Cours de Systèmes numériques

Ch. 4 : « Réutilisabilité »

Nicolas.Schroeter@hefr.ch

Design «réutilisabilité»

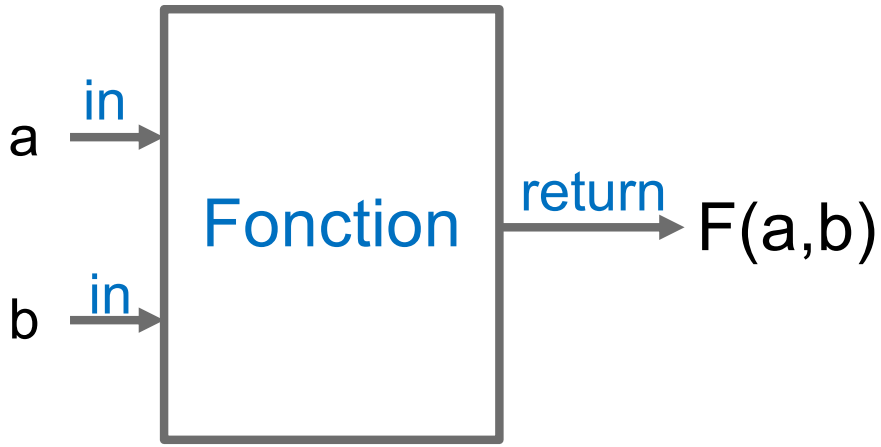
- Pourquoi:
  - Augmenter la productivité, gain de temps de développement
  - Limiter la génération d'erreurs (bugs)
  - Disposer d'une bibliothèque de descriptions ré-utilisable dans plusieurs projets
- Comment:
  - Ecrire des descriptions paramétrables
  - Profiter des mécanismes avancés du langage VHDL

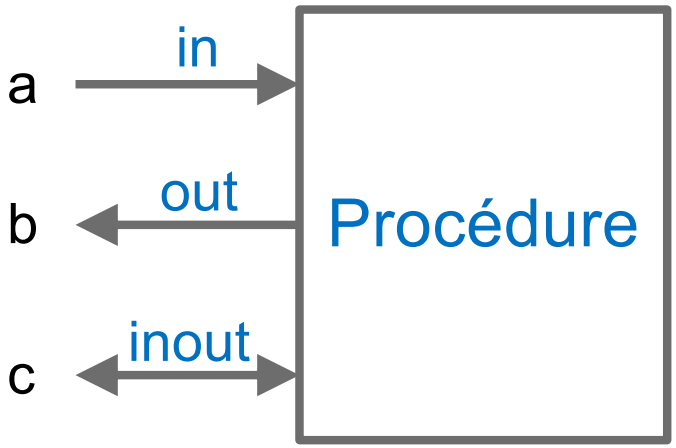
Mécanismes avancés du langage VHDL

- Sous-programmes:
  - Fonctions
  - Procédures
- Package
- Les attributs pour les arrays
- Généricité
- Tableaux non-contraints
- Instructions avancées:
  - Instruction séquentielle:
    - **for** … **loop**
  - Instructions concurrentes:
    - **for** … **generate**
    - **If** … **generate**

Sous-programmes

- Définis par une série d'instructions séquentielles
- Ne devraient contenir que de la logique combinatoire (pas de mémoire)
- Les sous-programmes peuvent être définis dans les parties déclaratives des architectures, process, autres sous-programmes ou principalement dans les packages body.





Fonctions

- Syntaxe de déclaration:

```
function nom[(liste_des_parametres)] return type is
    zone_declarative: constantes, variables, types, sous-
    programmes
begin
    zone d'instructions séquentielles
    return valeur_de_type; --plusieurs return possibles
end [function] [nom];
```

```
un_parametre_fonction :== [constant | variable | signal]
                          ID : [in] type [:= expression];
```

- L'utilisation des classes variable et signal permet de restreindre chaque paramètre à la classe spécifiée.
- Si la classe du paramètre est omise, alors elle est du type constant, constant étant compatible avec toutes les classes.
- Un paramètre peut avoir une valeur par défaut.

Fonctions

- Exemple de déclaration:

```
function max(a, b : integer) return integer is
begin
    if a > b then
      return a;
    end if;
    return b;
end function max;
```

- Appel de la fonction:

```
resultat := max(12, var1); --appel par position des paramètres
sig1 <= max(b => sig2, a => 12); -- appel par nom
```

- L'appel de la fonction peut se faire depuis toutes les instructions concurrentes ou séquentielles qui incluent une expression.

