Le langage VHDL

Alain Vachoux
Laboratoire de Systèmes Microélectroniques
alain.vachoux@epfl.ch

Ce document est une introduction aux aspects principaux du langage VHDL. Il ne prétend pas être exhaustif, mais doit fournir une base suffisante pour démarrer des activités de modélisation et de simulation de circuits numériques. Une liste de références est fournie pour approfondir le sujet.

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 2

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 3

Qu'est ce que VHDL?

- **Langage de description de systèmes matériels**
  - Comportement
  - Structure
  - Documentation
- **Développement de modèles exécutables**
  - Simulation
  - Synthèse (sous-ensemble)
- **Modèle logiciel**
  - Langage fortement typé
  - Modularité
  - Extensibilité
- **Standard IEEE (réf. IEEE Std 1076-2002)**
- **Supporté par tous les outils EDA**

Le langage VHDL permet la description des aspects les plus importants d'un système matériel (*hardware system*), à savoir son comportement, sa structure et ses caractéristiques temporelles. Par système matériel, on entend un système électronique arbitrairement complexe réalisé sous la forme d'un circuit intégré ou d'un ensemble de cartes.

Le ***comportement*** définit la ou les fonctions que le système remplit (p. ex. le comportement d'un microprocesseur comporte, entre autres, des fonctions arithmétiques et logiques).

La ***structure*** définit l'organisation du système en une hiérarchie de composants (p. ex. un microprocesseur est constitué d'une unité de contrôle et d'une unité opérative; cette dernière est elle-même, entre autres, constituée d'un composant réalisant les opérations arithmétiques entières et d'un composant réalisant les opérations arithmétiques en virgule flottante).

Les ***caractéristiques temporelles*** définissent des contraintes sur le comportement du système (p. ex. les signaux d'un bus de données doivent être stables depuis un temps minimum donné par rapport à un flanc d'horloge pour qu'une opération d'écriture dans la mémoire soit valable).

Un modèle VHDL est ***exécutable***, c.à.d. qu'il est possible de lui appliquer des stimulis (également décrits en VHDL) et d'observer l'évolution des signaux du modèle dans le temps par ***simulation***. La définition du langage précise les règles d'évaluation de l'état d'un modèle.

Le langage VHDL est aussi utilisé pour la ***synthèse***, par exemple pour dériver automatiquement un circuit à base de portes logique optimisé à partir d'une description au niveau RTL (*Register-Transfer Level*) ou algorithmique. Toute description VHDL légale n'est pas forcément synthétisable.

Le langage VHDL est défini par le standard IEEE 1076. La dernière révision de la norme date de 2002.

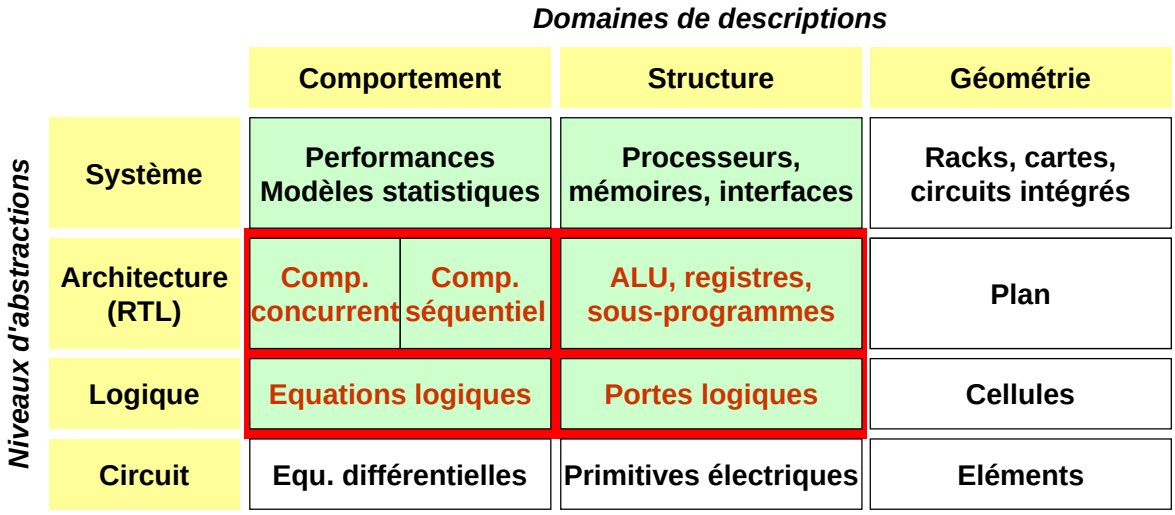
Histoire de VHDL

1980	**Début du projet VHDL financé par le US DoD**
1985	**Première version 7.2 publique**
1987	**Première version du standard IEEE Std 1076-1987**
1993	**Mise à jour du standard (IEEE Std 1076-1993)**
2002	**Mise à jour du standard (IEEE Std 1076-2002)**

le langage VHDL est un standard IEEE depuis 1987 sous la dénomination IEEE Std. 1076-1987 (VHDL-87). Il est sujet à révision tous les cinq ans. Une première révision, qui corrige certaines incohérences de la version initiale et qui ajoute de nouvelles fonctionnalités, a eu lieu en 1994 (IEEE Std. 1076-1993 ou VHDL-93). La dernière révision est celle de 2002 (IEEE Std. 1076-2002 ou VHDL-2002) .

L'IEEE (Institute of Electrical and Electronics Engineers, http://www.ieee.org et http://standards.ieee.org/) est un organisme international qui définit entre autres des normes pour la conception et l'usage de systèmes électriques et électroniques.

Domaine d'application



On considère trois ***domaines de description*** (ou vues). Les vues *Comportement* et *Structure* on déjà été introduites. La vue *Géométrie* décrit les caractéristiques physiques du système matériel.

A chaque ***niveau d'abstraction*** correspond un degré de détail. Le niveau *Système* est le moins détaillé et le niveau *Circuit* est le plus détaillé. Le degré de détail dépend de

- la ***représentation du temps***: causalité, cycles/"ticks" d'horloge, valeurs entières, valeurs réelles
- la ***représentation des données***: jetons, valeurs entières ou réelles, mots binaires.

Le langage VHDL est particulièrement adapté à la description de systèmes matériels au niveau *RTL* (*Register Transfer Level*) et au niveau *Logique*. Il peut être aussi utilisé au niveau *Système* et au niveau *Circuit* avec quelques limitations.

VHDL ne permet pas de prendre en compte les aspects géométriques d'un système matériel.

Comportement concurrent vs. séquentiel (1/3)

- **Exemple: multiplieur, modèle algorithmique**

```
procedure sa_mult (A, B, Z)
-- OPA, OPB     opérandes d'entrée, tailles N, indices N-1:0
-- MRES         résultat, taille 2*N, indices 2*N-1:0
-- REGA         registre local, taille N, indices N-1:0
-- REGB         registre local, taille 2*N, indices 2*N-1:0
-- ACC          accumulateur, taille 2*N, indices 2*N-1:0
-- stop         flag de fin d'opération
begin
    REGA := OPA; REGB := OPB;   -- charger les registres
    ACC := 0;                   -- initialiser l'accumulateur
    stop := FALSE;
    loop
        exit when stop;
        if REGA(0) = '1' then
            ACC := ACC + REGB;
        end if;
        REGA := '0' & REGA(N-1:1);      -- décalage de REGA à droite
        REGB := REGB(2*N-2:0) & '0';    -- décalage de REGB à gauche
        stop := (REGA = 0);             -- stop?
    end loop;
    MRES := ACC;  -- résultat disponible
end;
```

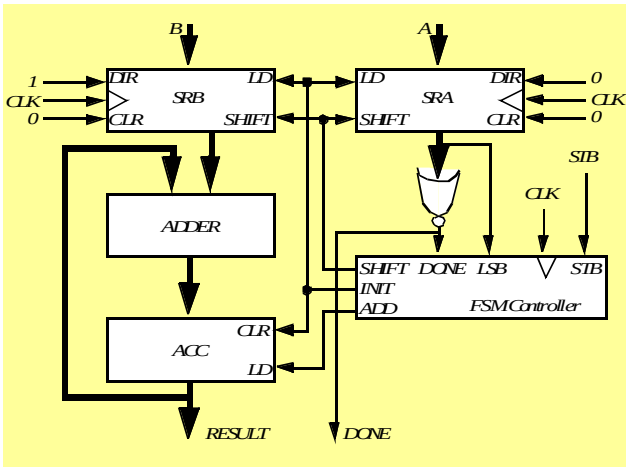

Un modèle algorithmique (p.ex. d'un multiplieur à additions et décalages) est constitué d'un ensemble d'instructions qui doivent être exécutées dans l'ordre donné. On parle de comportement ***séquentiel***.

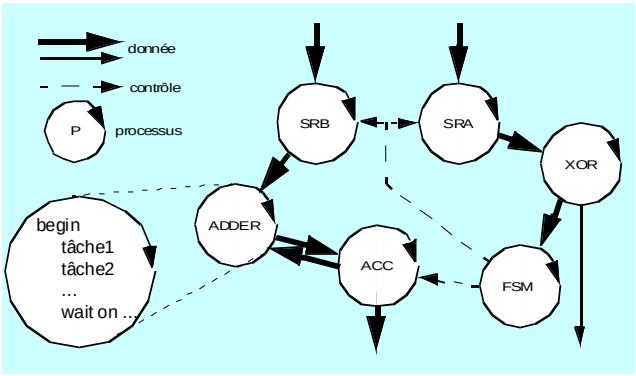
Le modèle ci-dessus n'est ***pas*** un modèle VHDL, mais un pseudo-code relativement proche. Il définit un certain nombre de ***variables*** (OPA, OPB, MRES, …, stop) qui ne correspondent pas nécessairement à des objets physiques (registres, signaux).

Il est possible de développer un modèle VHDL réalisant l'algorithme ci-dessus et de le simuler. Par contre, il n'est pas garanti que le modèle VHDL soit synthétisable.

Comportement concurrent vs. séquentiel (2/3)

- **Exemple: multiplieur, modèle RTL**





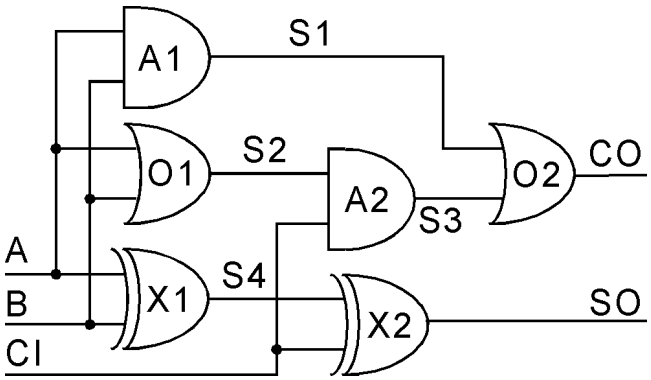
Un modèle RTL est principalement basé sur une description ***concurrente*** du comportement du système matériel. On identifie des composants/blocs qui doivent remplir une fonction particulière (registres, additionneur, séquenceur/contrôleur, etc.) et des ***signaux*** communicants entre ces blocs. Les fonctions des blocs peuvent être décrites de manière séquentielle. Par contre, les blocs peuvent être considérés comme des ***processus concurrents asynchrones***.

Un ***processus*** définit une séquence d'opérations qui sont exécutées en fonction d'événements auxquels il est *sensible* (*triggering conditions*). Les opérations sont usuellement des opérations logiques ou arithmétiques avec un contrôle du flot d'exécution (condition, boucle). Dès qu'un processus est activé il exécute ses instructions jusqu'à un point d'arrêt. Le point d'arrêt est une instruction particulière qui dit essentiellement d'attendre jusqu'à ce qu'il y ait un nouvel événement sur les signaux sensibles du processus. L'exécution d'un processus est cyclique: la séquence d'instructions recommence au début une fois la dernière instruction exécutée. Chaque processus peut être activé de manière concurrente et asynchrone. Un "super processus" (séquenceur, *scheduler*) permet de contrôler l'activation des processus lors de la simulation. Un réseau de processus est interconnecté par des signaux qui sont des fonctions à valeurs discrètes d'une variable entière représentant le temps.

Par exemple, un flanc montant du signal d'horloge CLK et un signal SHIFT actif va simultanément effectuer un décalage à gauche du registre SRB et un décalage à droite du registre SRA. L'effet sera effectivement simultané pour l'utilisateur, même si en réalité les deux processus SRA et SRB sont activés l'un après l'autre dans un ordre quelconque par le simulateur. Une fois les opérations effectuées chaque processus se met en état de veille jusqu'à ce qu'un nouvel événement le réactive à nouveau.

Comportement concurrent vs. séquentiel (3/3)

- **Exemple: additionneur, niveau logique**



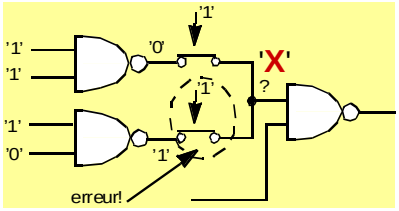
Un modèle logique (*gate-level model*) se base sur un ensemble de primitives logiques (ET, OU, NON, etc.) disponibles sous forme d'opérateurs booléens ou de composants (portes). Les signaux (A, B, CI, CO, SO, S1, S2, S3, S4) sont de mêmes types que ceux d'un modèle RTL.

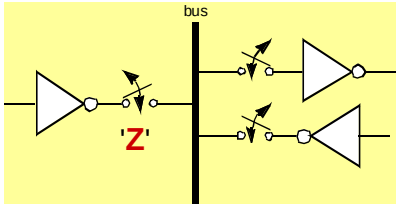
On peut aussi considérer chaque porte ou opérateur logique comme un processus concurrent. En simulation, la sortie d'une porte n'est réévaluée que si un changement d'état à ses entrées le requiert.

Les portes logiques sont usuellement représentatives de cellules d'une bibliothèque de cellules standard pour une technologie donnée (p.ex. CMOS 0.18µ).

Modèle logique (1/2)

- **Représentation des données: signaux logiques**
  - 2 états: '**0**' (vrai), '**1**' (faux)
  - 4 états: '**0**', '**1**', '**X**' (indéfini/conflit), '**Z**' (haute impédance/déconnexion)





  - N états…

- **Représentation du temps: valeur entière**
  - Multiple entier d'une résolution minimum (MRT)
  - Ex.: MRT = 1 fs => t_{max} = 2 μs (codage sur 32 bits) ou 3 h (64 bits)

- **Comportement: expressions logiques**
  - Logique combinatoire: opérateurs booléens (NON, ET, OU, etc.)
  - Logique séquentielle: horloge, délais, mémorisation des états (flip-flop, latch)

La simulation logique permet la vérification de la fonctionnalité est des caractéristiques temporelles de circuits logiques de très grande taille (circuits VLSI - *Very Large Scale Integrated circuits* - 50'000 à 1'000'000 transistors). Ceci est possible car les données et le temps sont modélisés de manière abstraite.

Les données sont représentées comme des signaux logiques ne pouvant prendre qu'un nombre fini d'***états***. Aux moins deux états sont requis: '0' (état vrai) et '1' (état faux). D'autres états sont en pratique nécessaire: l'état 'X' permet de représenter un conflit et l'état 'Z' permet de modéliser la déconnexion temporaire d'un composant connecté à un bus.

Le temps est représenté comme une valeur entière multiple d'un pas de résolution minimum (MRT - *Minimum Resolvable Time*). Comme le temps est codé dans le simulateur avec un nombre fini de bits, le choix du MRT va influencer le domaine des valeurs de temps considérées. Par exemple, un MRT de 1 fs permettra de représenter le temps jusqu'à 2 μs pour un codage sur 32 bits et jusqu'à 3 h pour un codage sur 64 bits. De plus, un délai de 2.5 ns sera interprété comme valant 2 ns si le MRT est de 1 ns et le choix d'un MRT ≤ 100 ps est requis pour représenter correctement ce délai.

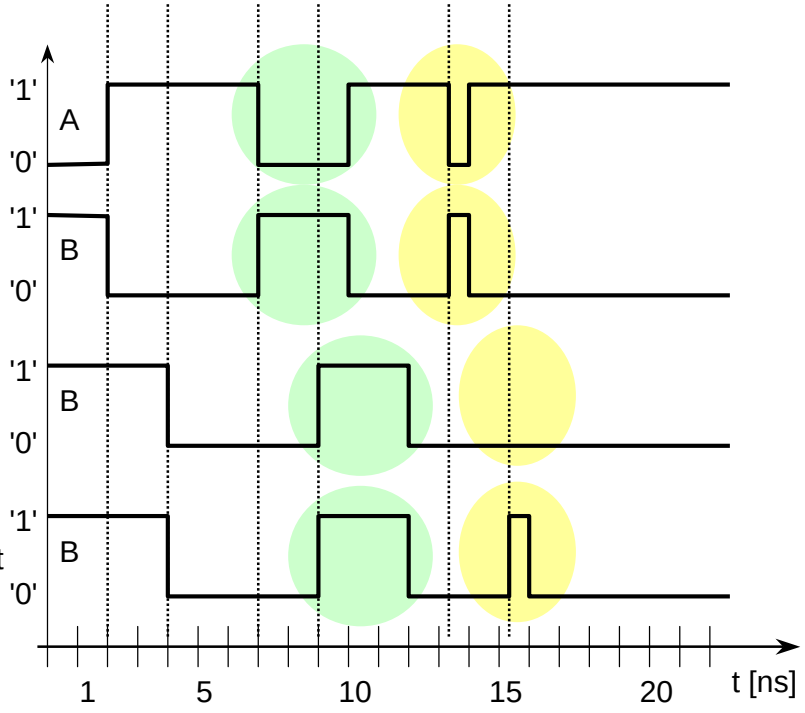
Le calcul du comportement revient à évaluer des expressions logiques combinatoires (NON, ET, OU, etc.) ou séquentielles (avec notion d'horloge, de délais et de mémorisation des états; p. ex. flip-flop ou latch).

