VHDL - Logique programmable

Partie 5

- **Description comportementale (les processus)**
- **Itération d'instructions concurrentes**

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. Description comportementale 3
1.1. La notion de processus 3
1.1.1. Propriétés générales des processus 3
1.1.2. Le processus convient à tous les types de logique 4
1.1.3. Les processus sont concurrents 5
1.2. Structure d'un processus 7
1.3. Activation d'un processus 8
1.4. Instruction séquentielle d'assignation de signal <= 9
1.4.1. Comportement d'un processus du point de vue des signaux 10
1.4.2. Règles pour les signaux 11
1.5. Instruction séquentielle d'assignation de variable := 13
1.5.1. Comportement d'un processus du point de vue des variables 14
1.5.2. Règles pour les variables 14
1.5.3. Applications des variables 14
1.5.4. Exemples de comportement des variables 15
1.6. Instruction séquentielle if … then … else … 17
1.6.1. Influence de l'ordre des instructions 19
1.6.2. Multiplexeur 4 vers 1, liste de conditions non exhaustive, avec clause `else` 20
1.6.3. Multiplexeur 4 vers 1, clause `else` manquante 21
1.6.4. Multiplexeur 4 vers 1, clause `else` manquante, mais valeur par défaut 22
1.6.5. Multiplexeur 4 vers 1, conditions non mutuellement exclusives 23
1.7. Instruction séquentielle case … when … 24
1.7.1. Multiplexeur (4 bits) 4 vers 1, liste de valeurs de sélecteur non exhaustive, avec mot réservé `others` 25
1.7.2. Double multiplexeur (4 bits) 4 vers 1, séquence d'instructions 26
1.7.3. Démultiplexeur 1 vers 8 27
1.8. Instruction séquentielle for … loop … 29
1.9. Instruction séquentielle while … loop … 31
2. Itération d'instructions concurrentes 33
2.1. Forme itérative 33
2.2. Forme conditionnelle 34
2.3. Exemples en mode itératif 35
2.3.1. Instanciations multiples d'un même composant 35
2.3.2. Exemple1 de répétition d'assignations concurrentes de signaux 36
2.3.3. Exemple 2 de répétition d'assignations concurrentes de signaux 37

1.Description comportementale

1.1. La notion de processus

La description comportementale fait appel à des processus.

1.1.1. Propriétés générales des processus

- **Du point de vue d'une architecture utilisatrice, un processus prend la place d'une instruction concurrente**

- **D'un point de vue « interne », un processus est constitué d'instructions séquentielles, à l'instar des instructions d'un langage impératif classique (C, …)**
  - **Pour le concepteur, ces instructions séquentielles permettent l'élaboration d'un raisonnement (algorithme), « comme si » la logique décrite était exécutée par un simulateur sur un ordinateur**
  - **L'outil de synthèse, quant à lui, se sert de cette description pour générer des connexions physiques de blocs logiques dans un circuit électronique**
  - **Les variables d'un processus ne sont pas toutes synthétisables, certaines (par ex. des compteurs) peuvent être transformées par le compilateur en signaux réels (registres), d'autres (par ex. les indices de boucle, les états intermédiaires) ne sont pas matérialisées car elles ne servent qu'à la logique de l'algorithme**

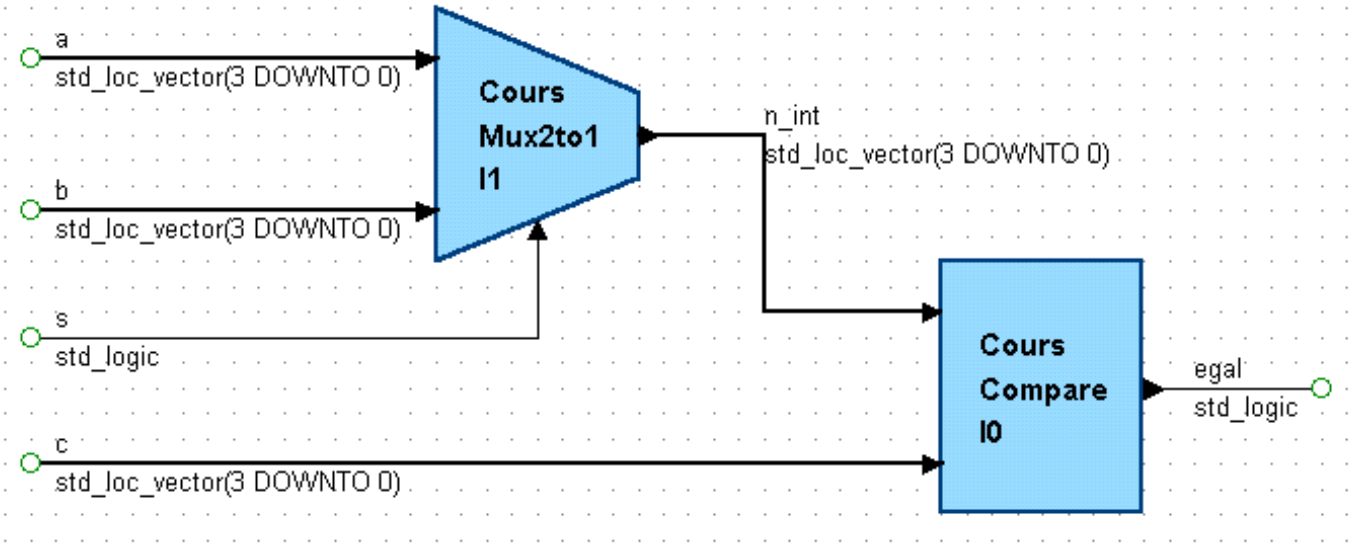
1.1.2. Le processus convient à tous les types de logique

```
mux:  x <= (d and     s(0) and     s(1)) or
           (c and not s(0) and     s(1)) or
           (b and     s(0) and not s(1)) or
           (a and not s(0) and not s(1));
```