Procédures

- Syntaxe de déclaration:

```
procedure nom[(liste_des_parametres)] is
    zone_declarative: constantes, variables, types, sous-
    programmes
begin
    zone d'instructions séquentielles
end [procedure] [nom];
```

```
un_parametre :== [constant | variable | signal]
                 ID : [in|out|inout] type [:= expression];
```

- L'utilisation des classes variable et signal permet de restreindre chaque paramètre à la classe spécifiée.
- Un paramètre peut avoir une valeur par défaut.

Mode	Classe par défaut
in*	constant
out	variable
inout	variable

* *Mode par défaut*

Procédures

- Exemple de déclaration:

```
procedure max(a, b : in integer; result : out integer) is
begin
    if a > b then
      result := a;
    end if;
    result := b;
end procedure max;
```

- Appel de la procédure:

```
max(12, var1, resultat); --appel par position des paramètres
max(result => resultat, b => var1, a => 12); -- appel par nom
```

- L'appel de la procédure peut se faire depuis toutes les instructions concurrentes ou séquentielles.

Accès aux signaux/variables dans les procédures

- L'accès aux signaux/variables, qui ne sont pas passés par paramètre, dépend de l'endroit de déclaration de la procédure:

Endroit, partie déclarative	Signaux de l'architecture	Variables du processus englobant
De l'architecture	Lecture uniquement; Pour écrire un signal, il faut le passer en paramètre (modes out ou inout)	Pas applicable
Du processus	Lecture et écriture	Lecture et écriture
Du package	Pas d'accès	Pas d'accès

Package

- Un package sert à partager du code dans un projet
- Un package est composé de :
  - La spécification
  - Son corps
- La spécification d'un package présente tout ce qu'exporte le package:
  - Constantes
  - Types, sous-types
  - Sous-programmes
  - Déclarations de composants
  - …
- Par défaut, les packages sont placés dans la bibliothèque work.
- L'importation d'un package se fait par la clause use.

Package body

- Le corps du package (package body) contient:
  - La définition des sous-programmes exportés
  - Déclarations locales: types, constantes…
  - Pas de déclaration de signaux !!!
- La définition du corps du package est optionnelle si aucun sous-programme n'est explicité dans la spécification.

Package

- Exemple de spécification d'un package

```
package exemple_pkg is
    type state_t is (init, read, write, done);
    constant MEMSIZE : integer :=256;
    constant RESET_ACTIVE : std_logic :='0';
    type memoire_t is array(0 to MEMSIZE-1) of std_logic_vector(15 downto 0);
    component MonComposant is
        port (
            A_i,B_i: in std_logic;
            C_o: out std_logic
        );
    end component;
    function min(a,b: in integer) return integer;
    function max(a,b: in integer) return integer;
end exemple_pkg;
```

Package body

```
package body exemple_pkg is
    function min(a,b: in integer) return integer is
    begin
        if a<b then return a;
        else return b;
        end if;
    end min;
    function max(a,b: in integer) return integer is
    begin
        if a>b then return a;
        else return b;
        end if;
    end max;
end exemple_pkg;
```

Descriptions paramétrables avec un package

- Partage de constantes pour toutes les descriptions

  Par exemple:

  - Pour le codage des états
  - Pour définir la taille de vecteurs, aussi dans la déclaration des ports de l'entité
  - Etc.
- Déclarer un sous-type dans un package

Rappel: Attributs d'un tableau

- Les attributs suivants sont définis pour n'importe quel type de tableau:
  - LEFT: l'indice le plus à gauche de l'intervalle
  - RIGHT: l'indice le plus à droite de l'intervalle
  - HIGH: l'indice le plus grand dans l'intervalle
  - LOW: l'indice le plus petit dans l'intervalle
  - RANGE: sous-type des indices, intervalle entre l'attribut LEFT et RIGHT
  - REVERSE_RANGE: intervalle inverse de RANGE
  - LENGTH: nombre d'éléments du tableau