Modèle logique (2/2)

♦ **Types de délais**

B = not A

- Délai nul
- Délai inertiel
  Δ_i = 2 ns
- Délai de transport
  Δ_p = 2 ns



La prise en compte des délais permet une vérification plus réaliste du comportement.

Le mode de ***délai nul*** considère le cas idéal où seule la fonctionnalité est vérifiée. Tout changement d'état est répercuté immédiatement.

Le mode de ***délai inertiel*** prend en compte le fait qu'un circuit ne va pas réagir immédiatement et que toute impulsion (transition '0', '1', '0' ou '1', '0', '1') dure suffisamment longtemps pour que l'effet d'un changement d'état soit observable.

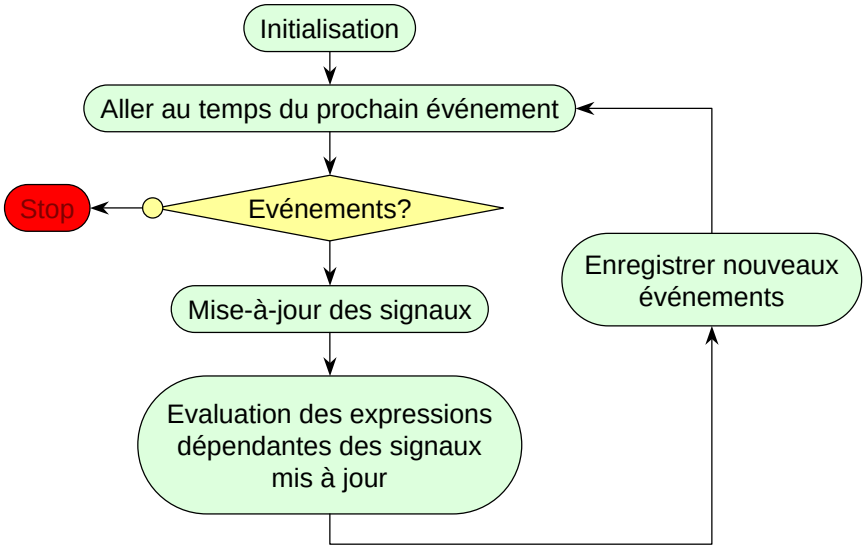
Le mode délai ***transport*** modélise une réponse fréquentielle de largeur de bande infinie pour laquelle toute impulsion, quelle que soit sa durée, est transmise.

Il est aussi plus réaliste de considérer des valeurs de délais différentes pour les transitions montantes '0' -> '1' et pour les transitions descendantes '1' -> '0'.

Ces trois types de délais peuvent être spécifiés dans un modèle VHDL. Les modalités d'application seront détaillées plus loin.

Simulation logique (1/2)

- **Simulation dirigée par les événements**
  - Etat initial pas nécessairement cohérent (pas de propagation d'états en t = 0)



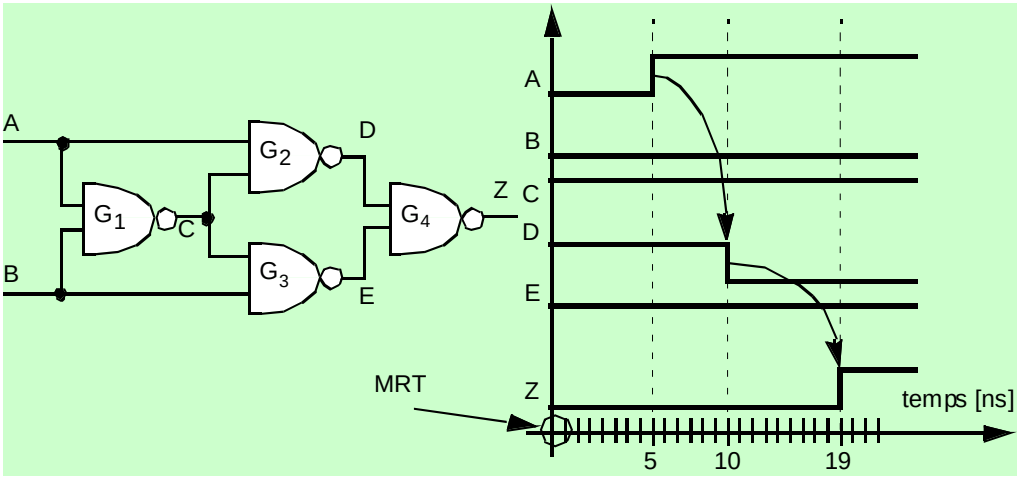
La simulation logique ***dirigée par les événements*** (*event-driven logic simulation*) minimise le nombre d'évaluations logiques à effectuer pour calculer l'état du modèle à un instant donné. Un ***événement*** est un changement d'état qu'il s'agit de répercuter de manière ***sélective*** que sur les expressions (portes) concernées par cet événement. Le temps simulé avance en fonction de l'ordonnancement temporel des événements.

La simulation démarre par l'affectation de valeurs initiales à tous les signaux. Il faut noter que l'état initial du modèle n'est pas nécessairement un état cohérent, ou stable car il n'y a pas encore de propagation des valeurs aux entrées primaires du modèle. Le temps de simulation est ensuite avancé jusqu'au moment du prochain événement prévu. Les valeurs des signaux ayant un événement à ce moment-là sont mises à jour et toutes les instructions ou les composants du modèle concernés par ces mises à jour sont réévalués. Ceci aboutit potentiellement à de nouveaux événements sur des signaux, au même instant (si mode de délai nul) ou à des temps futurs (si mode de délai inertiel ou de transport). La boucle se répète ainsi jusqu'à ce qu'il n'y ait plus d'événements à propager dans le modèle.

La définition du langage VHDL inclut la description formelle d'un ***cycle de simulation canonique*** qui régit la simulation de tout modèle VHDL. Ceci garantit que la simulation d'un même modèle VHDL, stimulé par les mêmes signaux, produira les mêmes résultats, quelle que soit l'implémentation utilisée.

Simulation logique (2/2)

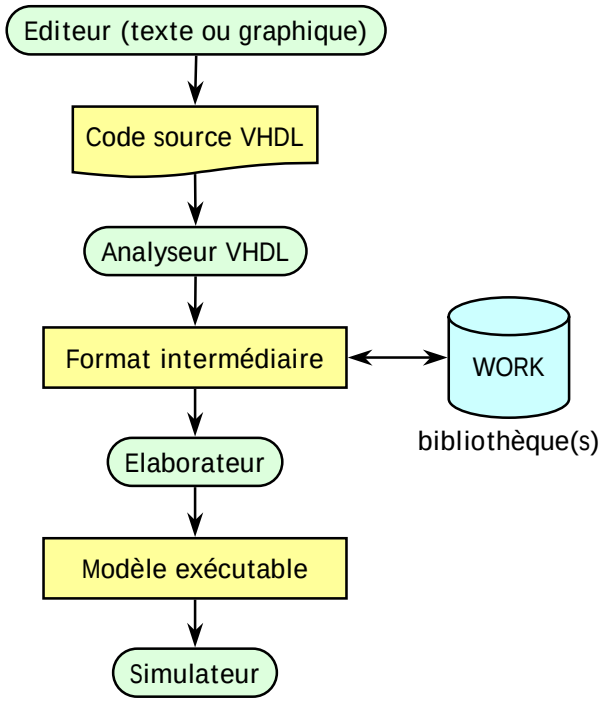
- **Exemple: Z = A ⊕ B**
  - Délai inertiel, $\Delta_{'0' \rightarrow '1'}$ = 9 ns, $\Delta_{'1' \rightarrow '0'}$ = 5 ns



A titre d'exemple simple, considérons un circuit logique réalisant la fonction A ≠ B (ou A ⊕ B) dont la sortie vaut '1' ou la valeur VRAIE si les entrées A et B sont différentes et '0' ou la valeur FAUSSE si les entrées sont les mêmes. Un chronogramme illustrant l'évolution des signaux logiques du circuit est également donné en supposant un changement de valeur sur l'entrée A. Les modèles des portes logiques tiennent comptent d'un délai de changement '0' à '1' de 9 ns et d'un délai de changement de '1' à '0' de 5 ns. Le MRT est de 1 ns.

L'événement sur A au temps courant t_c = 5 ns va forcer une réévaluation de la sortie de la porte G2 et prévoir un événement sur D à $t_c + \Delta_{'1' \rightarrow '0'}$ = 10 ns. L'événement sur D, au temps courant t_c = 10 ns, va forcer une réévaluation de la sortie de la porte G4 et prévoir un événement sur Z à $t_c + \Delta_{'0' \rightarrow '1'}$ = 19 ns. Dans ce cas de figure, les sorties des portes G1 et G3 n'ont pas besoin d'être réévaluées.

Utilisation de VHDL



La création du code source VHDL peut être faite au moyen d'un éditeur de texte ou d'outils graphiques permettant de décrire la structure du système à modéliser sous la forme de schémas ou de diagrammes de blocs et son comportement sous la forme de machines d'états, de chronogrammes ou de tables de vérité.

L'***analyseur*** (ou ***compilateur***) vérifie la syntaxe d'une description VHDL. Il permet la détection d'erreurs locales, qui ne concernent que de l'unité compilée. Plusieurs techniques d'analyse sont actuellement utilisées par les outils du marché. L'approche ***compilée*** produit directement du code machine, ou, dans certains cas, du code C qui sera lui-même compilé. L'objet binaire est alors lié au code objet du simulateur. Cette approche réduit le temps de simulation au détriment du temps d'analyse. L'approche ***interprétée*** transforme le code source en un pseudo-code qui est interprété par le simulateur. Cette approche réduit le temps d'analyse au détriment du temps de simulation.

Chaque concepteur possède une ***bibliothèque de travail*** (*working library*) de nom logique WORK (le nom est standard) dans laquelle sont placés tous les modèles compilés. Le lien du nom logique avec l'emplacement physique de la bibliothèque dépend de l'outil de simulation ou de synthèse utilisé. Il est aussi possible de faire référence, en mode de lecture seule, à d'autres bibliothèques, des ***bibliothèques de ressources***, contenant d'autres modèles ou des utilitaires. Plusieurs bibliothèques peuvent être actives simultanément. Chaque bibliothèque contient une collection de modèles mémorisés dans un format intermédiaire. Elle contient également un certain nombre de relations et d'attributs liant, si nécessaire, les différents modèles entre eux.

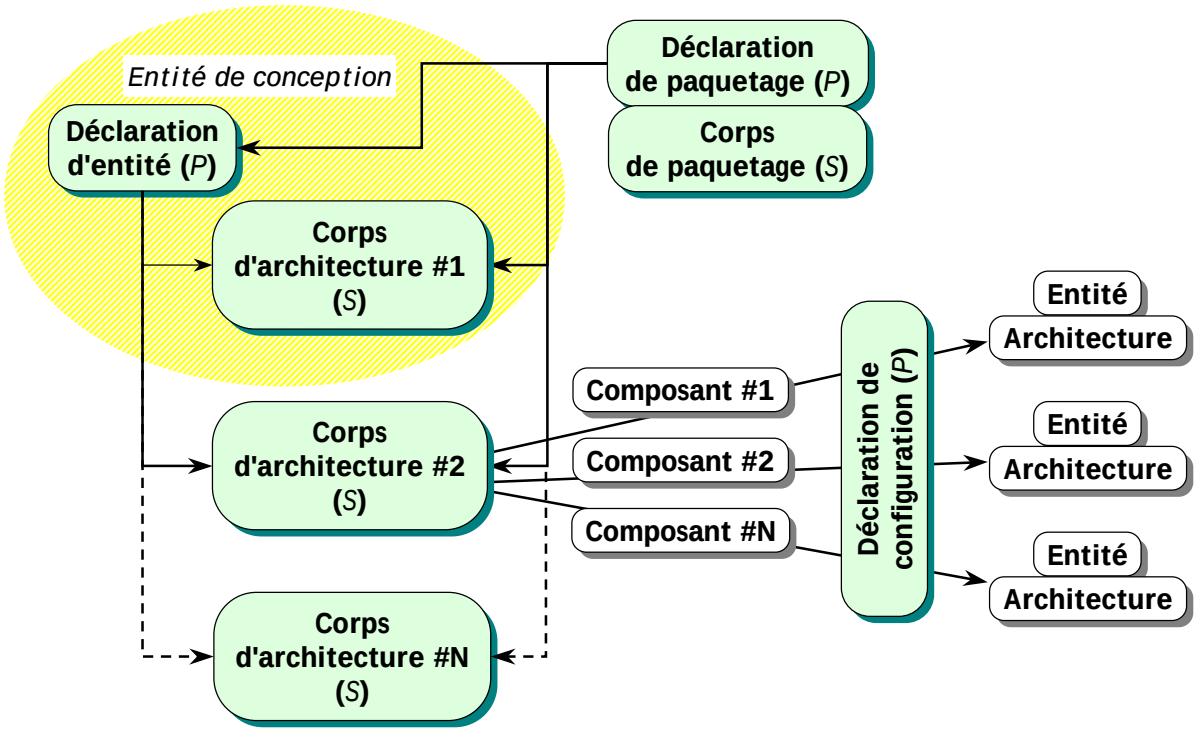
L'***élaborateur*** a pour tâche de créer un modèle exécutable à partir de modules compilés séparément et de détecter des erreurs globales.

Le ***simulateur*** calcule comment le système modélisé se comporte lorsqu'on lui applique un ensemble de stimuli. L'environnement de test peut également être écrit en VHDL: il peut être lui-même vu comme un système définissant les stimuli et les opérations à appliquer aux signaux de sortie pour les visualiser (sous forme texte ou graphique). Le simulateur permet aussi le déverminage (*debugging*) d'un modèle au moyen de techniques analogues à celles proposées pour les programmes écrits en Pascal, C ou Ada: simulation pas à pas, visualisation de variables, de signaux, modification interactive de valeurs, etc.

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 15

Unités de conception



L'***unité de conception*** (*design unit*) est le plus petit module VHDL compilable séparément. Le code source VHDL décrivant une unité de conception est stocké dans un fichier appelé ***fichier de conception*** (*design file*).

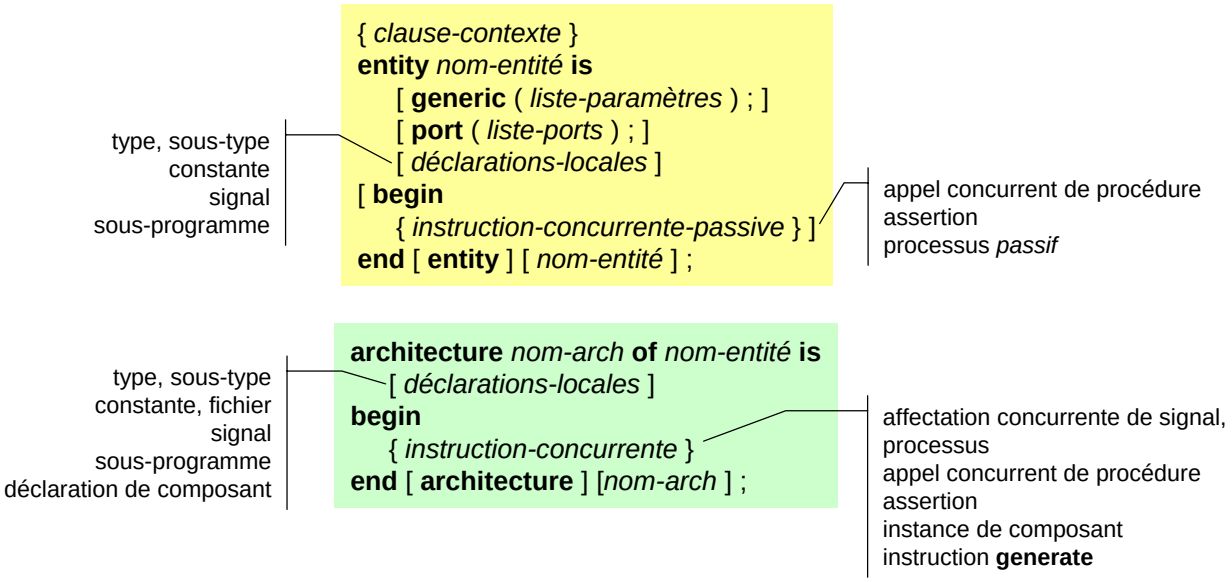
Il existe cinq types d'unités de conception:

- La ***déclaration d'entité*** (*entity declaration*) décrit la vue externe (ou l'interface) d'un composant matériel. Ceci inclut les paramètres et les ports.
- Le ***corps d'architecture*** (*architecture body*) décrit une vue interne d'un composant matériel sous la forme d'un comportement fonctionnel et/ou d'une hiérarchie de sous-composants. Il peut exister plusieurs architectures pour une même entité.
- La ***déclaration de configuration*** (*configuration declaration*) permet d'associer une paire entité-architecture pour chaque instance de composant dans une architecture.
- La ***déclaration de paquetage*** (*package declaration*) définit un ensemble de déclarations (p.ex. de types, de sous-programmes) qui peuvent être utilisés dans plusieurs unités de conception.
- Le ***corps de paquetage*** (*package body*) décrit les corps des déclarations définies dans la déclaration de paquetage (p.ex. les corps de sous-programmes).

