Le langage VHDL

Eduardo Sanchez
EPFL

- Livres conseillés:
  - John F. Wakerly
    Digital design (4th edition)
    Prentice Hall, 2005
  - Peter J. Ashenden
    The designer's guide to VHDL (3rd edition)
    Morgan Kaufmann, 2008
  - Peter J. Ashenden
    The student's guide to VHDL (2nd edition)
    Morgan Kaufmann, 2008
  - James R. Armstrong – F. Gail Gray
    VHDL design: Representation and synthesis (2nd edition)
    Prentice Hall, 2000
  - Jacques Weber – Maurice Meaudre
    Le langage VHDL: Du langage au circuit, du circuit au langage
    Masson, 2007
  - Roland Airiau – Jean-Michel Bergé – Vincent Olive – Jacques Rouillard
    VHDL: Langage, modélisation, synthèse (3ème édition)
    PPUR, 2003

VHDL

VHSIC
Very High-Speed Integrated Circuits

Hardware Description Language

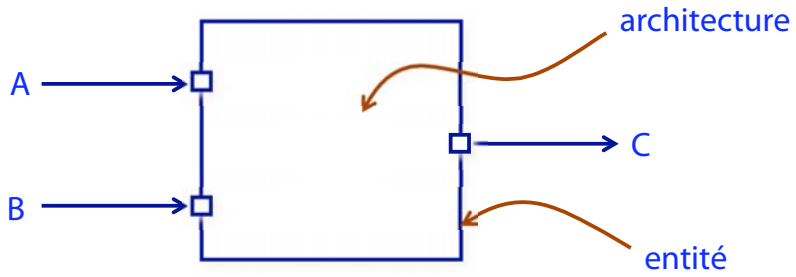
- Langage formel pour la spécification des systèmes digitaux, aussi bien au niveau comportemental que structurel
- Utilisation:
  - description des systèmes
  - simulation
  - aide à la conception
  - documentation
- Caractéristiques principales:
  - description à plusieurs niveaux
  - simulation activée par événements (*event-driven*)
  - modularité
  - extensibilité
  - langage général, fortement typé, similaire à Ada

Histoire

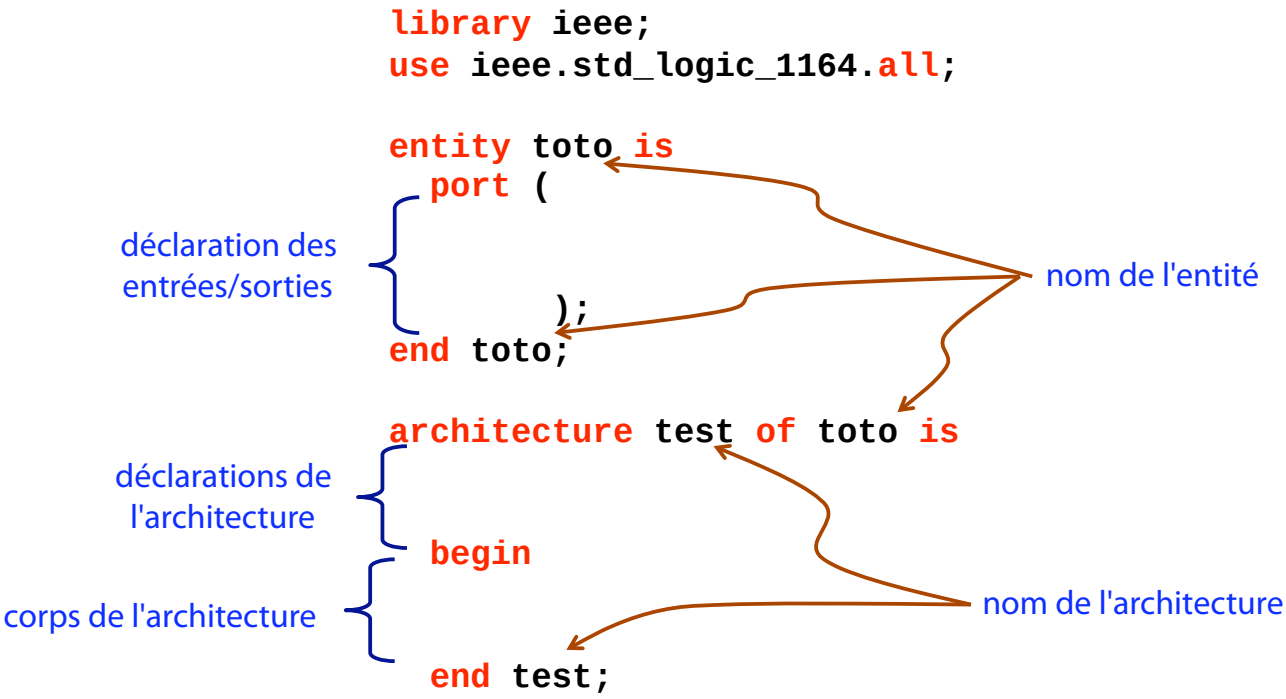
- 1980:
  Début du projet, financé par le DoD (400M $US)
- 1982:
  Contrat pour Intermetrics, IBM et Texas
- 1985:
  Version 7.2 dans le domaine public
- 1987:
  Standard IEEE 1076 (VHDL-87)
- 1993:
  Nouvelle version du standard (VHDL-93)
- 2001:
  Nouvelle version du standard (VHDL-2001)
- 2008:
  Nouvelle version du standard (VHDL-2008)

Entité et architecture

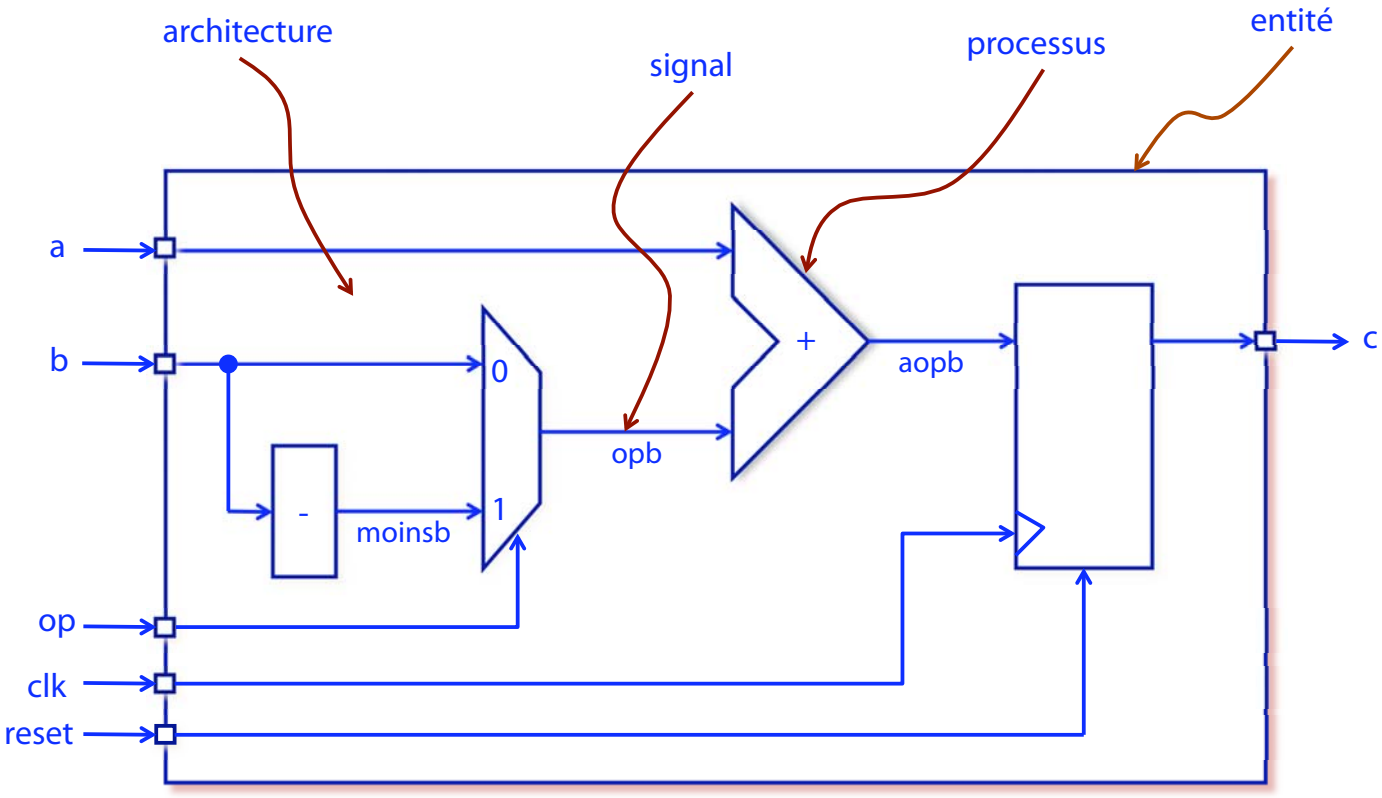
- VHDL nous intéresse en tant que langage pour la description, simulation et synthèse des systèmes digitaux
- Au plus haut niveau d'abstraction, un système digital est vu comme une "boîte noire", dont on connaît l'interface avec l'extérieur mais dont on ignore le contenu
- En VHDL la boîte noire est nommé entité (entity)
- Une entité doit toujours être associée avec au moins une description de son contenu, de son implémentation: c'est l'architecture