Descriptions paramétrables avec les attributs

- Les attributs :
  - Permettent de manipuler les array / tableaux
  - Sont nécessaires pour rendre les descriptions indépendantes de l'intervalle d'indices des tableaux
  - Sont à utiliser avec l'opérateur de concaténation & et la notation d'agrégat
  - Sont utiles pour les testbenchs
- Exemple d'utilisation des attributs :

```
Data : Std_Logic_Vector(7 downto 0);
signal V_SLR : Std_Logic_Vector(data'range);
--Décalage à droite :
V_SLR <= '0' & data(data'high downto data'low + 1);
```

Généricité

- La généricité en VHDL permet de configurer un composant avec des informations statiques
- Un composant dit générique est vue de l'extérieur comme un composant paramétrable, les valeurs des paramètres étant définies lors de chaque instanciation du composant
- A l'intérieur du composant, les paramètres génériques sont identiques à des constantes

Syntaxe:

```
entity exemple_genericite is
    generic (param1 [, autre_param]: type_param [:=valeur_par_defaut];
             param2 [, autre_param]: type_param [:=valeur_par_defaut];
             ...
             paramN [, autre_param]: type_param [:=valeur_par_defaut]
            );
```

Généricité: exemple

- Porte AND à N entrées

```
entity porteET is
    generic(NB_ENTREES: natural :=1);
    port (entrees_i : in std_logic_vector(NB_ENTREES-1 downto 0);
          sortie_o : out std_logic
    );
end porteET;

architecture comp of porteET is
begin
    process(entrees_i)
        variable resultat_v: std_logic;
    begin
        resultat_v:='1';
            for i in 0 to NB_ENTREES -1 loop
            resultat_v := resultat_v and entrees_i(i);
            end loop;
        sortie_o <= resultat_v;
    end process;
end comp;
```

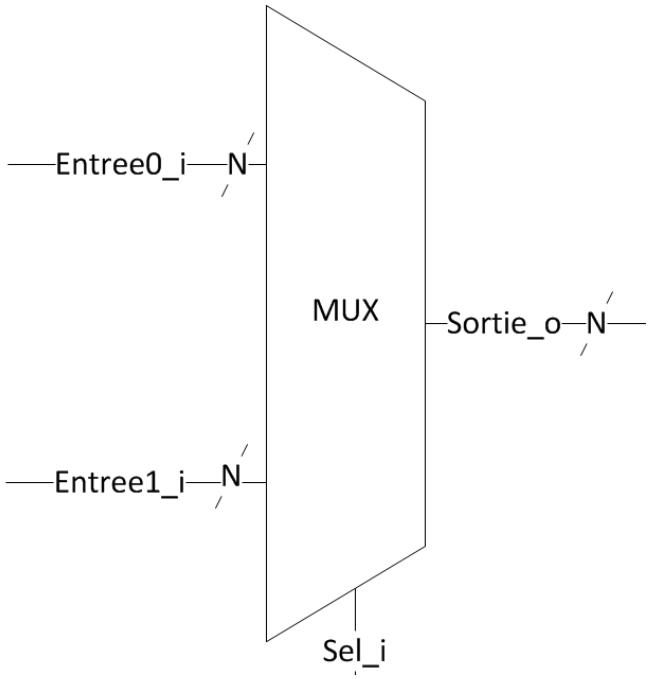
Généricité: exemple instancié

- Instanciation (port map) de la porte AND

```
architecture struct of quelquechose is
    component porteET is
        generic (NB_ENTREES: natural :=2);
            port(entrees_i : in std_logic_vector(NB_ENTREES-1 downto 0);
                 sortie_o : out std_logic
            );
    end component;
    signal les_entrees_s : std_logic_vector(7 downto 0);
    signal la_sortie_s : std_logic;
    ...
begin
    porte: porteET
        generic map (NB_ENTREES => 8)
        port map(
            entrees_i => les_entrees_s,
            sortie_o => la_sortie_s
        );
    ...
end struct;
```

