VHDL - Logique programmable

Partie 2- La structure d'un programme VHDL

Denis Giacona

ENSISA
École Nationale Supérieure d'Ingénieur Sud Alsace
12, rue des frères Lumière
68 093 MULHOUSE CEDEX
FRANCE

Tél. 33 (0)3 89 33 69 00

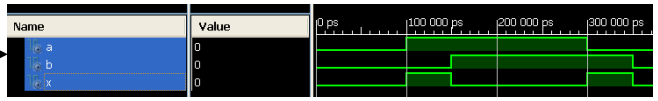
1. Éléments fondamentaux du langage VHDL

1.1. Les cibles du langage VHDL

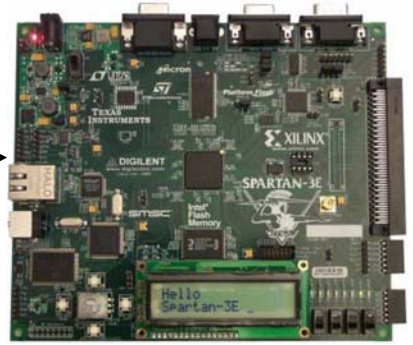
Des instructions pour décrire les signaux de test

Des instructions pour décrire le comportement du système
- fonctions combinatoires
- fonctions séquentielles

Logiciel de simulation



Logiciel de synthèse
(configuration de circuits logiques programmables)



1.2. Quelques caractéristiques du langage

- Syntaxe **complexe**
  - pour aider le concepteur, les outils de développement proposent des modèles VHDL ***(templates)*** et des convertisseurs de schémas en code VHDL

- Langage **strict** par rapport aux types et aux dimensions des données
  - avantage : élimination d'un grand nombre d'erreurs de conception dès la compilation

- Très bonne **portabilité**
  - à condition d'écrire un code indépendant de la technologie

- De nombreux **styles** de description
  - ☝ tous les styles ne conviennent pas à toutes les applications
  - ☝ tous les codes ne sont pas synthétisables

1.3. La bonne interprétation des instructions

✋ **Les instructions modélisent un câblage matériel** lorsqu'elles sont destinées à la **programmation/configuration** de circuits CPLD/FPGA.

✋ **Les instructions ressemblent à celles d'un langage impératif, mais,** alors que pour certaines, l'ordre d'écriture est déterminant, pour d'autres, l'ordre n'a pas d'importance.

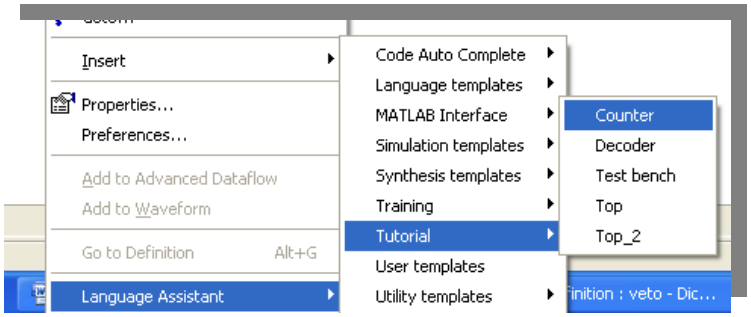
1.4. Deux conseils incontournables

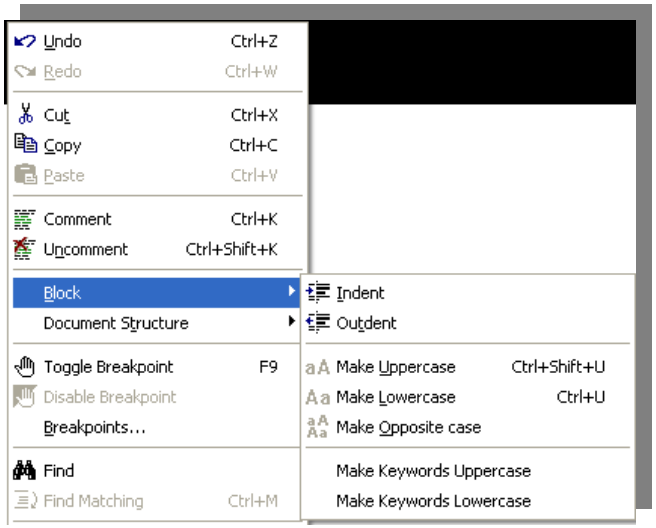
❶ S'appliquer sur la présentation

```
architecture arch_bcdcnt4_ar_en_comb of bcdcnt4_ar_en_comb is
signal count:std_logic_vector(3 downto 0);
begin
process(clk,ar)
begin
if ar='1' then
count<=(others=>'0');
elsif (clk'event and clk='1') then
if en='1' then
if count<x"9" then
count<=count+1;
else
count<=x"0";
end if;
end if;
end if;
end process;
co<='1' when (count=x"9") else '0';
q<=count;
end arch_bcdcnt4_ar_en_comb;
```

```
architecture arch_bcdcnt4_ar_en_comb of bcdcnt4_ar_en_comb is
   signal count_i : std_logic_vector(3 downto 0);
begin
count_proc: process (clk, ar)
begin
  if ar = '1' then
    count_i <= (others => '0');
  elsif (clk'event and clk = '1') then
    if en = '1' then
      if count_i < x"9" then
        count_i <= count_i + 1;
      else
        count_i <= x"0";
      end if;
    end if;
  end if;
end process;
-- Elaboration du signal carry "co" hors processus (assignation combinatoire)
co     <= '1' when (count_i = x"9") else '0';
q      <= count_i;
end arch_bcdcnt4_ar_en_comb;
```

Les éditeurs comportent des assistants de langage et des outils de mise en forme.



