Libraries

Ici on déclare les bibliothèques et quels modules utilisés.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use ieee.fixed_pkg.all;
library std;
use std.textio.all;
```

Entité (entrées-sorties)

Ici on a le nom du composant et les entrées sorties qui peuvent être in, out ou inout.

```
entity combinatoire1 is
port (
    A, B, C : in std_logic;
    F : out std_logic
);
end combinatoire1;
```

Architectures logique combinatoire

Structurel

```
architecture structural of combinatoire1 is
component INV
    port (I : in std_logic; O : out std_logic);
end component;
component NAND2
    port (I0, I1 : in std_logic; O : out std_logic);
end component;
component XOR2
    port (I0, I1 : in std_logic; O : out std_logic);
end component;
signal NET18, NET37 : std_logic;
begin
    U1 : NAND2
        port map(
            I0 => NET37,
            I1 => A,
            O => F
        );
    U2 : XOR2
    port map (
            I0 => NET18,
            I1 => B,
            O => NET37
        );
    U3 : INV
        port map(
            I => C,
            O => NET18
        );
end structurale;
```

Data Flow

Opérateurs logiques

```
architecture flotDeDonnees1 of combinatoire1 is
begin
    F <= not(A and (B xor not(C)));
end flotDeDonnees1;
```

With-select

```
architecture flotDeDonnees2 of combinatoire1 is
signal entree : std_logic_vector(2 downto 0);
begin
    entree <= (A, B, C);
    with entree select
        F <=
            '1' when "000",
            '1' when "001",
            '1' when "010",
            '1' when "011",
            '0' when "100",
            '1' when "101",
            '1' when "110",
            '0' when "111",
            '0' when others;
end flotDeDonnees2;
```

When-else

```
architecture flotDeDonnees3 of combinatoire1 is
begin
    Y <= A when sA = '1' else B when sB = '1' else C ;
end flotDeDonnees3;
```

Behavioral (avec générique)

```
entity porteET is
    generic (
        W : positive := 8 -- le nombre d'entrées
    );
    port (
        I : in std_logic_vector(W - 1 downto 0);
        F : out std_logic
    );
end porteET;
architecture comportementale of porteET is
begin
    process (I)
    variable sortie : std_logic;
    begin
        sortie := '1';
        for k in W - 1 downto 0 loop
            sortie := sortie and I(k);
        end loop;
        F <= sortie;
    end process;
end comportementale;
```

```
architecture structural of top_level is
  -- Declaration du composant
  component porteET
    generic (
      W : positive := 4  -- le nombre d'entrees
    );
    port (
      I : in std_logic_vector(W - 1 downto 0);
      F : out std_logic
    );
  end component;

  -- Declaration des signaux
  signal input_signals : std_logic_vector(3 downto 0);
  signal output_signal : std_logic;
begin
  -- Instanciation de la porte AND_gate
  AND_instance: AND_gate
    generic map (
      W => 4  -- Specification du nombre d'entrees
    )
    port map (
      I => input_signals,
      F => output_signal
    );
  -- Autre logique de l'architecture...
end architecture structural;
```

Categories objects VHDL

Variable

- Les variables sont déclarées dans les processus ou les blocs de déclarations locales.
- Elles sont utilisées pour stocker des valeurs temporaires et effectuer des calculs localement dans un processus.
- Leur portée est limitée au processus ou au bloc dans lequel elles sont déclarées.
- Les variables peuvent être assignées plusieurs fois dans un processus, et les valeurs sont mises à jour immédiatement.
- Il est intéressant de nommer les variables d'une manière identique dans le code pour les reconnaître (ex : _var).
- Les variables nous donnent plus de flexibilités dans le code mais généralement moins de visibilité que les signaux.
- Les variables sont assignées à l'aide de « := ».