Structure d'un programme VHDL



Exemple



- Les entrées/sorties du système sont les ports de l'entité
- Chaque composant interne du système sera un processus (process) de l'architecture
- Une architecture est un ensemble de processus
- Les processus s'exécutent en parallèle
- Les processus de l'architecture sont interconnectés par le biais des signaux (signal)
- Quelques notes sur la syntaxe d'un programme VHDL:
  - pas de différentiation entre majuscules et minuscules
  - format libre
  - toute phrase termine par un point virgule
  - le début d'un commentaire est signalé par un double trait ("--"). Le commentaire termine avec la fin de ligne

bibliothèque IEEE

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

entité

```
entity exemple is
  port (a, b           : in std_logic_vector(3 downto 0);
        op, clk, reset : in std_logic;
        c              : out std_logic_vector(3 downto 0));
end exemple;
```

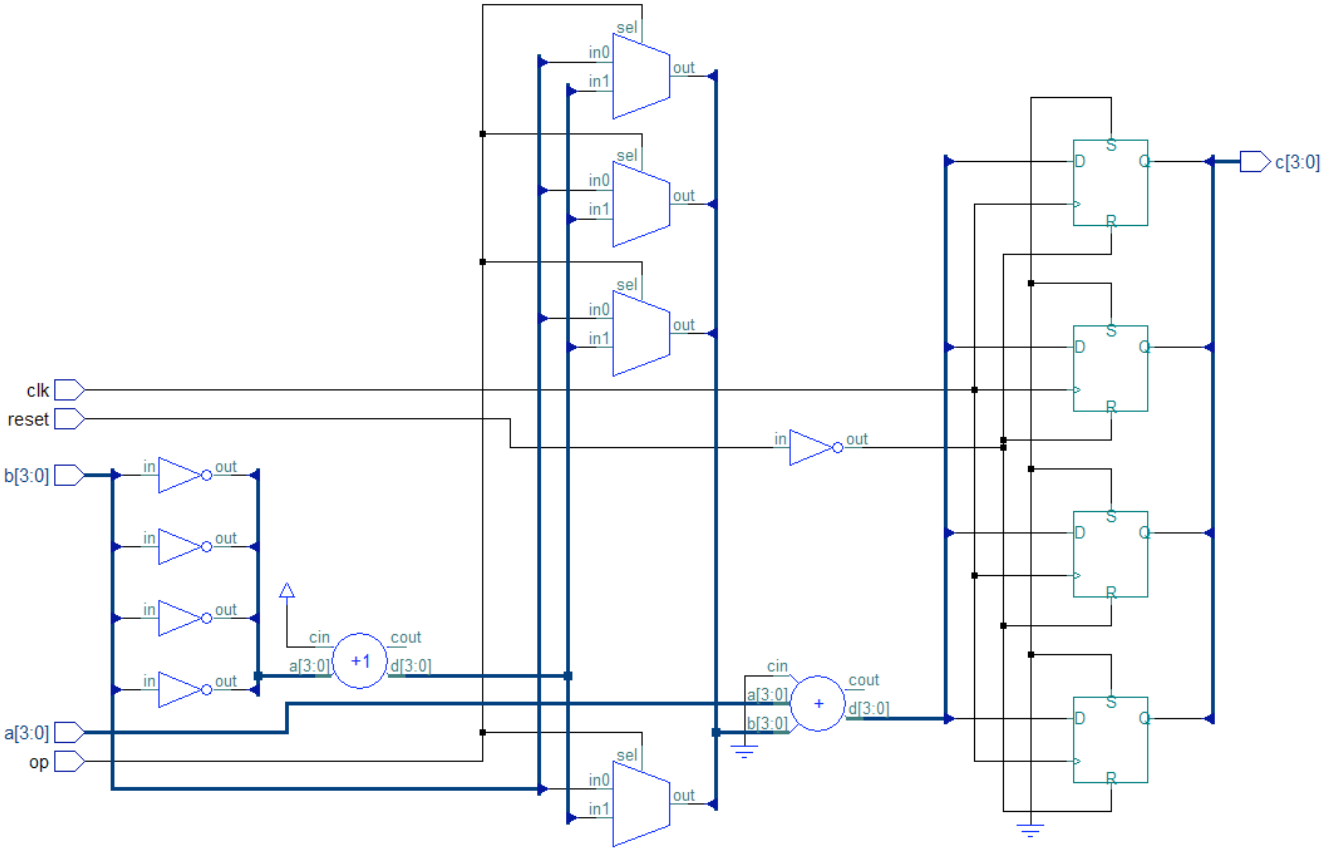
architecture

```
architecture test of exemple is
  signal moinsb, opb, aopb : std_logic_vector(3 downto 0);
begin
moinsb <= not b + "0001";

opb <= b when op='0'
         else moinsb;

aopb <= a + opb;

process (reset, clk)
  begin
  if reset='0'
    then c <= "0000";
    else if (clk'event and clk='1')
           then c <= aopb;
         end if;
  end if;
  end process;
end test;
```

processus implicites

processus explicite

- Résultat de la synthèse:



Données traitées par VHDL

- Toute donnée traitée par VHDL doit être déclarée comme constante, variable ou signal
- Constantes:

```
constant pi : real := 3.1416;
    constant index_max : integer is 10*N;
```

- Variables:
  valeur modifiable immédiatement par une affectation (:=)

```
variable stop : boolean;
    variable etat : CodeDEtat := ST0;
```

- Signaux:
  modélisation de l'entrée/sortie d'un dispositif. C'est une forme d'onde qui change avec le temps: la modification a lieu à la prochaine itération de la simulation (retard delta)

- VHDL est un langage fortement typé: toute donnée doit être déclaré avant utilisation, en indiquant son type
- Les types prédéfinis sont:
  - scalaire:
    integer
    real
    enumerated
    physical
  - composé:
    array
    record
  - pointeur:
    acces
  - I/O:
    file

- On peut également créer de nouveaux types, en fonction des types prédéfinis. Par exemple:

```
type HuitBits is range 0 to 255;

   type CodeDEtat is (init, ST1, ST2, exit);

   type word is array (0 to 31) of bit;

   type EtatsLogiques is record
       valeur : integer range -127 to 128;
       force  : integer;
       end record;
```


- Pour la synthèse, les types de données les plus utilisés sont **std_logic**, pour les données à un bit, et **std_logic_vector**, pour les bus
- Ces types ne sont pas prédéfinis: pour les utiliser, il faut déclarer le paquet (package) **std_logic_1164**, qui fait partie de la bibliothèque (library) IEEE:

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

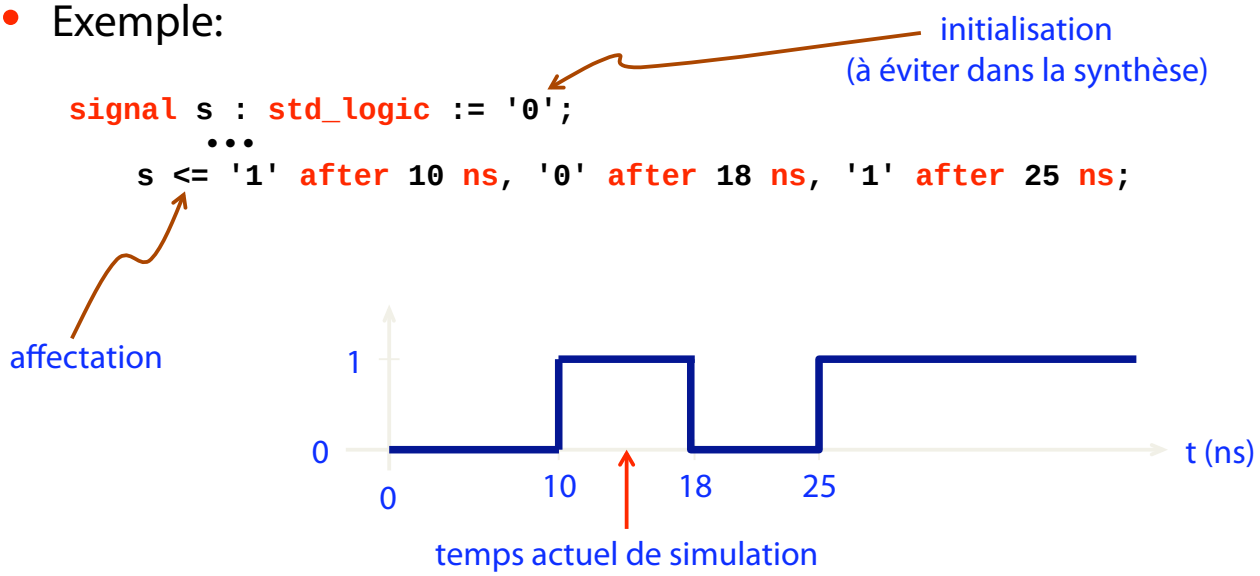
- Une donnée de type std_logic possède une valeur parmi neuf possibles:
  - **'U'** uninitialized
  - **'X'** forcing unknown
  - **'0'** forcing 0
  - **'1'** forcing 1
  - **'Z'** high impedance
  - **'W'** weak unknown
  - **'L'** weak 0 (pull-down)
  - **'H'** weak 1 (pull-up)
  - **'-'** don't care
- Pour une affectation, les valeurs utilisées sont: **'X'**, **'0'**, **'1'**, **'Z'**

Opérateurs

- Opérations logiques:
  - **and or nand nor xor xnor**
- Opérations de comparaison:
  - **= /= < <= > >=**
- Opérations de décalage
  - **sll srl sla sra rol ror**
- Opérations d'addition:
  - **+ - &**
- Opérations de signe:
  - **+ -**
- Opérations de multiplication:
  - *** / mod rem**
- Opérations diverses:
  - **not abs ****

Signaux (introduction)

- Pour connecter les différents composants d'un système, VHDL utilise les signaux (signal), équivalent des fils ou câbles d'interconnexion dans le monde réel
- Exemple:



- On peut associer certains attributs aux signaux, qui produisent une valeur. Deux exemples d'attributs sont:
  - **s'event**:
    vrai si un événement arrive pendant le delta présent (c'est-à-dire, si le signal **s** change de valeur)
  - **s'active**:
    vrai si une transaction arrive pendant le delta présent (c'est-à-dire, si le signal **s** est évalué, qu'il change ou pas de valeur)
- Un signal est toujours global à une architecture donnée

Processus (*process*)

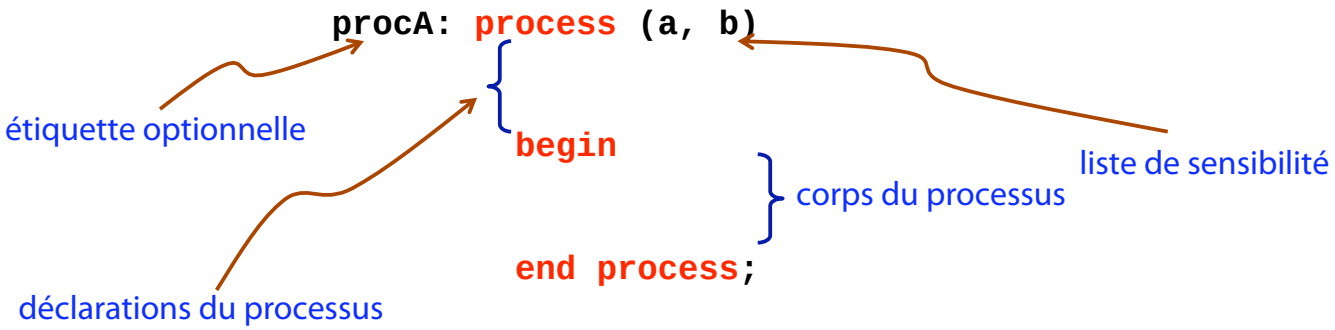
- Une architecture en VHDL est un ensemble de processus exécutés en parallèle (en concurrence)
- L'ordre relatif des processus à l'intérieur d'une architecture n'a pas d'importance
- Il existe deux types de processus:
  - le processus implicite ou phrase concurrente
  - le processus explicite

```
architecture toto of test is
begin
c <= a and b;
z <= c when oe='1' else 'Z';
seq: process (clk, reset)
     begin
       .
       .
     end process;
end toto;
```

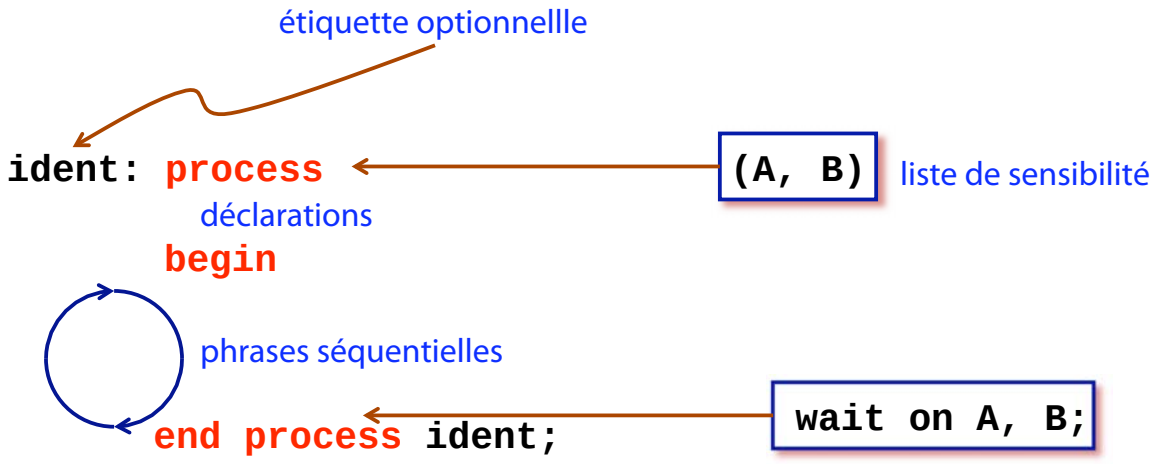
processus implicites

processus explicite

- Un processus explicite est un ensemble de phrases exécutées séquentiellement: à l'intérieur d'un processus l'ordre des phrases a donc une importance
- Un processus ne peut pas être déclaré à l'intérieur d'un autre processus



- Les phrases séquentielles d'un processus s'exécutent en boucle infinie
- L'exécution de la boucle s'arrête seulement lors d'un **`wait on`** sur une liste de signaux: l'exécution redémarre lorsque l'un des signaux de la liste change de valeur
- La plupart des outils de synthèse n'acceptent qu'un **`wait`** par processus, placé au début ou à la fin du processus
- Un **`wait on`** à la fin d'un processus peut être remplacé par une liste de sensibilité, placée juste après le mot clé **`process`**. La liste de sensibilité est incompatible avec un **`wait`**: c'est l'un ou l'autre
- Tout signal dont le changement de valeur a une influence sur le processus doit apparaître dans la liste de sensibilité. Dans le cas contraire, la simulation pourrait donner des résultats faux puisqu'elle ne serait pas enclenchée. Pour éviter des erreurs, on peut mettre dans la liste de sensibilité tous les signaux testés ou apparaissant à droite d'une affectation, à l'intérieur du processus

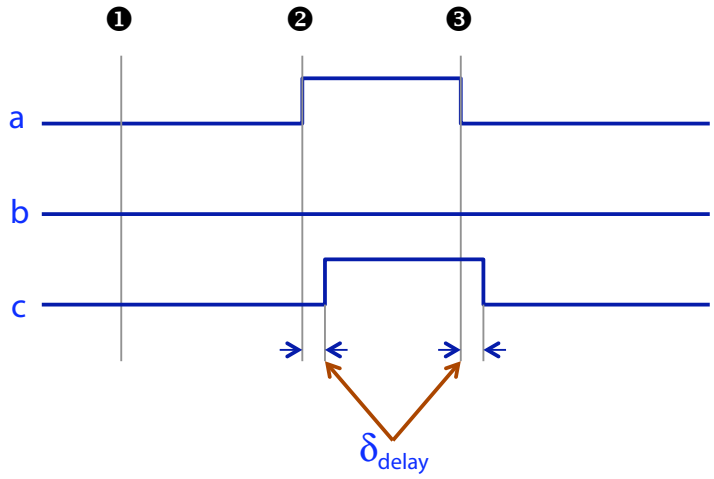


- L'évaluation des signaux à l'intérieur d'un processus se fait séquentiellement, mais l'affectation des nouvelles valeurs se fait au même moment: pendant le **`wait`**

- Exemple 1:

```
entity toto is
  port (a: in std_logic;
        b: out std_logic);
end toto;

architecture basic of toto is
  signal c: std_logic;
begin
  process (a)
  begin
  c <= a;
  if c='1'
    then b <= a;
    else b <= '0';
  end if;
  end process;
end basic;
```

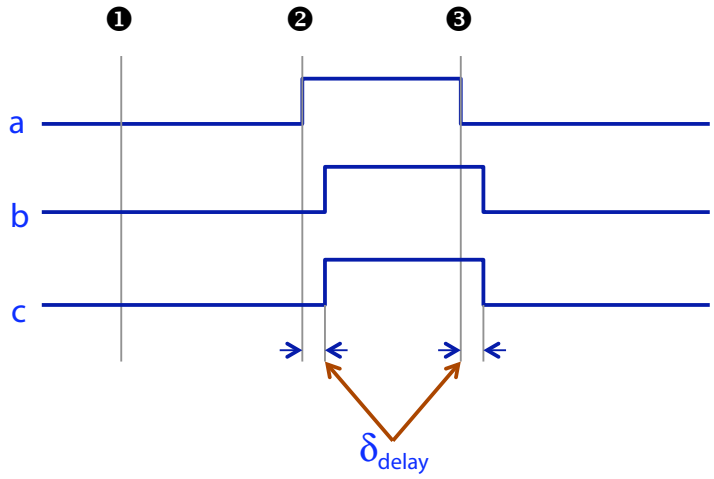

- Le signal **c** est affecté à **a** au début du processus, mais sa valeur est réellement mise à jour seulement à la fin du processus. Pour cette raison, lors du **if…then** la valeur de **c** qui est testée est celle qui avait le signal à la fin de l'exécution précédente du processus
- Supposons le cas de la figure suivante, avec **a** et **b** égaux à **'0'** au début de la simulation (temps ❶):



- Au temps ❷, **a** a changé. Comme le processus est sensible à **a**, le simulateur redémarre l'exécution du processus. Le signal **c** est affecté à **a**, et la condition **(c='1')** est testée. Le résultat du test sera **false**, puisque **c** est toujours égal à **'0'**: la mise à jour à la valeur **'1'** n'aura lieu qu'au temps ❷ + δ_{delay} , c'est-à-dire à la fin du processus. A la fin de la première exécution du processus, **b** sera donc toujours égal à **'0'**
- Au temps ❸, **a** passe à **'0'**, ce qui fait redémarrer l'exécution du processus. Comme la valeur de **c** est mise à jour seulement au temps ❸ + δ_{delay}, **c** garde sa valeur précédente (**'1'**). La condition **(c='1')** est maintenant vraie, et **b** est affecté à la valeur de **a** à la fin du processus, c'est-à-dire à **'0'**. En conclusion, **b** reste toujours à **'0'** pendant la simulation