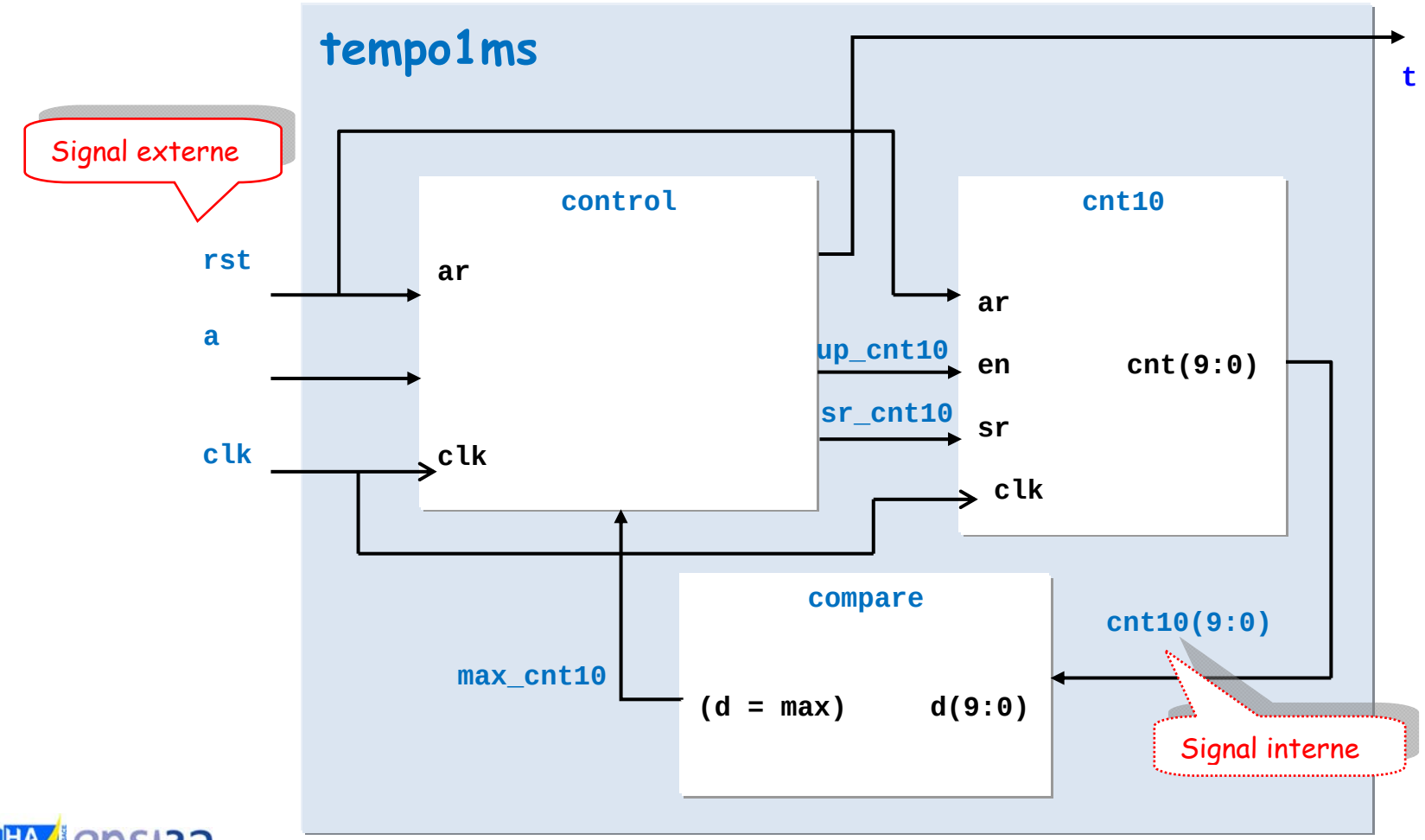


❷ Appliquer une règle de dénomination des identificateurs

- **Langue anglaise, minuscules et caractère _**
- **Quelques exemples :**
  - Horloge : **clk**
  - Reset : **rst**
  - Reset synchrone (mise à '0') : **sr**
  - Reset asynchrone : **ar**
  - Preset synchrone (mise à '1') : **sp**
  - Preset asynchrone : **ap**
  - Reset synchrone actif à l'état bas : **nsr**
  - Read/write actif à l'état bas : **nrw**
  - Décodeur 1 parmi 8 : **dec1of8**
  - Multiplexeur 2 vers : **mux2to1**
  - Registre de mémorisation 4 bits : **reg4**
  - Registre à décalage 4 bits : **shreg4**
  - Compteur 8 bits (binaire naturel) : **count8, cnt8**
  - Entrée incrémentation d'un compteur 8 bits : **up_count8, up_cnt8**
  - Sortie retenue d'un compteur 8 bits : **co_count8, co_cnt8**
  - Compteur BCD : **bcdcount, bcdcnt**
  - Compteur (signal interne) : **count_int, count_i**
  - Étiquette de processus de comptage : **count_proc**

1.5. Le principe du couple entité - architecture

Tout bloc logique est décrit par un couple (entity, architecture)



Le code VHDL correspondant au temporisateur **tempo1ms** comporte deux parties :

- une déclaration d'**entité** qui définit les entrées-sorties
- une **architecture** qui détermine le comportement

```
-- déclaration des entrées-sorties
entity tempo1ms is
    ...
end tempo1ms;
```

Déclaration des signaux externes

```
-- description du comportement
architecture tempo1ms_arch of tempo1ms is
    ...
    begin
    -- description du compteur
    cnt10_proc: process (rst,clk) …
    -- description du contrôleur
    control_proc: process (rst, clk) …
    -- description du comparateur
    compare: max_cnt10 <= '1' when …

end tempo1ms_arch;
```

Déclaration des signaux internes

1.6. Les commentaires

```
-- Un commentaire commence par deux tirets consécutifs et s'arrête à la fin de la ligne

/* Ici commence un bloc de commentaire.
    Autorisé uniquement avec la norme VHDL-2008 */
```

1.7. Les identificateurs

Les identificateurs sont des appellations d'objets du langage (données et types).

- Ils sont constitués de caractères alphabétiques (26 lettres), numériques (10 chiffres décimaux) et du caractère souligné _ ; les lettres accentuées sont exclues
- Le premier caractère doit être une lettre
- Les lettres majuscules et minuscules sont équivalentes
- Le dernier caractère doit être différent de _
- Deux _ à la file sont interdits
- Le nom ne doit pas être un mot réservé
- La longueur d'un mot est quelconque (mais une ligne maximum)

1.8. Les objets données : signal, constante, variable

```
signal inc : std_logic;
```

Les **signaux** portent les informations des liaisons d'entrée, des liaisons de sortie et des liaisons internes.

```
constant max : std_logic_vector(9 downto 0) := "1111100111";
```

Les **constantes** reçoivent leur valeur au moment de leur déclaration.

```
variable temp : integer range 0 to 999 := 0 ;
```

Les **variables** ne sont déclarées et utilisées que dans les **processus**, les **fonctions** et les **procédures**.
L'assignation initiale est facultative.

1.9. Les types des objets données

1.9.1. Les types prédéfinis par tous les compilateurs

integer : entier négatif ou positif

natural : entier positif ou nul

positive : entier positif

bit : énuméré dont les deux seules valeurs possibles sont '0' et '1'

bit_vector : composite tableau représentant un vecteur de bits

boolean : énuméré dont les deux valeurs possibles sont false et true

real : flottant compris entre -1.0E38 et 1.0E38

1.9.2. Les types complémentaires (inclus dans la bibliothèque IEEE 1164)

std_logic : 9 valeurs décrivant tous les états d'un signal logique

'U' : non initialisé**
'X' : niveau inconnu, forçage fort**
'0' : niveau 0, forçage fort
'1' : niveau 1, forçage fort
'Z' : haute impédance *
'W' : niveau inconnu, forçage faible**
'L' : niveau 0, forçage faible
'H' : niveau 1, forçage faible
'-' : quelconque (**don't care**) *

(**) Utile pour la simulation

(*) Utile pour la synthèse

std_logic_vector : vecteur de std_logic

Pour utiliser ces types il faut inclure les directives suivantes dans le code source.

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

Remarques concernant les synthétiseurs

- '0' et 'L' sont équivalents
- '1' et 'H' sont équivalents
- '-' est très utile pour la simplification des équations
- 'U', 'X', 'W' sont interdits
- Les types std_logic et std_logic_vector sont les plus utilisés.