```
variable temp_var : integer := 0;
variable compte_var : natural range 0 to in_signal'length;
```

Signal (port)

- Les signaux sont déclarés dans la section d'architecture et les entrées/sorties (ports) des composants sont toujours des signaux.
- Ils représentent les connexions entre différents composants dans votre design.
- Les signaux peuvent être utilisés pour transférer des données entre des processus ou des blocs d'architecture.
- Les valeurs des signaux ne sont mises à jour qu'à la fin d'un cycle de simulation, ce qui signifie qu'ils représentent le comportement de matériel réel.
- Malgré qu'un signal ne soit pas local à un processus (donc il peut être lu dans plusieurs processus concurrents), il NE PEUT pas être assigné dans 2 processus différents. Cela correspondrait à brancher deux sorties de portes logiques.
- Ils sont assignés à l'aide de « <= ». Attention, si on initialise un signal (dans la zone de déclaration, on utilisera « := » pour assigner une valeur initiale au signal).

```
signal data_signal : std_logic;
port (
    i_in : in std_logic_vector(W - 1 downto 0);
    o_out : out std_logic );
```

Constantes (génériques)

- Les constantes sont déclarées dans la section de l'architecture ou dans le bloc de déclarations globales. Les génériques d'un composant sont aussi des constantes paramétrables.
- Elles sont utilisées pour définir des valeurs constantes ou paramétrables qui restent inchangées pendant toute la durée de la simulation ou de la synthèse (ne peuvent être réassignées).
- Leur portée est globale à l'architecture ou au composant dans lequel elles sont déclarées.
- Elles sont généralement utilisées pour définir des paramètres de conception (ex : largeur de bus);
- Il est intéressant de nommer les variables d'une manière identique dans le code pour les reconnaître (ex : _cnst pour constante ou _G pour générique).
- Elles sont assignées à l'aide de « := ».

```
constant name_cnst : real := 3.14159;
generic (
    WIDTH_G : integer := 8;
    DEPTH_G : integer := 16);
```

Agrégats

- On parle d'assignation positionnelle lorsqu'on assigne la valeur des éléments d'un vecteur dans l'ordre

```
variable Data1 : bit_vector (0 to 3) := ('0','1','0','1'); (ici
bit_vector(0) = '0' et bit_vector(3) = '1')
variable Data2 : bit_vector (3 downto 0) := ('0','1','0','1'); (ici
bit_vector(3) = '0' et bit_vector(0) = '1')
```

- On parle d'assignation nommée lorsqu'on nomme l'indice avant de donner sa valeur

```
variable Data3 : bit_vector (0 to 3) := (1=>'1',0=>'0',3=>'1',2=>'0');
constant vecteurs : tableau_pixelRGB := (
(0,0,0),(1,1,1),(2,2,2));
type StatusRecord is record
   Code : Integer;
   Name : String (1 to 4);
end record;
variable StatusVar : StatusRecord := (Code => 57, Name =>
"MOVE");
```

- On peut aussi utiliser une gamme d'indices et/ou utiliser le choix « others »

```
DataBus <= (14 downto 8 => '0', others => '1');
DataBus <= (others => 'Z');
```

Types en VHDL

Types prédéfinis

catégorie	type ou sous-type	source de la définition	valeurs
scalaires	boolean	type prédéfini	FALSE et TRUE
	bit	type prédéfini	'0' et '1'
	character	type prédéfini	256 caractères de la norme ISO 8859-1, avec des abréviations reconnues et certaines qui sont propres à VHDL Les 128 premiers sont les caractères ASCII.
	integer	type prédéfini	plage minimale de $-2^{31}+1$ à $2^{31}-1$
	natural	sous-type prédéfini	0 à $2^{31}-1$
	positive	sous-type prédéfini	1 à $2^{31}-1$
	real	type prédéfini	typiquement $-1.7014111E\pm308$ à $1.7014111E\pm308$
	std_logic	Package std_logic_1164	'U' : valeur inconnue, pas initialisée 'X' : valeur inconnue forcée '0' : 0 forcé '1' : 1 forcé 'Z' : haute impédance (pas connecté) 'W' : inconnu faible 'L' : 0 faible 'H' : 1 faible '-' : peu importe (don't care)
composés	bit_vector	type prédéfini	tableau de bit
	string	type prédéfini	tableau de character
	std_logic_vector	Package std_logic_1164	tableau de std_logic
	unsigned	Package numeric_std	tableau de std_logic, interprété comme un nombre binaire non signé
	signed	Package numeric_std	tableau de std_logic, interprété comme un nombre binaire signé en complément à deux

- Les types en rouge ne sont pas synthétisables ou non recommandé pour la synthèse à part exceptions (comme si utilisé comme constante).
- Pour les opérations logiques, on utilise généralement les std_logic ou std_logic_vector.
- Pour les opérations arithmétiques, on utiliser les signed et unsigned.

Types spécifiés

- Pour définir un nouveau type, on peut utiliser « type » ou « sub-type »
- Les types composés sont définis à l'aide de « array » ou « record ».
- Il est intéressant de nommer les types d'une manière identique dans le code pour les reconnaître (ex : _type).