L'***entité de conception*** (*design entity*) est l'abstraction de base en VHDL. Elle représente une portion d'un système matériel possédant une interface entrée-sortie et une fonction bien définies. Une entité de conception est constituée d'une déclaration d'entité et d'un corps d'architecture correspondant. Une entité de conception peut représenter un système matériel à plusieurs niveaux de complexité: un système entier, un sous-système, une carte, un circuit intégré, une cellule complexe (p.ex. ALU, mémoire, etc.), une porte logique.

Une unité de conception est qualifiée de ***primaire*** (P) ou de ***secondaire*** (S). Une unité primaire doit être analysée (compilée) avant son unité secondaire correspondante. Toute déclaration faite dans une unité primaire (p.ex. dans une entité) est visible dans toute unité secondaire correspondante (p.ex. dans une architecture).

Entité de conception



Règles de description de la syntaxe:

- Les mots réservés sont indiqués en gras. P. ex.: **entity**, **begin**.
- Les mots en italique décrivent des catégories d'instructions. P ex.: *nom-entité*, *instruction-concurrente*.
- Les termes entre parentèses droites ([…]) sont optionnels.
- Les termes entre accolades ({…}) peuvent être répétés zéro ou plusieurs fois.
  La notation "*terme* {, …}" indique qu'un terme au moins doit être spécifié et que plusieurs termes doivent être séparés par des virgules.
- Une liste de termes séparés par des barres (|) indique qu'un terme de la liste doit être sélectionné.
- Toute autre ponctuation (p.ex. parenthèses rondes, virgules, point-virgules, etc.) doit apparaître telle quelle dans le code source.

Bibliothèques de conception

- **Clause de contexte**

  library *nom-bibliothèque* {, …} ;
  use *sélection* {, …} ;

- **Noms de bibliothèques: identificateurs logiques**
  - Association à des répertoires physiques en-dehors du modèle VHDL
- **Bibliothèques prédéfinies**
  - WORK
  - STD (paquetages STANDARD et TEXTIO)
- **Clause de contexte implicite**

  library std, work;
  use std.standard.**all**;

- **Clause use**
  - Paquetage STANDARD définit le type standard integer
  - La déclaration d'une variable de ce type devrait formellement être:

    variable v: std.standard.integer;

    mais, grâce à la clause de contexte implicite on peut écrire:

    variable v: integer;

Un modèle VHDL ne considére que des bibliothèques logiques, c.à.d. des noms simples. L'association de noms de bibliothèques à des répertoires physiques (p.ex. des répertoires Unix) est faite en-dehors du modèle VHDL et le mécanisme dépend de l'outil utilisé.

VHDL supporte deux bibliothèques logiques prédéfinies:

- La bibliothèque WORK stocke les unités de conception analysées. C'est la seule bibliothèque dans laquelle il est possible d'écrire (modification d'unités existantes ou insertion de nouvelles unités). Toutes les autres bibliothèques ne peuvent être accédées qu'en mode de lecture.
- La bibliothèque STD contient deux unités: le paquetage STANDARD qui définit les types prédéfinis du langage et leurs opérateurs associés, et le paquetage TEXTIO qui définit les types et les sous-programmes pour manipuler des fichiers textes.

Un modèle VHDL peut faire référence à un nombre quelconque de bibliothèques dans une clause de contexte. La clause de contexte est typiquement déclarée juste avant la déclaration d'entité. Tout modèle VHDL possède une clause de contexte implicite qui déclare les bibliothèques standard WORK et STD.

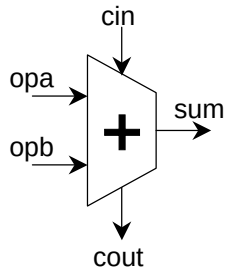
La clause **use** permet de spécifier les noms (de types, de sous-programmes, etc.) déclarés dans un paquetage sans qu'il soit nécessaire de mentionner le chemin d'accès complet, p.ex.

nom-bibliothèque.nom-paquetage.nom-type

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 19

Additionneur 1 bit: Déclaration d'entité




```
entity add1 is
    generic (
        TP: time := 0 ns  -- temps de propagation
    );
    port (
        signal opa, opb, cin: in   bit;    -- opérandes, retenue entrante
        signal sum, cout     : out bit     -- somme, retenue sortante
    );
end entity add1;
```


Une déclaration d'entité définit la vue externe, ou l'interface, d'un composant matériel. On lui attribue un nom, ici add1, qui fait référence au type de composant.

Les données d'interface incluent (optionnellement) des ***paramètres génériques*** (*generic parameters*), représentant des constantes dont les valeurs peuvent être différentes pour chaque instance du composant, et des ***ports*** (*ports*), représentant les canaux de communicationpar lesquels des ***signaux*** transitent entre le composant et le monde extérieur.

Les paramètres génériques et les ports possèdent des ***types*** qui définissent les valeurs qu'il peuvent prendre (paramètres) ou qui transitent par les canaux (ports). VHDL possède un certain nombre de types prédéfinis. Par exemple, le type time représente des valeurs de temps entières multiple d'un MRT (1 fs par défaut) et le type bit représente une valeur logique à deux états '0' ou '1' (les apostrophes sont requises et représentent un caractère).

Un paramètre générique peut posséder une valeur par défaut (ici 0 ns pour le paramètre TP). Il sera ainsi possible de déclarer une instance du composant add1 en ommettant toute référence à ce paramètre.

Un port possède un ***mode*** qui définit la direction dans laquelle les données transitent. Les modes les plus importants sont les modes **in** (donnée entrante), **out** (donnée sortante) et **inout** (donnée entrante ou sortante).

Notes:

- Le mot réservé **signal** peut être omis dans la déclaration de ports (classe par défaut). Un signal est un objet particulier en VHDL qui possède des caractéristiques temporelles.
- Une déclaration d'entité vide est légale et est typiquement utilisée pour un modèle de test. P. ex.:

```
entity tb_add1 is
end entity tb_add1;
```

- Une déclaration d'entité peut être compilée séparément et placée dans la bibliothèque WORK. Elle n'est cependant pas simulable tant qu'elle n'est pas associée à une architecture pour former une entité de conception.
- Toute chaîne de caractère précédée de deux tirets (--) est un commentaire.

Additionneur 1 bit: Architecture "flot de données"

- **Description d'un comportement logique combinatoire**
- **Equations logiques**

$$S = A \oplus B \oplus Cin$$
$$Cout = (A \bullet B) + (A \bullet Cin) + (B \bullet Cin)$$

- **Architecture**

```
architecture dfl of add1 is
begin
    sum <= opa xor opb xor cin after TP;
    cout <= (opa and opb) or (opa and cin) or (opb and cin) after TP;
end architecture dfl;
```

- **Entité de conception add1(dfl)**

Le nom de l'architecture, dfl, fait référence au ***style de description*** "flot de données" (*dataflow*) utilisé pour décrire le comportement de l'additionneur, à savoir un ensemble d'opérations logiques appliquées à des signaux. Ce style est adapté à la description de comportements logiques combinatoires.

L'architecture fait explicitement référence à la déclaration d'entité add1. Toutes les déclaration faites dans l'entité sont ainsi utilisables (visibles) sans redéclaration dans l'architecture. La notation add1(dfl) définit ainsi une entité de conception simulable.

Chaque équation logique est exprimée dans l'architecture comme une ***affectation concurrente de signal*** (*concurrent signal assignment*) dénotée par le signe "<=". Les expressions à droite de ce signe utilisent des opérateurs logiques prédéfinis (**and**, **or**, **xor**) et leurs évaluations produisent des valeurs de type bit. Noter que l'usage d'opérateurs n'implique aucune structure hiérarchique.

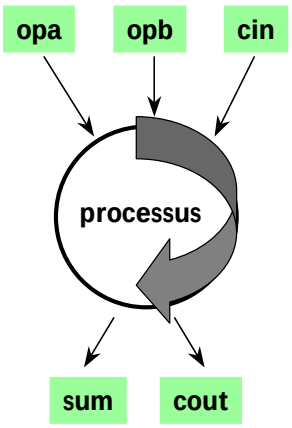
L'affectation de nouvelles valeurs aux signaux sum et cout est faite avec un certain délai dont la valeur est définie par le paramètre générique TP. Ceci est spécifié par la clause **after** à la fin de chaque instruction. Si TP = 0 ns, les affectations auront *apparemment* lieu immédiatement.

La sémantique d'affectation de signal en VHDL est complexe et sera vue en détail plus loin. Ce qu'il faut retenir à ce stade est qu'***un signal ne prend jamais sa nouvelle valeur immédiatement*** (même et surtout en mode de délai nul). Cette caractéristique permet de modéliser correctement la concurrence naturelle des systèmes matériels.

Dans l'exemple ci-dessus, chaque instruction d'affectation est en fait un ***processus*** qui n'est activé (dont l'expression de droite est réévaluée) que si un événement (un changement de valeur) survient sur ***au moins un*** des signaux référencés dans l'expression de droite. Ainsi, l'ordre dans lequel ces instructions sont spécifiées n'a aucune influence sur le résultat de la simulation.

Additionneur 1 bit: Architecture "algorithmique"

- **Description à l'aide d'un comportement séquentiel**




```
architecture algo of add1 is
begin
    process (opa, opb, cin)
        variable tmp: integer;
    begin
        tmp := 0;
        if opa = '1' then tmp := tmp + 1; end if;
        if opb = '1' then tmp := tmp + 1; end if;
        if cin = '1'  then tmp := tmp + 1; end if;
        if tmp > 1  then cout <= '1' after TP;
                    else  cout <= '0' after TP; end if;
        if tmp mod 2 = 0 then sum <= '0' after TP;
                                 else sum <= '1' after TP; end if;
    end process;
end architecture algo;
```

- **Entité de conception add1(algo)**

Le style de description "algorithmique" consiste à décrire le comportement du composant sous la forme d'une séquence d'instructions. Ceci signifie que l'ordre dans lequel ces instructions sont spécifiées est important.

L'architecture algo utilise une ***instruction processus*** explicite (*process statement*) dont le corps inclut des instructions séquentielles qui ne sont exécutées que si un événement survient sur au moins l'un des signaux dits ***sensibles***. Ici il s'agit des signaux **opa**, **opb** et **cin** spécifiés dans une ***liste de sensibilité*** (*sensitivity list*) juste après le mot réservé **process**.

Le processus de l'exemple stoppe lorsque le flot d'exécution atteint la dernière instruction du processus. Il attend alors sur un autre événement et recomencera l'exécution à partir de la première instruction après le mot réservé **begin**. Des modalités d'exécution avec des conditions d'activation plus complexes sont possibles et seront présentées plus loin.

Le corps du processus déclare une ***variable*** tmp qui permet de mémoriser des résultats intermédiaires. Les variables sont des objets particuliers en VHDL qui ne peuvent être utilisés que dans un corps de processus. L'***affectation de variable*** est dénotée par le signe ":=". Contrairement à un signal, une variable prend sa nouvelle valeur immédiatement après l'affectation.

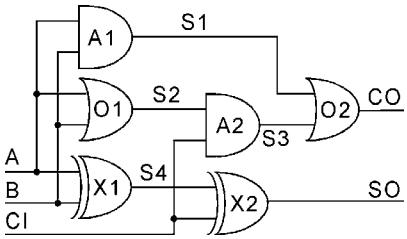
Comme un processus est créé avant le début de la simulation (élaboration) et n'est détruit qu'à la fin de la simulation, ***une variable conserve sa valeur entre deux activations successives du processus***. C'est pourquoi il est nécessaire, dans l'exemple, d'affecter la valeur 0 à la variable tmp au début du processus pour obtenir un résultat correct en simulation. Dans d'autres cas, la rémanence d'une variable est acceptable car elle représente un état interne.

VHDL dispose de toutes les instructions séquentielles essentielles: instruction conditionnelle, sélective, de boucle.

Il est recommandé d'utiliser le plus possible l'affectation de variable dans le corps d'un processus et d'affecter les résultats des calculs à des signaux au dernier moment (juste avant que le processus ne stoppe).

Additionneur 1 bit: Architecture "structurelle" (1/2)

- **Description d'une hiérarchie de composants interconnectés**



- **Instanciation directe**
- **Entité de conception add1(str)**

```
library gates;
architecture str of add1 is
    signal s1, s2, s3, s4: bit;
begin
    A1: entity gates.and2d1(dfl)
            generic map (TPR => TP)
            port map (i1 => opa, i2 => opb, o => s1);
    A2:  entity gates.and2d1(dfl)
            generic map (TPR => TP)
            port map (i1 => s2, i2 => cin, o => s3);
    O1: entity gates.or2d1(dfl)
            generic map (TP)
            port map (opa, opb, s2);
    O2: entity gates.or2d1(dfl)
            generic map (TP)
            port map (s3, s1, cout);
    X1: entity gates.ex2d1(dfl)
            generic map (TPR => TP)
            port map (o => s4, i1 => opa, i2 => opb);
    X2: entity gates.ex2d1(dfl)
            generic map (TP)
            port map (s4, cin, sum);
end architecture str;
```


Le style de description "structurel" consiste à décrire un modèle sous la forme d'une interconnexion de composants communiquant par l'intermédiaire de signaux.

L'architecture str utilise le mécanisme dit d'***instanciation directe*** (*direct instantiation*) qui permet d'inclure directement une entité de conception dans le modèle. Par analogie avec un circuit imprimé, on "soude" chaque instance d'entité de conception (de composant discret) dans l'architecture (sur la carte).

VHDL permet aussi l'utilisation d'un mécanisme d'instanciation indirecte basé sur la déclaration de composants et l'association à des entités de conception au moyen de configurations séparées. Ceci sera présenté plus loin dans le chapitre relatif à la description de structures.

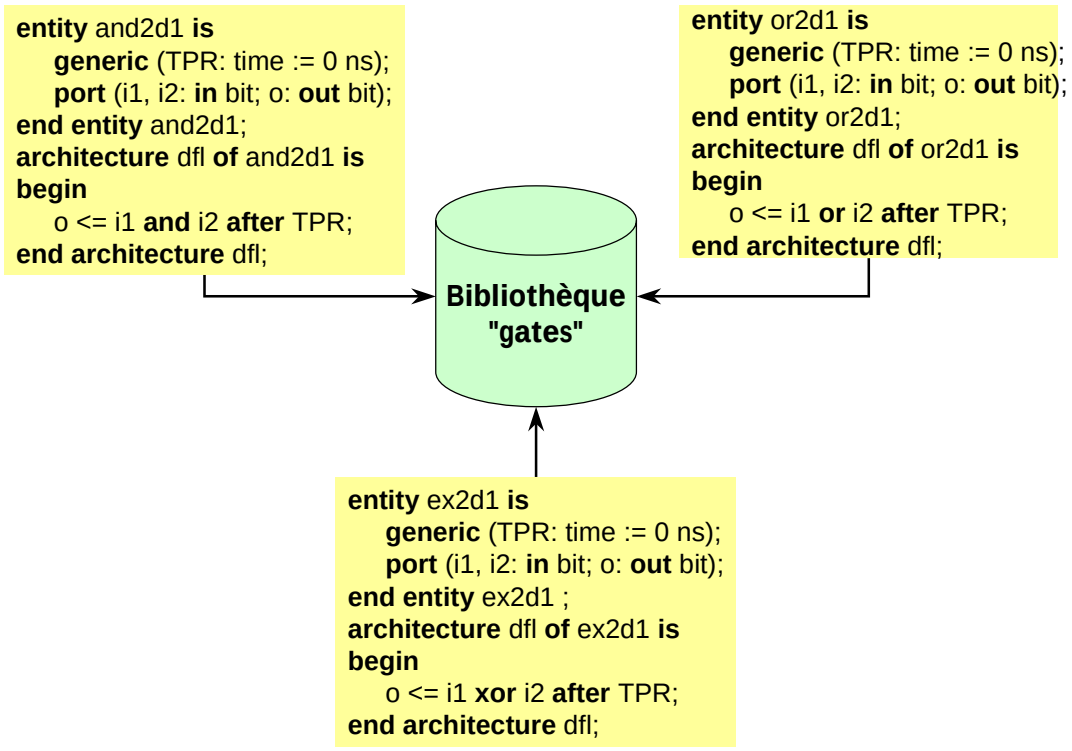
Chaque instance de composant possède une étiquette unique (p.ex. A1 et A2 pour les deux instances de portes AND) et associe (*map*) les paramètres génériques et les ports ***actuels*** (c.à.d. déclarés dans l'entité de conception add1(str)) à leurs équivalents ***formels*** (c.à.d. déclarés au niveau de l'entité de conception instanciée).

L'association peut être faite par ***association de noms*** (*named association*) sous la forme "*formel => actuel*", p.ex. pour les instances A1 et A2, ou par ***association positionnelle*** (*positional association*) pour laquelle seuls les éléments actuels sont spécifiés dans l'ordre de déclaration des éléments formels, p.ex. pour les instances O1 et O2. L'association par noms est préférable car elle documente mieux comment chaque instance est connectée.