1.9.3. Les types définis par l'utilisateur

Exemple : Description d'une table de vérité à l'aide d'un tableau

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

entity hexa_to_7seg is port (
        hexain:         in      bit_vector (3 downto 0);
        seg:            out     bit_vector (0 to 6));
end hexa_to_7seg;

architecture arch_hexa_to_7seg of hexa_to_7seg is
        type segment is (a, b, c, d, e, f, g);
        type truth_table_4 is array (bit, bit, bit, bit) of
                                bit_vector(a to g);
```

```
        constant hexa_7seg_table : truth_table_4 :=
-- outputs abcdefg (segments)       hexa inputs
-----------------------------------------------
                ((((("1111110",             -- 0000
                    "0110000"),             -- 0001
                   ("1101101",              -- 0010
                    "1111001")),            -- 0011
                  (("0110011",              -- 0100
                    "1011011"),             -- 0101
                   ("1011111",              -- 0110
                    "1110000"))),           -- 0111
                 (((("1111111",             -- 1000
                    "1111011"),             -- 1001
                   ("1110111",              -- 1010
                    "0011111")),            -- 1011
                  (("1001110",              -- 1100
                    "0111101"),             -- 1101
                   ("1001111",              -- 1110
                    "1000111"))));          -- 1111
begin
seg <= exa_7seg_table(hexain(3),hexain(2),hexain(1),hexain(0));
end arch_hexa_to_7seg;
```

1.9.4. Conversions de type

Les environnements de développement fournissent en général des **paquetages** comportant des fonctions de conversion de type. Par exemple, la bibliothèque de **Xilinx ISE** contient :

- **conv_integer (a)**
  - pour convertir un std_logic_vector **a** en un integer
- **conv_unsigned (x,n)**
  - pour convertir un std_logic_vector, integer, unsigned ou signed **x** en un unsigned de **n** bits (réalise un changement de taille)
- **conv_signed (x,n)**
  - pour convertir un std_logic_vector, integer, signed ou unsigned **x** en un signed de **n** bits (réalise un changement de taille)
- **conv_std_logic_vector (x,n)**
  - pour convertir un integer, unsigned ou signed **x** en un std_logic_vector de **n** bits

Pour utiliser ces fonctions, il suffit d'accéder au paquetage **std_logic_arith** de la bibliothèque **ieee**

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

1.9.5. Usage courant du type integer

- pour un **signal** servant de compteur (logique séquentielle synchrone) ; **note** : en synthèse, la valeur courante de comptage doit être convertie en **std_logic_vector** avant d'être appliquée sur une sortie

- pour une **variable** servant d'indice pour un **std_logic_vector**

1.10. Les littéraux

Les littéraux sont les représentations de valeurs attribuées aux objets données et aux objets types.

- **Les entiers décimaux**

```
1234
1_520_473                    -- pour améliorer la lisibilité
```

- **Les bits**

```
'0', '1'                     -- type bit
'0', '1' ,'U', 'X', 'H', 'L', 'W', 'Z', '-'   -- type std_logic
```

- **Les vecteurs de bits**

```
"1010"                       -- représentation binaire
O"12"                        -- représentation octale
X"A"                         -- représentation hexadécimale
```

- **Les caractères**

```
'a'
```

- **Les chaînes de caractères**

```
"ERREUR",  "ERREUR " & "N° "
```

1.11. Les opérateurs

1.11.1. Opérateurs prédéfinis de construction d'expressions

- Les opérateurs **relationnels** permettent de comparer des opérandes de même type et d'indiquer l'égalité, l'inégalité ou leur relation d'ordre. Ils sont utilisés dans des instructions de **test**. Ils rendent une valeur **booléenne** (**false** ou **true**).

  = **<** **<=** **>** **>=** **/=**

- Les opérateurs **logiques** sont définis pour les types boolean, bit, std_logic et les vecteurs correspondants

  and **or** **not** **nand** **nor** **xor**

  ✋ **and n'est pas prioritaire par rapport à or** (utiliser des parenthèses)

- Les opérateurs **arithmétiques** sont définis pour les types entiers et réels

  + **-** ***** **/** ****** **mod** **rem** **abs**

- Les opérateurs de **concaténation** et **d'agrégation** groupent des éléments de même type

  & **(, … ,)**

- Les opérateurs de **décalage** et de **rotation** sont définis sur le type bit_vector

  sll **srl** **sla** **sra** **ror** **rol**

1.11.2. Opérateurs d'assignation et d'association

- **Assignation de variable** `nom_variable := expression ;`
  - à l'intérieur d'un processus

- **Assignation de signal** (dynamique) `nom_signal <= expression ;`
  - dans la zone d'instructions d'une architecture

- **Assignation de signal** (statique) `signal nom_signal_interne : type_signal := valeur_initiale ;`
  - dans la zone de déclaration d'une architecture

- **Association** de signaux `nom_signal_formel => nom_signal_effectif ;`
  - connecteur de signaux dans une instanciation de composant

1.12. Les déclarations et les assignations des signaux vecteurs

gauche droite

```
signal a : std_logic_vector (0 to 7);
signal b : std_logic_vector (7 downto 0);

a <= "11110000";
b <= "11110000";
```

MSB (toujours à gauche) LSB (à droite)

a(0)	a(1)	a(2)	a(3)	a(4)	a(5)	a(6)	a(7)
1	1	1	1	0	0	0	0

MSB (toujours à gauche) LSB (à droite)

b(7)	b(6)	b(5)	b(4)	b(3)	b(2)	b(1)	b(0)
1	1	1	1	0	0	0	0

☞ La notation downto est la plus conventionnelle pour les applications de synthèse.

1.13. Surcharge d'opérateurs

❑ Les opérations ne sont pas admises sur des données de types différents sauf si l'on redéfinit l'opérateur (ce procédé est appelé surcharge).

```
function "+" (a: integer; b: bit) return integer is
begin
  if (b = '1') then
    return a + 1 ;
  else
    return a ;
  end if ;
end "+" ;

signal o, t     : integer range 0 to 255 ;
signal b        : bit ;
...
t    <=   o +  5 +  b ;
```

Opérateur + prédéfini pour les entiers

Opérateur + défini par le concepteur

❑ **Les environnements de programmation comportent des paquetages dans lesquels certains opérateurs sont redéfinis.**

- Dans l'outil **Xilinx ISE**, l'addition (**+**), la soustraction (**-**), la multiplication et la division de vecteurs sont définies dans le paquetage **std_logic_unsigned** de la bibliothèque **ieee**

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

❑ ***A propos de la multiplication et de la division :***

- L'opérateur de multiplication * est synthétisable (c.-à-d. que le compilateur est capable de produire une structure matérielle RTL)
- L'opérateur de division / n'est pas synthétisable de façon générale

```
xint  <=  conv_integer(a) / 10;
x  <=  conv_std_logic_vector (xint,16);
```

Parce qu'il s'agit d'une division par 10, ce code génère, avec Xilinx ISE, l'erreur suivante :

Operator <DIVIDE> must have constant operands or first operand must be power of 2

➔ On peut décrire la division à l'aide d'un système séquentiel que l'on intègre dans un composant VHDL et que l'on range dans une bibliothèque.

1.14. Les attributs

Les **attributs** sont des propriétés spécifiques que l'on peut associer aux signaux et aux types. La valeur d'un attribut peut être exploitée dans une expression.

- **Attribut valeur** sur des types scalaires ou des éléments (signaux, constantes, variables) de type scalaire

  'left, 'right, 'high, 'low, 'length

- **Attribut fonction** sur des types discrets ordonnés

  'pos, 'val, 'succ, 'pred, 'leftof, 'rightof

- **Attribut fonction** sur des signaux

  'event

- **Attribut intervalle** sur un signal dimensionné

  'range, 'reverse_range

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

entity attributs is port(
  vector_dwn      : in   std_logic_vector(15 downto 0);
  vector_up : in   std_logic_vector(0 to 7);
  x         : out  std_logic_vector(7 downto 0);
  y         : out  std_logic_vector(0 to 11);
  z         : out  std_logic_vector(7 downto 0));
end attributs;

architecture arch_attributs of attributs is
begin
x(0) <= vector_dwn(vector_dwn'left);
x(1) <= vector_dwn(vector_dwn'right);
x(2) <= vector_up(vector_up'left);
x(3) <= vector_up(vector_up'right);
x(4) <= vector_dwn(vector_dwn'high);
x(5) <= vector_dwn(vector_dwn'low);
x(6) <= vector_up(vector_up'high);
x(7) <= vector_up(vector_up'low);
y(vector_up'range)              <= "00001111";
z(vector_up'reverse_range)    <= "00110011";

end arch_attributs;
```