- Exemple 2:

```
entity toto is
  port (a: in std_logic;
        b: out std_logic);
end toto;

architecture basic2 of toto is
  signal c: std_logic;
begin
  process (a)
  begin
  c <= a;
  if c='0'
    then b <= a;
    else b <= '0';
  end if;
  end process;
end basic2;
```



- Les signaux gardent leur ancienne valeur jusqu'au moment du **`wait`**. Lorsque le même signal reçoit plusieurs affectations à l'intérieur du même processus, seulement la dernière affectation aura un effet réel

Phrases séquentielles

-

```
if condition then
 phrases
 { elsif condition then
 phrases }
 [ else
 phrases ]
end if;
```

-

```
case opcode is
      when X"00" => add;
      when X"01" => subtract;
      when others => illegal_opcode;
   end case;
```

-
```
loop
      faire;
   end loop;
```
-
```
while toto < tata loop
      toto := toto + 1;
   end loop;
```
-
```
   for item in 1 to last_item loop
     table(item) := 0;
   end loop;
```
-
```
   next [ label ] [ when condition ];
```
-
```
   exit [ label ] [ when condition ];
```


Processus implicites

- Les phrases suivantes sont des processus implicites (ou phrases concurrentes):
  - l'affectation inconditionnelle d'un signal
  - l'affectation conditionnelle d'un signal (équivalent d'un **`case`**).
    Exemple:

```
toto <= x when a=1 else
        y when a=2 else
        "0000";
```

  - le **`generate`** (équivalent d'un **`for`**).
    Exemple:

```
G1: for i in 0 to 5 generate
      result(i) <= ena and InputSignal(i);
   end generate;
```

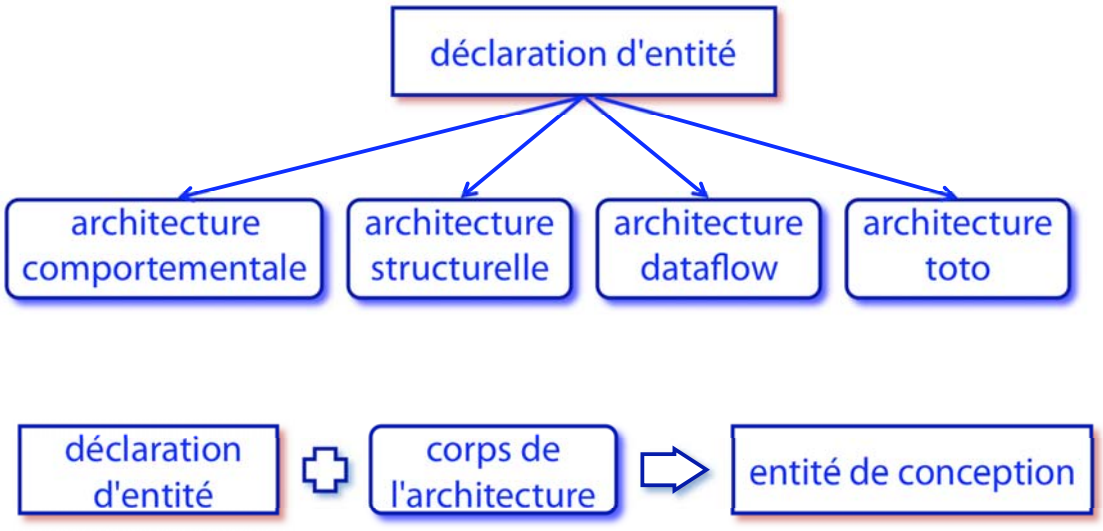
- Une phrase concurrente est un processus à part entière, avec un **`wait on`** implicite sur les signaux placés à droite de l'affectation. La phrase concurrente:

```
toto <= a + b;
```

est, par exemple, tout à fait équivalente au processus explicite:

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


Structure d'un programme

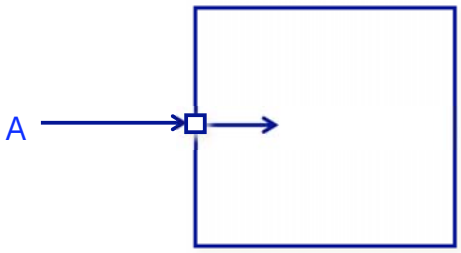


Déclaration d'entité

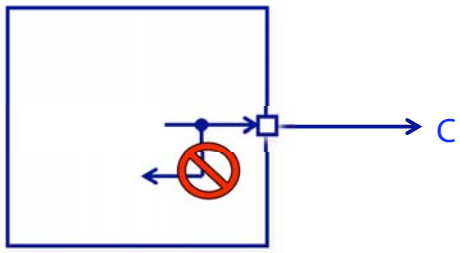
- C'est la vue externe du système, avec une déclaration de ses ports (canaux de communication entrée/sortie, pour le transport des signaux)
- Exemple:


```
entity comparateur is
  port (A, B : in std_logic;
        C    : out std_logic);
end comparateur;
```

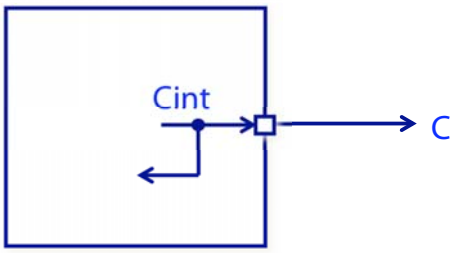

- Les ports sont toujours des signaux, avec un type et un mode associés. Le mode indique la direction de l'information et si le port peut être lu à l'intérieur de l'entité

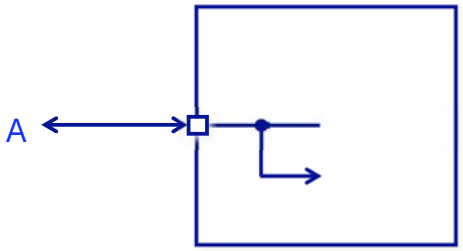


Mode **in**
Le signal peut être lu à l'intérieur de l'entité (mais ne peut pas être modifié)



Mode **out**
Le signal ne peut pas être lu
à l'intérieur de l'entité





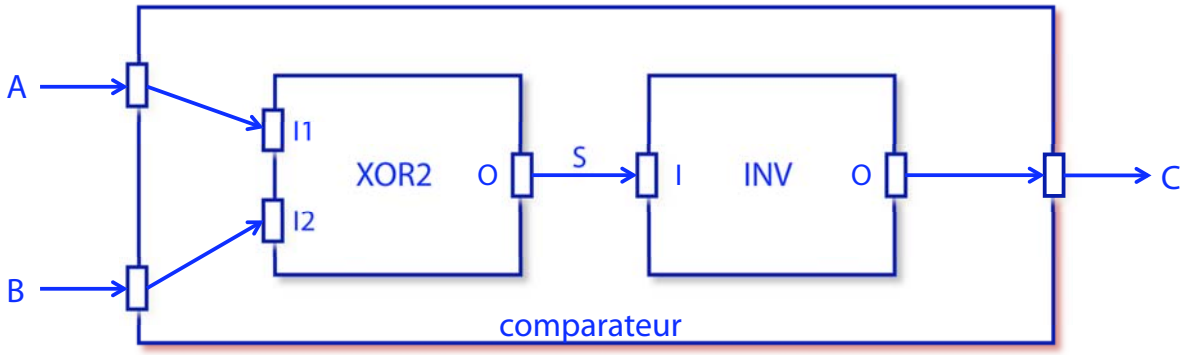
Mode **inout**
Le signal peut être lu et modifié
à l'intérieur de l'entité

- Exemple:

```
entity nonet is
  port (a, b      : in std_logic;
        z, zbarre : out std_logic);
end nonet;

architecture fausse of nonet is
  begin
    z <= a and b;
    zbarre <= not z;
  end fausse;

architecture correcte of nonet is
  signal resultat : std_logic;
  begin
    resultat <= a and b;
    z <= resultat;
    zbarre <= not resultat;
  end correcte;
```

Corps de l'architecture

- C'est la vue interne du système
- Plusieurs vues sont possibles pour la même conception, dont les principaux styles sont:
  - structurel: un assemblage de sous-blocs, similaire à la liste d'interconnexions d'un schéma logique (*netlist*)
  - dataflow: équations logiques
  - comportemental: algorithmes
- Exemple:




```
architecture comportementale of comparateur is
  begin
    process (A, B)
      begin
      if (A = B) then
        C <= '1';
      else
        C <= '0';
      end if;
    end process;
  end comportementale;
```

```
architecture dataflow of comparateur is
  begin
    C <= not (A xor B);
  end dataflow;
```



```
architecture structurelle of comparateur is

  component XOR2
    port (O : out std_logic; I1, I2 : in std_logic);
  end component;
  component INV
    port (O : out std_logic; I : in std_logic);
  end component;

  signal S : std_logic;

  begin
    C1 : XOR2 port map (O => S, I1 => A, I2 => B);
    C2 : INV port map (C, S);
  end structurelle;
```

- Avant d'utiliser (instancier) un composant, il doit être déclaré
- La déclaration d'un composant est similaire à la déclaration d'une entité: c'est simplement la liste des ports de sa boîte noire
- Pour pouvoir simuler ou synthétiser un composant, il doit exister ailleurs une paire entité-architecture qui le décrive. Le nom du composant doit être le même que celui de l'entité correspondante. Les noms des signaux du composant doivent être identiques aux noms des signaux de l'entité correspondante
- Chaque instance d'un composant doit posséder une étiquette. Chaque signal dans la liste des ports du composant (les noms formels) doit être connecté à un signal dans l'architecture (les noms réels). Ces associations peuvent être explicites ou implicites (données par l'ordre dans la liste)

- Exemple:


```
u4: OR2 port map (i1 => toto,
                  i2 => titi,
                  y => y);

u4: OR2 port map (toto,
                  titi,
                  y);
```

