
كلية العلوم بتونس
FACULTE DES SCIENCES DE TUNIS

Plan

Plan

Types d'objets

Les types

- VHDL possèdes les différents types d'objets suivants :
  - Scalaires dont la valeur est constituée d'un seul élément
  - Composites dont la valeur comprend plusieurs éléments (tableaux, enregistrements)
  - Accès ou pointeurs
  - Fichiers

Types d'objets

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

  - Les vecteurs de bits

```
"10100"          -- représentation binaire
O"1234"          -- représentation octale
X"ABCD"          -- représentation hexadécimale
```

  - Les caractères

```
'a'
```

  - Les chaînes de caractères

```
"ERREUR"
"ERREUR " & "N° "
```

Types d'objets

Constantes

- Déclarer un objet dont la valeur ne peut pas être modifiée
- Syntaxe

```
CONSTANT nom_de_constante : type := valeur;
```

  - Le type de données `constant` peut être employé à la place des types signal ou variable
  - Il est possible de définir le type des autres données à partir de cette donnée constante

- Exemple

```
CONSTANT N : INTEGER := 8;
SIGNAL B : STD_LOGIC_VECTOR(N-1 DOWNTO 0);
```

Types d'objets

Tableaux

- Le type Array réunit des objets de même type
- Un tableau se caractérise par :
  - Sa dimension
  - Le type de l'indice dans chacune de ses dimensions
  - Le type de ses éléments
- Chaque élément est accessible par sa position (indice)
- Syntaxe

```
TYPE nom_du_tableau IS ARRAY(intervalle) of type_élément;
signal nom_du_signal : nom_du_tableau;
...
nom_du_signal(position)<=valeur;
```

Types d'objets

Les sous-types

- Association d'une contrainte à un type
- La contrainte est optionnelle
- Le simulateur vérifie dynamiquement la valeur de l'objet
- Les opérateurs définis pour le type sont utilisables pour le sous-type

```
subtype printemps is type_mois range mars to juin ;
subtype valeur is bit;
subtype octect is bit_vector ( 7 downto 0);
subtype byte is bit_vector (7 downto 0);
```

Types d'objets

Les agrégats

- Notation permettant de spécifier la valeur d'objets de type tableau ou article
- Dans un agrégat les éléments sont spécifiés par association de position, nommée ou mixte

```
variable client : personne ;
signal bus4 : bit_vector ( 3 downto 0 );
client := ("Dupont","Michel", 45, ( janv,1992)); -- positionnelle
bus4 <= ('1','0','1','1');
bus4 <= (0=>'1', 3=>'1', 2=>'0', 1=>'1'); -- nommée
signal r:bit_vector(31 downto 0):=(5=>'1',8 to 15=>'1',others=>'0');
signal m:memoire(1 to 2048,1 to 8):=(1 to 2048=>(1 to 8=>'1'));
reg <= ('1','0','1', 15 => '1', others =>'0'); -- mixte
```

Types d'objets

Les enregistrements (RECORD)

- Un enregistrement regroupe plusieurs éléments de types différents
- Syntaxe

```
type nom_du_type is record
  identifier : type_indication;
  ...
  identifier : type_indication;
end record;
```