Exercice 1: Multiplexeur N bits

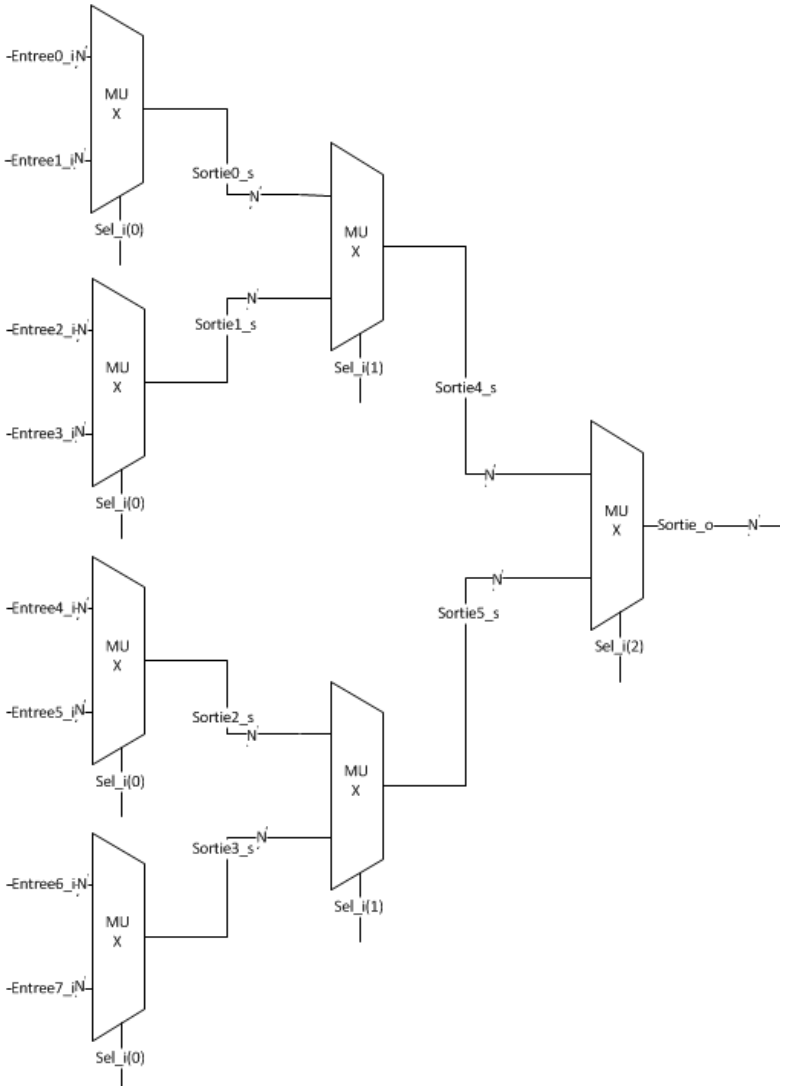
- Implémenter et tester un multiplexeur (Mux2) ou les entrées et la sortie ont N bits.



Exercice 2: Mux à 8 entrées

- En instanciant le mux2, implémentez un Mux à 8 entrées

- Remarque:
  Noter que pour le generic map, déclarer un generic dans le mux8 de façon à changer facilement la taille des bus entrées/sorties