Ces deux descriptions représentent la même fonction combinatoire

```
mux_proc: process (s, a, b, c, d)
begin
  x <= d;    -- valeur par défaut

  if s = "00" then
    x <= a;
  elsif s = "01" then
    x <= b;
  elsif s = "10" then
    x <= c;
  end if;
end process;
```

1.1.3. Les processus sont concurrents




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

entity deux_processus is port(
  a,b,c   : in std_logic_vector(3 downto 0);
  s       : in std_logic;
  egal    : out std_logic);
end deux_processus;

architecture arch_deux_processus of deux_processus is
signal n_int : std_logic_vector(3 downto 0);
begin
```

```
-- description du bloc multiplexeur
mux2to1_proc:  process (s,a,b)
begin
  case s is
    when '0'       => n_int <= a;
    when others    => n_int <= b;
  end case;
end process Mux2to1;
```

```
-- description du bloc comparateur
compare_proc:  process (n_int,c)
begin
  egal <= '0';
  if c = n_int then
     egal <= '1';
  end if;
end process Compare;

end arch_deux_processus;
```

```
DESIGN EQUATIONS
    /egal =
          /b(0) * c(0) * s
        + /b(1) * c(1) * s
        + /b(2) * c(2) * s
        + /b(3) * c(3) * s
        + b(0) * /c(0) * s
        + b(1) * /c(1) * s
        + b(2) * /c(2) * s
        + b(3) * /c(3) * s
        + /a(0) * c(0) * /s
        + /a(1) * c(1) * /s
        + /a(2) * c(2) * /s
        + /a(3) * c(3) * /s
        + a(0) * /c(0) * /s
        + a(1) * /c(1) * /s
        + a(2) * /c(2) * /s
        + a(3) * /c(3) * /s
```

1.2. Structure d'un processus

```
[étiquette :] process (liste_de_sensibilité)

              { déclaration_de_type
              | déclaration_de_constante
              | déclaration_de_variable
              | déclaration_d'alias}

          begin

              { instruction_d'assignation_de_signal
              | instruction_d'assignation_de_variable
              | instruction_if
              | instruction_case
              | instruction_for_loop}
              | instruction_while_loop}

          end process [étiquette];
```

1.3. Activation d'un processus

Il faut raisonner comme si le processus correspondait à un programme d'instructions séquentielles exécutées par un simulateur !

- Un processus est activé à chaque changement d'état de l'un quelconque des signaux auxquels il est déclaré sensible
- Une liste de sensibilité est constituée
  - pour les fonctions combinatoires : de tous les signaux lus (les entrées qui peuvent changer)
  - pour les fonctions séquentielles : de l'horloge et tous les signaux asynchrones (set, reset)

- Au moment de l'activation du processus, chaque signal, référencé dans la partie droite d'une instruction d'assignation, prend une valeur courante qu'il conservera tout au long du déroulement du processus

- Le processus se déroule à partir de la première instruction qui suit le mot clé **`begin`**

1.4. Instruction séquentielle d'assignation de signal <=

```
identificateur_signal  <=   expression_logique
                          | littéral
                          | concaténation_de_bits
                          | agrégat
                          | expression arithmétique;
```

1.4.1. Comportement d'un processus du point de vue des signaux

- Toute instruction d'assignation de signal porte sur la valeur courante des signaux qui se situent à droite de l'opérateur d'assignation **<=**
- Les instructions déterminent les signaux à modifier et planifient leur prochaine valeur
- Les nouvelles valeurs sont calculées au fur et à mesure des assignations
- L'attribution définitive des nouvelles valeurs est faite à la fin du processus, et au même moment pour tous les signaux modifiés par les instructions d'assignation

Assignation de tous les signaux, en même temps, à la fin du processus

Simule une simultanéité d'exécution; donc adapté à une description de matériel où tout est parallélisme

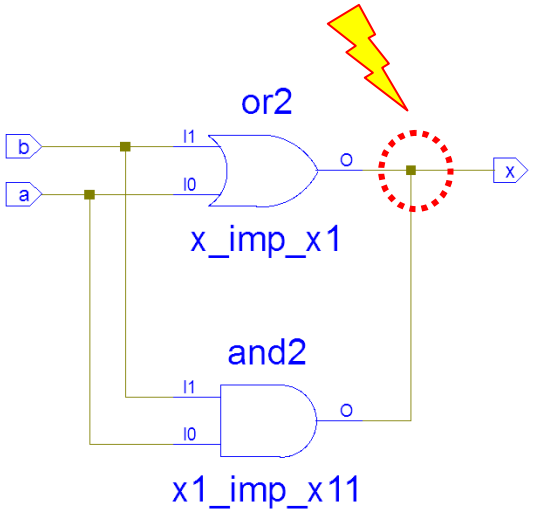
```
process (a,b,c)
begin