- Exemple

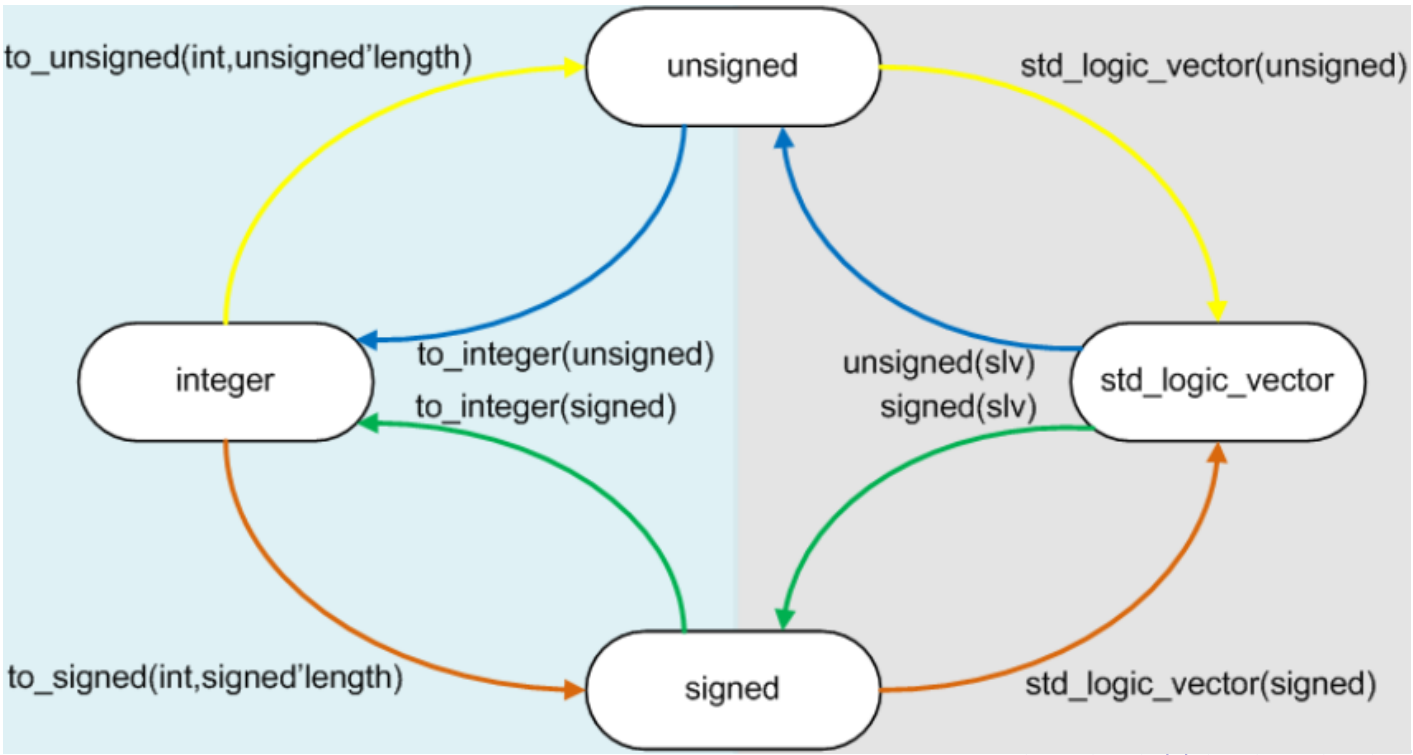
```
type PAQUET is record
  mot_unique : std_logic_vector (7 downto 0);
  data : std_logic_vector (23 downto 0);
  CRC : std_logic_vector( 5 downto 0);
  num : integer range 0 to 1023;
end record;
signal paq_rec : PAQUET;
...
paq_rec.CRC <= "111000";
```

Plan

Conversions

Conversions

- Le paquet `numeric_std` contient une variété de fonctions de conversion utiles
  - Les conversions vers le type entier permettent de réaliser des opérations arithmétiques : additionm soustraction, etc.
  - Les conversions vers le type `std_logic_vector` permettent d'obtenir les résultats en respectant les interfaces d'une unité de conception



Plan

Attributs

Présentation

- Un attribut est une caractéristique associée à un type ou à un objet
- Utile pour les tableaux
- Utile pour les signaux

Attributs

Attributs pré-définis pour les tableaux

- Rendre le code utilisant les tableaux plus générique
  - Pas besoin de connaître à l'avance la taille d'un tableau
- Les attributs suivants sont définis sur n'importe quel type tableau :
  - *LEFT* : élément le plus à gauche de l'intervalle de l'index
  - *RIGHT* : élément le plus à droite de l'intervalle de l'index
  - *HIGH* : élément le plus grand de l'index
  - *LOW* : élément le plus petit de l'index
  - *RANGE* : sous-type des indices, intervalle entre l'attribut LEFT et RIGHT
  - *REVERSE_RANGE* : intervalle inverse de RANGE
  - *LENGTH* : nombre d'éléments du tableau

Attributs

Exemple : attributs des tableaux

➥ Soit le code suivant :

```
type index1 is range 1 to 20;
type index2 is range 19 downto 2;
type vecteur1 is index1 of std_logic;
type vecteur2 is index2 of std_logic;
```