Solution 1: Mux à 2 entrées de N bits

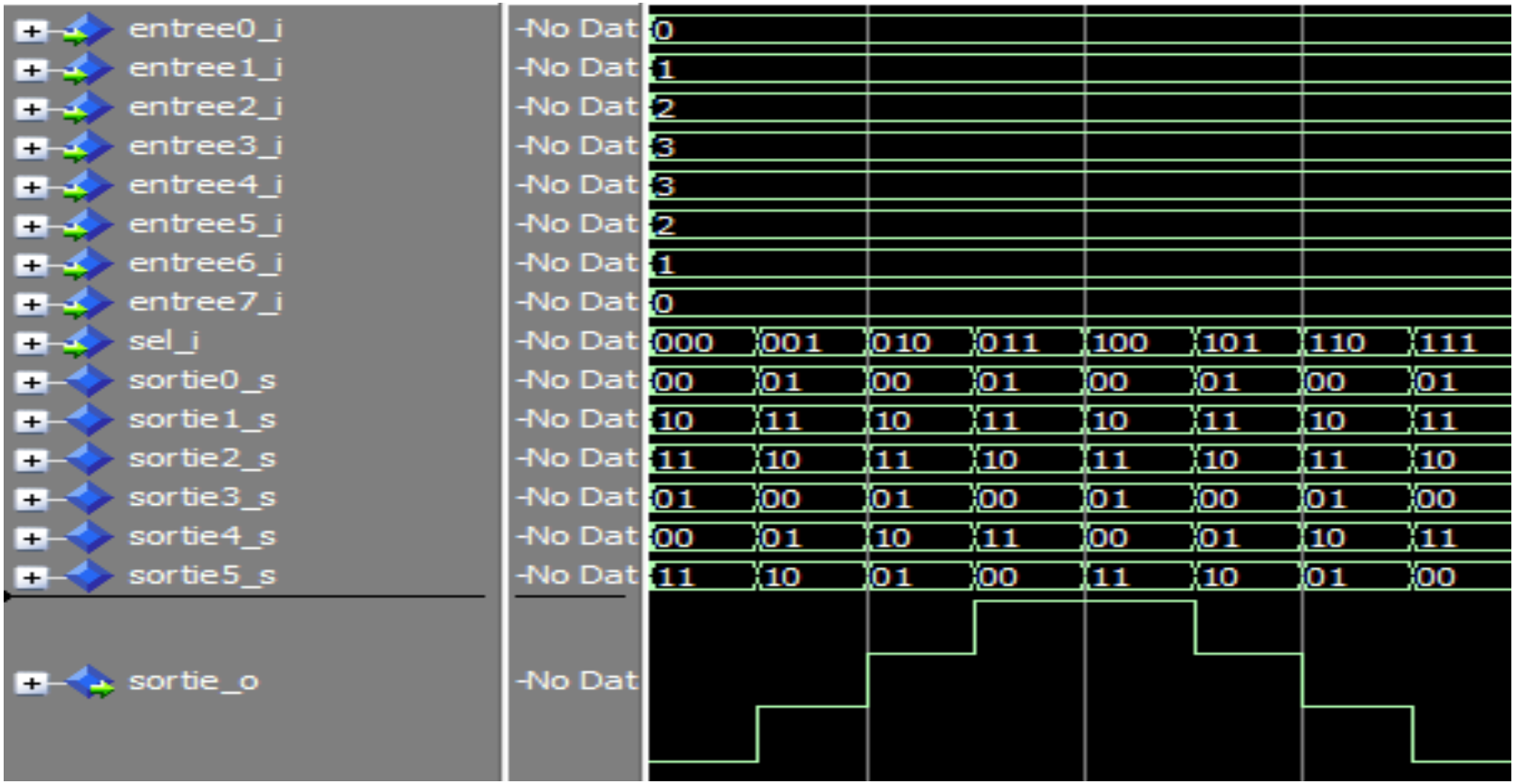
```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Numeric_Std.all;

entity mux2_generic is
    generic(SIZE: positive := 2);
    port(entree0_i : in std_logic_vector(SIZE-1 downto 0);
         entree1_i : in std_logic_vector(SIZE-1 downto 0);
         sel_i : in std_logic;
         sortie_o : out std_logic_vector(SIZE-1 downto 0)
    );
end mux2_generic;
architecture flot of mux2_generic is
begin
    sortie_o<= entree0_i when sel_i='0' else entree1_i;
end flot;
```

Solution 2: Mux à 8 entrées

```vhdl
library IEEE; use IEEE.Std_Logic_1164.all; use IEEE.Numeric_Std.all;
entity mux8_generic is
    generic(SIZE_mux8: positive := 8);
    port(entree0_i,entree1_i,entree2_i,entree3_i : in std_logic_vector(SIZE_mux8-1 downto 0);
         entree4_i,entree5_i,entree6_i,entree7_i : in std_logic_vector(SIZE_mux8-1 downto 0);
         sel_i : in std_logic_vector(2 downto 0);
         sortie_o : out std_logic_vector(SIZE_mux8-1 downto 0)
    );
end mux8_generic;

architecture behav of mux8_generic is
  signal sortie0_s,sortie1_s,sortie2_s : std_logic_vector(SIZE_mux8-1 downto 0);
  signal sortie3_s,sortie4_s,sortie5_s : std_logic_vector(SIZE_mux8-1 downto 0);

  component mux2_generic is
    generic(SIZE: positive := 2);
    port(entree0_i : in std_logic_vector(SIZE-1 downto 0);
         entree1_i : in std_logic_vector(SIZE-1 downto 0);
         sel_i : in std_logic;
         sortie_o : out std_logic_vector(SIZE-1 downto 0)
    );
  end component;

begin
  -- first level of mux
  mux1: mux2_generic
  generic map(SIZE => SIZE_mux8)
  port map(entree0_i => entree0_i,
           entree1_i => entree1_i,
           sel_i => sel_i(0),
           sortie_o => sortie0_s
  );
  mux2: mux2_generic
  generic map(SIZE => SIZE_mux8)
  port map(entree0_i => entree2_i,
           entree1_i => entree3_i,
           sel_i => sel_i(0),
           sortie_o => sortie1_s
  );
  mux3: mux2_generic
  generic map(SIZE => SIZE_mux8)
  port map(entree0_i => entree4_i,
           entree1_i => entree5_i,
           sel_i => sel_i(0),
           sortie_o => sortie2_s
  );
  mux4: mux2_generic
  generic map(SIZE => SIZE_mux8)
  port map(entree0_i => entree6_i,
           entree1_i => entree7_i,
           sel_i => sel_i(0),
           sortie_o => sortie3_s
  );
  -- second level of mux
  mux5: mux2_generic
  generic map(SIZE => SIZE_mux8)
  port map(entree0_i => sortie0_s,
           entree1_i => sortie1_s,
           sel_i => sel_i(1),
           sortie_o => sortie4_s
  );
  mux6: mux2_generic
  generic map(SIZE => SIZE_mux8)
  port map(entree0_i => sortie2_s,
           entree1_i => sortie3_s,
           sel_i => sel_i(1),
           sortie_o => sortie5_s
  );
  -- third level of mux
  mux7: mux2_generic
  generic map(SIZE => SIZE_mux8)
  port map(entree0_i => sortie4_s,
           entree1_i => sortie5_s,
           sel_i => sel_i(2),
           sortie_o => sortie_o
  );
end architecture;
```