  y1 <= a or b;
  y2 <= a and b;
  y3 <= a xor b xor c;

end process;
```

Les valeurs de ces signaux sont celles prises à l'activation du processus, cela pour toutes leurs occurrences

1.4.2. Règles pour les signaux

- Les signaux sont globaux pour l'architecture courante

- Pour une synthèse, ne jamais assigner un même signal dans plusieurs processus, cela induirait un court-circuit (sauf pour des signaux 3 états)

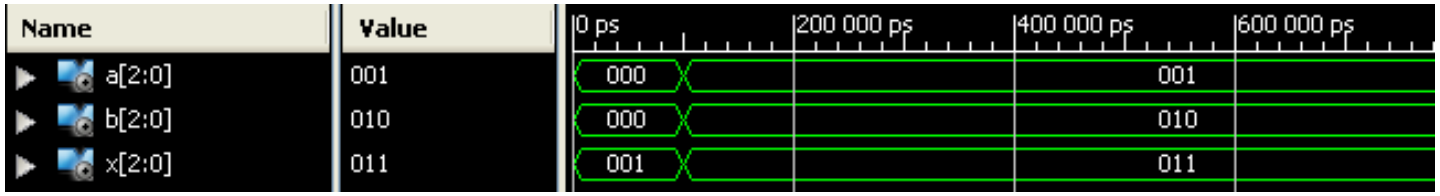


- Dans un processus combinatoire,
  - ne jamais faire plusieurs assignations sur le même signal
  - ne jamais lire et assigner le même signal
    - `x <= x and a;` `-- exemple à éviter !`

- Dans un processus séquentiel,
  - Ne jamais assigner un signal en dehors de l'instruction de contrôle de l'horloge **`clk`** ou de l'instruction **`set`** ou **`reset`** asynchrone

Cet exemple, juste pour comprendre ce qu'il se passe lorsqu'une séquence d'assignations inconditionnelles porte sur le même signal : c'est la dernière assignation qui l'emporte, donc la séquence ne sert à rien !

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity add_signal is
    Port ( a : in  STD_LOGIC_VECTOR (2 downto 0);
           b : in  STD_LOGIC_VECTOR (2 downto 0);
           x : out  STD_LOGIC_VECTOR (2 downto 0));
end add_signal;

architecture Behavioral of add_signal is
     signal t : std_logic_vector(2 downto 0) := "000";
begin

process (a,b)
begin
     t <= a + 1;
     t <= b + 1;
end process;

x <= t;

end Behavioral;
```



Cette instruction permet d'exporter le signal interne t vers la sortie **x**

1.5. Instruction séquentielle d'assignation de variable :=

```
identificateur_variable  :=    expression_logique
                             | littéral
                             | concaténation_de_bits
                             | agrégat
                             | expression arithmétique;
```

1.5.1. Comportement d'un processus du point de vue des variables

Dans un processus, les variables ne sont pas assignées de la même façon que les signaux.

- Les variables sont mises à jour immédiatement, dès leur assignation par un opérateur **:=**
- Au fur et à mesure, les modifications des variables se propagent vers les instructions suivantes (comme dans un langage classique)

1.5.2. Règles pour les variables

- Avant de lire une première fois une variable, il faut l'avoir assignée auparavant

1.5.3. Applications des variables

- Les variables facilitent les descriptions algorithmiques
- En synthèse : on limite en général leur usage à la gestion des boucles