Attribut	vecteur1	vecteur2
LEFT	1	19
RIGHT	20	2
HIGH	20	19
LOW	1	2
RANGE	1 to 20	19 downto 2
REVERSE_RANGE	20 downto 1	2 to 19
LENGTH	20	18

Attributs

Les attributs pré-définis pour les signaux

- Ils permettent de détecter des événements particuliers sur les signaux
- Syntaxe
  $S'event$ vrai seulement si un événement se produit sur S
  $S'active$ vrai seulement si une transaction se produit sur S
  $S'transaction$ signal de type bit qui change d'état à chaque transaction sur S
  $S'last\_event$ retourne le temps écoulé depuis le dernier évènement
  $S'last\_active$ retourne le temps écoulé depuis la dernière transaction
  $S'last\_Value$ retourne la dernière valeur du signal avant le dernier évènement
  $S'stable(T)$ vrai si le signal est n'a eu d'évènements (stable) durant le temps T
  $S'quiet(T)$ vrai si le signal est n'a eu de transaction (tranquille) durant le temps T

Plan

Vérification des circuits numériques

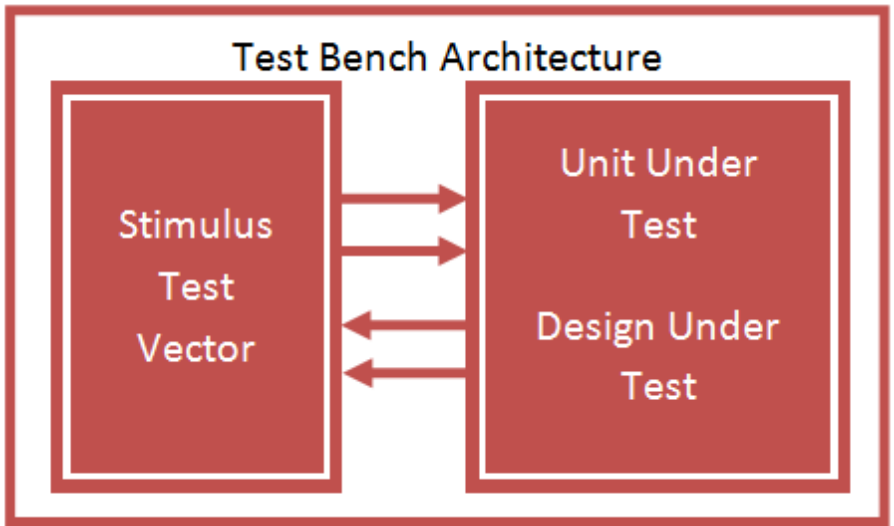
Concepts de base

- La vérification est un processus par lequel on vérifie qu'un design rencontre bien ses spécifications.
- Pour les circuits les plus simples, la vérification complète d'un circuit est un problème très difficile.
  - Vérifier toutes les séquences de valeurs possibles pour les entrées ? ? ?
- La vérification d'un circuit se repose sur la maîtrise des trois principes suivants :
  1. La compréhension de la spécification
  2. Le contrôle des entrées et de signaux internes du circuit à vérifier
  3. L'observation des sorties, des signaux internes et de l'état du circuit à vérifier.

Vérification des circuits numériques

Concepts de base

- La vérification d'un circuit peut être réalisé en VHDL
- Un banc de test (test bench) est un module ou un programme qui permet de :
  - Appliquer des vecteurs de test ou **stimuli** au circuit à vérifier (UUT : Unit Under TEST)
  - Observer et vérifier sa sortie dans le but de vérifier que le circuit rencontre ses spécifications : les réponses du circuit à un stimuli sont identiques aux réponses attendues
- L'écriture d'un banc d'essai en VHDL n'est pas restreinte par les mêmes contraintes que lors de la description d'un circuit



Vérification des circuits numériques

Écriture d'un banc de test

- Un banc d'essai doit effectuer les tâches suivantes :
  1. Instancier le circuit à vérifier ;
  2. Générer des vecteurs de test et les appliquer aux ports d'entrée du circuit ;
  3. [optionnel mais utile] : Générer automatiquement des réponses attendues aux vecteurs de test
  4. [optionnel mais utile] : Comparer les réponses du circuit aux celles attendues, et indiquer toute différence entre les deux par une condition d'erreur
  5. [optionnel mais utile] : lire des vecteurs de test et des réponses attendues d'un fichier, et enregistrer les réponses du circuit ainsi que les résultats de la vérification dans un autre fichier.