- La connexion des ports de sortie n'est pas obligatoire: un port de sortie laissé ouvert (non connecté) doit être associé à la valeur **open**
- L'instanciation d'un composant est un processus: il ne peut donc pas être fait à l'intérieur d'un autre processus

- A partir de VHDL-93, il est possible d'utiliser une syntaxe simplifié pour les architectures structurelles
- Pour notre exemple, cela donne:

```
architecture structurelle of comparateur is

  signal S : std_logic;

  begin
    C1 : entity work.XOR2(toto_arch)
           port map (O => S, I1 => A, I2 => B);
    C2 : entity work.INV(toto_arch)
           port map (C, S);
  end structurelle;
```

- Ce qui suppose que les composants **`XOR2`** et **`INV`** ont été déclarés dans la library **`work`**, et que l'on utilisera les architectures correspondantes données en parenthèse

Variables et signaux

- Les variables sont toujours locales à un processus: il n'y a pas de variables globales
- Les signaux sont toujours globaux à toute l'architecture. Un signal est donc connu par tous les processus d'une architecture donnée. Toutefois, un signal ne peut pas être modifié par plus d'un processus: en effet, cela voudrait dire que le signal a plus d'une valeur à un moment donné
- Contrairement aux signaux, les variables sont mises à jour de façon instantanée, au moment de l'affectation, sans retard possible

- Les phrases à l'intérieur d'un processus sont toujours exécutées en séquence, y compris les affectations de signaux. Toutefois, les signaux sont mis à jour en même temps, au moment d'un **`wait`**.
  En absence d'un **`wait`**, le processus est exécuté sans arrêt, sans que les signaux soient mis à jour.
  C'est-à-dire: lorsqu'on exécute les phrases à l'intérieur d'un processus, on utilise les valeurs initiales des signaux pour tous les calculs et, à la fin, on met à jour les nouvelles valeurs.
  Si, par exemple **`a=4`** et **`b=1`** et on a dans un processus:

```
b <= a;
c <= b;
```

  à la fin on aura **`b=4`** et **`c=1`**

- Exemple avec des variables:

```
entity toto is
end toto;

architecture var of toto is
  signal trigger, sum : integer := 0;
begin
  process
    variable var1 : integer := 1;
    variable var2 : integer := 2;
    variable var3 : integer := 3;
    begin
      wait on trigger;
      var1 := var2 + var3;
      var2 := var1;
      var3 := var2;
      sum <= var1 + var2 + var3;
    end process;
end var;
```

  Si **`trigger`** change à **`t=10`**, alors **`var1=5`**, **`var2=5`**, **`var3=5`** et à **`t=10+Δ`**, **`sum=15`**

- Exemple avec des signaux:

```
entity toto is
end toto;

architecture sig of toto is
  signal trigger, sum : integer := 0;
  signal sig1 : integer := 1;
  signal sig2 : integer := 2;
  signal sig3 : integer := 3;
  begin
    process
      begin
      wait on trigger;
      sig1 <= sig2 + sig3;
      sig2 <= sig1;
      sig3 <= sig2;
      sum <= sig1 + sig2 + sig3;
    end process;
end sig;
```

Si **`trigger`** change à **`t=10`**, tous les signaux sont mis à jour à **`t=10+Δ`** : **`sig1=5`**, **`sig2=1`**, **`sig3=2`** et **`sum=6`**

- Exemple à l'intérieur d'un programme principal:

```
a <= b;
b <= c;
c <= d;
```

En supposant que la période de simulation est de 10 ns, que les valeurs initiales des variables sont **`a=1`**, **`b=2`**, **`c=3`**, **`d=0`**, et qu'un événement **`d=4`** arrive, le timing de l'exécution serait:

temps	delta	a	b	c	d
0	+0	1	2	3	0
10	+0	1	2	3	4
10	+1	1	2	4	4
10	+2	1	4	4	4
10	+3	4	4	4	4

Synthèse d'un système combinatoire

- Additionneur de deux nombres à deux bits:


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

entity AddLog is
  port (x1, x0, y1, y0 : in std_logic;
        s2, s1, s0     : out std_logic);
end AddLog;
```

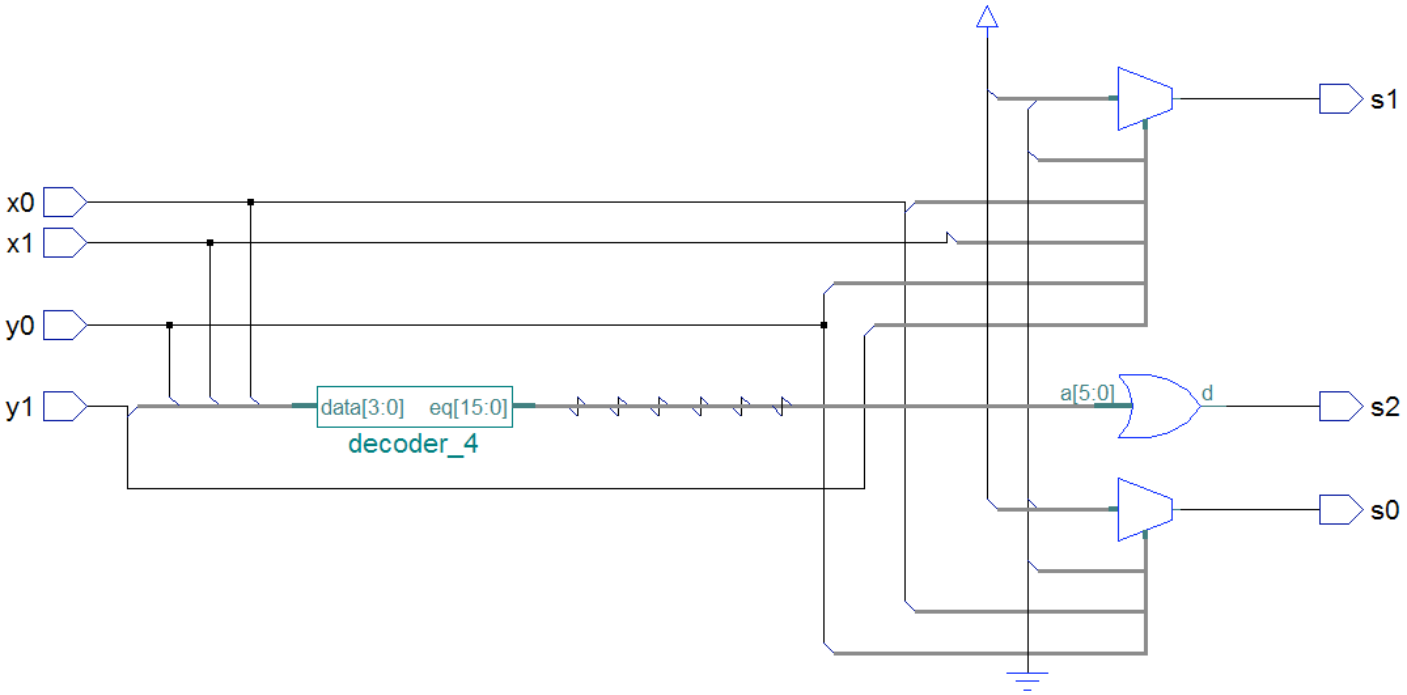


```
architecture table of AddLog is
  signal entree: std_logic_vector(3 downto 0);
begin
  entree <= x1 & x0 & y1 & y0;
  process (entree)
  begin
    case entree is
      when "0000" => s2 <= '0';
                     s1 <= '0';
                     s0 <= '0';
      when "0001" => s2 <= '0';
                     s1 <= '0';
                     s0 <= '1';
      when "0010" => s2 <= '0';
                     s1 <= '1';
                     s0 <= '0';
      when "0011" => s2 <= '0';
                     s1 <= '1';
                     s0 <= '1';
```

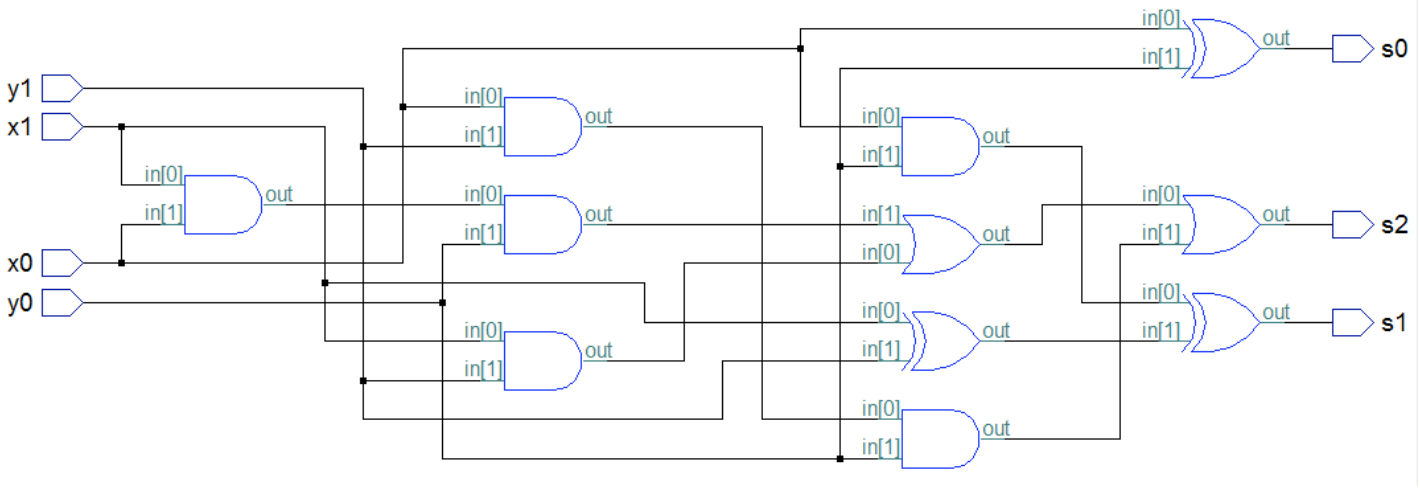
```
        when "0100" => s2 <= '0';
                       s1 <= '0';
                       s0 <= '1';
        when "0101" => s2 <= '0';
                       s1 <= '1';
                       s0 <= '0';
        when "0110" => s2 <= '0';
                       s1 <= '1';
                       s0 <= '1';
        when "0111" => s2 <= '1';
                       s1 <= '0';
                       s0 <= '0';
        when "1000" => s2 <= '0';
                       s1 <= '1';
                       s0 <= '0';
        when "1001" => s2 <= '0';
                       s1 <= '1';
                       s0 <= '1';
        when "1010" => s2 <= '1';
                       s1 <= '0';
                       s0 <= '0';
        when "1011" => s2 <= '1';
                       s1 <= '0';
                       s0 <= '1';
```