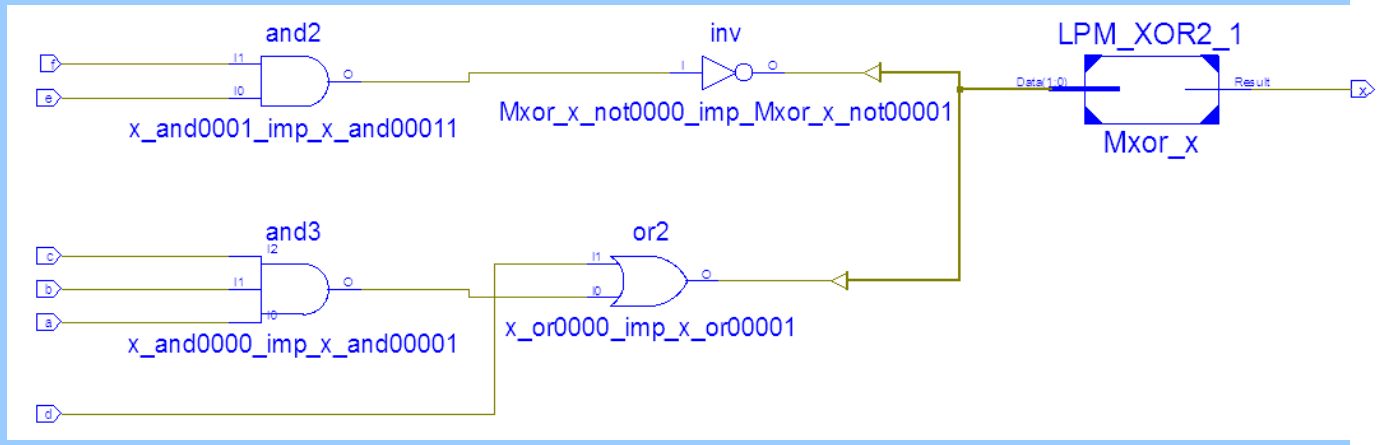
1.5.4. Exemples de comportement des variables

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity add_variable is
    Port ( a : in  STD_LOGIC_VECTOR (2 downto 0);
           x : out  STD_LOGIC_VECTOR (2 downto 0));
end add_variable;

architecture Behavioral of add_variable is
begin
process (a)
     variable t : std_logic_vector(2 downto 0) := "000";
begin
     t := a + 1;
     t := t + 1;
     t := t + 1;

     x <= t;
end process;
end Behavioral;
```

Les modifications de la variable **t** se propagent d'instruction en instruction

Cette instruction permet d'exporter la viariable t vers la sortie **x**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity comb_variable is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           c : in  STD_LOGIC;
           d : in  STD_LOGIC;
           e : in  STD_LOGIC;
           f : in  STD_LOGIC;
           x : out  STD_LOGIC);
end comb_variable;

architecture Behavioral of
comb_variable is

begin

process (a,b,c,d,e,f)
     variable temp_v : std_logic;
begin
     temp_v := a and b;
     temp_v := temp_v and c;
     temp_v := temp_v or d;
     temp_v := temp_v xor (e nand f);
     x <= temp_v;
end process;

end Behavioral;
```



1.6. Instruction séquentielle if … then … else …

```
 if condition then
    séquence_d'instructions_séquentielles

{elsif condition then
    séquence_d'instructions_séquentielles}

[else
    séquence_d'instructions_séquentielles]

 end if ;
```

Condition ::= expression booléenne (de valeur `true` ou `false`) comportant :

- ☞ des opérateurs relationnels: `=`, `/=`, `<`, `<=`, `>`, `>=`
- ☞ des opérateurs logiques: `and`, `or`, `xor`

Règles

- **A tout moment, le signal prend la valeur correspondant à la première condition évaluée à `true`. ⇨ L'ordre de déclaration des conditions permet éventuellement de définir des priorités pour les signaux en jeu**

- **Les conditions listées ne sont pas nécessairement mutuellement exclusives (c.-à-d. tous les signaux en jeu n'apparaissent pas obligatoirement dans l'expression d'une condition)**

- **La liste des conditions n'est pas nécessairement exhaustive (une liste explicite de tous les cas n'est pas obligatoire); si elle ne l'est pas :**
  - **si le mot clé `else` est présent, il rassemble les conditions manquantes et détermine les actions communes à entreprendre**
  - **sinon, lorsque aucune condition n'est vérifiée, il y a mémorisation implicite pour tous les signaux qui ont été assignés dans les branches**