Plan

Clauses d'attente

Clause `wait`

- L'instruction wait permet d'attendre
  1. Qu'un des signaux d'une liste soit modifié
  2. Qu'une condition soit vérifiée
  3. Qu'un certain temps se soit écoulé
- Le premier évènement qui arrive termine l'instruction `wait`
- Syntaxe

```
[label:] wait [on sensitivity_list]
         [until condition]
         [for time_expression];
```

Avec

```
sensitivity_list = signal1, signal2, ...
time_expression = une valeur de temps, de type time;
```

Clauses d'attente

Type `time`

- `time` est un type physique
  - Caractérisé par son unité de base, l'intervalle de ses valeurs et ses éventuelles sous-unités

```
type time is range -(2**63-1) to (2**63+1);

units
fs; -- unité de base
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
ms = 1000 us;
sec = 1000 ms;
min = 60 sec;
hr = 60 min;
end units;
```

Clauses d'attente

Utilisation de `wait`

```
constant CLK_PERIOD: time := 10 ns; -- constante de type time

wait; -- attente infinie
wait for 10 ns; -- attente de 10 ns
wait for CLK_PERIOD; -- attente de 10 ns
wait on SignalA; -- attente que SignalA change de valeur
wait on SignalA until SignalA='1'; -- attente que SignalA = '1'
wait on SignalA for 100 ns; -- SignalA = '1' au maximum 100 ns
wait until rising_Edge(clk); -- attente d'un flanc montant de clk
```

Avertissement

Séparer la valeur et l'unité par un espace

Clauses d'attente

`wait` : génération des signaux

- Les signaux d'entrée de l'entité à tester doivent être forcés par le testeur.
- L'utilisation de l'instruction `wait` est cruciale.
- Autre possibilité : clause `after`.

```
process
begin
        c<='0';
        wait for 10ns;
        c<='1';
        wait for 10ns;
        c<='0';
        wait for 40ns;
        c<='1';
        wait;
end process;
```

Clauses d'attente

`wait` : génération du signal horloge

- Pour un système synchrone, un processus du testeur est responsable de générer une horloge

```
constant CLK_PERIOD: time:=40 ns;
signal clk: std_logic;
...
clk_process: process
begin
        clk<='0';
        wait for CLK_PERIOD/2;
        clk<='1';
        wait for CLK_PERIOD/2;
end process;
```

Clauses d'attente

`wait` : génération du reset

- Pour un système synchrone, un processus du testeur doit être responsable de générer un reset.
  - Pour éviter que les bascules ne se trouvent dans l'état indéfini `'U'`.

- Génération d'un reset asynchrone

```
constant CLK_P:time:=40 ns;
...
reset_process: process
begin
        nreset<='0';
        wait for CLK_P/4;
        nreset<='1';
        wait;
end process;
```

- Génération d'un reset synchrone

```
constant CLK_P:time:=40 ns;
...
reset_process: process
begin
        nreset<='0';
        wait for CLK_P;
        nreset<='1';
        wait;
end process;
```

Plan

Notion du temps

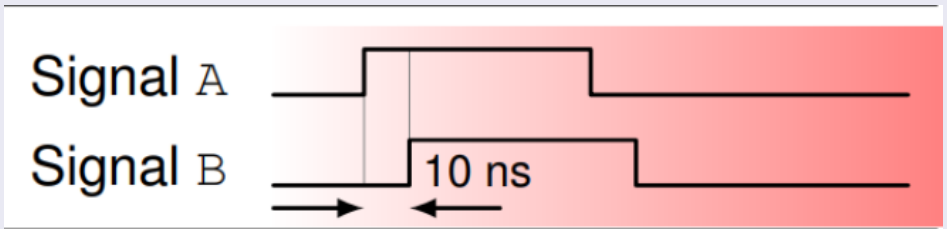
La clause `after`

- Le VHDL définit la notion de temps
- Possibilité de modéliser le comportement réel des portes logiques
  - Retard sur la mise à jour des sorties
  - Retard sur les fils
- La clause `after` permet de simuler l'écoulement du temps
  - Elle est intégrée dans la notion de signal
  - Syntaxe

```
signal_name <= expression after delay;
```