```
        when "1100" => s2 <= '0';
                       s1 <= '1';
                       s0 <= '1';
        when "1101" => s2 <= '1';
                       s1 <= '0';
                       s0 <= '0';
        when "1110" => s2 <= '1';
                       s1 <= '0';
                       s0 <= '1';
        when "1111" => s2 <= '1';
                       s1 <= '1';
                       s0 <= '0';
        when others => null;
    end case;
  end process;
end table;
```

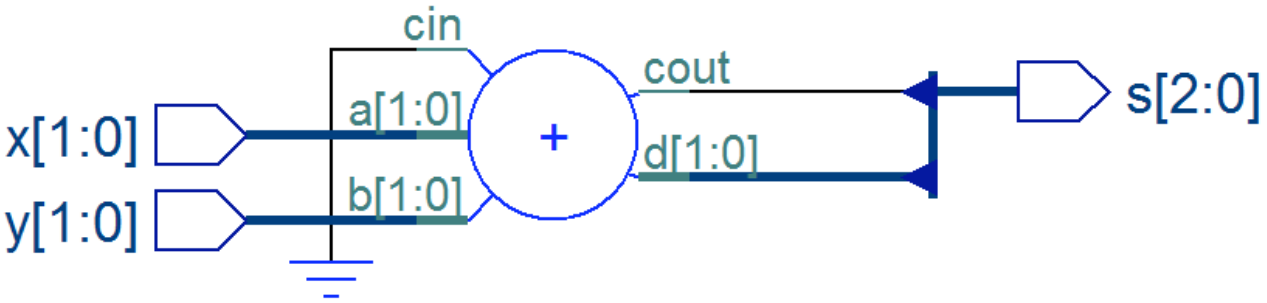



```
architecture equations of AddLog is
begin
  s2 <= (x1 and y1) or (x1 and x0 and y0) or (x0 and y1 and y0);
  s1 <= (x0 and y0) xor (x1 xor y1);
  s0 <= x0 xor y0;
end equations;
```



- Description fonctionnelle:

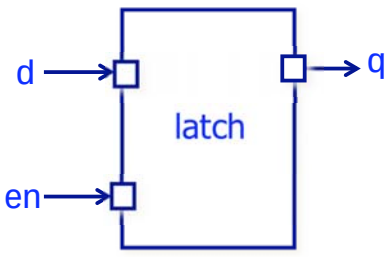
```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_arith.all;
  use ieee.std_logic_unsigned.all;

entity AddFonc is
  port (x, y : in std_logic_vector(1 downto 0);
        s    : out std_logic_vector(2 downto 0));
end AddFonc;

architecture comportement of AddFonc is
begin
  s <= ('0' & x) + ('0' & y);
end comportement;
```



Synthèse d'un élément de mémoire

- Un signal garde son ancienne valeur entre deux exécutions d'un processus: cette propriété de mémoire implicite peut être utilisé pour implémenter des éléments de mémoire (*latch*)

```
process (en, d)
  begin
  if en='1'
    then q <= d;
  end if;
end process;
```

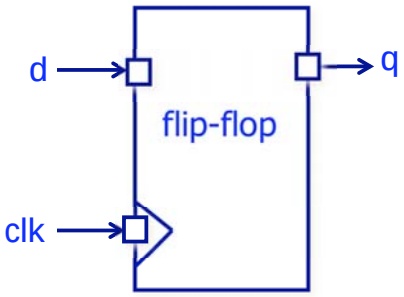


- Cette propriété peut produire des éléments de mémoire non voulus. Dans la plupart des cas, le système résultant sera fonctionnellement correct, mais plus lent et plus grand que nécessaire
- Pour des systèmes rapides et efficaces, on doit faire attention à:
  - spécifier complètement toutes les branches d'une condition à l'intérieur d'un processus: si la branche **`else`** d'une condition n'est pas spécifiée, alors les signaux modifiés à l'intérieur du **`if…then`** gardent leur dernière valeur en générant un élément de mémoire
  - spécifier complètement toutes les possibilités de modification d'un signal à l'intérieur d'un processus: par exemple, spécifier complètement les valeurs pour tous les signaux modifiés à l'intérieur d'une phrase **`case`**. Ceci peut être fait plus facilement en mettant des valeurs par défaut à tous les signaux au début du processus: de cette façon, on modifie seulement les signaux nécessaires dans chaque sélection du **`case`** (rappelez-vous que les valeurs des signaux ne sont mises à jour qu'à la fin du processus)

Synthèse d'un registre

- La façon utilisée couramment pour générer une bascule (*flip-flop*) est:

```
process (clk)
begin
  if clk'event and clk='1'
      then q <= d;
  end if;
end process;
```



- Reset asynchrone:

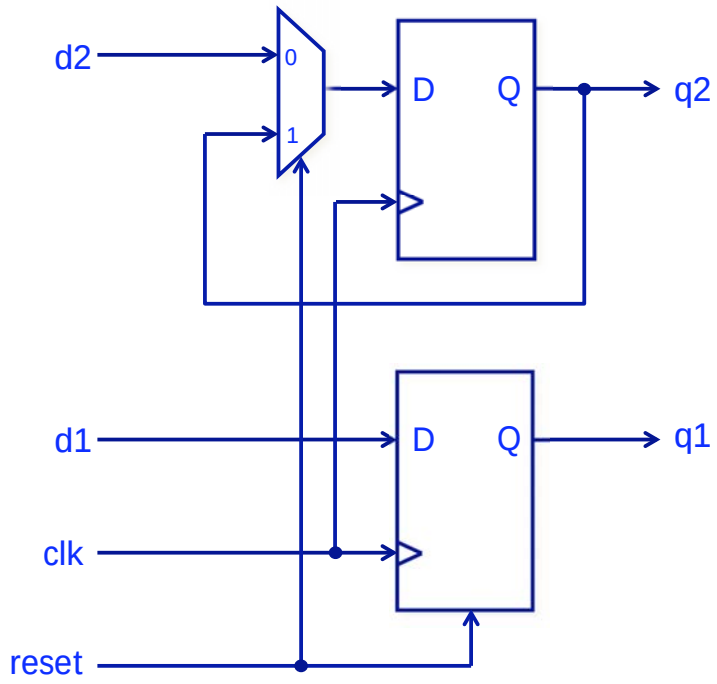
```
process (clk, reset)
begin
  if reset='1'
    then q <= '0';
    elsif clk'event and clk='1'
      then q <= d;
  end if;
end process;
```

- Reset synchrone:

```
process (clk)
begin
  if clk'event and clk='1'
    then if reset='1'
           then q <= '0';
           else q <= d;
         end if;
  end if;
end process;
```

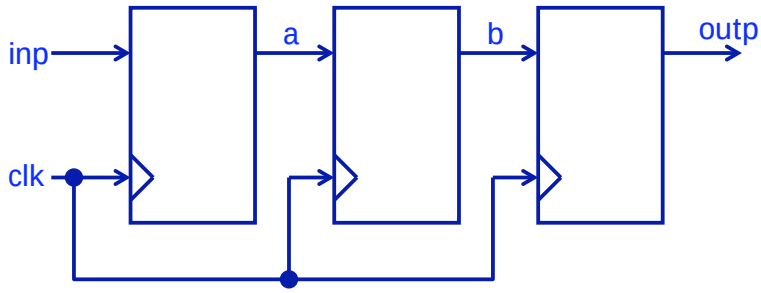

- Exemple:

```
architecture toto of test is
begin
  process (clk, reset)
  begin
  if reset='1'
    then q1 <= '0';
    elsif clk'event and clk='1'
      then q1 <= d1;
           q2 <= d2;
  end if;
  end process;
end toto;
```



- Exemple:

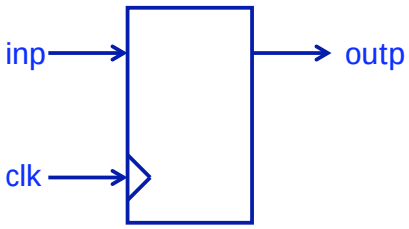
```
process (clk)
  variable a, b : std_logic;
begin
  if (clk'event and clk='1')
    then outp <= b;
         b := a;
         a := inp;
  end if;
end process;
```



Une variable ne garde pas sa valeur entre deux exécutions d'un processus. Comme, dans cet exemple, les variables sont utilisées avant d'avoir reçu une valeur, il faut chercher leurs valeurs précédentes dans des des bascules

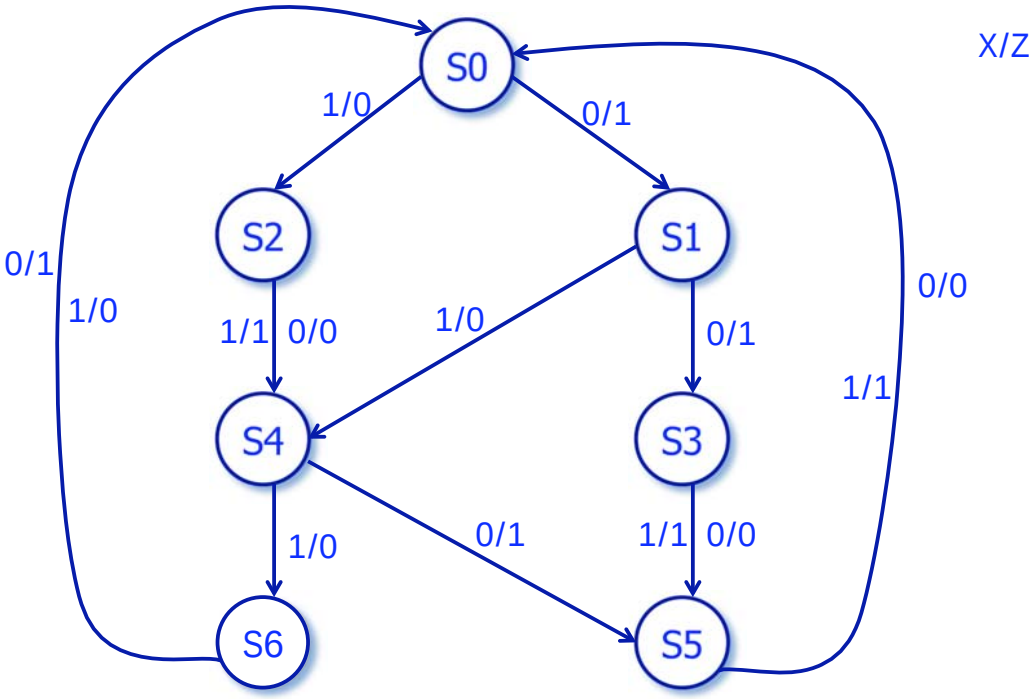
- Exemple:

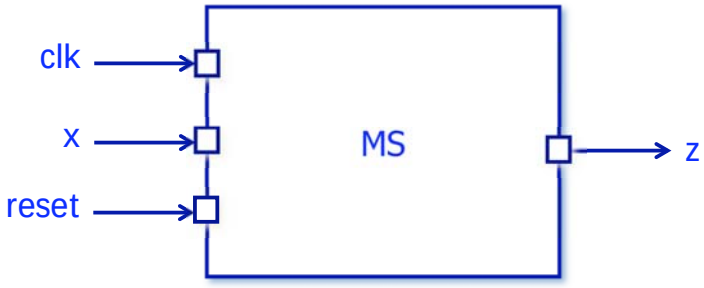
```
process (clk)
  variable a, b : std_logic;
begin
  if (clk'event and clk='1')
    then a := inp;
         b := a;
         outp <= b;
  end if;
end process;
```



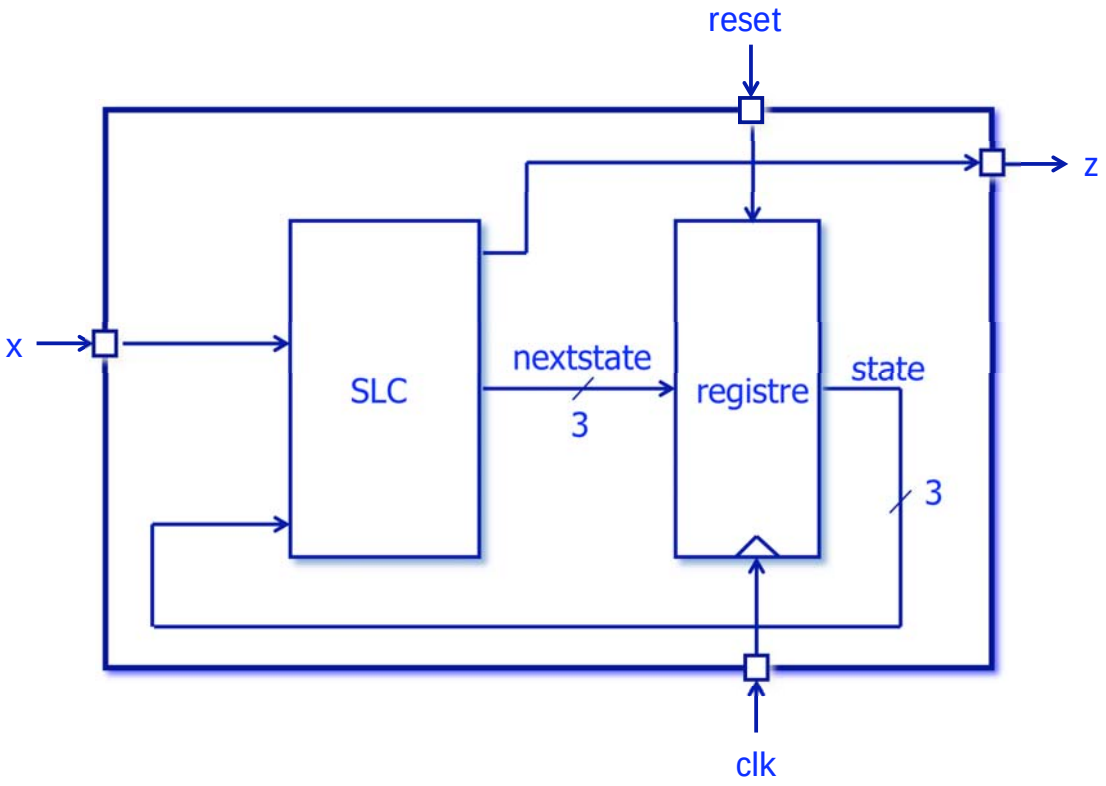
Ici, les variables sont de simples fils: elles ne génèrent aucun élément

Synthèse d'une machine séquentielle





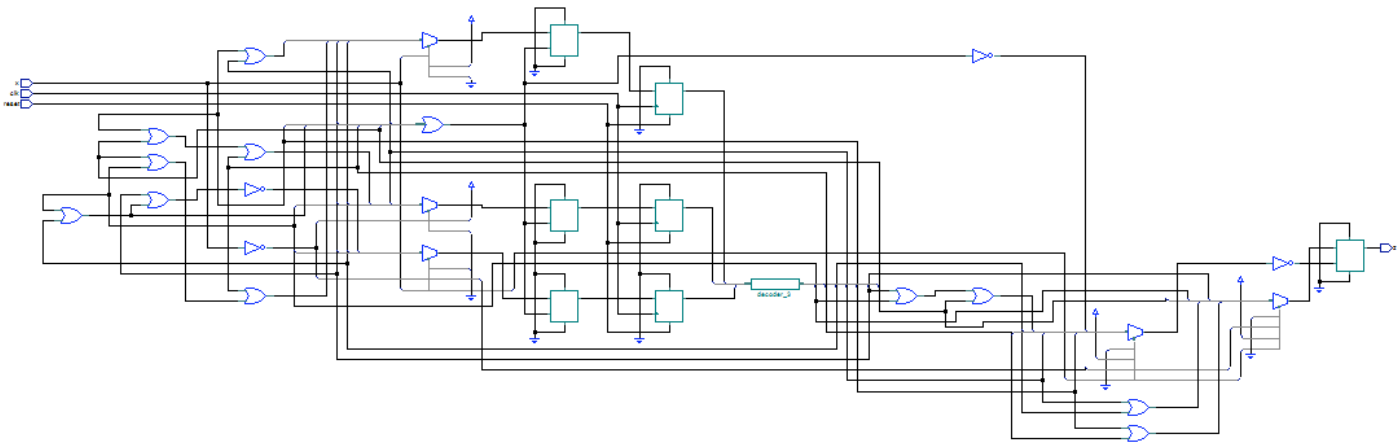

```
library ieee;
  use ieee.std_logic_1164.all;

entity MS is
  port (x, clk, reset : in std_logic;
        z             : out std_logic);
end MS;
```




```
architecture graphe of MS is
  signal state, nextstate: integer;
begin
  process (state, x)
  begin
    case state is
      when 0 =>
        if x='0' then z <= '1';
                      nextstate <= 1;
                 else z <= '0';
                      nextstate <= 2;
        end if;
      when 1 =>
        if x='0' then nextstate <= 3;
                 else z <= '0';
                      nextstate <= 4;
        end if;
      when 2 =>
        nextstate <= 4;
        if x='1' then z <= '1';
        end if;
      when 3 =>
        nextstate <= 5;
        if x='0' then z <= '0';
        end if;
```

```
    when 4 =>
      if x='0' then z <= '1';
                    nextstate <= 5;
               else z <= '0';
                    nextstate <= 6;
      end if;
    when 5 =>
      nextstate <= 0;
      if x='0' then z <= '0';
               else z <= '1';
      end if;
    when 6 =>
      nextstate <= 0;
      if x='0' then z <= '1';
      end if;
    when others => null;
  end case;
end process;
```