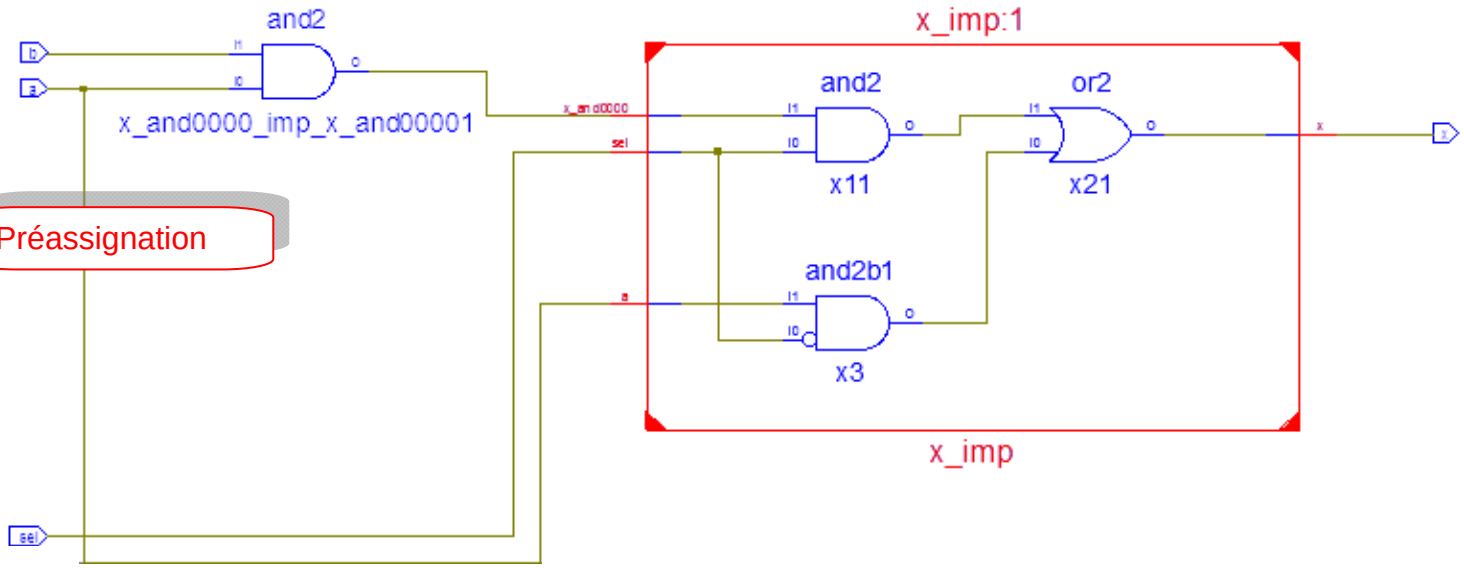
- **Si un signal reçoit une assignation dans une branche `if` ou `elsif`, alors il doit aussi recevoir une assignation**
  - **soit dans toutes les autres branches (y compris la branche `else`)**
  - **soit dans une instruction d'assignation qui précède l'instruction `if`**

  Sinon il y a mémorisation implicite (en logique séquentielle asynchrone, ou en logique séquentielle synchrone s'il est fait référence à un front d'horloge)

1.6.1. Influence de l'ordre des instructions

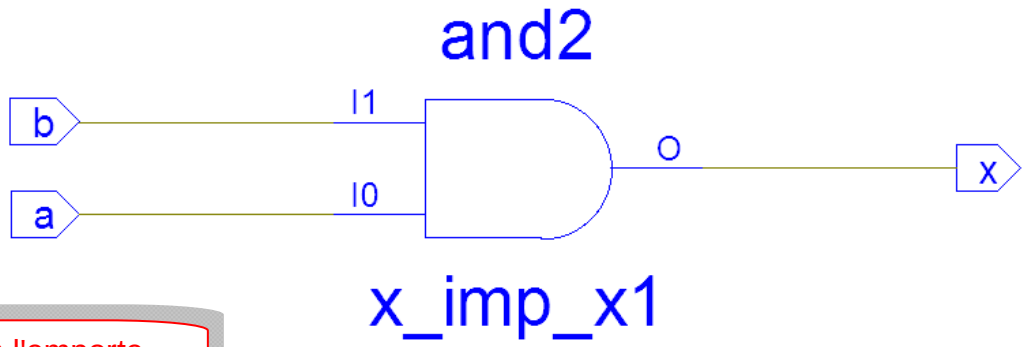
```
process(a,b,sel)

begin
     x <= a and b;
     if sel = '0' then
          x <= a;
     end if;
end process;
```

Préassignation




```
process(a,b,sel)

begin
     if sel = '0' then
          x <= a;
     end if;
     x <= a and b;
end process;
```

Cette dernière instruction l'emporte



1.6.2. Multiplexeur 4 vers 1, liste de conditions non exhaustive, avec clause `else`

```
entity mux4to1_v5 is port(
  a, b, c, d : in std_logic;
  s          : in std_logic_vector(1 downto 0);
  x          : out std_logic);
end mux4to1_v5;

architecture arch_mux4to1_v5 of mux4to1_v5 is
begin

process (s, a, b, c, d)
begin
if s = "00" then
    x <= a;
elsif s = "01" then
    x <= b;
elsif s = "10" then
    x <= c;
else                        -- liste non exhaustive
    x <= d;
end if;
end process;

end arch_mux4to1_v5;
```

```
DESIGN EQUATIONS

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

1.6.3. Multiplexeur 4 vers 1, clause `else` manquante

❒ Mémorisation implicite

```
entity mux4to1_v5 is port(
  a, b, c, d : in std_logic;
  s          : in std_logic_vector(1 downto 0);
  x          : out std_logic);
end mux4to1_v5;

architecture arch_mux4to1_v5 of mux4to1_v5 is
begin
  process (s, a, b, c, d)
    begin
      if s = "00" then
        x <= a;
      elsif s = "01" then
        x <= b;
      elsif s = "10" then
        x <= c;       -- clause else manquante
      end if;
end process;
end arch_mux4to1_v5;
```

```
DESIGN EQUATIONS

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

Si s = "11", le signal x conserve la valeur qu'il avait au moment de l'activation du processus.

