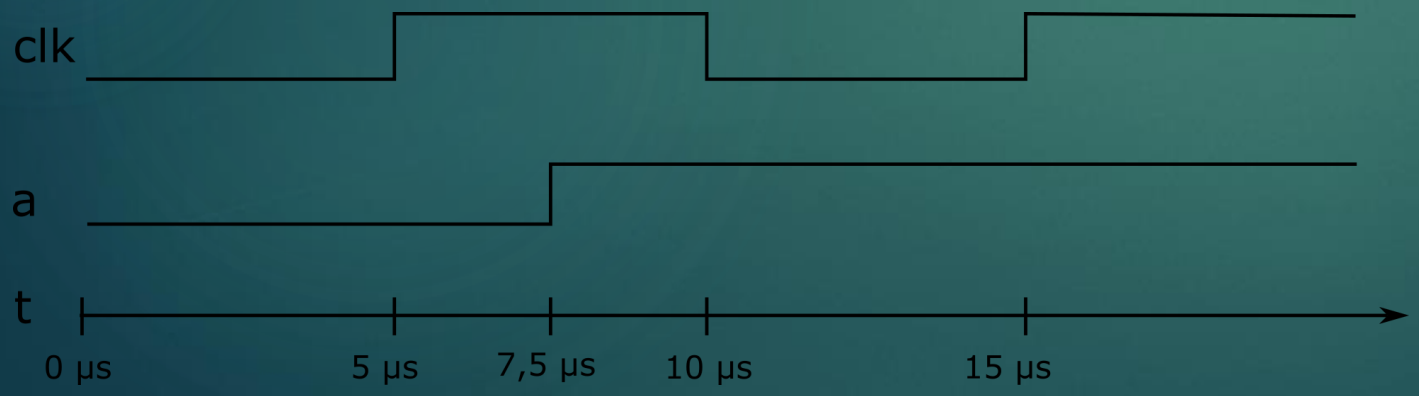
Contenu du process




```
architecture ar of testbench is

    signal a : std_logic;
    signal b : std_logic_vector(7 downto 0);

begin

    uut:entity work.XXX port map(…);

    process
    begin
        a <= '0';
        b <= "00000000";
        wait for 10 us;

        a <= '1';
        b <= "01010101";
        wait for 5 us;

        b <= "11110000";
        wait;

    end process;

end architecture;
```

Cas particulier : signal carré

- Pour les signaux carrés, il peut être fastidieux de répéter le changement de valeur à chaque fois : exemple d'une horloge

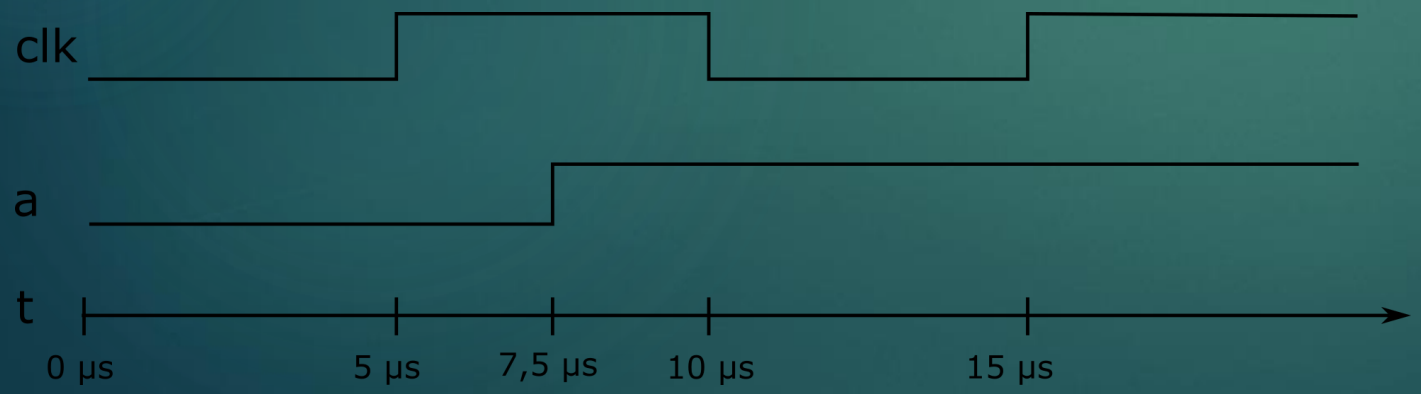



```
architecture ar of testbench is
    signal clk : std_logic;
    signal a   : std_logic;
begin
    process
    begin
        a   <= '0';
        clk <= '0';
        wait for 5 us;

        clk <= '1';
        wait for 2.5 us;

        a   <= '1';
        wait for 2.5 us;

        clk <= '0';
        wait for 5 us;

        clk <= '1';
        wait for 5 us;

        wait;

    end process;
end architecture;
```