```
  process (clk, reset)
  begin
    if reset='1'
      then state <= 0;
      else if clk='1' and clk'event
             then state <= nextstate;
           end if;
    end if;
  end process;

end graphe;
```

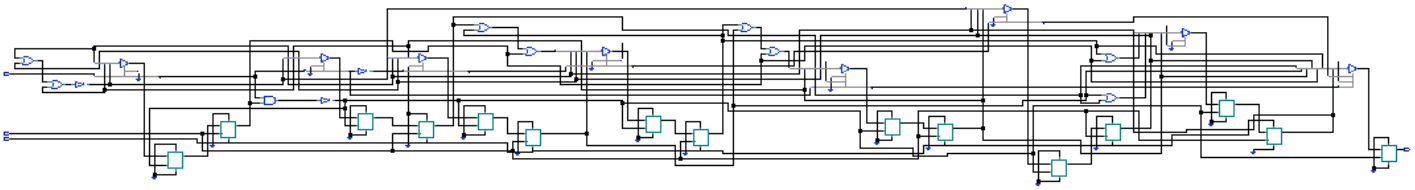



```
architecture synt1 of MS is
  type s_type is (S0, S1, S2, S3, S4, S5, S6);
  signal state, nextstate: s_type;
begin
  process (state, x)
  begin
    case state is
      when S0 =>
        if x='0' then z <= '1';
                      nextstate <= S1;
                 else z <= '0';
                      nextstate <= S2;
        end if;
      when S1 =>
        if x='0' then z <= '1';
                      nextstate <= S3;
                 else z <= '0';
                      nextstate <= S4;
        end if;
      when S2 =>
        if x='0' then z <= '0';
                      nextstate <= S4;
                 else z <= '1';
                      nextstate <= S4;
        end if;
```



```
    when S3 =>
      if x='0' then z <= '0';
                    nextstate <= S5;
               else z <= '1';
                    nextstate <= S5;
      end if;
    when S4 =>
      if x='0' then z <= '1';
                    nextstate <= S5;
               else z <= '0';
                    nextstate <= S6;
      end if;
    when S5 =>
      if x='0' then z <= '0';
                    nextstate <= S0;
               else z <= '1';
                    nextstate <= S0;
      end if;
    when S6 =>
      if x='0' then z <= '1';
                    nextstate <= S0;
      end if;
  end case;
end process;
```

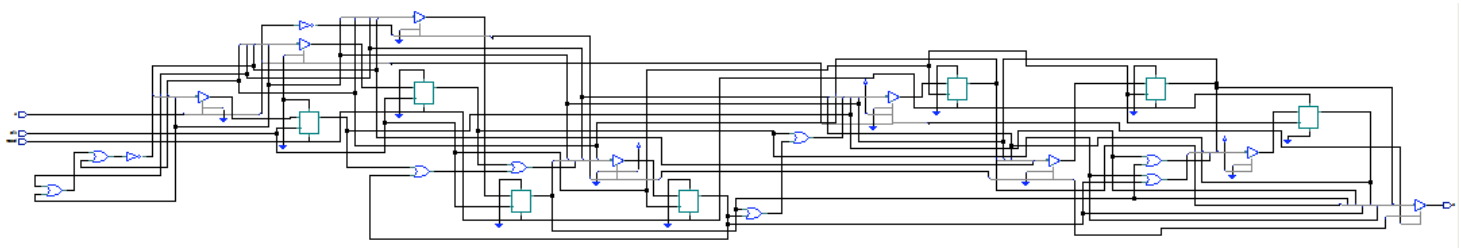
```
process (clk, reset)
begin
  if reset='1'
    then state <= S0;
    else if clk='1' and clk'event
           then state <= nextstate;
         end if;
  end if;
end process;

end synt1;
```

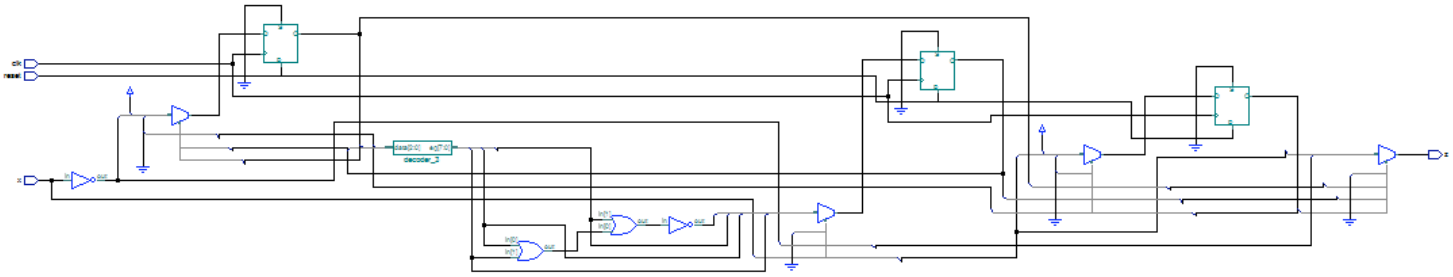



```
architecture synt2 of MS is
  type s_type is (S0, S1, S2, S3, S4, S5, S6);
  signal state, nextstate: s_type;

begin
  process (state, x)
  begin
    z <= '0';
    nextstate <= S0;
    case state is
      when S0 =>
        if x='0' then z <= '1';
                      nextstate <= S1;
                 else nextstate <= S2;
        end if;
      when S1 =>
        if x='0' then z <= '1';
                      nextstate <= S3;
                 else nextstate <= S4;
        end if;
      when S2 =>
        nextstate <= S4;
        if x='1' then z <= '1';
        end if;
```

```
    when S3 =>
      nextstate <= S5;
      if x='1' then z <= '1';
      end if;
    when S4 =>
      if x='0' then z <= '1';
                    nextstate <= S5;
                else nextstate <= S6;
      end if;
    when S5 =>
      if x='1' then z <= '1';
      end if;
    when S6 =>
      if x='0' then z <= '1';
      end if;
  end case;
end process;
```

```
process (clk, reset)
 begin
   if reset='1'
     then state <= S0;
     elsif clk='1' and clk'event
       then state <= nextstate;
   end if;
 end process;

end synt2;
```



Procédures

- Il existe trois modes de passage pour les paramètres des procédures:
  - **`in`**: le paramètre est passé par valeur et ne peut donc pas être modifié par la procédure
  - **`out`**: aucune valeur n'est transmise à la procédure mais le paramètre peut être modifié par la procédure
  - **`inout`**: le paramètre est passé par référence et il peut donc être lu et modifié

- Exemple:

```
type byte is array (7 downto 0) of std_logic;
    •••
procedure ByteToInteger (ib: in byte; oi: out integer) is
  variable result : integer := 0;
  begin
    for i in 0 to 7 loop
      if ib(i) = '1' then
        result := result + 2**i;
      end if;
    end loop;
    oi := result;
  end ByteToInteger;
```


Fonctions

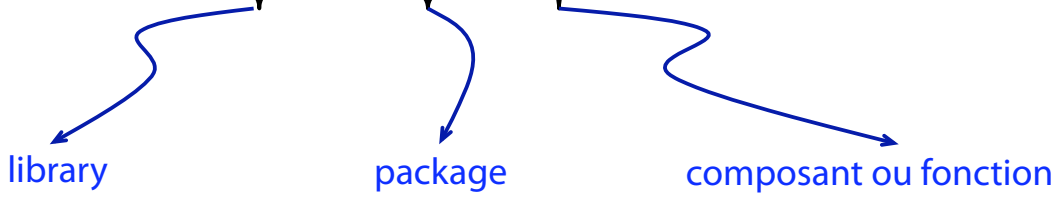
- Tous les paramètres d'une fonction sont de mode **`in`**: en dehors de ses variables locales, une fonction ne peut modifier que la valeur retournée. En plus, les paramètres ne peuvent pas être de type variable: ils ne peuvent être que des constantes ou des signaux
- Exemple:

```
function f (a, b, c : std_logic) return std_logic is
  variable x : std_logic;
  begin
    x := ((not a) and (not b) and c);
    return x;
  end f;
```

Packages et libraries

- On peut mettre dans un **package** des composants et des fonctions qui pourront être utilisés par la suite, par d'autres programmes
- Des *packages* peuvent être mis à l'intérieur d'une **library**
- Pour utiliser un *package*, il faut le déclarer, ainsi que sa *library*:

```
library bitlib;
        use bitlib.bit_pack.all;
```



- Il y a eux bibliothèques qui sont visibles implicitement et qui n'ont donc pas besoin d'être déclarées pour être utilisées:
  - **std**: contient les différents types et opérateurs qui font partie du standard VHDL
  - **work**: c'est la bibliothèque de travail par défaut
- Généralement, un *package* est utilisé pour déclarer des composants, qui pourront être utilisés par tous les systèmes qui fassent référence au *package* par la suite

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

package toto is
      .
      .
      .
end toto;
```

déclarations des composants

- Le *package* doit être ajouté à la bibliothèque de travail (**`work`**) avant de pouvoir être utilisé. Pour la plupart des outils de conception, ceci implique compiler le *package* avant de compiler le système qui l'utilise
- Des types et des sous-types peuvent également être déclarés dans les *packages*

Generics

- C'est une façon de spécifier des paramètres pour un composant: à chaque fois que le composant est utilisé, on peut donner des valeurs différentes pour ces paramètres
- Exemple:

```
entity nand2 is
  generic (trise, tfall : time;
           load         : natural);
  port (a, b : in std_logic;
        c    : out std_logic);
end nand2;
                •
                •
                •
component nand2 is
  generic (trise : time := 3 ns;
           tfall : time := 2 ns;
           load  : natural := 1);
  port (a, b : in std_logic;
        c    : out std_logic);
end component;
```

Testbench

- Après avoir développé un code VHDL, et avant de le synthétiser pour un dispositif donné, il peut être simulé pour vérifier son comportement fonctionnel
- Pour la simulation, il est courant de créer un programme spécial, appelé un *testbench*, composé de trois parties:
  - un générateur de vecteurs de test
  - le système à tester
  - un moniteur, pour examiner les réponses de la simulation