1.6.4. Multiplexeur 4 vers 1, clause `else` manquante, mais valeur par défaut

```
entity mux4to1_v6 is port(
a, b, c, d:  in std_logic;
s:       in std_logic_vector(1 downto 0);
x:       out std_logic);
end mux4to1_v6;

architecture arch_mux4to1_v6 of mux4to1_v6 is
begin
  process (s, a, b, c, d)
  begin
    x <= d;       -- valeur par défaut
    if s = "00" then
      x <= a;
    elsif s = "01" then
      x <= b;
    elsif s = "10" then
      x <= c;
    end if;
  end process;
end arch_mux4to1_v6;
```

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

Si `s = "11"`, le signal x conserve sa valeur (mémorisation implicite); il prend donc la valeur qui lui a été assignée par l'instruction `x <= d` juste avant l'instruction `if`.

1.6.5. Multiplexeur 4 vers 1, conditions non mutuellement exclusives

❒ Priorité

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

entity mux4to1_v7 is port(
  a, b, c : in std_logic;
  s       : in std_logic_vector(1 downto 0);
  x       : out std_logic);
end mux4to1_v7;


architecture arch_mux4to1_v7 of mux4to1_v7 is
begin
  process (s, a, b, c)
    begin
      if s(0) = '0' then -- conditions non exclusives
        x <= a;
      elsif s(1) = '0' then
        x <= b;
      else
        x <= c;
      end if;
  end process;
end arch_mux4to1_v7;
```

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

1.7. Instruction séquentielle case … when …

```
case sélecteur is
    {when valeur_sélecteur =>
        séquence_d'instructions_séquentielles}
end case ;
```

sélecteur ::= identificateur_de_signal | variable_entière | variable_d'un_type_énuméré
valeur_sélecteur ::= intervalle_discret | littéral | choix_multiple | concaténation | agrégat | others
choix_multiple ::= valeur { | valeur}
intervalle_discret ::= valeur_basse to valeur_haute | valeur_haute downto valeur_basse | nom_signal'range

- Toutes les valeurs de sélecteur doivent être mutuellement exclusives
- Si une liste explicite de toutes les valeurs de sélecteur n'est pas fournie, les valeurs manquantes doivent être rassemblées par le mot réservé `others`
- L'opérateur | sert à composer un choix multiple
- Si un signal reçoit une assignation dans une branche, alors il doit aussi recevoir une assignation
  - soit dans toutes les autres branches (y compris la branche `others`)
  - soit dans une instruction d'assignation qui précède l'instruction `case`,

    sinon il y a **mémorisation implicite** (en logique séquentielle asynchrone, ou en logique séquentielle synchrone s'il est fait référence à un front d'horloge)

1.7.1. Multiplexeur (4 bits) 4 vers 1, liste de valeurs de sélecteur non exhaustive, avec mot réservé `others`

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

entity mux4to1_4bits_v2 is port(
  a, b, c, d : in std_logic_vector(3 downto 0);
  s          : in std_logic_vector(1 downto 0);
  x          : out std_logic_vector(3 downto 0));
end mux4to1_4bits_v2;

architecture arch_mux4to1_4bits_v2 of mux4to1_4bits_v2 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_ mux4to1_v2;
```

```
DESIGN EQUATIONS
    x(3) =
          /s(1) * /s(0) * a(3)
        + s(1) * /s(0) * c(3)
        + /s(1) * s(0) * b(3)
        + s(1) * s(0) * d(3)
    x(2) =
          s(1) * s(0) * d(2)
        + s(1) * /s(0) * c(2)
        + /s(1) * s(0) * b(2)
        + /s(1) * /s(0) * a(2)
    x(1) =
          s(1) * s(0) * d(1)
        + s(1) * /s(0) * c(1)
        + /s(1) * s(0) * b(1)
        + /s(1) * /s(0) * a(1)
    x(0) =
          s(1) * s(0) * d(0)
        + s(1) * /s(0) * c(0)
        + /s(1) * s(0) * b(0)
        + /s(1) * /s(0) * a(0)
```

1.7.2. Double multiplexeur (4 bits) 4 vers 1, séquence d'instructions

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

entity two_mux4to1_4bits_v3 is port(
  a, b, c, d : in std_logic_vector(3 downto 0);
  s          : in std_logic_vector(1 downto 0);
  x,y        : out std_logic_vector(3 downto 0));
end two_mux4to1_4bits_v3;

architecture arch_two_mux4to1_4bits_v3 of two_mux4to1_4bits_v3 is
begin
  process (s, a, b, c, d)
    begin
      case s is
        when "00" =>
          x <= a;   -- séquence d'instructions
          y <= d;
        when "01" =>
          x <= b;
          y <= c;
        when "10" =>
          x <= c;
          y <= b;
        when others =>
          x <= d;
          y <= a;
      end case;
  end process;

end arch_two_mux4to1_4bits_v3;
```

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

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

    x(2) =
          d(2) * s(0) * s(1)
        + c(2) * /s(0) * s(1)
        + b(2) * s(0) * /s(1)
        + a(2) * /s(0) * /s(1)

    x(3) =
          d(3) * s(0) * s(1)
        + c(3) * /s(0) * s(1)
        + b(3) * s(0) * /s(1)
        + a(3) * /s(0) * /s(1)

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

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

    y(2) =
          a(2) * s(0) * s(1)
        + b(2) * /s(0) * s(1)
        + c(2) * s(0) * /s(1)
        + d(2) * /s(0) * /s(1)

    y(3) =
          a(3) * s(0) * s(1)
```