Cas particulier : signal carré

- Pour les signaux carrés, il peut être fastidieux de répéter le changement de valeur à chaque fois : exemple d'une horloge
- Pour ces signaux, on préfèrera la syntaxe ci-contre
  - Notez l'initialisation du signal clk !
  - *Question bonus : quelle serait la valeur initiale de clk si on ne l'initialisait pas ?*

```
architecture ar of testbench is
    signal clk : std_logic := '0';
    signal a   : std_logic;
Begin

    clk <= not clk after 5 us;

    process
    begin
        a <= '0';
        wait for 7.5 us;

        a <= '1';
        wait;

    end process;
end architecture;
```



HIÉRARCHIE DE COMPOSANT

INSTANCIATION DE COMPOSANTS

SIMULATION ET TESTBENCHES

Types avancés

Syntaxe avancée pour les `std_logic_vector`

- Il existe une syntaxe permettant d'affecter une valeur à un `std_logic_vector` dans différentes bases, en préfixant la valeur
  - Pour une valeur en hexadécimal, avec le préfixe « x » :
    - `Y <= x"ABCD";`
  - Pour une valeur en binaire, avec le préfixe « b » :
    - `Y <= b"0110010110000111";`
    - Mais m'sieur, quel intérêt ? On sait déjà faire en binaire sans préfixe…
    - Oui, mais cette syntaxe permet d'utiliser des underscores pour la lisibilité :
    - `Y <= b"0110_0101_1000_0111";`

Syntaxe avancée pour les `std_logic_vector`

- On a déjà vu la possibilité d'accéder à un bit individuel, ou à un sous-range, d'un vecteur
  - Par exemple `s(2)` ou `y(4 downto 2)`
- Dans une affectation, il est possible d'utiliser des opérateurs de position pour adresser certains bits
- Par exemple, dans un vecteur `V` de 32 bits, si on souhaite mettre les bits n°1 et n°10 à `'1'`, et tous les autres à `'0'`, la version « brute » serait :
  - `V <= "00000000000000000000001000000010";`
- Même avec la version « lisible », on aurait :
  - `V <= b"0000_0000_0000_0000_0000_0100_0000_0010";`
- Avec l'opérateur positionnel, la syntaxe est la suivante :
  - `V <= (1 => '1', 10 => '1', others => '0');`
  - On voit immédiatement quels bits sont à 1, moins de risque de se tromper de position
- Et on peut même aller plus loin : pour positionner *tous* les bits d'un vecteur à '0', quelle que soit sa taille :
  - `V <= (others => '0');`
  - *Ce sera très utile au S6 pour les génériques (si vous choisissez la coloration informatique embarquée)*

Types `Signed` et `Unsigned`

- Un `std_logic_vector` n'est qu'un vecteur de bits, chacun des bits étant indépendant des autres
  - Il n'est donc pas possible (du moins pas « proprement ») de réaliser des opérations arithmétiques sur ce type
- Les types `Signed` et `Unsigned` sont également des vecteurs de bits, mais qui représentent des valeurs, respectivement signée et non signée
- Ces types définissent les opérations mathématiques usuelles, notamment l'adition et la soustraction
  - En dehors de ça, ils peuvent se manipuler exactement comme des `std_logic_vector`
- Pour pouvoir utiliser ces types, il faut inclure la bibliothèque `numeric_std` en plus de `std_logic_1164`
  - `use IEEE.numeric_std.all;`

Comment utiliser les types `signed` et `unsigned`

- Il est nécessaire de « caster » entre les types
  - En VHDL, cela se fait de la manière ci-contre
- Ensuite, on peut faire des opérations simplement
- On peut ajouter/soustraire
  - des `signed` ou des `unsigned` entre eux
    - Attention, on ne peut pas mixer !
  - Des `signed` ou des `unsigned` avec des `integer`

C'est un `integer` !

Euh… c'est quoi ce 2 ?