```
type entiers_s20_type is range 0 to 19;
type matrice_reelle_s10_type is array (1 to 10) of real;
type tableau_type is array (natural range <>) of
std_logic_vector(2 downto 0);
constant vecteurs_cnst : tableau_type :=
("000", "001", "010", "011", "100", "101", "110", "111");
type nom_mois_type is (janvier, février, mars, avril, mai, juin,
juillet, aout, septembre, octobre, novembre, décembre);
type state_type is (IDLE, START, RUN, STOP);
type date_rec_type is record
    jour : integer range 1 to 31;
    mois : nom_mois_type;
    année : positive range 1 to 3000;
end record;
constant confederation : date_rec_type := (1, juillet, 1867);
```

Conversion

Conversion explicite

- **to_integer** et **to_unsigned** : convertir un type signé (**signed**) ou entier naturel (**natural**) en entier (**integer**) ou en non-signé (**unsigned**).
- **to_signed** et **to_unsigned** : convertir un entier (**integer**) ou un naturel (**natural**) en signé (**signed**) ou non-signé (**unsigned**).
- Convertir un entier en string ou vice-versa avec **to_string** ou **to_integer**.

```
str_value <= to_string(int_value);
```

- Conversion de longueur de vecteur (**resize**)

```
signal input_vector : std_logic_vector(3 downto 0) ;
signal output_vector : std_logic_vector(7 downto 0);
output_vector <= resize(input_vector, 8);
```

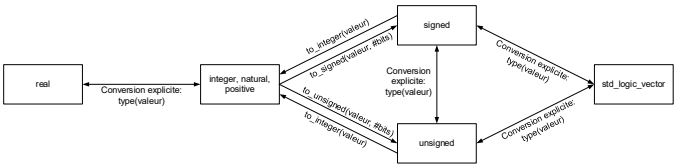
Conversion implicite

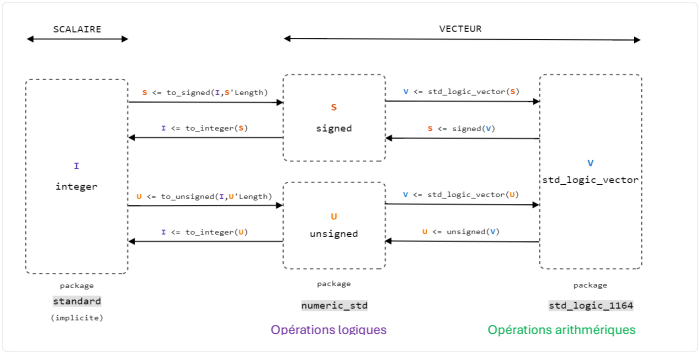
- On peut faire de la conversion de type dans certains cas compatible par exemple, convertir un std_logic_vector en entier non-signé

```
index := to_integer(unsigned(gray_out))
```

- Conversion pour affichage. Par exemple, **integer'image**(k) retourne une chaîne de caractères représentant l'entier k