1.7.3. Démultiplexeur 1 vers 8

☹ Proposition n°1 (incorrecte)

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

entity demux1to8_v4 is port(
  i : in std_logic;
  s : in std_logic_vector(2 downto 0);
  o : out std_logic_vector(7 downto 0));
end demux1to8_v4;

architecture archdemux1to8_v4 of demux1to8_v4 is
begin

process (i,s)
begin
  case s is
    when o"0" => o(0) <= i;   -- sélecteur exprimé en octal
    when o"1" => o(1) <= i;
    when o"2" => o(2) <= i;
    when o"3" => o(3) <= i;
    when o"4" => o(4) <= i;
    when o"5" => o(5) <= i;
    when o"6" => o(6) <= i;
    when o"7" => o(7) <= i;
    when others => o <= x"00";
  end case;
end process;
end archdemux1to8_v4;
```

```
DESIGN EQUATIONS (for CPLDs)
    o(0).D = i
    o(0).LH = /s(0) * /s(1) *
/s(2)

    o(1).D = i
    o(1).LH = s(0) * /s(1) *
/s(2)

    o(2).D = i
    o(2).LH = /s(0) * s(1) *
/s(2)

    o(3).D = i
    o(3).LH = s(0) * s(1) *
/s(2)

    o(4).D = i
    o(4).LH = /s(0) * /s(1) *
s(2)
```

L'instruction `when o"0" => o(0) <= i` ne mentionne pas les sorties `o(7 downto 1)`. Celles-ci conservent donc leur valeur (mémorisation implicite par l'intermédiaire de latches D).

☺ Proposition n°2 (correcte)

```
process (i,s)
begin
  o <=  x"00";      -- initialisation du vecteur de sortie

  case s is         -- modification de la composante sélectionnée
    when o"0" => o(0) <= i;
    when o"1" => o(1) <= i;
    when o"2" => o(2) <= i;
    when o"3" => o(3) <= i;
    when o"4" => o(4) <= i;
    when o"5" => o(5) <= i;
    when o"6" => o(6) <= i;
    when o"7" => o(7) <= i;
    when others => o <= x"00";
  end case;
end process;
```

La clause `others` est obligatoire, sinon erreur de syntaxe

`"A value is missing in case"`

1.8. Instruction séquentielle for … loop …

```
[étiquette :]
for variable_de_boucle in intervalle_discret loop

   {séquence_d'instructions}

end loop [étiquette];
```

intervalle_discret ::= valeur_basse `to` valeur_haute
| valeur_haute `downto` valeur_basse
| nom_signal`'range`

- La variable de boucle n'a pas à être déclarée et peut être utilisée à l'intérieur de la boucle
- La séquence peut contenir des instructions conditionnelles ou d'autres boucles
- ☝ L'intervalle doit être fixé au moment de la compilation (ne peut pas être dépendant d'une entrée du système)

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

entity boucle_for is
  port(
       entree : in std_logic_vector(7 downto 0);
       valid  : in std_logic;
       sortie : out std_logic_vector(7 downto 0));
end boucle_for;


architecture arch_boucle_for of boucle_for is
begin

process (entree, valid)
begin

elaboration_sortie:
for i in entree'range loop
    sortie(i) <= valid and entree(i);
end loop elaboration_sortie;

end process;

end arch_boucle_for;
```

```
DESIGN EQUATIONS
    sortie(0) = entree(0) * valid

    sortie(1) = entree(1) * valid

    sortie(2) = entree(2) * valid

    sortie(3) = entree(3) * valid

    sortie(4) = entree(4) * valid

    sortie(5) = entree(5) * valid

    sortie(6) = entree(6) * valid

    sortie(7) = entree(7) * valid
```

1.9. Instruction séquentielle while … loop …

```
[étiquette :]
while condition loop

    {séquence_d'instructions}

end loop [étiquette];
```

- Tant que la condition est vraie (`true`) la séquence d'instructions est répétée.
- ☝ La condition doit être statique (fixée au moment de la compilation)