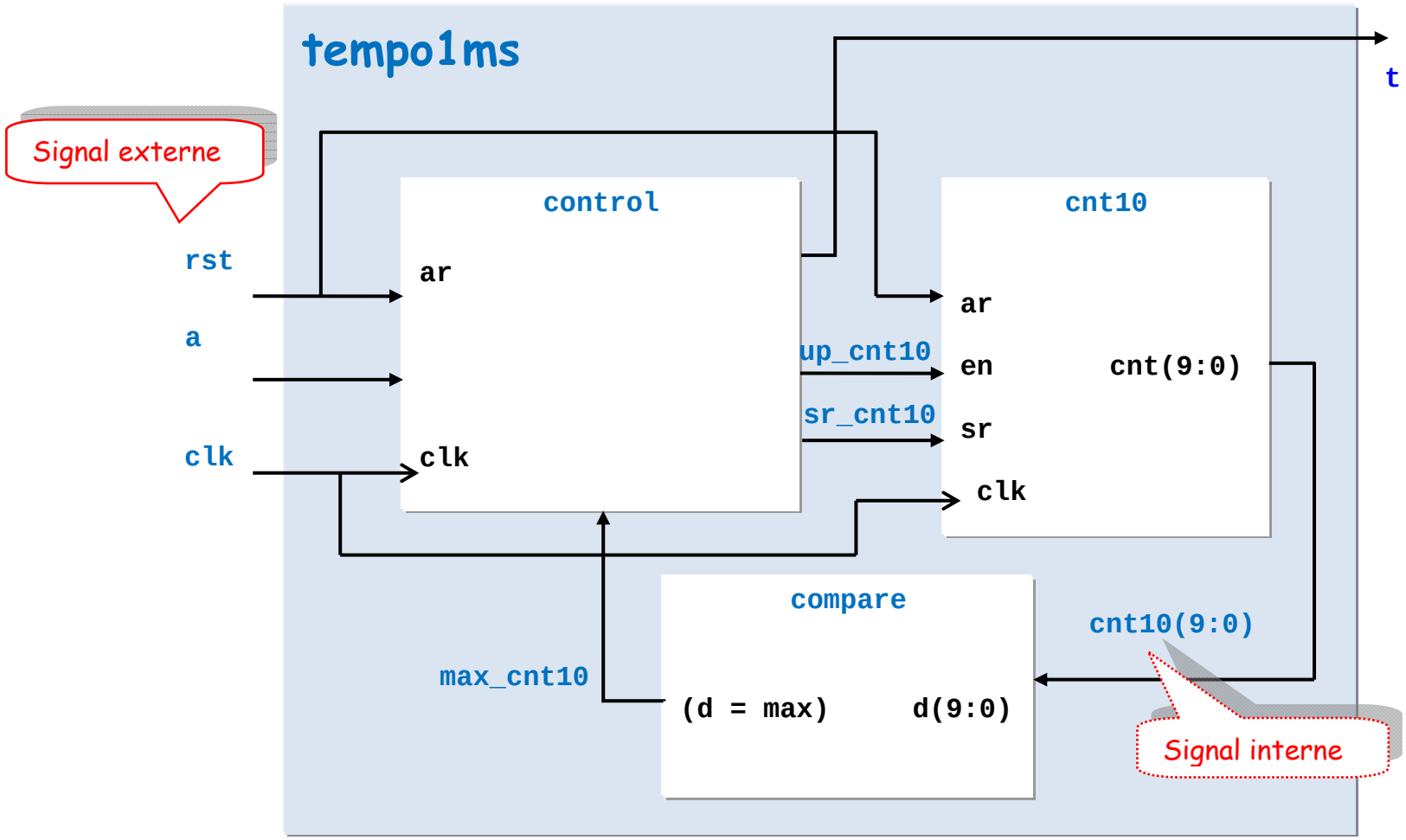
```
x(0) = vector_dwn(15)
x(1) = vector_dwn(0)
x(2) = vector_up(0)
x(3) = vector_up(7)
x(4) = vector_dwn(15)
x(5) = vector_dwn(0)
x(6) = vector_up(7)
x(7) = vector_up(0)

y(0) = GND
y(1) = GND
y(2) = GND
y(3) = GND
y(4) = VCC
y(5) = VCC
y(6) = VCC
y(7) = VCC

z(0) = VCC
z(1) = VCC
z(2) = GND
z(3) = GND
z(4) = VCC
z(5) = VCC
z(6) = GND
z(7) = GND
```