```
signal real_value : real := 3.14;
variable str_value : string(1 to 10);
str_value := real'image(real_value);
```





Nombre à virgule fixe

En VHDL 2008, des packages ont été ajoutés pour permettre les opérations sur les nombres à virgule fixe. Le package fixed_pkg permet de manipuler les données à virgule fixe prédéfinis (et fixed_generic_pkg pour des données à virgule fixe paramétrables).

Types pour nombre à virgule fixe

Les deux types principaux ajoutés sont :

- **ufixed** : nombre non-signé sous le format ufixed(n downto -m) qui aura n+1 bits pour la partie entière et m bits pour la partie fractionnaire.
- **sfixed** : nombre signé sous le format sfixed(n downto -m) qui aura n+1 bits pour la partie entière (incluant le signe) et m bits pour la partie fractionnaire.

Fonction pour nombre à virgule fixe

- Il existe plusieurs fonctions disponibles pour traiter les styles à virgules fixes.
- Pour ces fonctions, les arguments qui ont une valeur par défaut sont facultatifs.
- Le style de débordement (**overflow_style**)
  - **fixed_wrap** pour débordement normal (défaut)
  - **fixed_saturate** pour la saturation à la valeur maximale
- Le style d'arrondi (**rounding**)
  - **fixed_truncate** pour la fonction plancher (défaut)
  - **fixed_round** pour l'arrondi
- Les arguments *integer_width* et *fractional_width* peuvent être remplacé par un objet de la taille voulue.
- Les fonctions importantes sont les suivantes :
  - **function to_real**(x : **sfixed**; rounding : **fixed_round_style**:= fixed_truncate) **return real**;
  - **function to_integer**(x : **sfixed**; rounding : **fixed_round_style**:= fixed_truncate) **return integer**;
  - **function resize**(x : **sfixed**; integer_width : **integer**; fractional_width : **integer**; overflow_style : **fixed_overflow_style** := fixed_wrap; rounding_style : **fixed_round_style** := fixed_truncate) return **sfixed**;
  - **function saturate**(x : **sfixed**; integer_width : **integer**; fractional_width : **integer**) return **sfixed**;
  - **function to_sfixed**(x : **real**; integer_width : **integer**; fractional_width : **integer**; rounding_style : **fixed_round_style** := fixed_truncate) return **sfixed**;
  - Les fonctions **to_ufixed** et **to_sfixed** ont le même format.

```
variable s1 : sfixed(3 downto -4);
variable compte : ufixed(1 downto -2);
s1 := to_sfixed(3.1415926, s1);
s1 := to_sfixed(3.1415926, 3, -4);
compte := resize(compte + 0.25, compte, fixed_wrap, fixed_round);
```

Attributs

- **'range** : renvoie la plage d'index d'un vecteur/tableau
- **'length** : renvoie la longueur d'un vecteur/tableau
- **'left** : renvoie la borne gauche de l'index d'un vecteur/tableau/type
- **'right** : renvoie la borne droite de l'index d'un vecteur/tableau/type
- **'high** : Renvoie l'index le plus élevé d'un vecteur/tableau/type
- **'low** : Renvoie l'index le plus bas d'un vecteur/tableau/type
- **'event** : Utilisé pour détecter un changement sur un signal
- **'image(x)** : convertir une valeur d'un type spécifié en sa représentation textuelle (string)
- **'value(s)** convertir une représentation textuelle (string) en une valeur du type spécifié
- **'val(n)** : renvoie le nième élément dans un type spécifié
- **'pos(n)** : renvoie la position d'un élément dans un type spécifié;

```
avec : signal my_vector : std_logic_vector(7 downto 0);
vector_range := my_vector'range;  -- Donne 7 downto 0
vector_length := my_vector'length;  -- Donne 8
left_index := my_vector'left;  -- Donne 7
right_index := my_vector'right;  -- Donne 0
high_index := my_vector'high;  -- Donne 7
low_index := my_vector'low;  -- Donne 0
clk_event_detected := clk'event;  -- Vrai si clk a changé à cette simulation
report "Length: " & integer'image(vector_length);
current_state <= state_type'Val(index);
state_index <= state_type'Pos(current_state);
current_state <= state_type'Value(state_string);
state_string <= state_type'Image(current_state);
charout <= to_unsigned(character'pos(message(index)), 8);
```

Fonctions et procédures

- Les fonctions et procédures peuvent être définies dans la partie déclarative d'une architecture ou bien dans un package. Elles peuvent aussi être surchargée.

Fonctions

- Une fonction est un sous-programme qui retourne un résultat unique (scalaire ou composée)
- Elle peut accepter des paramètres en entrée, mais ces paramètres ne peuvent pas être modifiés par la fonction.
- Une fonction est appelé dans une expression, comme si toutes les étapes étaient réalisées instantanément (n'accepte aucun comportement temporel ou délai).