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

entity ajouter_deux is
port(inc : in  std_logic;
     I   : in  std_logic_vector(31 downto 0);
     O   : out std_logic_vector(31 downto 0) );
end entity;

architecture ar of ajouter_deux is

    signal i_u : unsigned(31 downto 0);
    signal o_u : unsigned(31 downto 0);

begin

    i_u <= unsigned(i);
    o   <= std_logic_vector(o_u);

    o_u <= i_u + 2 when inc = '1' else
           i_u;

end architecture;
```

Type `integer`

- Le type `integer` est disponible de base, sans appel à une bibliothèque
- Il permet de représenter une valeur numérique « simple »
- C'est utile :
  - Dans des opérations arithmétiques sur les `signed`/`unsigned`
  - Dans des opérations nécessitant un itérateur, par exemple des boucles ou des `generate`
    - On ne verra pas ces deux opérations ce semestre
  - Pour représenter un range dans des composants génériques
    - On ne verra pas la notion de composant générique ce semestre

- Attention, quelques restrictions
- Un `integer` est représenté matériellement par un vecteur de 32 bits, et possède un signe
  - Il ne peut donc varier qu'entre -2^{32} et $2^{32}-1$
- Si on souhaite déclarer un signal avec le type `integer`, il est souhaitable de déclarer sa plage de variation
  - `signal s : integer range 0 to 255;`
  - Cela permet au synthétiseur, de déterminer le nombre de bits à utiliser pour représenter le signal, et ne pas utiliser de bits inutiles
  - La plage définie ne peut pas être plus importante que la plage par défaut
- Pour transformer un `signed`/`unsigned` en `integer`, il faut utiliser des opérations de conversion permettant de préciser le nombre de bits
  - Par exemple : `integer` vers `signed`
    - `signed_sig <= to_signed(integer_sig, <taille>);`
  - Dans l'autre sens : `signed` vers `integer`
    - `integer_sig <= to_integer(signed(std_logic_sig));`
  - Pour caster depuis/vers `std_logic_vector`, il faut passer par `signed` ou `unsigned`

Bonus

Instanciation : autres méthodes

- Syntaxe complète de l'instanciation,
  - Notamment utile dans le cas où il y a plusieurs architectures et qu'on souhaite indiquer laquelle on utilise

```
<nom d'instanciation>:entity work.<nom du composant utilisé>(<nom de l'architecture utilisée>)
  port map(<liens>);
```

Instanciation : autres méthodes

- Il existe deux autres méthodes d'instanciation
  - En utilisant un `component` dans un `package`
  - En utilisant un `component` directement dans l'`architecture`
- Ces deux méthodes permettent
  - De simplifier la syntaxe de l'instanciation
    - `<nom>:entity work.<composant>(<archi>) port map`
    - Devient alors simplement
    - `<nom>:<composant> port map`
  - D'inclure dans un composant VHDL un composant écrit dans un autre langage (notamment, un composant Verilog)
- Mais elle nécessite une étape supplémentaire par rapport à la méthode vue dans le cours

Instanciation avec un package

- Pour réutiliser le composant A dans un composant B
- 1) On crée un fichier package, et on copie-colle l'entité de A dans le package, en remplaçant « `entity` » par « `component` »
  - On peut placer tous ses composants dans le même package
- 2) On fait appel au package dans le fichier du composant B avec la syntaxe ci-contre
  - Tous les fichiers compilés appartiennent par défaut à la bibliothèque `work`

Fichier a.vhd

```
entity A is
port(I : in  std_logic;
     O : out std_logic
    );
end entity;

architecture ar of A is
begin
    (…)
end architecture;
```

Fichier mon_package.vhd

```
package mon_package is

    component A is
    port(I : in  std_logic;
         O : out std_logic
        );
    end component;

end package;
```

Fichier b.vhd

```
library work;
use work.mon_package.all;
```

Instanciation directe dans l'architecture

- Cette méthode consiste à faire la même chose que l'instanciation avec package, mais en plaçant le composant directement avant le `begin` de l'architecture du composant B
- Avantage : plus besoin de package
- Inconvénient : si on réutilise un composant dans plusieurs autres composants, il faut faire la copie plusieurs fois, et penser à maintenir à jour toutes les copies si on modifie l'interface du composant réutilisé

Ressources

- Les types en VHDL
- Opérations de conversion entre les types