L'interconnexion interne des instances de composants est effectuée par la déclaration de signaux locaux s1, s2, s3 et s4 et par l'association de ces signaux (actuels) aux ports formels des instances.

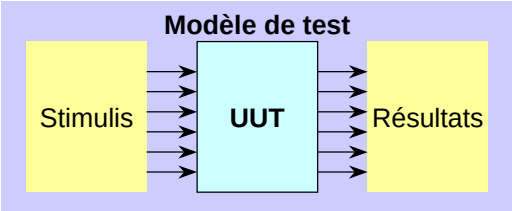
Les entités de conceptions instanciées sont supposées avoir été préalablement analysées et stockées dans une bibliothèque de nom logique gates. La déclaration de bibliothèque (clause **library**) est requise pour rendre cette bibliothèque visible dans l'architecture.

Additionneur 1 bit: Architecture "structurelle" (2/2)



Ces modèles de portes logiques sont extrêmement simplifiés. Les modèles VHDL fournis par les fonderies pour les bibliothèques de cellules standard tiennent compte de beaucoup plus de caractéristiques physiques comme des temps de propagation dépendant du type de transition et des chemins pris par les signaux dans les cellules. Les modèles sont normalement développés en fonction de la norme IEEE 1076.4 (VITAL).

Additionneur 1 bit: Modèle de test (1/3)



op1	op2	ci	sum	co
'0'	'0'	'0'	'0'	'0'
'0'	'1'	'0'	'1'	'0'
'1	'0'	'0'	'1'	'0'
'1'	'1'	'0'	'0'	'1'
'0'	'0'	'1'	'1'	'0'
'0'	'1'	'1'	'0'	'1'
'1	'0'	'1'	'0'	'1'
'1'	'1'	'1'	'1'	'1'

```
entity tb_add1 is
end entity tb_add1;

architecture bench of tb_add1 is
    signal op1, op2, ci, sum, co: bit;
begin

    -- unité à tester
    UUT: entity work.add1(dfl)
                generic map (TP => 1.2 ns)
                port map (
                    opa => op1,
                    opb => op2,
                    cin   => ci,
                    sum => sum,
                    cout => co);

    -- stimulis
    ci <= not ci after 20 ns;
    op1 <= not op1 after 10 ns;
    op2 <= not op2 after 5 ns;

end architecture bench;
```


Un modèle de test est un modèle fermé en ce sens qu'il ne possède pas de communication avec le monde extérieur. Il instancie typiquement l'unité à tester (UUT - *Unit Under Test*), définit un ensemble de stimulis qui vont exercer l'unité à tester et inclut des instructions pour interpréter les résultats de la simulation.

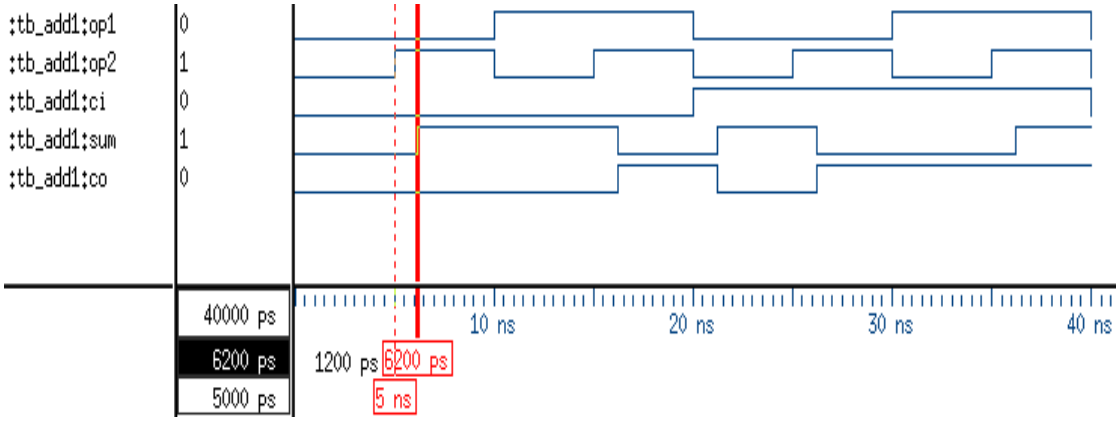
Dans l'exemple ci-dessus, l'unité à tester est l'entité de conception add1(dfl) (le modèle "flot de données" de l'additionneur 1 bit) avec un temps de propagation de 1.2 ns. Les stimulis sont définis par trois affectations concurrentes de signaux de telle manière que la table de vérité de l'additionneur soit vérifiée. Ces affectations réalisent de simples commutations entre état '0' et '1' à des fréquences doublant à chaque fois.

Les valeurs initiales des signaux ci, op1 et op2 valent toutes '0' selon la règle VHDL que tout signal ou variable de type énuméré (le type bit est un type à deux états définis dans l'ordre '0', '1') prend comme valeur initiale par défaut la valeur la plus à gauche de l'énumération, soit '0' dans notre cas.

La vérification que la table de vérité est bien satisfaite doit se faire par examen visuel des formes d'ondes obtenues en simulation.

Additionneur 1 bit: Modèle de test (2/3)

- **Résultats de la simulation du modèle de test**



A. Vachoux, v2.0, 2003 Le langage VHDL - 26

Additionneur 1 bit: Modèle de test (3/3)

- **Modèle plus complet**

```
entity tb_add1 is
end entity tb_add1;

architecture bench2 of tb_add1 is
    signal op1, op2, ci, sum, co: bit;
begin
    UUT: entity work.add1(dfl)
                generic map (TP => 1.2 ns)
                port map (
                    opa => op1,
                    opb => op2,
                    cin  => ci,
                    sum => sum,
                    cout => co);
    Stimulus_check: process
        type table_elem is record
            x, y, ci, co, s: bit;
        end record;
        type table is array (0 to 7) of table_elem;
    …
```

```
        …
        constant TT: table :=
            (-- x -- y -- ci ------ co -- s --
              ('0',  '0',  '0',       '0',   '0'),
              ('0',  '0',  '1',       '0',   '1'),
              ('0',  '1',  '0',       '0',   '1'),
              ('0',  '1',  '1',       '1',   '0'),
              ('1',  '0',  '0',       '0',   '1'),
              ('1',  '0',  '1',       '1',   '0'),
              ('1',  '1',  '0',       '1',   '0'),
              ('1',  '1',  '1',       '1',   '1'));
    begin
        for i in TT'range loop
            op1 <= TT(i).x;
            op2 <= TT(i).y;
            ci <= TT(i).ci;
            wait for 5 ns;
            assert co = TT(i).co and sum = TT(i).s;
        end loop;
        wait;  -- stop définitif du processus
    end process Stimulus_check;
end architecture bench2;
```


Un modèle de test plus élaboré pour le modèle "flot de données" de l'additionneur 1 bit est donné ci-dessus. Il a toujours pour but de vérifier la table de vérité, mais il inclut en plus un mécanisme de vérification automatique des résultats de la simulation avec émission d'un message lorsque une entrée de la table n'est pas vérifiée.

Le processus appelé Stimulus_check définit une structure de donnée permettant de mémoriser la table de vérité et les résultats attendus. Un tableau (*array*) constant de huit éléments appelé TT contient un test par élément. Chaque élément est un enregistrement (*record*) de cinq champs définissant un jeu de stimulis et les résultats attendus pour ces stimulis.

Le corps du processus parcourt chaque élément du tableau toutes les 5 ns. Il affecte les stimulis définis dans les trois premiers champs, applique les stimulis sur l'unité à tester et vérifie que les signaux de sortie ont les valeurs attendues. Si ce n'est pas le cas, l'instruction d'assertion (**assert**) telle quelle est spécifiée dans le modèle va simplement produire un message sur la console indiquant que l'assertion a été violée. Noter que le temps d'attente entre l'application de deux jeux de stimulis successifs doit être plus grand ou égal au temps de propagation de l'unité à tester. Une fois que le processus a parcouru tous les éléments du tableau TT, il stoppe définitivement.

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 28

Types et sous-types

- **Type = ensemble de valeurs + opérateurs associés**
- **Sous-type = type dérivé d'un type de base (type parent) avec contraintes**
- **4 classes de types**
  - Types scalaires: entier, réel, énuméré, physique
  - Types composites: tableaux, enregistrements
  - Pointeurs
  - Fichiers
- **Déclaration de type:** **type** *nom-type* **is** *définition* ;
- **Déclaration de sous-type:** **subtype** *nom-soustype* **is** *définition* ;
- **Types et sous-types prédéfinis dans le paquetage STANDARD**

VHDL est un langage fortement typé: tout objet appartient explicitement ou implicitement à un type ou à un sous-type. Ceci signifie aussi qu'il n'est pas possible de mélanger des objets de types différents dans une expression sans spécifier des conversions explicites. Des objets appartenant à un sous-type sont toutefois compatibles avec des objets appartenant au type de base (ou type parent).

VHDL possède un certain nombre de types et de sous-types prédéfinis. Leurs déclarations sont incluses dans le paquetage STANDARD de la bibliothèque STD. La clause de contexte implicite permet de faire référence à ces (sous-)types sans qu'il soit nécessaire de spécifier le chemin complet (STD.STANDARD.*(sous-)type*).

Types numériques

♦ Types et sous-types numériques prédéfinis

```
type integer is implementation defined; -- au moins 32 bits: [-2147483647, + 2147483647]
type real    is implementation defined; -- au moins [-1.0e38, +1.0e38]

subtype natural  is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

♦ Exemples de types et sous-types non prédéfinis

```
type memory_size is range 1 to 2048;                  -- intervalle montant
subtype word_index is natural range 31 downto 0;      -- intervalle descendant
type signal_level is range -15.0 to 15.0;
subtype probability is real range 0.0 to 1.0;
```

♦ Opérateurs

- Arithmétiques: + – (unaire & binaire) * / **abs** **mod** **rem** (seulement pour types entiers) ** (puissance entière)
- Relationnels: = /= < <= > >=

A. Vachoux, v2.0, 2003 Le langage VHDL - 30

Types énumérés

- **Enumération de toutes les valeurs possibles**
- **Types énumérés prédéfinis**

```
type bit is ('0', '1');                    -- caractères
type boolean is (false, true);             -- identificateurs
type character is 256 caractères du jeu ISO 8859-1;
type severity_level is (NOTE, WARNING, ERROR, FAILURE);
```

- **Exemples de types et sous-types non prédéfinis**

```
type logic4 is ('0', '1', 'X', 'Z');  -- surcharge avec type bit
type states is (IDLE, INIT, CHECK, ADD, SHIFT);
type mixed is (FALSE, '1', '2', '3', IDLE); -- surcharges avec types bit, boolean et states
```

- **Qualification de type**

```
logic4'('1')      ≠   bit'('1')
states'(IDLE)     ≠   mixed'(IDLE)
```

- **Opérateurs**
  - Relationnels: = /= < <= > >=
  - Logiques: **and or nand nor xor xnor not** (types bit & boolean seulement)

L'ordre dans lequel les litéraux sont spécifiés est important car chaque élément de la liste est référencé par un nombre entier indiquant sa position (*position number*). Le premier élément a la position 0. Ceci permet de définir des opérateurs relationnels.

Des déclarations de types énumérés différentes peuvent inclure le même identificateur ou le même caractère. ce mécanisme est appelé ***surcharge*** (*overloading*) . Comme il n'est pas toujours évident de déterminer par le contexte à quel type énuméré l'identificateur ou le caractère appartient, une ***qualification de type*** est nécessaire en cas de surcharge.

Types physiques

- **Types entiers avec facteurs d'échelle**
- **Type et sous-type prédéfinis**

```
type time is range implementation defined
    units
        fs; -- unité primaire et résolution temporelle
        ps = 1000 fs;
        ns = 1000 ps;
        …
        hr = 60 min;
    end units;
subtype delay_length is time range 0 fs to time'high;
```

```
-- exemples de litéraux:
5 ns
ms
8.25 ns (= 8250 ps)
5.6943 ps (= 5694 fs)
```

- **Opérateurs**
  - Arithmétiques: + – (unaire & binaire) * / **abs**
  - Relationnels: = /= < <= > >=
- **Le type time est utile pour spécifier des délais et pour gérer la simulation dirigée par les événements**

Les types physiques ont été introduits en VHDL pour représenter des valeurs de quantités physiques telles que le temps, une longueur, une masse ou une tension électrique en permettant de spécifier des facteurs d'échelle. En pratique, seul le type prédéfini time a conservé une utilité.

Le type time représente le temps simulé. La ***limite de résolution*** (MRT) par défaut est la fs (unité primaire). Les valeurs de temps plus petites que la limite de résolution sont ramenées à zéro. Un simulateur peut permettre de spécifier une limite de résolution secondaire comme un multiple entier de l'unité primaire (p.ex. 100 ps ou 1 ns). Ceci permet de simuler un plus grand intervalle de temps, au prix d'une perte de précision temporelle.

Il est possible de convertir un litéral de type time en un litéral de type integer et réciproquement en utilisant respectivement la division ou la multiplication:

```
10 ms / ms = 10
25 * ns = 25 ns
```

Types tableaux (1/2)

- **Collection d'éléments de même type**
- **Types prédéfinis**
  - Non contraints

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
```

- **Exemples de types non prédéfinis**

```
type word is array (31 downto 0) of bit;     -- intervalle descendant
type memory is array (0 to 255) of word;     -- intervalle montant
type truth_table is array (bit, bit) of bit;
```

- **Accès aux éléments d'un tableau**

```
-- soit W un objet de type word, M de type memory, et TT de type truth_table
W(15)              -- bit d'indice 15
M(122)             -- mot d'indice 122
M(122)(8)          -- bit d'indice 8 du mot d'indice 122
TT('0', '1')       -- bit à la ligne 1 et colonne 2
-- tranches
W(31 downto 16)    -- 16 bits les plus significatifs
M(129 to 255)      -- portion haute de la mémoire
```


Un type tableau définit une collection d'éléments dont les valeurs sont du même type de base. La position de chaque élément dans le tableau est définie par un (tableau mono-dimensionnel) ou plusieurs (tableau multi-dimensionnel) indices de types scalaires.

Le domaine d'indices d'un tableau peut être ascendant (**to**) ou descendant (**downto**). Le tableau est dit ***contraint*** (*constrained*). Un type tableau peut aussi être ***non contraint*** (*unconstrained*): son domaine d'indice n'est pas spécifié à la déclaration de type et la notation "<>" (*box*) est utilisée. La spécification du domaine d'indice actuel devra alors être faite dans la déclaration de l'objet appartenant à ce type.

Types tableaux (2/2)

- **Opérateurs**
  - Relationnels: = /= < <= > >=
  - Logiques*: **and** **or** **nand** **nor** **xor** **xnor** **not**
  - Décalage et rotation*: **sll** **srl** **sla** **sra** **rol** **ror**

    (* tableaux à 1 dimension de types bit ou boolean seulement; opérations logiques effectuées bit à bit)

  - Concaténation: & (tableaux à 1 dimension seulement)

- **L'opérateur de concaténation peut émuler les opérateurs de décalage et de rotation**

```
-- soit B un tableau de type bit_vector(7 downto 0):
"00" & B(7 downto 2)    = B srl 2
B(6 downto 0) & '0'     = B sll 1
B(6 downto 0) & B(0)    = B sla 1
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 34

Types enregistrements

- **Collection d'éléments nommés, ou champs, dont les valeurs peuvent être de types différents**
- **Exemples de types non prédéfinis**

```
type memory_bus is record
    addr : bit_vector(15 downto 0);
    data : bit_vector(7 downto 0);
    read, write: bit;
    enable : boolean;
end record memory_bus;

type complex is record
    real_part, imag_part: real;
end record complex;
```

- **Accès aux éléments d'un enregistrement**

```
-- soit MB un objet de type memory_bus et Z de type complex
MB.addr                  -- tout le tableau addr
MB.addr(7 downto 0)      -- tranche du tableau addr
MB.data(7)               -- élément d'indice 7 du tableau addr
Z.real_part, Z.imag_part -- nombres réels
```


Un type enregistrement définit une collection d'éléments (ou de champs) nommés dont les valeurs peuvent être de types différents. Chaque nom d'élément doit être unique. Les noms d'éléments permettent de les sélectionner (par la notation *nom-enregistrement.nom-élément*) et de les manipuler séparément.

Types fichiers

- **Information stockée dans des fichiers externes**
- **Type prédéfini (dans paquetage TEXTIO)**

```
type text is file of string ;
```

- **Exemples de types non prédéfinis**

```
type word is bit_vector(7 downto 0);
type word_file is file of word;
type real_file is file of real;
```

- **Opérations implicitement déclarées**

```
type file_type is file of elem_type;

procedure read (file f: file_type; value: out elem_type);
function endfile (file f: file_type) return boolean;
procedure write (file f: file_type; value: in elem_type);
procedure file_open (file f: file_type; external_name: in string;
                     open_kind: in file_open_kind := read_mode);
procedure file_open (status: out file_open_status; file f: file_type;
                     external_name: in string;
                     open_kind: in file_open_kind := read_mode);
procedure file_close (file f: file_type);
```


Un type fichier définit une information qui est stockée dans des fichiers externes. Les éléments d'un fichier peuvent être de n'importe quel type scalaire, enregistrement ou tableau mono-dimensionnel.

A chaque déclaration de type fichier sont associées implicitement les opérations suivantes: ouverture du fichier (file_open), fermeture du fichier (file_close), lecture du fichier (read), détection de la fin de fichier (endfile), écriture dans le fichier (write).

Un fichier peut être ouvert selon plusieurs modes définis par le type file_open_kind:

```
type file_open_kind is (read_mode, write_mode, append_mode);
```

Le paramètre optionnel status permet de connaître l'état d'une opération d'ouverture. Les états possibles sont définis par le type file_open_status:

```
type file_open_status is (open_ok, status_error, name_error, mode_error);
```

Le type fichier prédéfini text est déclaré dans le paquetage standard TEXTIO dans la bibliothèque STD. Des procédures de lecture et d'écriture d'éléments de texte (chaînes de caractères, nombres, etc.) sont également fournies par ce paquetage. L'usage du paquetage TEXTIO requiert la déclaration de la clause **use** suivante:

```
use STD.TEXTIO.all;
```

Expressions et opérateurs

- **Expression = formule pour calculer une valeur**
- **Priorité des opérateurs (de la plus basse à la plus haute)**
  - Logiques: **and** **or** **nand** **nor** **xor** **xnor**
  - Relationnels: = /= < <= > >=
  - Décalage et rotations: **sll** **srl** **sla** **sra** **rol** **ror**
  - Addition: + - &
  - Signe (unaires): + -
  - Multiplication: * / **mod** **rem**
  - Divers: ** **abs** **not**
- **Exemples**

```
-- soient A et B de type integer
B /= 0 and A/B > 1  -- court-circuit
A**2 + B**2
4*(A + B)
(A + 1) mod B
```

```
-- soient A, B et C de type boolean
A and B and C -- évaluation de gauche à droite
              -- idem pour or, xor and xnor
A nor (B nor C) -- parenthèses requises, nor pas associatif
              -- idem pour nand
```

```
-- soient PI et R de type real
PI*(R**2) / 2
2.0*PI*R
```


Une expression est une formule qui spécifie comment calculer une valeur. Une expression est constituée d'opérandes (termes) et d'opérateurs.

Les termes d'une expression sont typiquement des valeurs litérales (p.ex. B"001101") ou des identificateurs représentant des objets (constantes, variables, signaux).

Les opérateurs sont associés à des types. Chaque opérateur appartient à un ***niveau de précédance*** ou de priorité qui définit l'ordre dans lequel les termes de l'expression doivent être évalués. L'usage de parenthèse permet de rendre les niveaux de priorité explicites ou de changer les niveaux par défaut.

Les opérateurs logiques **and**, **nand**, **or** et **nor** utilisent un ***mécanisme de court-circuit*** lors de l'évaluation. L'opérateur **and**/**nand** n'évalue pas le terme de droite si le terme de gauche s'évalue à la valeur '0' ou false. L'opérateur **or**/**nor** n'évalue pas le terme de droite si le terme de gauche s'évalue à la valeur '1' ou true.

Les opérateurs relationnels ne peuvent s'appliquer que sur des opérandes de même type et retournent toujours une valeur de type boolean. Les opérandes doivent être d'un type scalaire ou de type tableau mono-dimensionnel avec éléments d'un typediscret (entier ou énuméré). Les opérateurs "=" et "/=" ne peuvent pas avoir des opérandes de type fichier.

Objets et constantes

- **Objet = élément nommé ayant des valeurs d'un type donné**
- **4 classes d'objets: constantes, variables, signaux, fichiers**
- **Une constante possède une valeur fixe durant la simulation**
- **Déclaration de constante** **constant** *nom-const* {, ... } : *soustype* [:= *expression*] ;
- **Exemples de déclarations**

```
constant PI: real := 3.1416;
constant CLK_PERIOD: time := 20 ns;
constant STROBE_TIME: time := CLK_PERIOD/2; -- = 10 ns
constant MAX_COUNT: positive := 255;

-- initialisation avec un aggrégat
constant BV0: bit_vector(15 downto 0) := (others => '0');
constant TT: truth_table: (others => (others => '0'));
constant MEMBUS: memory_bus :=
                        (addr => X"00A7", data => X"FF",
                         read => '0', write => '1', enable => '1');
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 38

Variables

- **Simple conteneur dont la valeur peut changer en cours de simulation**
- **Déclaration de variable** **variable** *nom-var* {, ... } : *soustype* [:= *expression*] ;
- **Valeur initiale par défaut = soustype'left**
- **Exemples de déclarations**

```
                                -- valeur initiale:
variable count: natural;        -- 0
variable done: boolean;         -- false
variable lev: signal_level;     -- -15.0
variable rval: real;            -- -1.0e-38
```

```
-- avec valeur initiale explicite
variable state: states := IDLE;
variable start, stop: time := STROBE_TIME + 2 ns;
```


Une variable possède une valeur initiale qui est déterminée à partir de sa déclaration (valeur explicite) ou à partir de son type (valeur par défaut ou implicite). Dans ce dernier cas, la règle est la suivante: Soit T le type scalaire de la variable, alors la valeur initiale par défaut est la valeur la plus à gauche des valeurs prises par le type. Par exemple: -2_147_483_648 pour le type integer, -1.0e-38 pour le type real, '0' pour le type bit, false pour le type boolean. Si T est un type composite, la règle s'applique séparément à chaque type des éléments.

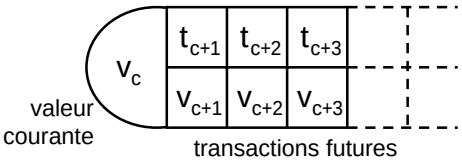
Une variable peut changer de valeur en cours de simulation au moyen d'une ***instruction d'affectation de variable*** (instruction séquentielle). Les instructions sont présentées plus loin.

Signaux

- **Représentent des formes d'ondes logiques sous forme de paires temps/valeur**
- **Objets essentiels à la modélisation de comportements concurrents**
- **Déclaration de signal** **signal** *nom-sig* {, ... } : *soustype* [:= *expression*] ;
- **Exemples de déclarations**

```
signal S: bit_vector(15 downto 0);      -- valeur initiale = (others => '0')
signal CLK: bit := '1';                 -- valeur initiale explicite
signal reset, strobe, enable: boolean;  -- valeur initiale = false
```

- **Un signal possède un structure de donnée complexe (pilote)**



Les signaux sont les objets de base pour la modélisation et la simulation de systèmes matériels. Leur structure (pilote - *driver*) permet de modéliser les caractéristiques temporelles des signaux réels et de prendre en compte les différents modes de délais (nul, inertiel, transport). Le pilote d'un signal stocke sa valeur courante et les ***transactions*** futures prévues sur ce signal sous forme de paires temps/valeur. Les valeurs de temps sont de type time. Un signal peut être de n'importe quel type scalaire ou composite.

La règle pour déterminer la valeur initiale d'un signal est identique à celle valable pour une variable.

Un signal peut changer de valeur en cours de simulation au moyen d'une ***instruction d'affectation de signal*** (instruction séquentielle ou concurrente). Les instructions sont présentées plus loin.

Fichiers

- **Représentent un stockage externe**
- **Déclaration de fichier**

  file *nom-fichier* {, … } : *type-fichier* [[**open** *mode-ouverture*] **is** *nom-logique*] ;

- **Exemples de déclarations**

```
type integer_file is file of integer;

file file1, file2: integer_file;
-- fichiers pas automatiquement ouverts; appel à la procédure file_open requis

file file3: integer_file is "intdata";
-- fichier ouvert en mode lecture et associé au nom logique "intdata"

file file4: integer_file open write_mode is "intdata";
-- fichier ouvert en mode écriture
```

- **Fermeture de fichiers**
  - A la fin de la simulation
  - Au moyen de la procédure file_close

Un fichier est un objet représentant un dispositif de stockage externe qui peut conserver de l'information au-delà d'une simulation. Un objet fichier ne peut être que d'un type fichier.

Un déclaration de fichier définit au moins un ou plusieurs fichiers et leur type. Le mode d'ouverture du fichier et le nom logique du fichier.

Un fichier peut être ouvert selon plusieurs modes définis par le type file_open_kind:

```
type file_open_kind is (read_mode, write_mode, append_mode);
```

Le mode d'ouverture par défaut est read_mode.

Le nom logique optionnel est une chaîne de caractères. L'association du nom logique à un fichier physique dépend du système d'exploitation et ne fait ainsi pas partie du modèle.

Dans le cas où le mode d'ouverture et le nom logique sont omis, la procédure prédéfinie file_open doit être exécutée explicitement dans le modèle avant tout accès à un élément du fichier. L'état de l'ouverture peut être vérifié en examinant la valeur du paramètre status de type file_open_status:

```
type file_open_status is (open_ok, status_error, name_error, mode_error);
```

Un fichier est automatiquement fermé à la fin de la simulation. La procédure prédéfinie file_close permet de fermer un fichier explicitement.

Un fichier ne peut pas être la cible d'une affectation. Il ne peut être que passé en paramètre à un sous-programme.

Attributs prédéfinis (1/3)

- **Attribut = information sur un élément du langage**
  - Par exemple: longueur d'un signal bus = bus'length
- **Permettent d'écrire des modèles généraux**
- **Notation**
  - Elément: type ou objet
  - Attribut: type, intervalle, valeur, fonction ou signal

élément ' attribut

- **Attributs scalaires utiles**

type address **is** integer **range** 7 **downto** 0;

address'left = 7	address'image(5) = "5"
address'right = 0	address'value("4") = 4
address'low = 0	address'high = 7

type MVL4 **is** ('U', '0', '1', 'Z');

MVL4'pos('U') = 0	MVL4'image('Z') = "Z"
MVL4'pos('Z') = 3	MVL4'value("0") = '0'
MVL4'val(2) = '1'	MVL4'succ('1') = 'Z'
MVL4'pred('0') = 'U'	

time'image(5 ns) = "5000000 fs"
time'value("250 ms") = 250 ms
time'val(634) = 634 fs
time'pos(2 ps) = 2000

A. Vachoux, v2.0, 2003 Le langage VHDL - 42

Attributs prédéfinis (2/3)

- **Attributs tableaux utiles**

```
type word is bit_vector(31 downto 0);
type memory is array (7 downto 0) of word;
variable mem: memory;
```

-- objet	type	valeur
mem'low	memory'low	0
mem'high	memory'high	7
mem'left	memory'left	7
mem'right	memory'right	0
mem(3)'low	word'low	0
mem(3)'high	word'high	31
mem'length	memory'length	8
mem(3)'length	word'length	32
mem'range	memory'range	7 **downto** 0
mem'reverse_range	memory'reverse_range	0 **to** 7
mem(3)'range	word'range	31 **downto** 0

A. Vachoux, v2.0, 2003 Le langage VHDL - 43

Attributs prédéfinis (3/3)

- **Attributs signaux utiles**
  - Objets signaux implicites:

S'delayed (T)	Signal ayant même valeur que S mais retardé de T unités de temps ($T \geq 0$ ns)
S'stable(T)	Signal de type boolean valant TRUE si aucun événement n'est arrivé sur S durant T unités de temps ($T \geq 0$ ns), et FALSE sinon

  - Fonctions:

S'event	Fonction à valeur de type boolean valant TRUE si un événement est arrivé sur S durant le cycle de simulation courant, et FALSE sinon
S'last_event	Fonction à valeur de type time valant le temps écoulé depuis le dernier événement sur S
S'last_value	Fonction à valeur du type de S valant la valeur de S avant le dernier événement sur S

A. Vachoux, v2.0, 2003 Le langage VHDL - 44

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 45

Domaines d'instructions

- **Instructions concurrentes**
  - Instructions s'exécutant dans des flots séparés
  - Base pour la modélisation de systèmes matériels
  - Requièrent un gestionnaire d'événements
- **Instructions séquentielles**
  - Instructions s'exécutant dans un ordre donné
  - Similaires à celles existant dans les langages de programmation
- **Cycle de simulation canonique**

```
entity E is
    -- déclarations de paramètres, ports…
begin
    -- instructions concurrentes passives...
end entity E;

architecture A of E is
    -- déclarations…
begin
    -- instructions concurrentes...
    P1: process
        -- déclarations…
    begin
        -- instructions séquentielles...
    end process P1;
    -- instructions concurrentes...
    P2: process
        -- déclarations…
    begin
        -- instructions séquentielles...
    end process P2;
    -- instructions concurrentes...
end architecture A;
```


VHDL considère deux domaines d'instructions disjoints. Chaque domaine possède ses instructions légales qu'il n'est pas permis d'utiliser dans l'autre domaine (exception: instruction d'affectation de signal).

Les ***instructions concurrentes*** (*concurrent statements*) permettent de modéliser des comportements dirigés par les événements qui peuvent s'exécuter de manière asynchrone. Les instructions concurrentes ne peuvent être utilisées que dans la partie exécutable d'une déclaration d'entité (sous réserve qu'elles soient ***passives***, c.à.d. qu'elles ne modifient pas l'état du modèle) ou dans la partie exécutable d'une architecture.

Les ***instructions séquentielles*** (*sequential statements*) permettent de modéliser des comportements procéduraux constitués d'un suite d'action à exécuter en séquence. Les instructions séquentielles ne peuvent être utilisées que dans le corps d'un processus ou d'un sous-programme.

VHDL définit un ***cycle de simulation canonique*** qui précise comment ces différentes instructions sont exécutées.

Processus

- **Instruction concurrente la plus fondamentale**
  - Toute instruction concurrente peut s'exprimer de manière équivalente à l'aide d'un processus

- **Syntaxe**

constante
type
variable
sous-programme

[*étiquette* :] **process** [(*nom-signal* { , ... })] [**is**]
{ *déclaration* }
begin
{ *instruction-séquentielle* }
end process [*étiquette*] ;

- **Vie et mort d'un processus**
  - Contexte local créé dans la phase d'élaboration
  - Activé/stoppé durant la simulation
  - Contexte local détruit à la fin de la simulation

- **Pas un sous-programme!**

Le processus est la forme la plus fondamentale d'une instruction concurrente. Les contextes locaux de tous processus d'un modèle sont créés avant la simulation dans la phase d'élaboration. Un processus peut être activé ou stoppé un nombre quelconque de fois durant une simulation. Son contexte local est détruit à la fin de la simulation. C'est pourquoi ***les variables déclarées conservent leur valeur d'une activation à une autre***. Ceci est à opposer aux sous-programmes dont les contextes locaux sont détruits à la fin de leur exécution et reconstruits à chaque nouvelle exécution.

La partie exécutable d'un processus ne peut contenir que des instructions séquentielles.

Modalités d'activation d'un processus

- **Seulement si un événement survient sur au moins un signal *sensible***
- **Soit liste de sensibilité**

```
process (S1, S2, ...)
begin
    ...
end process;
```

, soit instruction wait

```
process
begin
    ...
    wait ...;
    ...
end process;
```

- **Instruction wait**
  - Instruction séquentielle
  - Plusieurs instructions dans un processus possible

```
-- liste de sensibilité
wait on S1, S2, S3;

-- délai (de type time)
wait for 10 ns;

-- condition
wait until clk = '1';
wait on clk until clk = '1';     -- équiv. au précédant
wait on reset until clk = '1';  -- plus sensible sur clk!
```

```
-- forme générale
wait on S1, S2 until en = '1' for 15 ns;

-- stop définitif (pas de réactivation possible)
wait;
wait until next_event;
    -- si: variable next_event: boolean;
```


La partie exécutable d'un processus n'est activée que si un événement (changement d'état) survient sur au moins un signal dit ***sensible***.

Les signaux sensibles d'un processus peuvent être spécifiés de deux manières ***mutuellement incompatibles***: la liste de sensibilité ou l'instruction **wait**. Il est interdit d'utiliser les deux formes dans le même processus.

La liste de sensibilité est une liste de noms de signaux entre parenthèses spécifiée juste après le mot réservé **process**.

L'instruction **wait** est une instruction séquentielle qui peut prendre plusieurs formes:

- La forme **wait on** est équivalente à une liste de sensibilité.
- La forme **wait for** permet de spécifier un délai d'attente de type time.
- La forme **wait until** permet de spécifier une condition. Une liste de sensibilité constituée de tous les signaux référencés dans la condition est implicitement considérée. La liste de sensibilité peut être rendue explicite, mais dans ce cas tout événement sur un signal référencé dans la condition mais pas dans la liste de sensibilité sera ignoré en simulation.
- La forme générale **wait on** ... **until** ... **for** ... réactive le processus de toute façon après le délai spécifié.
- L'instruction **wait** suspend le processus de manière définitive si l'instruction ne fait référence à aucun signal.

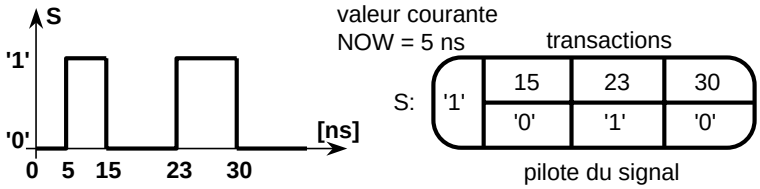
Instruction d'affectation de signal (1/2)

♦ Forme générale

[*étiquette* :]
nom-signal <= [*mode-délai*] *expression-valeur* [**after** *expression-temps*] {, … } ;

♦ Génération d'une forme d'onde

```
-- signal S: bit;
stimulus: process
begin
    wait for 5 ns;
    S <= '1', '0' after 10 ns, '1' after 18 ns, '0' after 25 ns;
    wait; -- forever
end process stimulus;
```




```
-- signal S: bit;
stimulus: process
    begin
        wait for 5 ns;
        S <= '1';
        wait for 10 ns;
        S <= '0';
        wait for 8 ns;
        S <= '1';
        wait for 7 ns;
        S <= '0';
        wait;
end process stimulus;
```


L'affectation à un signal est représentée par le signe "<=".