```
-- Déclaration d'une fonction
function add_ numbers(a : integer; b : integer) return integer is
begin
  return a + b;
end function;
-- Utilisation de la fonction dans une architecture
architecture Behavioral of ExampleFunction is
  signal result : integer;
begin
  process
  begin
    result <= add_numbers(5, 3);  -- Appel de la fonction
    wait;  -- Arrêt du processus
  end process;
end Behavioral;
```

Procédures

- La procédure est un sous-programme avec une liste de paramètres ou non. Lors de l'appel d'une procédure, les signaux disponibles à l'endroit où la procédure est appelée peuvent être lus et modifiés (attention les signaux en VHDL ne peuvent être modifiés dans 2 processus différents).
- Si des éléments sont déclarés dans la partie déclarative d'une procédure, ils ne seront visibles qu'à l'intérieur de cette procédure (perdant sa valeur en sortant de la procédure).
- Si on ajoute des paramètres à la procédure, ils peuvent avoir les modes in, out et inout. L'avantage des paramètres est qu'à chaque appel de procédure, les entrants seront différents (donc on peut utiliser la même procédure pour modifier des signaux différents). Si aucun type n'est précisé pour les paramètres, ils seront des variables. Pour qu'ils agissent comme des signaux, on doit mettre le mot « signal » devant le paramètre.
- La procédure est appelée dans un énoncé concurrent. Elle peut être appelé à l'intérieur ou à l'extérieur d'un processus. Dans ce dernier cas, la procédure agit comme un processus et s'exécute en parallèle avec les autres processus.
- La procédure peut accepter des comportements temporels ou des délais. Attention, l'ajout de délais dans une procédure est réservé au « testbench » puisque ce genre d'instructions ne sont pas synthétisables.

```
-- Déclaration d'une procédure
procedure add_numbers_proc(a : in integer; b : in integer;
result : out integer) is
begin
  result := a + b;
end procedure;
-- Utilisation de la procédure dans une architecture
architecture Behavioral of ExampleProcedure is
  signal result : integer;
begin
  process
  begin
    add_ numbers_proc(5, 3, result);  -- Appel de la procédure
    wait;  -- Arrêt du processus
  end process;
end Behavioral;
```

Déclarations

Conditionnelles et sélectives

Data Flow

```
out_signal <= '1' when (condition) else '0';

with sel_signal select
out_signal <=
    A when "...",
    B when "...",
    C when  others;
```

Behavioral (process)

```
if variable/signal > condition then
    out_signal <= '1';
else
    out_signal <= '0';
end if;

case sel_signal is
    when "..."  =>  out_signal <= A;
    when "..."  =>  out_signal <= B ;
    when others =>  out_signal <= C ;
end case ;
```

Boucles

Behavioral (process)

```
for k in 3 downto 0 loop
... (use k in code as an integer in code)
end loop;

for k in in_signal 'range loop
... (use k in code as an integer and in_signal as a signal in code)
end loop;

ind_var := 0;
while (ind_var <= 3) loop
...
ind_var := ind_var + 1;
end loop;
```

Éléments de mémoire

On ne fait pas d'opérations logiques sur les signaux d'horloge et de réinitialisation (reset)!

Bascule

- Sans reset

```
process (CLK) is
begin
    if (CLK = '1' and CLK'event) then
    -- if (rising_edge(CLK)) then
        Q <= D;
    end if;
end process;
```

- Avec reset asynchrone actif bas (préconisé)