  - Exemple

```
B <= A after 10 ns;
```



Notion du temps

Délais inertial et transport

- Le VHDL définit la notion de temps
- Possibilité de modéliser le comportement réel des portes logiques
  - Retard sur la mise à jour des sorties
  - Retard sur les fils
- La clause `after` permet de simuler l'écoulement du temps

Vérification des circuits numériques

Afficher un message sur la console

- Il existe deux types de délai
  1. Inertiel (par défaut) : les impulsions d'une durée inférieure au délai indiqué sont supprimées du signal affecté

```
S <= inertial A after 10 ns; -- Explicite
S <= A after 10 ns; -- Par défaut
```

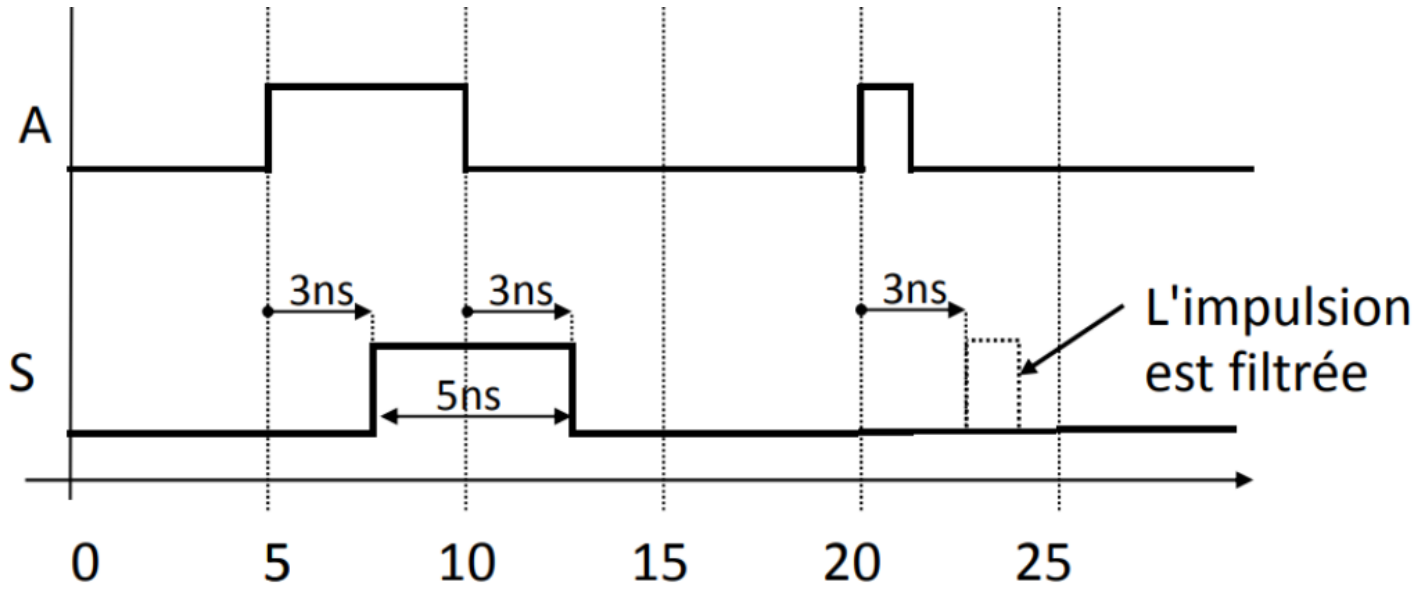
  2. Transport : retard pur

```
S <= transport A after 10 ns;
```