La partie droite de l'instruction doit au minimum spécifier une valeur compatible avec le type du signal affecté ou une expression dont l'évaluation fournit une valeur de ce type.

La combinaison "*expression-valeur* **after** *expression-temps*" constitue un ***élément de forme d'onde***. Elle peut être spécifiée plusieurs fois, à la condition que les valeurs de temps (du type time) soient croissantes. Chaque élément de forme d'onde est enregistré dans le ***pilote*** (*driver*) du signal comme une ***transaction*** qui sera appliquée dans le futur.

Le processus stimulus commence par attendre un délai initial de 5 ns avant d'affecter une forme d'onde au signal S. La valeur initiale par défaut du signal S est '0'. Les valeurs de temps sont des délais ***relatifs*** à l'instant auquel l'instruction est exécutée. Le processus stoppe ensuite définitivement. Les transactions futures enregistrées dans le pilote du signal créeront des événements et des réévaluations d'autres parties du modèle. La forme de gauche définit la forme d'onde en une seule instruction. La forme de droite utilise plusieurs instructions d'affectation et **wait** pour obtenir le même résultat.

La fonction NOW est prédéfinie en VHDL et retourne le temps de simulation courant.

Les différents modes de délais sont présentés plus loin.

Instruction d'affectation de signal (2/2)

```
entity xor_gate is
    port (A, B: in bit; Z: out bit);
end entity xor_gate;
```

- **Instr. séquentielle**

```
architecture proc1 of xor_gate is
begin
    process (A, B)
    begin
        Z <= A xor B after 2 ns;
    end process;
end architecture proc1;
```

```
architecture proc2 of xor_gate is
begin
    process
    begin
        Z <= A xor B after 2 ns;
        wait on A, B;
    end process;
end architecture proc2;
```

- **Instr. concurrente**

```
architecture dfl of xor_gate is
begin
    Z <= A xor B after 2 ns;
end architecture dfl;
```


L'instruction d'affectation de signal a la même forme et la même interprétation en simulation, qu'elle soit séquentielle (à gauche) ou concurrente (à droite).

Un processus avec une liste de sensibilité explicite est équivalent à un processus avec une instruction **wait on** juste avant le **end process**.

Exemples de processus

♦ Générateur d'horloge

```
-- signal clk: bit := '0';
clk_gen: process (clk)
begin
    clk <= not clk after 5 ns;  -- 100 MHz
end process clk_gen;
```

♦ Elément asynchrone

```
-- signal A, B, Q: bit;
MullerC: process
begin
    wait until A = '1' and B = '1';
    Q <= '1';
    wait until A = '0' and B = '0';
    Q <= '0';
end process MullerC;
```

♦ Latch

```
entity latch is
    port (en, d: in bit; q: out bit);
end entity latch;

architecture bhv of latch is
begin
    process (en, d)
    begin
        if en = '1' then
            q <= d;
        end if;
    end process;
end architecture bhv;
```

♦ Flip-flop

```
entity dff is
    port (clk, d: in bit; q: out bit);
end entity dff;

architecture bhv of dff is
begin
    process
    begin
        wait until clk = '1';
        q <= d;
    end process;
end architecture bhv;
```


Le processus du générateur d'horloge est sensible sur tout événement sur le signal clk. Il ne stoppe jamais en simulation car il force le signal à commuter entre ses deux états possibles '0' et '1' avec un certain délai, ce qui crée toujours une nouvelle transaction dans le futur.

Le modèle de flip-flop n'est sensible que sur le flanc montant du signal clk. A l'instant où le flanc montant est détecté, l'entrée d du flip-flop est échantillonnée et transmise sur la sortie q. Tout changement d'état sur l'entrée d ne peut être détecté qu'à cet instant.

Le modèle de latch est sensible sur deux signaux: en (*enable*) et d (*data*). Tant que le signal en est actif (égal à '1'), tout événement sur le signal d sera répercuté directement sur la sortie q (mode transparent). La latch est verouillée dès que le signal en est inactif (égal à '0').

Le modèle d'élément asynchrone utilise plusieurs instructions **wait**. Ce style de description permet de modéliser des machines à états finis.

Signal ou variable?

```
entity shiftreg is
    generic (W: positive := 8);  -- largeur du registre
    port (clk, din: in bit; dout: out bit);
end entity shiftreg;
```

```
architecture sig of shiftreg is
    signal reg: bit_vector(W-1 downto 0);
begin
    process
    begin
        wait until clk = '1';
        reg(W-1) <= din;
        reg(W-2 downto 0) <= reg(W-1 downto 1)
        dout <= reg(0);
    end process;
end architecture sig;
```

- **Ordre des affectations de signaux *pas* important**

```
architecture var of shiftreg is
begin
    process
        variable reg: bit_vector(W-1 downto 0);
    begin
        wait until clk = '1';
        dout <= reg(0);
        reg(W-2 downto 0) := reg(W-1 downto 1)
        reg(W-1) := din;
    end process;
end architecture var;
```

- **Ordre des affectations de variables important**

Le choix d'utiliser un signal ou une variable dépend de plusieurs points:

- ***Domaine d'application***
  Un signal est global à toute l'architecture. Une variable est locale à un processus. Seuls les signaux peuvent communiquer entre plusieurs processus dans une architecture et avec le monde extérieur (par l'intermédiaire des ports d'interface).
- ***Sémantique d'affectation***
  Une variable prend sa nouvelle valeur au moment de l'exécution d'une instruction d'affectation. Un signal ne prend ***jamais*** sa nouvelle valeur à un tel moment. Ce point sera détaillé plus loin.
- ***Efficacité de simulation***
  Une variable n'est qu'un simple conteneur et n'utilise donc qu'une place limitée dans la mémoire du simulateur. Un signal requiert une structure de donnée plus importante (pilote) et une gestion plus complexe à cause de ses caractéristiques temporelles et des aspects concurrents du modèle.

L'exemple ci-dessus illustre deux modèles corrects d'un registre à décalage. Pour représenter le contenu du registre, l'architecture sig utilise un signal, alors que l'architecture var utilise une variable. Dans le premier cas, l'ordre des instructions d'affectation de signal n'est pas importante car à chaque cycle d'exécution du processus (déclenché par un flanc montant du signal clk) les valeurs des signaux reg(…) référencés dans les parties droites des affectations sont toujours celles calculée au ***cycle précédant***. Dans le second cas, l'ordre d'affectation des variables est important car les variables reg(…) sont mises à jour immédiatement.

Instructions séquentielles (1/3)

♦ Instruction conditionnelle

```
[ étiquette : ]
if expr-booléenne then
    { instr-séquentielle }
{ elsif expr-booléenne then
    { instr-séquentielle } }
[ else
    { instr-séquentielle } ]
end if [ étiquette ] ;
```

```
-- Exemple:
Max: if A > B then
    vmax := A;
elsif A < B then
    vmax := B;
else
    vmax := integer'low;
end if Max;
```

♦ Instruction sélective

```
[ étiquette : ]
case expression is
    when choix { | ... } => { instr-séquentielle }
    { when choix { | ... } => { instr-séquentielle } }
end case [ étiquette ] ;
choix ::= { expr-simple | intervalle | others }
```

```
-- Exemple:
case int is  -- soit int du type integer
    when 0          => V:= 4; S <= '1' after 5 ns;
    when 1 | 2 | 7  => V:= 6; S <= '1' after 10 ns;
    when 3 to 6     => V:= 8; S <= '1' after 15 ns;
    when9           => null;  -- pas d'opération
    when others     => V := 0; S <= '0'; -- tous les autres cas
end case;
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 53

Instructions séquentielles (2/3)

- **Instructions de boucle**

```
-- boucle générale
[ étiquette : ] loop
    { instr-séquentielle }
end loop [ étiquette ] ;
```

```
-- boucle infinie
loop
    wait until CLK = '1';
    q <= d after 5 ns;
end loop;

-- boucle avec sortie
L: loop
    exit L when value = 0;
    value := value / 2;
end loop L;
```

```
-- boucle while
[ étiquette : ]
while condition loop
    { instr-séquentielle }
end loop [ étiquette ] ;
```

```
L: while i < str'length loop
    next L when i = 5;
    i := i + 1;
end loop L;
```

- **Les instructions next et exit permettent de contrôler la boucle**

```
-- boucle for
 [ étiquette : ]
for identificateur in intervalle loop
    { instr-séquentielle }
end loop [ étiquette ] ;
```

```
for i in 15 downto 0 loop
    vector(i) := i*2;
end loop;

L1: for i in 15 downto 0 loop
    L2: for j in 0 to 7 loop
        exit L1 when i = j;
            tab(i,j) := i*j + 5;
    end loop L2;
end loop L1;
```

- **L'indice de boucle ne doit pas être déclaré**

Les instructions de boucle permettent de décrire des comportements répétitifs:

- La ***boucle générale*** permet de décrire des boucles infinies.
- La ***boucle while*** permet d'effectuer des itérations de manière conditionnelle.
- La ***boucle for*** permet d'effectuer un nombre fini d'itérations. L'îndice de boucle est implicitement déclaré et il est considéré comme une constante pendant une itération. Un objet de même nom que l'indice de boucle déclaré en-dehors de la boucle est localement caché dans le corps de la boucle. L'indice de boucle n'existe pas en-dehors du corps de la boucle.

Les instructions **next** et **exit** permettent de contrôler le comportement de la boucle:

- L'instruction **next** stoppe l'itération courante et démarre l'itération suivante.
- L'instruction **exit** stoppe l'itération et sort de la boucle. Le flot d'exécution reprend à la première instruction après la boucle.

Les deux instructions peuvent être exécutées conditionnellement. Par défaut, les deux instructions affectent la boucle courante. Elles peuvent toutefois affecter des boucles imbriquées de niveau plus haut en faisant référence à des étiquettes.

Instructions séquentielles (3/3)

♦ Instructions d'assertion et de rapport

[*étiquette* :] **assert** *expr-booléenne*
    [**report** *expression-chaîne*]
    [**severity** *niveau-sévérité*] ;

[*étiquette* :] **report** *expression-chaîne*]
    [**severity** *niveau-sévérité*] ;

```
assert value <= MAX_VALUE
    report "value too large" severity ERROR;

assert resistance > 0.0;
    -- rapport par défaut: "Assertion violation"
    -- sévérité par défaut: ERROR

-- formes équivalentes
assert FALSE
    report "Value = " & value_type'image(value)
    severity NOTE;

report "Value = " & value_type'image(value);
```


Les instructions d'assertion sont utiles pour inclure des vérifications dans un modèle. Une assertion peut être n'importe quelle expression légale qui s'évalue à une valeur de type boolean. Si la valeur est false, une violation d'assertion est émise et des actions particulières sont exécutées:

- Un message est produit sur la console. Le contenu du message peut être spécifié dans l'instruction d'assertion sous la forme d'une chaîne de caractères. Par défaut, le message "Assertion violation" est produit.
- Chaque violation d'assertion peut être associée à un niveau de sévérité dont les valeurs possibles sont définies par le type prédéfini severity_level:

  type severity_level **is** (NOTE, WARNING, ERROR, FAILURE);

  Le niveau de sévérité est aussi mentionné dans le message produit dans la console. Le niveau de sévérité par défaut est NOTE.

L'instruction **report** permet d'afficher un message dans la console indépendemment de toute condition d'assertion. Le niveau de sévérité est NOTE.

Le traitement du niveau de sévérité dépend du simulateur. En général, le simulateur permet de spécifier le niveau de sévérité à partir duquel la simulation doit stopper en cas de violation.

Instruction concurrente conditionnelle

- **Format**

[*étiquette* :] *nom-signal* <= [*mode-délai*]
{ *forme-onde* **when** *expr-booléenne* **else** }
forme-onde [**when** *expr-booléenne*] ;

- **Exemple**

```
muxc:
z <= d0 after 2 ns when sel1 = '0' and sel0 = '0' else
     d1 after 2 ns when sel1 = '0' and sel0 = '1' else
     d2 after 2 ns when sel1 = '1' and sel0 = '0' else
     d3 after 2 ns when sel1 = '1' and sel0 = '1';
```

```
-- processus équivalent
muxc: process (d0, d1, d2, d3, sel1, sel2)
begin
    if sel1 = '0' and sel2 = '0' then
        z <= d0 after 2 ns;
    elsif sel1 = '0' and sel2 = '1' then
        z <= d1 after 2 ns;
    elsif sel1 = '1' and sel2 = '0' then
        z <= d2 after 2 ns;
    elsif sel1 = '1' and sel2 = '1' then
        z <= d3 after 2 ns;
    end if;
end process muxc;
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 56

Instruction concurrente sélective

- **Format**

[*étiquette* :] **with** *expression* **select**
nom-signal <= [*mode-délai*] { *forme-onde* **when** *choix* { | ... } , }
forme-onde **when** *choix* { | ... } ;

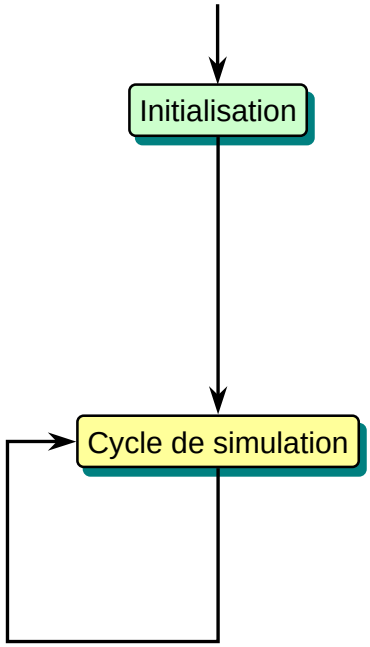
- **Exemple**

```
muxs:
with sel select    -- sel: bit_vector(1 downto 0)
    z <= d0 after 2 ns when "00",
         d1 after 2 ns when "01",
         d2 after 2 ns when "10",
         d3 after 2 ns when "11";
```

```
-- processus équivalent
muxs: process (d0, d1, d2, d3, sel)
begin
    case sel is
        when bit_vector'("00") => z <= d0 after 2 ns;
        when bit_vector'("01") => z <= d1 after 2 ns;
        when bit_vector'("10") => z <= d2 after 2 ns;
        when bit_vector'("11") => z <= d3 after 2 ns;
    end case;
end process muxs;
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 57

Initialisation et cycle de simulation



I1. Affecter la valeur initiale à chaque variable et signal.

I2. T_c = 0 ns, δ = 0.

I3. Exécuter tous les processus jusqu'à leur suspension.

I4. Déterminer le temps du prochain événement T_n selon S4.

S1. $T_c = T_n$.

S2. Mettre à jour les signaux.

S3. Exécuter tous les processus sensibles aux signaux mis à jour.

S4. Déterminer le temps du prochain événement T_n:
- **Si transactions au temps courant, $\delta = \delta + 1$ -> S2**
- **Si plus de transactions OU temps = time'high -> STOP**
- **Sinon, T_n = temps prochaine transaction, δ = 0.**

S5. Exécuter tous les processus retardés (*postponed*).

S6. -> S1.

La simulation d'un modèle VHDL débute par une phase d'***initialisation*** et est suivie par l'exécution répétitive d'un ***cycle de simulation***.

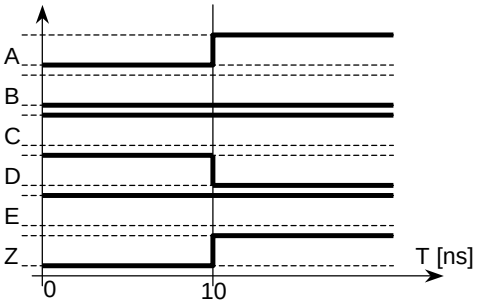
Durant la phase d'initialisation, chaque signal et variable reçoit une valeur initiale qui est soit déterminée par son type, soit par l'expression qui accompagne sa déclaration. Le temps de simulation courant (T_c) est ensuite mis à zéro, chaque processus est activé et ses instructions sont exécutées jusqu'à la première instruction **wait** rencontrée. Chaque processus est finalement suspendu à l'endroit de l'instruction **wait**. L'exécution des instructions des processus génère usuellement des transactions dans des pilotes de signaux. Une affectation de signal à délai nul génère une transaction à traiter au temps de simulation courant (***délai ou cycle delta***). Une affectation de signal à délai non nul génère une transaction à traiter à un temps futur. Finalement, le temps du prochain événement à traiter (T_n) est déterminé: soit le temps courant (cycle delta), soit un temps futur, soit la fin de la simulation.

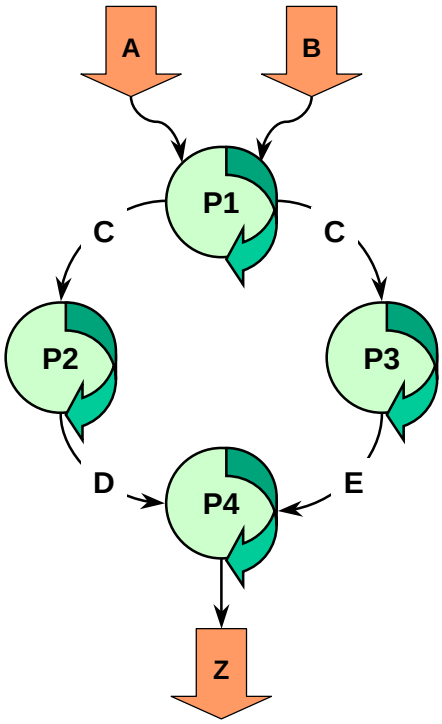
Le cycle de simulation proprement dit commence par affecter le temps du prochain événement au temps courant. Toutes les transactions prévues à ce temps sont alors appliquées, ce qui se traduit par une éventuelle mise à jour de la valeur courante de signaux. Une transaction aboutissant à un changement de valeur d'un signal définit un événement sur ce signal. Tous les processus sensibles aux signaux qui ont subit un événement sont ensuite réactivés et leurs instructions exécutées jusqu'à la prochaine instruction **wait**. Cette étape génère normalement de nouvelles transactions sur des signaux, soit au temps de simulation courant (***délai ou cycle delta***), soit à un temps futur. Si ce n'est pas le cas, ou si l'on a atteint la valeur maximum du temps représentable pour la base de temps utilisée, la simulation stoppe.

Un signal ne prend jamais sa nouvelle valeur immédiatement, mais toujours au début du cycle de simulation suivant. Cette règle permet d'obtenir les mêmes résultats de simulation quelque soit l'ordre d'exécution des processus.

Exemple de simulation à délai nul (1/2)