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

entity boucle_while is
  port(
       entree :     in std_logic_vector(7 downto 0);
       valid :      in std_logic;
       sortie :     out std_logic_vector(7 downto 0));
end boucle_while;


architecture arch_boucle_while of boucle_while is
begin

process (entree, valid)
  variable i : integer;

begin

i := 7;
elaboration_sortie:
while i >= 0 loop
     sortie(i) <= valid and entree(i);
     i := i - 1;
end loop elaboration_sortie;

end process;

end arch_boucle_while;
```

```
DESIGN EQUATIONS
    sortie(0) = entree(0) * valid

    sortie(1) = entree(1) * valid

    sortie(2) = entree(2) * valid

    sortie(3) = entree(3) * valid

    sortie(4) = entree(4) * valid

    sortie(5) = entree(5) * valid

    sortie(6) = entree(6) * valid

    sortie(7) = entree(7) * valid
```

2.Itération d'instructions concurrentes

- L'instruction **generate** permet de réitérer un ensemble d'instructions concurrentes
- Elle s'utilise dans la zone des instructions concurrentes

2.1. Forme itérative

```
étiquette : for variable in intervalle_discret generate

          {instruction_concurrente}

           end generate [étiquette] ;
```

intervalle_discret ::= valeur_basse **to** valeur_haute
| valeur_haute **downto** valeur_basse
| nom_signal**'range**

2.2. Forme conditionnelle

```
étiquette :  if condition generate

             {instruction_concurrente}

              end generate ;
```

Cette instruction permet :

- **dans le mode itératif, de générer plusieurs répliques d'un composant ou d'équations logiques**
- **dans le mode conditionnel, de générer sous certaines conditions un composant particulier ou des équations particulières**

En synthèse, la forme conditionnelle est toujours utilisée à l'intérieur d'une forme itérative; la condition est alors dépendante de la variable de boucle. Dans cette forme il n'y a ni clause `else`, ni clause `elsif`. La variable de boucle n'a pas à être déclarée.

2.3. Exemples en mode itératif

2.3.1. Instanciations multiples d'un même composant

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

entity ex_anti_rebonds is
     port(
     inter          :in std_logic_vector(0 to 15);
     inter_filtre   :out std_logic_vector(0 TO 7));
end ex_anti_rebonds;


library lib_cours;
     use lib_cours.p_latch.all;

architecture arch_ex_anti_rebonds of ex_anti_rebonds is

alias intera : std_logic_vector(0 to 7) is inter(0 to 7);
alias interb : std_logic_vector(0 to 7) is inter(8 to 15);

begin

filtrage_entrees :
for i in 0 to 7 generate
     entree_i : c_latch_rs port map (intera(i), interb(i),
                inter_filtre(i));
end generate;

end arch_ex_anti_rebonds;
```

```
Equations générées

inter_filtre(0) =
          inter(8) * inter_filtre(0).CMB + /inter(0)

    inter_filtre(1) =
          inter(9) * inter_filtre(1).CMB + /inter(1)

    inter_filtre(2) =
          inter(10) * inter_filtre(2).CMB + /inter(2)

    inter_filtre(3) =
          inter(11) * inter_filtre(3).CMB + /inter(3)

etc...
```

2.3.2. Exemple1 de répétition d'assignations concurrentes de signaux

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

entity boucle_generate is
  port(
       entree : in std_logic_vector(7 downto 0);
       valid  : in std_logic;
       sortie : out std_logic_vector(7 downto 0));
end boucle_generate;


architecture arch_boucle_generate of boucle_generate is
begin


elaboration_sortie:
for i in entree'range generate
    sortie(i) <= valid and entree(i);
end generate elaboration_sortie;


end arch_boucle_generate;
```

```
DESIGN EQUATIONS
    sortie(0) = entree(0) * valid

    sortie(1) = entree(1) * valid

    sortie(2) = entree(2) * valid

    sortie(3) = entree(3) * valid

    sortie(4) = entree(4) * valid

    sortie(5) = entree(5) * valid

    sortie(6) = entree(6) * valid

    sortie(7) = entree(7) * valid
```

2.3.3. Exemple 2 de répétition d'assignations concurrentes de signaux

❒ Démultiplexeur 1 vers 8

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

library lib_cours;
use lib_cours.p_conv_functions.all;


entity demux1to8_v6 is port(
  i :    in std_logic;
  s : in std_logic_vector(2 downto 0);
  o : out std_logic_vector(7 downto 0));
end demux1to8_v6;

architecture archdemux1to8_v6 of demux1to8_v6 is
begin

demux:
for n in 0 to 7 generate
  o(n) <= i when n = f_slv_to_i(s) else
          '0';
end generate demux;

end archdemux1to8_v6;
```

```
DESIGN EQUATIONS

    o(0) = i * /s(0) * /s(1) * /s(2)

    o(1) = i * s(0) * /s(1) * /s(2)

    o(2) = i * /s(0) * s(1) * /s(2)

    o(3) = i * s(0) * s(1) * /s(2)

    o(4) = i * /s(0) * /s(1) * s(2)

    o(5) = i * s(0) * /s(1) * s(2)

    o(6) = i * /s(0) * s(1) * s(2)

    o(7) = i * s(0) * s(1) * s(2)
```