Vérification des circuits numériques

Exemple : délai inertiel

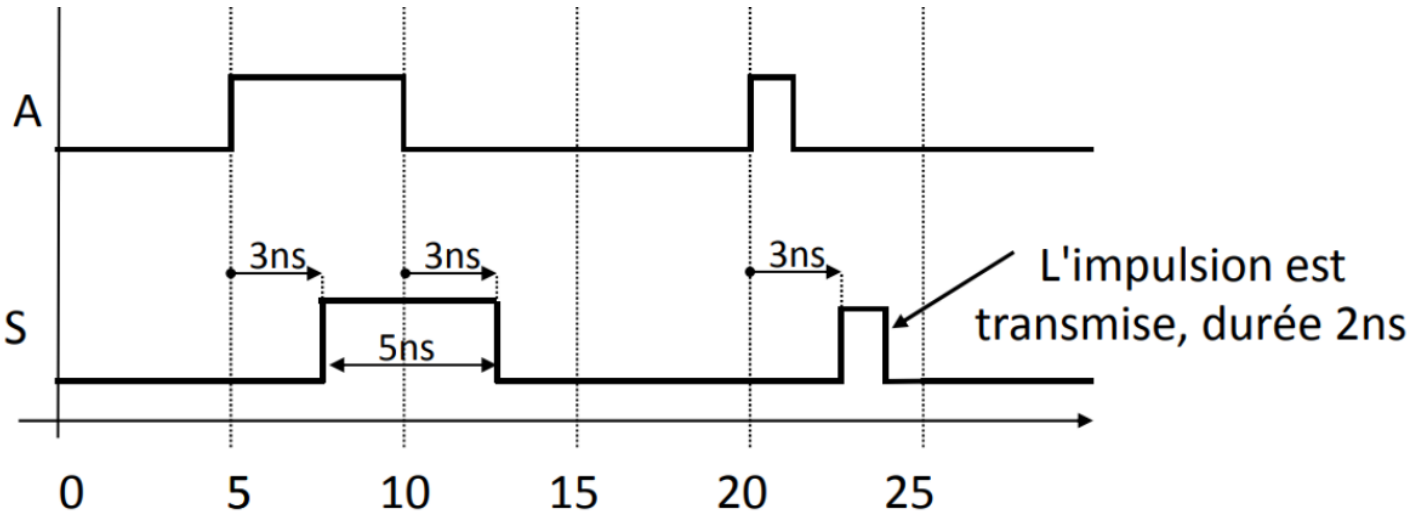
```
A <= '0', '1' after 5 ns, '0' after 10 ns,'1' after 20 ns,
                '0' after 22 ns;
S <= A after 3 ns; --delai inertial
```



Vérification des circuits numériques

Exemple : délai transport

```
A <= '0', '1' after 5 ns, '0' after 10 ns,'1' after 20 ns,
                '0' after 22 ns;
S <= transport A after 3 ns;
```



Plan

Vérification des circuits numériques

Les assertions

- L'instruction `assert` est utilisée dans les simulations et les bancs de test pour :
  - Observer les valeurs renvoyées par l'UUT
  - Comparer les valeurs renvoyées par l'UUT contre les valeurs attendues
- Elle teste la valeur d'une condition booléenne et affiche le message spécifié si la condition est fausse
- Syntaxe

```
assert condition [report message_string]
                          [severity severity_level];
```

- Les niveaux de gravité sont utilisés comme suit :
  - `note` est utilisé pour afficher des informations lors de la simulation
  - `warning` est utilisé dans des situations dans lesquelles la simulation peut être poursuivie, mais les résultats peuvent être imprévisibles
  - `error` est utilisé lorsque la violation d'assertion représente une erreur qui détermine un comportement incorrect du modèle ; la simulation sera arrêtée
  - `failure` : est utilisé lorsque la violation d'assertion représente une erreur fatale, telle que la division par zéro ou adressage d'un tableau avec un index qui dépasse la plage autorisée. La simulation sera arrêté

Plan

Manipulation des fichiers

Écriture d'un banc de test

- Pour la vérification, les fichiers sont utiles pour :
  - Récupérer des données de type stimuli
  - Récupérer des données de référence
  - Fournir les résultats observés
  - Fournir un rapport de simulation
- Pour les descriptions de spécifications
  - Lecture de données (ex : fichier de programmation d'une mémoire)
- Le paquetage TEXTIO offre
  - Le type FILE
  - Les fonctions nécessaires à l'accès aux fichiers

Manipulation des fichiers

Écriture d'un banc de test

- Le type `FILE` représente de l'information stockée dans un fichier externe
- Déclaration d'un objet

```
FILE nom_fichier : type_fichier;
```

- Le type de contenu du fichier doit être défini

```
TYPE Type_Fichier IS FILE OF Quelque_Chose;
```