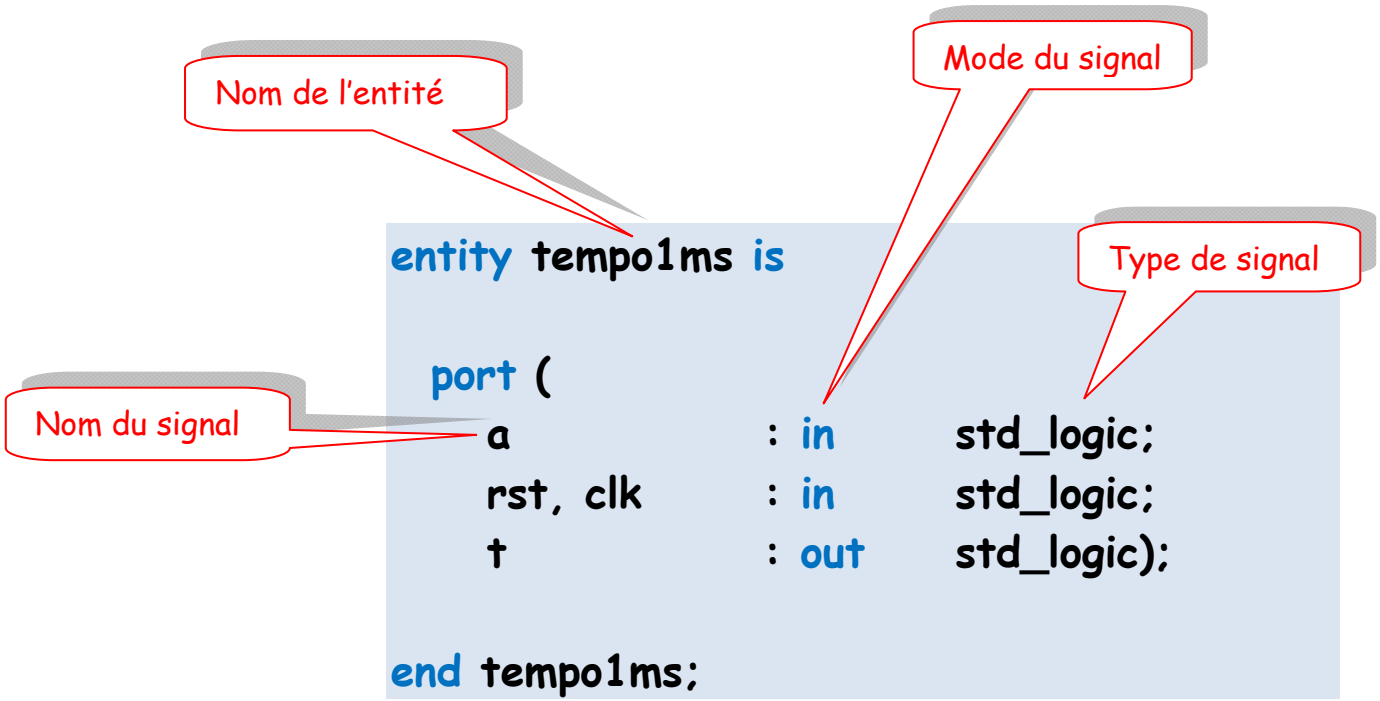
2. La déclaration d'entité

- La déclaration d'entité décrit une interface externe, unique, à la manière des entrées-sorties d'une boîte noire
- La déclaration comprend :
  - éventuellement, une liste de paramètres génériques
  - la définition des ports, c.-à-d. une liste de signaux d'entrée et de signaux de sortie ; chaque signal possède un nom, un mode et un type

2.1. Description d'une entité non générique

```
entity nom_entité is

port (
   [signal] nom_signal {,nom_signal}: [mode] type_signal
 {;[signal] nom_signal {,nom_signal}: [mode] type_signal}
);

end [entity] nom_entité;
```


tempo1ms
t
Signal externe
control
cnt10
rst
ar
a
up_cnt10
en
cnt(9:0)
sr_cnt10
sr
clk
clk
clk
compare
cnt10(9:0)
max_cnt10
(d = max)
d(9:0)
Signal interne

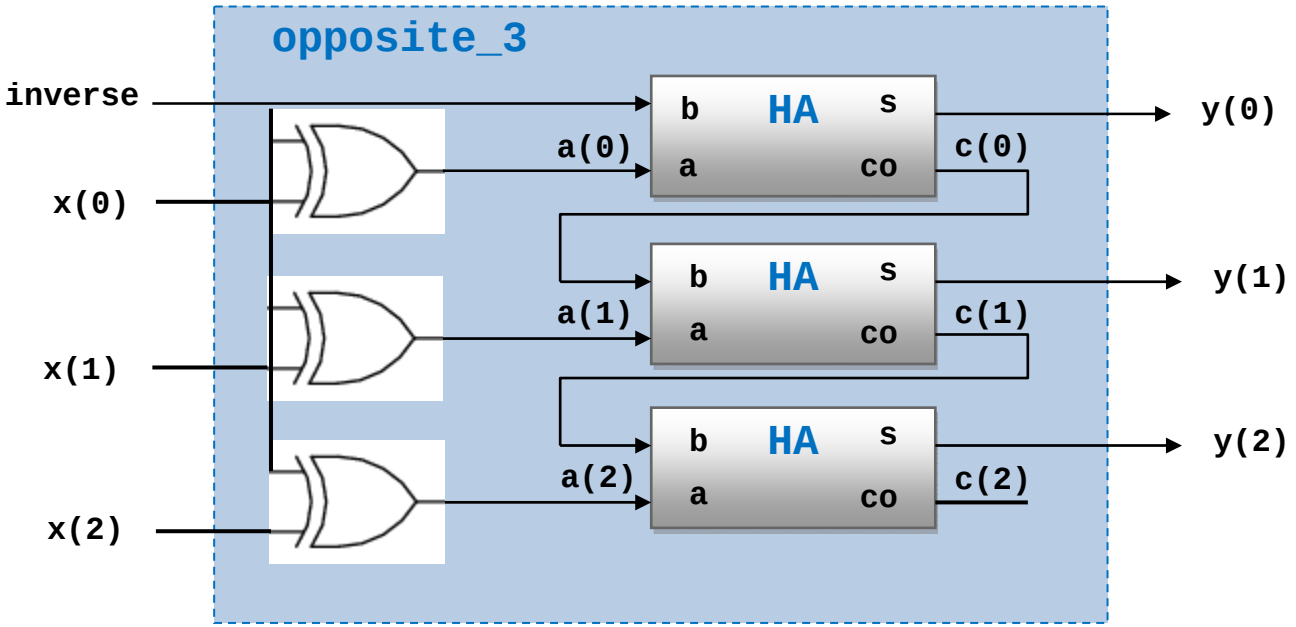

Nom de l'entité
Mode du signal
Type de signal
Nom du signal
entity tempo1ms is
port (
a : in std_logic;
rst, clk : in std_logic;
t : out std_logic);
end tempo1ms;

2.2. Description d'une entité générique

```
entity nom_entité is
generic (
     nom_paramètre {, nom_paramètre }: type_paramètre
          [ := expression_statique]
     {; nom_paramètre {, nom_paramètre }: type_paramètre
          [ := expression_statique]}
port (
   [signal] nom_signal {,nom_signal}: [mode] type_signal
 {;[signal] nom_signal {,nom_signal}: [mode] type_signal}
);

end [entity] nom_entité;
```

❑ Exemple : calcul de l'opposé arithmétique

Schéma de principe pour une fonction de dimension 3



Paramètre générique

Valeur par défaut

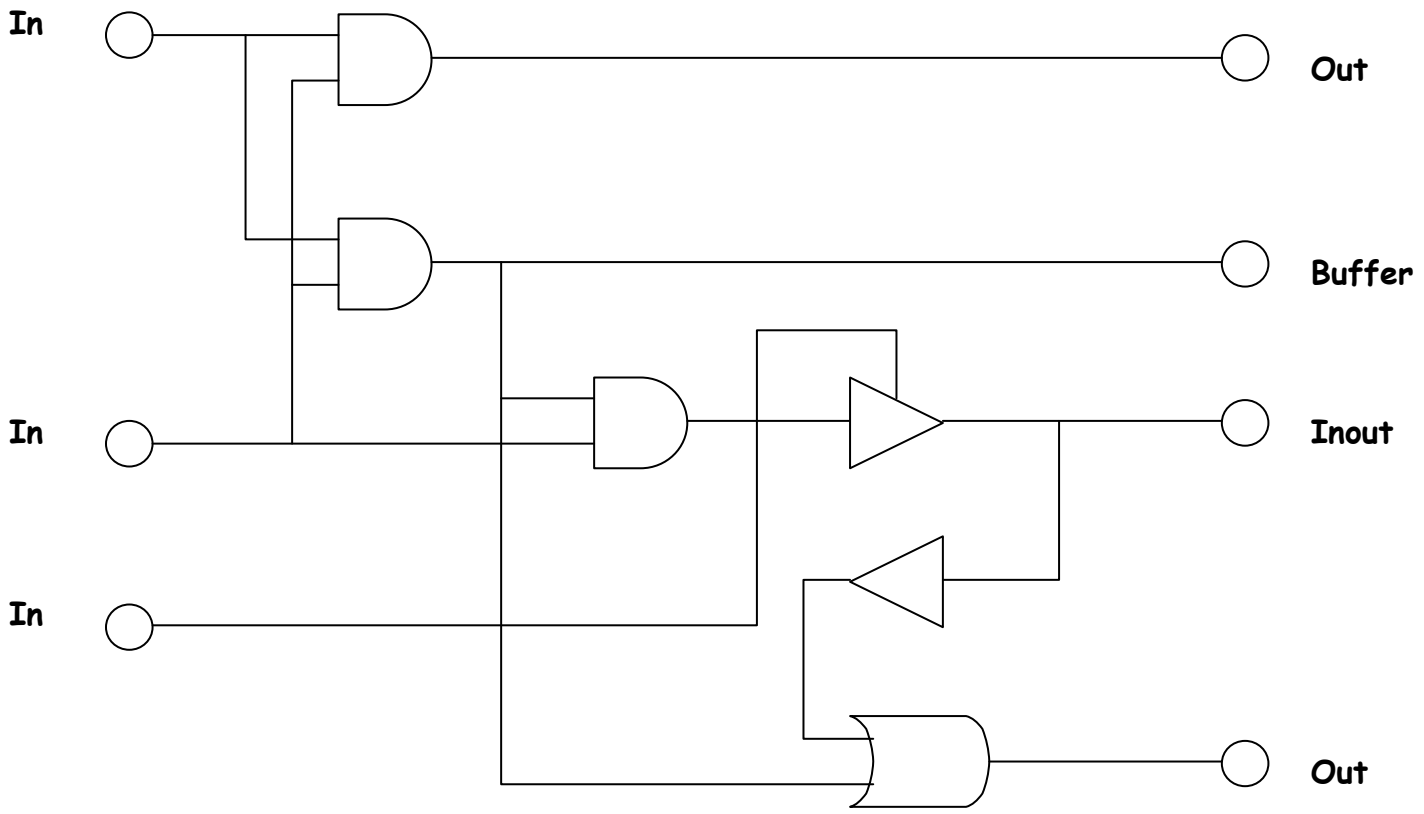
```
entity opposite_n is

    generic (n : integer := 3);
    port (
      x             : in  std_logic_vector(n-1 downto 0);
      inverse       : in  std_logic;
      y             : out  std_logic_vector (n-1 downto 0));

end opposite_n;
```