```
entity noteq is
    port (A, B: in bit; Z: out bit);
end entity noteq;

architecture dfl of noteq is
    signal C, D, E: bit;
begin
    P1: C <= A nand B;
    P2: D <= A nand C;
    P3: E <= C nand B;
    P4: Z <= D nand E;
end architecture dfl;
```





Pour illustrer le fonctionnement du délai delta (*delta cycle*), considérons à nouveau le circuit logique de la réalisant une fonction OU exclusif.

Chaque instruction d'affectation de signal concurrente est équivalente à un processus sensible sur les signaux référencés dans la partie de droite de l'affectation. Le modèle définit ainsi quatre processus concurrents P1 (sensible sur A et B), P2 (sensible sur A et C), P3 (sensible sur C et B) et P4 (sensible sur D et E), chacun d'eux affectant une valeur à un seul signal. On aura ainsi après élaboration six pilotes créés dénotés P_A, P_B, P1_C, P2_D, P3_E et P4_Z.

Exemple de simulation à délai nul (2/2)

T [ns]	δ	A	B	P1_C	P2_D	P3_E	P4_Z
0	0	'0' ('1' @ 10)	'0'	'0' ('1' @ δ1)	'0' ('1' @ δ1)	'0' ('1' @ δ1)	'0' ('1' @ δ1)
	1	'0'	'0'	'1'	'1'	'1'	'1' ('0' @ δ2)
	2	'0'	'0'	'1'	'1'	'1'	'0'
10	0	'1'	'0'	'1' ('1' @ δ1)	'1' ('0' @ δ1)	'1'	'0'
	1	'1'	'0'	'1'	'0'	'1'	'0' ('1' @ δ2)
	2	'1'	'0'	'1'	'0'	'1'	'1'

Initialisation:

I1. Tous les signaux ont la valeur '0' (valeur initiale par défaut).

I2. T_c = 0 ns, δ = 0 (δ est l'indice de cycle delta).

I3. L'exécution des quatres processus (dans un ordre quelconque) jusqu'à leur suspension va créer des transactions dans les pilotes P1_C, P2_D, P3_E et P4_Z. Comme les affectations de signaux sont à délai nul, les transactions sont ***prévues au cycle delta suivant*** (@ δ1).
En supposant que l'entrée primaire A soit affectée dans un modèle de test à l'expression "A <= '1' **after** 10 ns", une transaction est aussi créée dans le pilote P_A avec un délai de 10 ns (@ 10).

I4. Comme il y a des transactions prévues au temps courant, T_n = 0.

Cycles de simulation:

S1. T_c = 0 ns et δ = 1.

S2. (T_c = 0, δ = 1) Mise à jour des signaux C, D, E et Z.

S3. Exécuter les processus P2, P3 et P4.

S4. P4 crée une transaction à délai nul sur P4_Z.

S2. (T_c = 0, δ = 2) Mise à jour du signal Z.

S3. Aucun processus à réévaluer.

S4. T_n = 10 ns.

S1. T_c = 10 ns et δ = 0.

S2. (T_c = 10, δ = 0) Mise à jour du signal A.

S3. Exécuter les processus P1 et P2.

S4. Transactions sur P1_C et P2_D (délai delta).

S2. (T_c = 10, δ = 1) Mise à jour du signal D.

S3. Exécuter le processus P4.

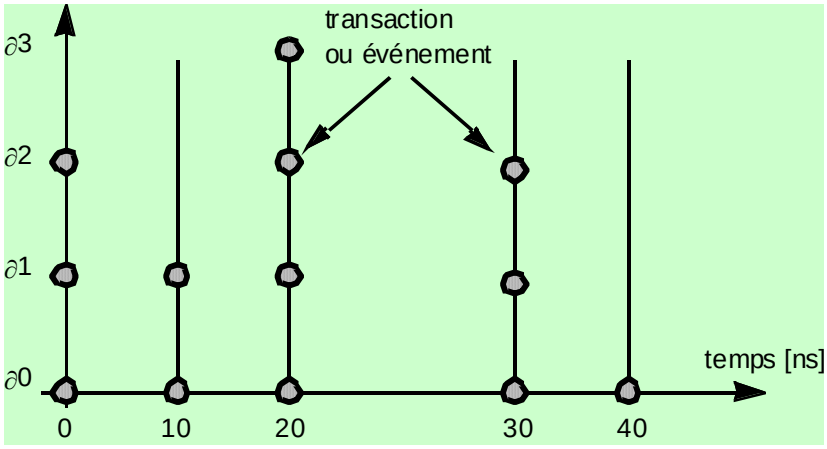
S4. Transaction sur P1_Z (délai delta).

S2. (T_c = 10, δ = 2) Mise à jour du signal Z.

S3. Aucun processus à réévaluer.

S4. T_n = ? ns (on continue…).

Relation entre temps simulé et itération delta



Les délais delta induisent des itérations sans que le temps simulé avance. Les délais delta garantissent le déterminisme: les valeurs des signaux à la fin d'un cycle d'itérations delta ne dépendent pas de l'ordre dans lequel les processus sont exécutés, et donc dans lequel les signaux sont évalués.

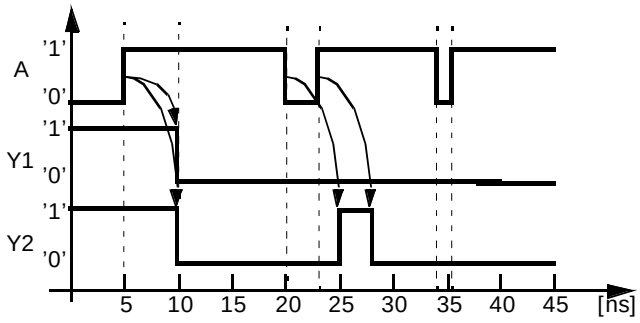
Mode de délai inertiel

- **Mode de délai par défaut**
- **Format**

nom-signal <= [[**reject** *durée-rejection*] **inertial**]
expression-valeur **after** *délai* { , … } ;

- **Exemples**

```
inv1: process (A) is
begin
    Y1 <= not A after 5 ns;  -- ou Y1 <= inertial not A after 5 ns;
    Y2 <= reject 2 ns inertial not A after 5 ns;
end process inv1;
```



Le délai inertiel modélise le fait qu'un système matériel réel ne peut pas changer d'état en un temps infiniment court. En réalité, il faut plutôt appliquer des signaux pendant un temps suffisamment long pour surmonter l'inertie inhérente dûe au mouvement des électrons. Le délai inertiel est le mécanisme par défaut en VHDL.

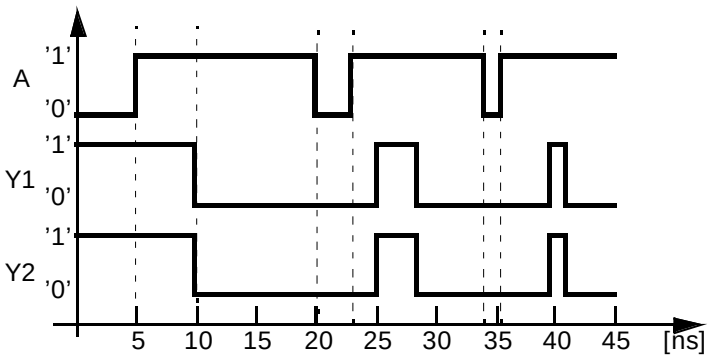
La durée du temps de rejection doit être positive et plus petite ou égale au délai inertiel. La spécification du temps de rejection permet de clairement distinguer entre le délai inertiel et les délais d'affectation d'une forme d'onde (ce qui n'est pas le cas en VHDL-87 puisque le délai de la première clause **after** sert aussi à définir le temps de rejection).

Mode de délai transport

- **Format** *nom-signal* <= **transport** *expression-valeur* **after** *délai* { , … } ;

- **Exemples**

```
inv2: process (A) is
begin
    Y1 <= transport not A after 5 ns;
    Y2 <= reject 0 ns inertial not A after 5 ns;
end process inv2;
```



Le délai transport modélise un comportement idéal pour lequel toute impulsion, quelle que soit sa durée, est transmise.

Le mode de délai transport est équivalent au mode de délai inertiel avec un temps de rejection nul.

Dans le cas d'une affectation de signal générant une forme d'onde (p.ex. p.49), le mode de délai ne s'applique qu'au premier élément de la forme d'onde. Tous les autres éléments de la forme d'onde sont considérés en mode transport.

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 64

Déclaration de composant

- **Syntaxe**

```
component nom-composant [ is ]
    [ generic ( liste-paramètres ) ; ]
    [ port ( liste-ports ) ; ]
end component [ nom-composant ] ;
```

Exemple

```
component inv is
    generic (TD: delay_length := 0 ns);
    port (signal iin: in bit; signal iout: out bit);
end component inv;
```

- **Similaire à une déclaration d'entité, mais:**
  - Un composant définit ce dont on a besoin (demande)
  - Une entité définit ce qui est disponible (offre)
- **Le nom d'un composant, de ses paramètres et de ses ports ne doivent pas nécessairement être identiques à ceux d'une entité de conception particulière**
- **Une configuration permet d'associer une instance de composant à une entité de conception**

A la différence du mécanisme d'instanciation directe (voir p. 23), la ***déclaration de composant*** (*component declaration*) permet de disposer d'une interface entre ce dont le modèle à besoin et ce qui est disponible dans une bibliothèque de conception. Ceci permet de définir les modèles effectifs des composants en-dehors de l'architecture et donc d'offrir plus de souplesse dans les cas où plusieurs architectures de composants sont possibles.

Une déclaration de composant peut apparaître dans la partie déclarative d'une architecture ou dans une déclration de paquetage. Cette dernière forme sera présentée plus loin.

Les noms donnés au composant, à ses paramètres et à ses ports sont purement locaux à l'architecture dans laquelle le composant est instancié. C'est le rôle d'une ***configuration*** d'associer une instance de composant à une entité de conception dans une bibliothèque.

Instance de composant

- **Syntaxe**

étiquette : [**component**] *nom-composant*
[**generic map** (*liste-association-paramètres*)]
[**port map** (*liste-association-ports*)] ;

- **Exemple**

```
entity buffer is
    port (bin: in bit; bout: out bit);
end entity buffer;

architecture str of buffer is
    signal sint: bit;

    component inv is
        generic (TD: delay_length := 0 ns);
        port (i: in bit; o: out bit);
    end component inv;

begin
    inv1: component inv
            generic map (TD => 2.5 ns)
            port map (i => bin, o => sint);

    inv2: component inv
            generic map (TD => 3 ns)
            port map (i => sint, o => bout);
end architecture str;
```

associations locales

Une ***instance de composant*** (*component instantiation statement*) est une instruction concurrente. Elle définit une instance uniquement identifiable d'un composant (virtuel). L'unicité de l'instance est définie par son étiquette obligatoire (p.ex. inv1, inv2).

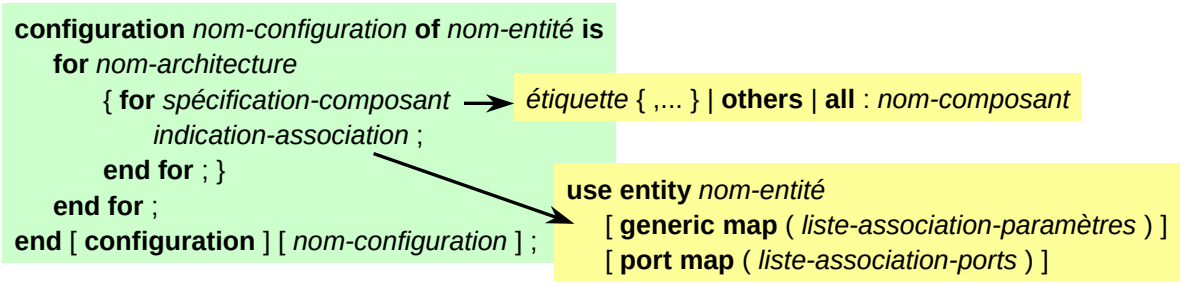
Les associations de paramètres (*generic map*) et de ports (*port map*) définissent des ***associations locales*** entre les noms utilisés dans la déclaration de composant (paramètres et ports formels) et les paramètres et ports effectifs de l'instance.

Les paramètres et les ports de mode **in** peuvent être associés à des valeurs litérales. P. ex.: 2.5 ns pour TD.

Le modèle du buffer n'est pas forcément simulable dans la mesure où les instances de composants ne sont pas encore associées à des entités de conceptions. Ces associations doivent être effectuées par une ***configuration***.

Déclaration de configuration

- **Unité de conception**
- **Syntaxe**



- **Configuration globale à une entité de conception identifiée par *nom-entité*(*nom-architecture*)**
- **Configuration hiérarchique possible**

Une ***déclaration de configuration*** (*configuration declaration*) est une unité de conception. Elle est donc compilable séparément.

Elle a pour but d'associer les instances de composants d'une entité de conception identifiée par *nom-entité*(*nom-architecture*) à des entités de conceptions disponibles dans une bibliothèque. Les associations peuvent être définies pour un nombre quelconque de niveaux hiérarchiques.

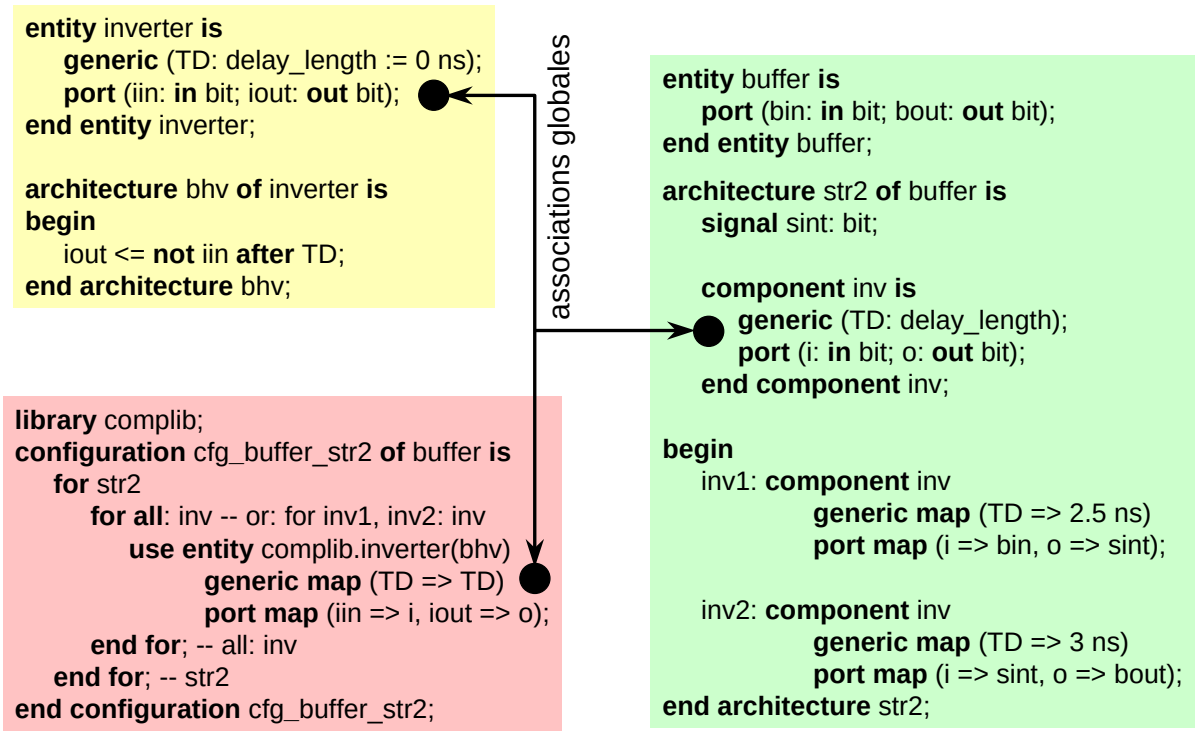
La syntaxe d'une déclaration de configuration est relativement complexe car elle peut prendre en compte tous les cas d'associations, parmi lesquels on peut citer:

- L'entité de conception à associer dispose de plusieurs architectures possibles (dans une même bibliothèque ou dans des bibliothèques différentes).
- Les noms du composant déclaré, de ses paramètres ou de ses ports sont différents de ceux de l'entité de conception à lui associer.
- La déclaration de composant déclare moins de paramètres ou de ports que la déclaration d'entité à lui associer.
- Les instances d'un composants nécessitent des associations de paramètres ou de ports différentes.
- L'architecture à configurer est hiérarchique.

La déclaration de configuration définit la vue globale pour la simulation. Son chargement dans le simulateur va forcer le chargement automatique de toutes les entités de conceptions associées aux instances de composants.

Une indication d'association peut associer des éléments de noms différents, mais il est nécessaire que leurs types soient compatibles.

Exemple de déclaration de configuration



A. Vachoux, v2.0, 2003 Le langage VHDL - 68

Configuration par défaut

- **Déclaration de composant et déclaration d'entité ont même signature**
  - Nom du composant et de l'entité identiques
  - Même liste de paramètres génériques, mêmes noms et types
  - Même liste de ports, mêmes noms, mêmes types, mêmes directions
- **Association implicite sans configuration**
- **Si plusieurs architectures existent pour une même entité, l'architecture analysée le plus récemment est considérée**

Exemple de configuration par défaut