- Le paquetage TEXTIO définit le type suivant :

```
TYPE text IS FILE OF string;
```

Manipulation des fichiers

Écriture d'un banc de test

- La gestion des fichiers s'effectue au travers du paquetage `TEXTIO`
  - Facilite la manipulation des fichiers
  - Définit les types et procédures nécessaires
  - Est inclus dans la bibliothèque standard
  - Est recommandé pour manipuler les fichiers
  - Est portable
- Les types suivants sont prédéfinis dans `TEXTIO`

```
type LINE is access string;
type TEXT is file of string;
type SIDE is (right,left);
subtype WIDTH is natural;
type file_type is file of elem_type;
type file_open_kind is (read_mode, write_mode, append_mode);
type file_open_status is (open_ok,status_error,name_error
                                              ,mode_error);
```

Manipulation des fichiers

Ouverture d'un fichier

- Les fichiers sont automatiquement
  - Ouverts en début de simulation
  - Fermés en fin de simulation
- Exemple

```
architecture ... is
-- Déclaration d'un fichier en lecture
file fr: TEXT open READ_MODE is "Nom_Du_Fichier1.txt";
-- Déclaration d'un fichier en lecture/écriture
file fw: TEXT open WRITE_MODE is "Nom_Du_Fichier2.txt";
process
begin
...
-- Les fichiers peuvent être directement accédés
...
end process;
end architecture;
```

Manipulation des fichiers

Ouverture explicite d'un fichier

- Il est possible de contrôler l'ouverture/fermeture des fichiers
  - Permet plus de flexibilité
  - La déclaration d'un fichier définit implicitement deux procédures

```
procedure file_open(file f:file_type; extern_name:in string;
open_kind: in file_open_kind:= read_mode);
procedure file_close(file f: file_type);
type file_open_kind is (read_mode, write_mode, append_mode);
```

Manipulation des fichiers

Ouverture explicite d'un fichier

```
architecture ... is
-- Déclaration d'un type de fichier
type file_real is file of real;
-- Déclaration d'un fichier en lecture
file fr: file_real;
-- Déclaration d'un fichier en lecture/écriture
file fw: TEXT;
process
begin
FILE_OPEN(fr, "Nom_du_Fichier1.txt", READ_MODE);
FILE_OPEN(fw, "Nom_du_Fichier2.txt", WRITE_MODE);
...
-- Les fichiers peuvent être accédés
...
FILE_CLOSE(fr);
FILE_CLOSE(fw);
end process;
end architecture;
```

Manipulation des fichiers

Accès aux fichiers : lecture

- Accès en lecture
  - Lecture d'une ligne dans un fichier

```
procedure readline(file f: TEXT; L: out LINE);
```

  - Détection de la fin de fichier

```
function endfile(file f: file_type) return boolean;
```

  - Lecture d'une valeur depuis une ligne

```
procedure read(line l: LINE; value: out elem_type);
```

  - Lecture d'une valeur depuis une ligne, avec statut

```
procedure read(line l: LINE; value: out elem_type;good: out boo
-- elem_type peut être :        bit,bit_vector,boolean,characte
```

Manipulation des fichiers : écriture

Accès aux fichiers

- Accès en écriture
  - Écriture d'une ligne dans un fichier

```
procedure writeline(file f: TEXT; L: inout LINE);
```

  - Écriture d'une valeur dans une ligne

```
procedure write(line l: inout LINE; value: in elem_type;
justified: in SIDE := right;field: in WIDTH :=0);
-- elem_type peut être:
bit,bit_vector,boolean,character,integer,real,string,time
-- justified peut être: RIGHT ou LEFT
-- field définit un nombre minimum de colonnes
```

Manipulation des fichiers

Fichiers prédéfinis : entrées/sorties

- Deux fichiers sont prédéfinis pour accéder au flux d'entrée et à celui de sortie

```
file input : TEXT open read_mode is "STD_INPUT";
file output : TEXT open write_mode is "STD_OUTPUT";
```

- Ils permettent
  - D'intéragir avec la simulation via le terminal de QuestaSim
  - D'intéragir avec la simulation via le terminal de l'OS, en lançant la simulation en ligne de commande
  - D'échanger des données avec une application qui aurait lancé la simulation

MERCI POUR VOTRE ATTENTION

Questions ?