Solution 2: Mux à 8 entrées

Simulation avec entrée 2 bits par exemple:



Descriptions paramétrables avec les tableaux non-contraints

- VHDL autorise la déclaration de tableau non-contraint, sans définition de l'intervalle des indices:

```
type std_logic_vector is array (NATURAL range <>) of std_logic;
```

- La taille du tableau est définie dans ce cas à l'instanciation
- Un tableau non-contraint n'est pas synthétisable.

Exemple description paramétrable avec les tableaux non-contraints

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity reg_nc is
    Port ( clk, rst : in STD_LOGIC;
           d : in STD_LOGIC_VECTOR;
           q : out STD_LOGIC_VECTOR);
end reg_nc;

architecture Behavioral of reg_nc is
    signal reg : std_logic_vector(d'range);
begin
    process(clk, rst)
    begin
        if rst = '1' then
            reg <= (others => '0');
        elsif rising_edge(clk) then
            reg <= d;
        end if;
    end process;
    q <= reg;
end Behavioral;
```

Exemple description paramétrable avec les tableaux non-contraints (2)

```
architecture Behavioral of reg8 is
    signal data, registre : std_logic_vector(15 downto 0);
begin
    reg_inst: entity work.reg_nc(Behavioral)
            port map (
                clk => clk,
                rst => rst,
                d => data,
                q => registre);

end Behavioral;
```

Instruction for … loop

- Instruction séquentielle, syntaxe:

```
[label:] for <identificateur> in <range> loop
    zone d'instructions séquentielles
end loop;
```

- `<identificateur>`:
  - Est auto-déclaré avec l'instruction `for` et est du type de `<range>`
  - Ne peut être modifié avec une affectation
  - Visible dans l'itération uniquement
- L'interprétation pour le synthétiseur des itérations est de répliquer du matériel (blocs)
- Pour être synthétisable, les bornes de l'intervalle doivent être fixes et ne sont pas paramétrables.

Instruction for … loop

- Exemple:

```
architecture behaviour of match_bits is
     signal a, b, matches : std_logic_vector(3 downto 0);
begin
    process(a, b)
    begin
       for i in a'range loop
          matches(i) <= a(i) xnor b(i);
       end loop;
    end process;
```

- Equivalent à:

```
    process(a, b)
    begin
       matches(3) <= a(3) xnor b(3);
       matches(2) <= a(2) xnor b(2);
       matches(1) <= a(1) xnor b(1);
       matches(0) <= a(0) xnor b(0);
    end process;
```

Instruction for … generate

- Permet de créer des composants à structures répétitives et des circuits itératifs:
  - Registres universels
  - Additionneur
  - Etc.
- Correspond à la version concurrente de `for loop`
- Syntaxe:

```
--Label obligatoire
label: for <identificateur> in <range> generate
    zone d'instructions concurrentes
end generate;
```

- Rappel: l'ordre, dans lequel les instructions concurrentes sont décrites, n'est pas important.

Instruction for … generate

- Exemple:

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

entity forgenerate is
    Generic(n : natural);
    Port ( clk, rst : in STD_LOGIC;
           d : in STD_LOGIC_VECTOR (n-1 downto 0);
           q : out STD_LOGIC_VECTOR (n-1 downto 0));
end forgenerate;

architecture Behavioral of forgenerate is
begin
    register_bank: for i in 0 to n-1 generate
        process(clk, rst)
        begin
            if rst = '1' then
                q(i) <= '0';
            elsif rising_edge(clk) then
                q(i) <= d(i);
            end if;
        end process register_bank;
    end generate;
end Behavioral;
```

Instruction if … generate

- Permet de déclarer des structures optionnelles
- Instruction concurrente
- Syntaxe:

```
--Label obligatoire
label: if <condition booléenne> generate
    zone d'instructions concurrentes
end generate;
```

- La condition booléenne est généralement exprimée avec les constantes génériques ou avec les identificateurs des boucles `for generate`.

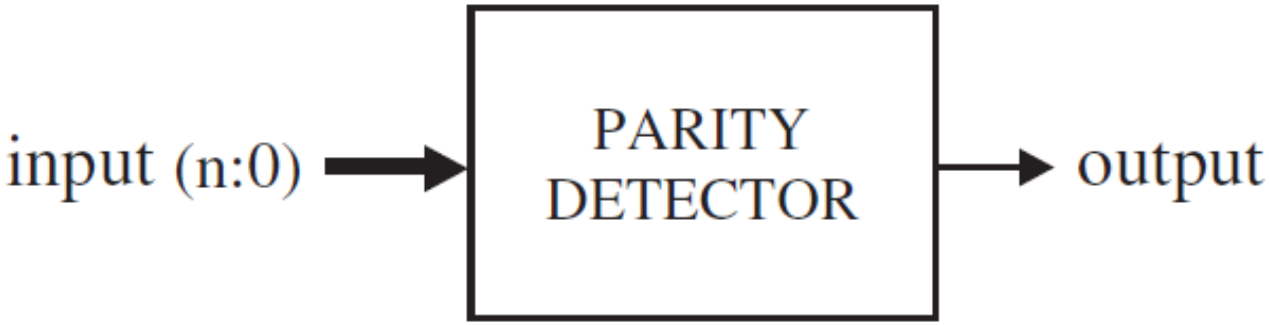
Instruction if … generate

- Exemple:

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

entity ifgenerate is
    generic (n : natural; store : boolean);
    port (a : in std_logic_vector (n-1 downto 0);
          clk : in std_logic;
          z : out std_logic_vector (n-1 downto 0));
end;
architecture behaviour of ifgenerate is
begin
    registre: if store generate
       process(clk)
       begin
          if rising_edge(clk) then
            z <= a;
          end if;
       end process;
     end generate;