```
process(CLK, reset) is
begin
    if (reset = '0') then
        Q <= '0';
    elsif (CLK = '1' and CLK'event) then
        Q <= D;
    end if;
end process;
```

- Avec reset synchrone actif bas (moins préconisé)

```
process(CLK, reset) is
begin
    if (CLK = '1' and CLK'event) then
        if (reset = '0') then
            Q <= '0';
        else
            Q <= D;
        end if;
    end if;
end process;
```

Loquet (moins préconisé)

```
process (G, D) is
begin
    if (G = '1') then
        Q <= D;
    end if;
end process;
```

Banc d'essai

Outils intéressants

- Utilisation de la clause **after** pour assigner un signal « x » temps plus tard

```
Z <= '0' after 0 ns, '1' after 40 ns;
```

- Utilisation de vecteur de tests et clause **wait for**

```
type vect_type is array (natural range <>) of std_logic_vector(2 downto 0);
constant vect_test : vect_type :=
                        ("000", "011", "010", "111");
...
for k in vect_test 'low to vect_test 'high loop
    (Z, Y, X) <= vecteurs(k);
    wait for 10 ns;
end loop;
```

Messages

- Utilisation de **report** pour afficher un message lors d'un test. On associe un niveau de sévérité au message. Il y en a quatre: Note, Warning, Error et Failure. La couleur du message affiché peut varier en fonction du niveau de sévérité, et la simulation peut être interrompue. Normalement la Note le Warning n'arrête pas la simulation tandis que Error et Failure l'arrêteront (mais c'est configurable dans le simulateur).

```
report "Comparing dec value = " & integer'image(index) severity
note; -- le & sert à concaténer
report "Fin de la simulation"  severity failure;
```

- L'énoncé **assert** est utilisé pour faire un test d'équivalence entre deux expressions (comme un if). Si les deux expressions ne sont pas équivalentes, alors l'énoncé report est exécuté.

```
assert to_integer(unsigned(Z)) = index report "Count Err"
severity warning;
assert false report "Fin de la simulation"  severity failure;
```

Banc d'essai de circuits synchrones

- Dans un circuit synchrone, en plus de générer les entrées régulières, il faut aussi générer les entrées spéciales clk et reset.

```
clk <= not clk after periode / 2;
reset <=
    '0' after 0 ns,
    '1' after 2*periode;
```

Tests pseudo-aléatoire

L'utilisation du package *ieee.math_real* permet l'utilisation de la fonction uniform qui permet de générer un valeur aléatoire entre 0.0 et 1.0 pour la génération de vecteur de tests aléatoires.

```
variable seed1 : positive := 1;
variable seed2 : positive := 2;
variable alea : real;
variable t : integer := -1;
begin
    uniform(seed1, seed2, alea); -- 0.0 < alea < 1.0
    aleatoire := floor(aleatoire * 255.0); -- 0.0 <= alea <= 255.0
    t := integer(alea); -- 0 <= t <= 255
```

Utilisation de fichiers

Les fichiers peuvent être utilisés pour injecter des vecteurs de tests à un banc de tests et pour capturer les résultats de tests. Pour ce faire il faut utiliser le package *std.textio.all*

Voici un exemple d'utilisation de fichier pour lecture :

```
constant filename : string := "premiers.txt";
file vecteurs : text open read_mode is filename;

process
variable tampon : line; -- tampon de lecture
begin
while not endfile(vecteurs) loop
    readline(vecteurs, tampon);
    if tampon(1 to 2) /= "--" then -- commentaires
        read(tampon, n); -- entier
        read(tampon, c); -- séparateur
        read(tampon, c); -- indication: P ou N
        l <= to_unsigned(n, 6);
        wait for 10 ns;
        assert ((c = 'P') = (F = '1') and (c = 'N') = (F = '0'))
        report "erreur entrée " & integer'image(n) severity error;
    end if;
end loop;
deallocate(tampon); -- relâcher la mémoire du tampon
report "simulation terminée" severity failure;
end process;
```

Voici un exemple d'utilisation de fichier pour l'écriture. Les fichiers ouverts dans VHDL sont automatiquement fermés à la fin de la simulation, mais vous pouvez les fermer explicitement si nécessaire.