2.3. Les modes des ports

VHDL définit quatre modes qui déterminent le sens de transfert d'une donnée au travers du port :

in	L'entité lit un signal d'entrée fourni par l'extérieur (ex : load, reset, clock, données unidirectionnelles)
out	L'entité fournit un signal de sortie, mais ne peut pas relire ce signal
buffer	L'architecture de l'entité fabrique un signal utilisable en sortie, qui peut aussi être relu par l'entité comme un signal interne (ex : sorties d'un compteur dont l'état doit être testé)
inout	Le signal est bidirectionnel : en sortie, il est fourni par l'entité; en entrée, il est fourni par l'extérieur. Ce mode autorise aussi le bouclage interne (ex : bus de données)

Par défaut, le mode d'un port est in.



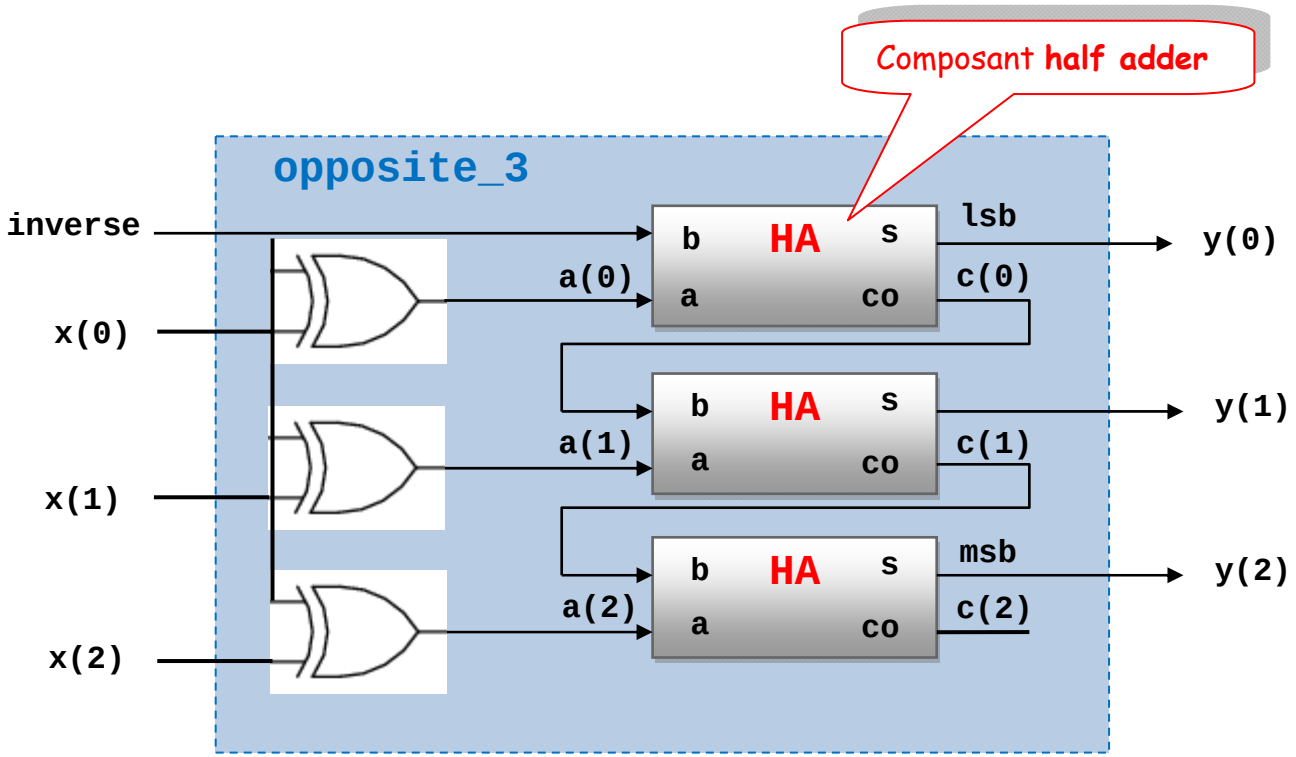
3. Le corps d'architecture

Le corps d'architecture décrit le fonctionnement interne du bloc logique

3.1. Syntaxe

```
architecture nom_architecture of nom_entité is
   { déclaration_de_composant
   | déclaration_de_constante
   | déclaration_de_signal_interne
   | déclaration_de_type
   | déclaration_d'alias}
begin
   { instruction_concurrente_d'assignation_de_signal
   | instruction_concurrente_d'instanciation_de_composant
   | instruction_concurrente_de_processus
   | instruction_de_génération}
end [architecture] [nom_architecture];
```

3.2. Déclarations dans l'architecture



```
architecture behavioral of opposite_3 is

-- déclaration de composant
     component ha port(
          a, b : in std_logic;
          co, s : out std_logic);
     end component;

-- déclaration des signaux internes
     signal a, c : std_logic_vector(n-1 downto 0);

-- déclaration d'alias
     alias lsb : std_logic is y(0) ;
     alias msb : std_logic is y(n-1) ;

begin
...

end behavioral ;
```

3.3. Instructions concurrentes

3.3.1. Propriétés

❑ **L'ordre d'écriture des instructions n'a pas d'importance (c'est le parallélisme)**

- Une instruction concurrente décrit une opération qui porte sur des signaux (entrée, interne) pour produire d'autres signaux (interne, sortie)
- Tous les signaux mis en jeu dans l'architecture sont **disponibles au même moment**