    combinatoire: if not store generate
        z <= a;
    end generate;
end;
```

Exercice 1: Détecteur de parité paire

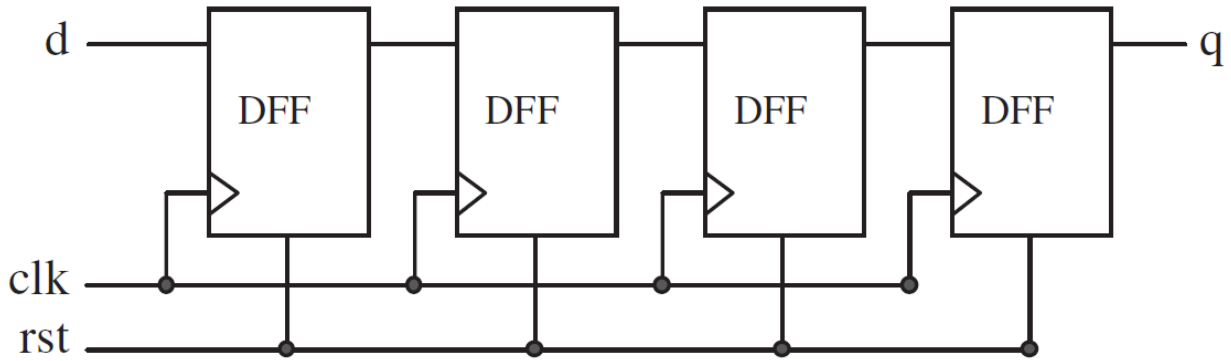
- Le circuit doit générer un ‘0’ en sortie lorsque le nombre de ‘1’ en entrée est pair, sinon ‘1’.



- Noter que le modèle contient un generic *n* qui définit la taille du bus d’entrée. La taille du bus d’entrée peut varier en changeant uniquement la valeur du paramètre n

Exercice 2: Registre à décalage

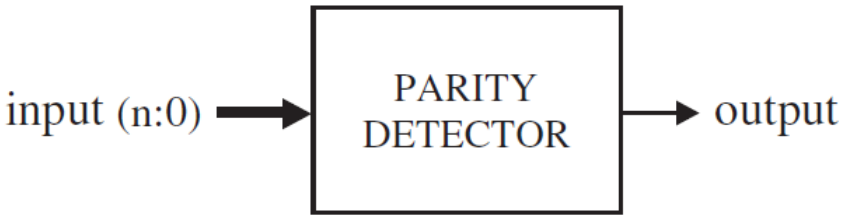
- Soit un registre à décalage, composé de flip-flop avec reset asynchrone.



- L'entrée d arrive à la sortie q après 4 flancs d'horloge
- Proposer 2 solutions:
  - Utiliser une constante generic sans `generate`
  - Utiliser une constante generic **avec** `generate`

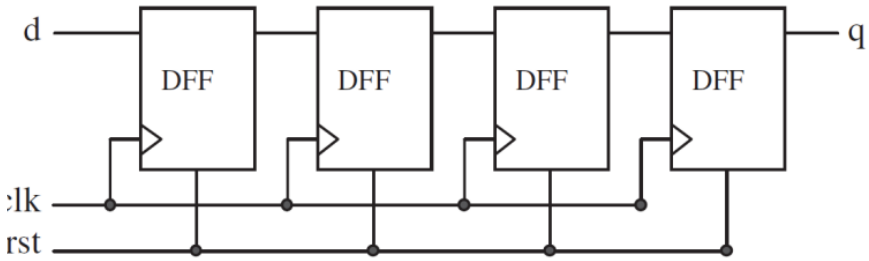
Solution 1: Détecteur de parité

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY parity_det IS
    GENERIC (n : INTEGER := 7);
    PORT ( input: IN std_logic_vector (n DOWNTO 0);
    output: OUT std_logic);
END parity_det;
-----------------------------------------
ARCHITECTURE parity OF parity_det IS
    BEGIN
    PROCESS (input)
        VARIABLE temp: std_logic;
    BEGIN
        temp := '0';
        FOR i IN input'RANGE LOOP
            temp := temp XOR input(i);
        END LOOP;
        output <= temp;
    END PROCESS;
END parity;
-----------------------------------------
```



Solution 2: Registre à décalage

```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY shiftreg IS
    GENERIC (n: INTEGER := 4); -- # of stages
    PORT (d, clk, rst: IN STD_LOGIC;
          q: OUT STD_LOGIC
    );
END shiftreg;
-------------------------------------------------
ARCHITECTURE behavior OF shiftreg IS
    SIGNAL internal: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
BEGIN
    PROCESS (clk, rst)
    BEGIN
        IF (rst='1') THEN
            internal <= (OTHERS => '0');
        ELSIF (clk'EVENT AND clk='1') THEN
            internal <= d & internal(internal'LEFT DOWNTO 1);
        END IF;
    END PROCESS;
    q <= internal(0);
END behavior;
-------------------------------------------------
```



Solution 2: avec generate

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

entity regdec_generate is
    Generic (n : positive := 4);
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           d : in STD_LOGIC;
           q : out STD_LOGIC);
end regdec_generate;

architecture Beh of regdec_generate is
  signal internal :
         std_logic_vector(n-1 downto 0);
begin
    q <= internal(n-1);
```

```
  reg_dec : for i in 0 to n-1 generate
    first: if i = 0 generate
      process(clk, rst)
      begin
        if rst = '1' then
          internal(0) <= '0';
        elsif rising_edge(clk) then
          internal(0) <= d;
        end if;
      end process;
    end generate;

    other: if i /= 0 generate
      process(clk, rst)
      begin
        if rst = '1' then
          internal(i) <= '0';
        elsif rising_edge(clk) then
          internal(i) <= internal(i-1);
        end if;
      end process;
    end generate;
  end generate;
end Beh;
```