```
file resultats : text open write_mode is "resultats.txt";

process
 variable tampon : line;  -- tampon d'écriture
 -- Écriture initiale dans tampon
 write(tampon, string'(" ** sortie de simulation, TB.vhd ** "));
 -- Écriture dans le fichier
writeline(resultats, tampon);

for k in 0 to 63 loop  -- boucle de vecteurs exhaustifs
    l <= to_unsigned(k, 6);
    wait for 10 ns;
    write(tampon, string'("temps: "));
    write(tampon, now, unit => ns);
    write(tampon, string'(", entier: ") & integer'image(k));
    write(tampon, string'(", sortie: ") & std_logic'image(F));
    writeline(resultats, tampon);
end loop;
write(tampon, string'(" ** simulation terminée ** "));
writeline(resultats, tampon);

file_close(resultats); -- facultatif
report "simulation terminée" severity failure;
end process;
```

Le tampon d'écriture ne peut être utiliser que pour une écriture. Pour l'utiliser 2 fois, il doit être copier :

```
write(tampon2, tampon.all);
```

Avec Vivado (pas dans tous les simulateurs), on peut utiliser « output » pour écrire à la console tel que :

```
writeline(output, tampon2);
```

Synthèse

- Attention, les bancs d'essai ne sont généralement pas synthétisables étant donné qu'ils utilisent une partie du VHDL qui ne l'est pas comme :
    - Énoncé **wait** (attention, l'utilisation de l'énoncé wait pourrait être synthétisable mais devrait être proscrit car le résultat de simulation pourrait différer du circuit réel synthétisé)
    - Clause **after**
    - Énoncé **assert** ou **report**
    - Types **real** et les fonctions/procédures du package **math_real,** à moins qu'elles ne soient utilisées pour des constantes.
    - Toutes les fonctions/procédures du package **textio**.
    - Autres modèles de haut niveau comme **sqrt** ou utilisation de boucles avec bornes variables.
    - La division, rem et mod (à moins d'une constante ou d'une puissance de 2)
    - Boucles d'exécutions qui ne prennent pas des valeurs statiques.

Mémoires

Tri-state buffer

L'utilisation d'un port **inout** nécessite l'instanciation d'un « tri-state buffer »

```
C : inout  STD_LOGIC;
...
process(A, B)
  begin
    -- C est une sortie lorsque B = 1
    if B = '1' then
    C <= A;
    -- C est une entrée lorsque B = 0
    else
    C <= 'Z';
    end if;
  end process;
```

Distributed RAM

La Distributed RAM est construite en utilisant les cellules de mémoire des LUTs (Look-Up Tables) dans le FPGA. Elle peut être inférée en utilisant le code qui suit :

```
-- Déclaration de la mémoire distribuée
 type ram_type is array (0 to 255) of std_logic_vector(7 downto 0);
 signal ram : ram_type := (others => (others => '0'));
```

Bloc RAM

Les Block RAM sont des blocs de mémoire intégrés dans le FPGA. Ils sont organisés en grandes cellules de mémoire avec une largeur/profondeur fixe, permettant de créer des structures de mémoire plus grandes/performantes.

- Elle peut être inférée en utilisant les primitives de Xilinx :

```
  -- Instanciation d'une primitive BRAM Xilinx
  signal ram : std_logic_vector(7 downto 0);
...
  ram_inst : block_ram
    port map (
      clka => clk,
      wea => we,
      addra => addr,
      dina => din,
      douta => dout );
```

- Elle peut être inférée en utilisant les attributs VHDL :

```
-- Déclaration de la RAM avec des attributs spécifiques
signal ram : std_logic_vector(7 downto 0) := (others => '0');

-- Attribut pour spécifier que la RAM doit utiliser les Block RAM
attribute ram_style : string;
attribute ram_style of ram : signal is "block";
```