❑ **Le corps d'architecture est décrit dans un ou plusieurs styles**

- Le style n'est pas imposé par le type de logique (combinatoire ou séquentielle) ou le type de traitement des données (parallèle ou séquentiel)
- Le style est **choisi** pour apporter **concision** ou par **préférence** personnelle

Trois styles peuvent coexister au sein d'une même architecture

3.3.2. Classification des styles de description

❑ Description flot de données : instructions concurrentes d'assignation de signal

- Description de la manière dont les données circulent de signal en signal, ou d'une entrée vers une sortie

- Trois types d'instructions :

```
étiquette : … <= …
étiquette : … <= … when … else …
étiquette : with … select … <= … when …
```

- L'ordre d'écriture des instructions d'assignation de signaux est quelconque

❑ Description structurelle : instructions concurrentes d'instanciation de composant

- Interconnexion de **composants** (**components**), à la manière d'un schéma, mais sous forme d'une **liste**
- Dans l'architecture utilisatrice, un composant peut être considéré comme une **boîte noire**
- Un composant est **instancié** à l'aide d'une instruction d'appel de composant :

étiquette : nom_composant **port map** (liste_des_entrées_et_sorties);

- L'ordre d'écriture des instructions d'instanciation de composants est quelconque
- Par définition, un composant est aussi un système logique (un **sous-sytème**) ; à ce titre, il doit aussi être décrit par un couple (entity, architecture) dans lequel sont définis ses entrées-sorties et son comportement
- Le corps d'architecture du composant (le **comportement**) est décrit selon un ou plusieurs styles ; par exemple, un composant peut faire appel à d'autres composants
- Un composant peut être rangé dans une **bibliothèque**

❑ Description comportementale : instructions concurrentes d'appel de processus

- Certains comportements peuvent être décrits de façon **algorithmique** ; il faut alors les définir comme des **processus**
- Dans l'architecture utilisatrice, un processus est considéré comme une instruction concurrente
- Instruction d'appel de processus :

étiquette: **process** déclarations **begin** instructions_séquentielles **end process**;

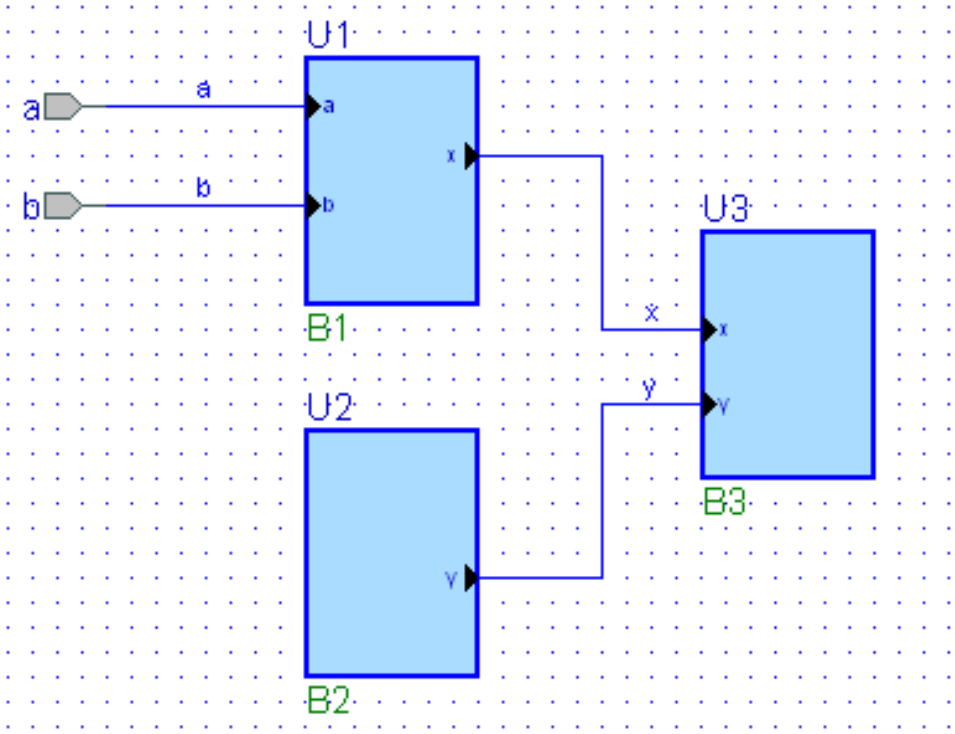
- L'ordre d'écriture des instructions d'appel de processus est quelconque
- Un processus **contient** des instructions **séquentielles** qui ne servent qu'à traduire simplement et efficacement, sous forme d'un algorithme, le comportement d'un sous-ensemble matériel
- **Des instructions séquentielles (un algorithme) peuvent décrire un système combinatoire ; a contrario, une instruction concurrente peut décrire un système séquentiel !!**
- À l'intérieur d'un processus, trois types d'instruction d'assignation de signaux et deux types d'instruction d'itération :

```
… <= …
if … then … else …
case … when …
for … loop …
while … loop …
```

- L'**ordre** d'écriture des instructions à l'intérieur du processus est **déterminant**

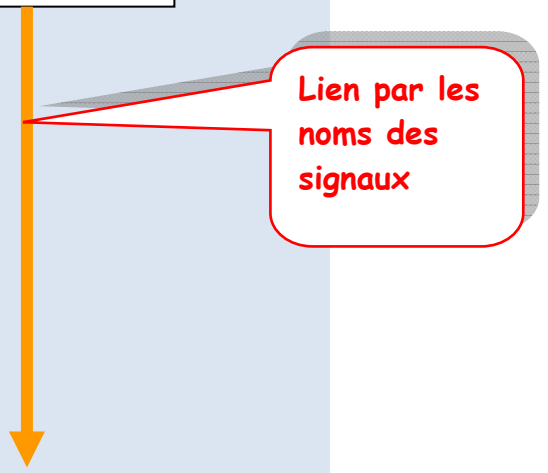
3.3.3. Exemple 1 : architecture comportant des styles différents

❑ **Conception schématique à l'aide d'un éditeur de schémas (par exemple** Active-HDL)

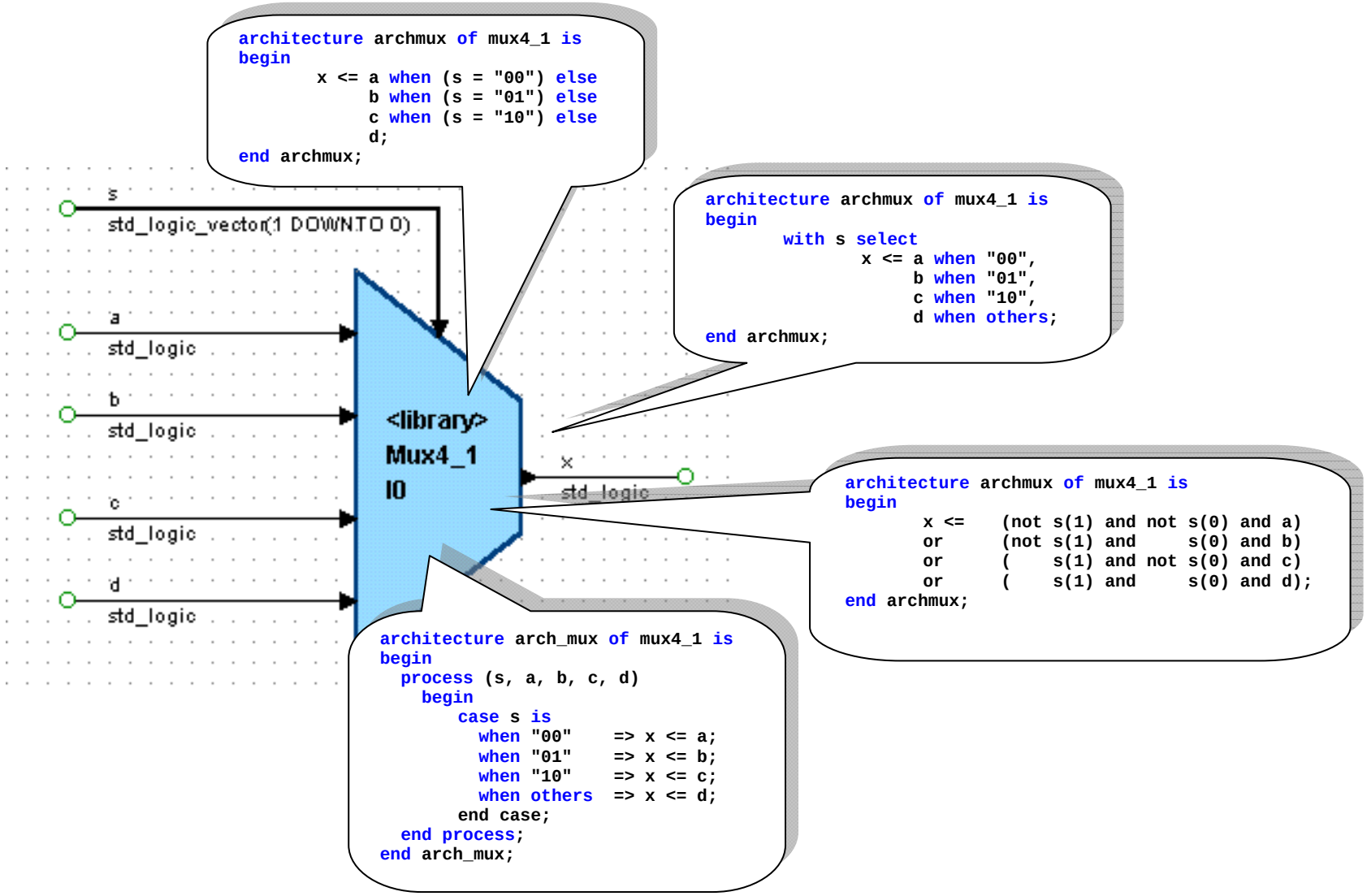


❑ Conception directe VHDL à l'aide d'un éditeur de texte

```
entity systeme is port(
...

architecture arch_systeme of systeme is
  signal x,y ...          -- liste des signaux internes
...
begin

-- B1 (Description par flot de données)

   -- Instructions concurrentes d'assignations de
   -- signaux

   B1 : x <= a and b;
   ...


-- B2 (Description comportementale)

   B2 : process (...)
   -- Instructions séquentielles
   y <= ...

   end process;


-- B3 (Description structurelle)

   B3 : composant_B3 port map (x, y,...);

end arch_systeme;
```



3.3.4. Exemple 2 : styles différents pour un même bloc logique




```
architecture archmux of mux4_1 is
begin
        x <= a when (s = "00") else
             b when (s = "01") else
             c when (s = "10") else
             d;
end archmux;
```

```
architecture archmux of mux4_1 is
begin
        with s select
                x <= a when "00",
                     b when "01",
                     c when "10",
                     d when others;
end archmux;
```

```
architecture archmux of mux4_1 is
begin
        x <=    (not s(1) and not s(0) and a)
        or      (not s(1) and     s(0) and b)
        or      (    s(1) and not s(0) and c)
        or      (    s(1) and     s(0) and d);
end archmux;
```

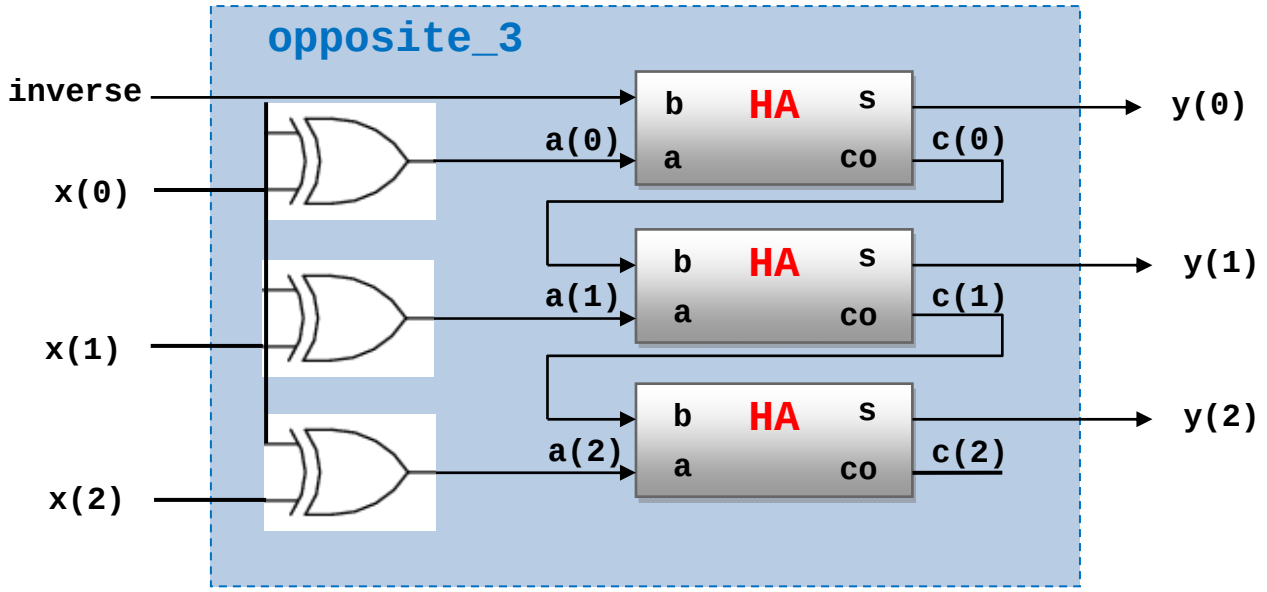
```
architecture arch_mux of mux4_1 is
begin
  process (s, a, b, c, d)
    begin
       case s is
         when "00"    => x <= a;
         when "01"    => x <= b;
         when "10"    => x <= c;
         when others  => x <= d;
       end case;
  end process;
end arch_mux;
```

❑ Résultat de compilation

Dans cet exemple, quel que soit le style de description employé, les équations générées par le compilateur sont les mêmes.

```
x    =    d *  s(0) *  s(1)
       + c * /s(0) *  s(1)
       + b *  s(0) * /s(1)
       + a * /s(0) * /s(1)
```

3.3.5. Exemple 3 : fonction opposé arithmétique opposite_n




```
begin
      a(0) <= x(0) xor inverse;
      ha0 : ha port map (a(0),inverse,c(0),y(0));

      haloop : for i in 1 to n-1 generate
            a(i) <= x(i) xor inverse;
            han : ha port map (a(i),c(i-1),c(i),y(i));
      end generate;
end Behavioral;
```