```
entity inverter is
    generic (TD: delay_length := 0 ns);
    port (iin: in bit; iout: out bit);
end entity inverter;

architecture bhv of inverter is
begin
    iout <= not iin after TD;
end architecture bhv;
```

```
entity buffer is
    port (bin: in bit; bout: out bit);
end entity buffer;

architecture str2 of buffer is
    signal sint: bit;

    component inverter is
        generic (TD: delay_length);
        port (iin: in bit; iout: out bit);
    end component inverter;

begin
    inv1: component inverter
            generic map (TD => 2.5 ns)
            port map (iin => bin, iout => sint);

    inv2: component inverter
            generic map (TD => 3 ns)
            port map (iin => sint, iout => bout);
end architecture str2;
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 70

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 71

Paramètres génériques (1/4)

- **Permettent d'écrire des modèles généraux**
- **Spécifiés dans l'entité par une clause generic**
- **Exemple:**
  additionneur générique N bits

```
entity addn is
    generic (
        TP: time := 0 ns;    -- temps de propagation
        NB: natural := 8     -- nb de bits des opérandes
    );
    port (
        opa, opb: in   bit_vector(NB-1 downto 0);
        cin       : in   bit;
        sum       : out bit_vector(NB-1 downto 0);
        cout      : out bit
    );
end entity addn;
```

```
architecture dfl of addn is
begin
  process (cin, opa, opb)
    variable ccin, ccout: bit;
    variable result: bit_vector(sum'range);
  begin
    ccout := cin;
    for i in sum'reverse_range loop
      ccin := ccout;
      result(i) := opa(i) xor opb(i) xor ccin;
      ccout := (opa(i) and opb(i))
                 or (ccin and (opa(i) or opb(i)));
    end loop;
    sum <= result after TP;
    cout <= ccout after TP;
  end process;
end architecture dfl;
```


Un paramètre générique définit une valeur constante qui peut être différente d'une instance de composant à l'autre. La valeur effective d'un paramètre générique peut aussi être définie dans une configuration.

La déclaration d'un paramètre générique se fait dans une déclaration d'entité au moyen d'une clause **generic**.

Les usages typiques de paramètres génériques sont:

- La spécification de paramètres temporels (délais, contraintes).
- La spécification de tailles de bus.

Le modèle de l'additionneur générique N bits est paramétré sur un temps de propagation aux signaux de sortie et sur la taille des opérandes. Il réalise l'addition bit à bit avec propagation de retenue.

Paramètres génériques (2/4)

- **Exemple: modèle de test d'un additionneur 32 bits**

```
entity tb_add32 is
end entity tb_add32;

architecture bench of tb_add32 is

    signal opa, opb, sum: bit_vector(31 downto 0);
    signal cin, cout: bit;

begin

    UUT: entity work.addn(dfl)
                generic map (       -- association par nom
                    TP => 2 ns,
                    NB => opa'length)
                port map (opa, opb, cin, sum, cout);

    -- stimulis…

end architecture bench;
```


La spécification des valeurs effectives de paramètres génériques se fait au niveau des instances de composants dans une architecture au moyen d'une clause **generic map**.

Comme pour l'association de ports, il est possible d'utiliser une association par nom ou par position. Si le paramètre a une valeur par défaut, il peut être soit omis de la clause **generic map** ou associé au mot réservé **open**. P. ex., dans le cas d'un additionneur 8 bits:

```
generic map (TP => 2 ns);
```

ou

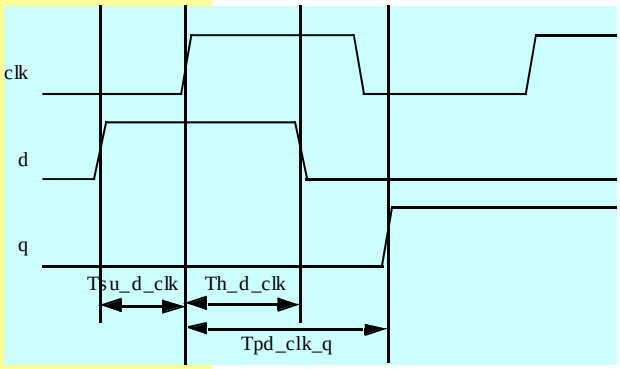
```
generic map (TP => 2 ns, NB => open);
```

Paramètres génériques (3/4)

- **Exemple: flip-flop D avec vérification de contraintes temporelles**

```
entity dff is
    generic (
        Tpd_clk_q,   -- temps de propagation
        Tsu_d_clk,   -- temps de pré-positionnement
        Th_d_clk: delay_length); -- temps de maintien
    port (clk, d: in bit; q: out bit);
end entity dff;
```

```
architecture bhv of dff is
begin
    behavior: q <= d after Tpd_clk_q when clk = '1' and clk'event;

    check_setup: process is
    begin
        wait until clk = '1';
        assert d'last_event >= Tsu_d_clk
            report "Setup violation";
    end process check_setup;

    check_hold: process is
    begin
        wait until clk'delayed(Th_d_clk) = '1';
        assert d'delayed'last_event >= Th_d_clk
            report "Hold violation";
    end process check_hold;
end architecture bhv;
```



Un exemple typique d'utilisation de paramètres génériques est la spécification de paramètres temporels permettant de caractériser le même comportement dans des conditions différentes. On considère ici le modèle d'un flip-flop D avec ses caractéristiques temporelles: temps de propagation, temps de pré-positionnement (*setup time*) et temps de maintien (*hold time*).

Dans le modèle, les caractéristiques temporelles (temps de propagation Tpd_clk_q, temps de pré-positionnement Tsu_d_clk et temps de maintien Th_d_clk sont définis comme des paramètres génériques. Le corps d'architecture est composé de trois processus concurrents. Le premier, nommé behavior, définit le comportement du composant en tenant compte du temps de propagation. Le deuxième, nommé check_setup, vérifie le temps de pré-positionnement. Le troisième, nommé check_hold, vérifie le temps de maintien.

Noter l'usage des attributs prédéfinis sur les signaux 'last_event et 'delayed. Ce dernier attribut permet de prendre en compte correctement les cas où les signaux clk et d changent de valeurs simultanément. S'delayed a la même valeur que S, mais retardé d'une itération delta.

Paramètres génériques (4/4)

- **Exemple: flip-flop D avec vérification de contraintes temporelles et processus passifs**

```
entity dff is
    generic (
        Tpd_clk_q,   -- temps de propagation
        Tsu_d_clk,   -- temps de pré-positionnement
        Th_d_clk: delay_length); -- temps de maintien
    port (clk, d: in bit; q: out bit);
begin
    check_setup: process is
    begin
        wait until clk = '1';
        assert d'last_event >= Tsu_d_clk
            report "Setup violation";
    end process check_setup;

    check_hold: process is
    begin
        wait until clk'delayed(Th_d_clk) = '1';
        assert d'delayed'last_event >= Th_d_clk
            report "Hold violation";
    end process check_hold;
end entity dff;
```

```
architecture bhv of dff is
begin
    behavior: q <= d after Tpd_clk_q when clk = '1' and clk'event;
end architecture bhv;
```


Comme les vérifications s'appliquent en fait à toute architecture relative à l'entité dff, il est possible de déplacer les instructions VHDL dans l'entité elle-même. VHDL autorise des instructions dans la déclaration d'entité pour autant qu'elles soient ***passives***: elles ne doivent pas modifier l'état du modèle et peuvent seulement lire l'état courant.

Instruction generate (1/3)

- **Instruction concurrente permettant d'encapsuler des instructions concurrentes telles que**
  - instances de composants
  - processus
  - affectation concurrente de signal
  - d'autres instructions **generate**
- **Equivalentes à des macro-instructions qui sont exécutées une seule fois avant de commencer la simulation**
- **Deux formes d'instructions generate**

étiquette :
for *identificateur* **in** *intervalle* **generate**
    [{ *déclaration-locale* }
begin]
    { *instruction-concurrente* }
end generate [*étiquette*] ;

forme itérative

étiquette :
if *expression-booléenne* **generate**
    [{ *déclaration-locale* }
begin]
    { *instruction-concurrente* }
end generate [*étiquette*] ;

forme conditionnelle

L'instruction **generate** est une instruction concurrente qui offre des fonctionnalités de type macro pour la description de comportements ou de structures régulières.

Elle peut encapsuler des déclarations locales et un nombre quelconque d'instructions concurrentes. Elle a pour effet de dupliquer ces déclarations et ces instructions, soit un nombre fixe de fois (forme itérative), soit selon une condition (forme conditionnelle). La dupliquation a lieu une fois avant de démarrer la simulation.

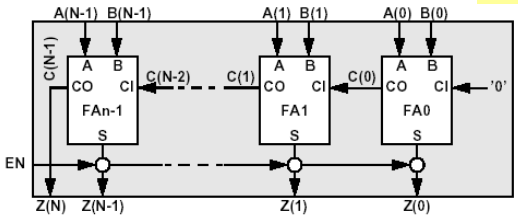
La forme itérative est utile pour décrire des comportements ou des structures régulières dont le nombre est élevé ou non spécifié a priori. Elle utilise une syntaxe similaire à celle de la boucle **for**. L'identificateur n'a pas besoin d'être déclaré et ne peut prendre que des valeurs discrètes dans un intervalle donné. La valeur de l'identificateur peut être lue dans le corps de l'instruction **generate**, mais il ne peut pas être modifié.

La forme conditionnelle est utile pour prendre en compte des cas limites (p.ex. les instances de composants communiquant directement avec les ports de l'architecture). Si l'expression booléenne ne s'évalue pas à la valeur true, les déclarations locales et les instructions encapsulées seront physiquement éliminées du code simulé.

Une instruction **generate** peut encapsuler d'autres instructions **generate**.

Instruction generate (2/3)

- **Exemple: additionneur N bits basé sur des cellules 1 bit**




```
entity addn is
    generic (wsize: positive := 8);
    port (
        en  : in   bit;
        a, b: in   bit_vector(wsize-1 downto 0);
        z   : out bit_vector(wsize downto 0));
end entity addn;
```

```
architecture str of addn is
    signal c: bit_vector(wsize-1 downto 0);
begin
    STAGES: for i in wsize-1 downto 0 generate
        signal s_unbuffered: bit;
    begin
        LSB: if i = 0 generate
            FA1: entity work.add1(dfl)
                port map (opa => a(0), opb => b(0), cin => '0',
                          sum => s_unbuffered, cout => c(0));
        end generate LSB;
        OTHERB: if i /= 0 generate
            FAi: entity work.add1(dfl)
                port map (opa => a(i), opb => b(i), cin => c(i-1),
                          sum => s_unbuffered, cout => c(i));
        end generate OTHERB;
        OUT_STAGE: process (en)
        begin
            if en = '1' then
                z(i) <= s_unbuffered;
            end if;
        end process OUT_STAGE;
    end generate STAGES;
    z(wsize) <= c(wsize-1);
end architecture str;
```


Cet exemple illustre plusieurs aspects de l'instruction **generate**:

- Les deux formes itérative et conditionnelle.
- L'usage d'une déclaration locale (le signal s_unbuffered).
- L'encapsulation d'instances de composants et d'un processus.

Instruction generate (3/3)

- **Déclaration de configuration (syntaxe étendue)**

```
configuration nom-configuration of nom-entité is
    for nom-architecture
        | étiquette-instruction-generate
        [ ( intervalle | expression-statique ) ]
        { for spécification-composant
            indication-association ;
        end for ; }
    end for ;
end [ configuration ] [nom-configuration ] ;
```

- **Exemple: additionneur N bits**

```
configuration cfg_addn_str of addn is
    for str
        for STAGES (wsize - 2 downto 0)
            for LSB
                ... configuration du composant
            end for; -- LSB
            for OTHERB
                ... configuration des composants
            end for; -- OTHERB
        end for; -- STAGES (wsize - 2 downto 0)
        for STAGES (wsize - 1)
            for OTHERB
                ... configuration des composants
            end for; -- OTHERB
        end for; -- STAGES (wsize - 1)
    end for; -- str
end configuration cfg_addn_str;
```


Une forme étendue de la déclaration de configuration est requise pour une modèle structurel utilisant des instructions **generate**.

Prenons l'exemple du modèle générique de l'additionneur N bits. Les instances 0 à N-2 de l'additionneur 1 bit doivent être associées à la même entité de conception et la dernière instance N-1 doit être associée à une autre entité de conception.

Une première partie définit la configuration des instances 0 à N-2. Une clause **for** inclut une spécification d'intervalle et deux sous-clauses permettent de traiter séparément la cellule LSB et les autres cellules. Une seconde partie définit la configuration de l'instance N-1 en ne spécifiant que l'étiquette OTHERB puisque la condition liée à l'étiquette LSB est fausse dans ce cas.

Sous-programmes

- **Encapsulent des instructions séquentielles**
- **Peuvent être déclarés dans un paquetage**
- **Procédures**
  - Domaines de déclaration et d'exécution:
    Processus ou sous-programme (forme séquentielle)
    Architecture (forme concurrente)
- **Fonctions**
  - Utilisées dans des expressions
  - Domaines de déclaration et d'exécution: processus ou sous-programme

Les sous-programmes permettent de modulariser la description du comportement d'un modèle. Ils favorisent aussi la réutilisation entre modèles.

Il existe deux sortes de sous-programmes:

- Les ***procédures*** encapsulent des instructions séquentielles. Elles peuvent prendre deux formes, en fonction de l'endroit où elles sont exécutées. La forme séquentielle est la forme usuelle utilisée dans les langages de programmation. La forme concurrente est une notation compacte équivalente à un processus exécutant la procédure dans son corps.
- Les ***fonctions*** encapsulent aussi des instructions séquentielles. Elles retournent un résultat et ne peuvent donc être utilisées que dans des expressions.

Un sous-programme peut être défini soit dans le modèle, près de l'endroit où il est exécuté, soit dans une unité de conception séparée appelée paquetage. Les paquetages seront présentés plus loin.

Le langage VHDL définit un ensemble de procédures et de fonctions standard. Par exemple, les opérateurs prédéfinis "+", "*", "**and**", "**or**", etc. sont définis comme des fonctions.

Procédures (1/4)

- **Syntaxe: déclaration de procédure**

procedure *nom-procedure* [(*liste-paramètres-formels*)] **is**
    { *déclaration-locale* }
begin
    { *instruction-séquentielle* }
end [**procedure**] [*nom-procédure*] ;

déclaration-locale : type, sous-type, constante, variable, déclaration de sous-programme

- **Classes de paramètres formels:**

[**constant**] *identificateur* { ,... } : [**in**] *sous-type* ;
[**variable**] *identificateur* { ,... } : [**in** | **out** | **inout**] *sous-type* ;
[**signal**] *identificateur* { ,... } : [**in** | **out** | **inout**] *sous-type* ;
[**file**] *identificateur* { ,... } : *sous-type* ;

- **Syntaxe: appel de procédure**

[*étiquette* :] *nom-procédure* [(*liste-paramètres-actuels*)] ;

- **Associations paramètre formel - paramètre actuel par nom ou par position**

- **Les déclarations locales sont (re)créées à chaque début d'exécution et détruites en fin d'exécution de la procédure**

Les paramètres formels d'une procédure peuvent être de ***classe*** *constante*, *variable*, *signal* ou *fichier*. Les trois premières classes de paramètres possèdent un ***mode*** (optionel) définissant la manière dont l'objet est accédé dans la procédure:

- Un paramètre de mode **in** ne peut qu'être lu. C'est le mode par défaut et la classe par défaut du paramètre est dans ce cas *constante*.
- Un paramètre de mode **out** ne peut qu'affecté. La classe par défaut du paramètre est dans ce cas *variable*.
- Le mode **inout** est une combinaison des deux modes précédants. La classe par défaut du paramètre est dans ce cas *variable*.

Les paramètres de classe *fichier* n'ont pas de mode. Le mode d'accès d'un fichier est défini dans sa déclaration.

L'appel de procédure a pour effet d'associer des paramètres effectifs aux paramètres formels de sa déclaration, de (re)créer les objets declarés localement et d'exécuter les instructions séquentielles dans l'ordre où elles sont spécifiées.

L'association des paramètres peut se faire par nom ou par position et doit respecter les règles suivantes:

- Pour un paramètre formel de classe *constante*, le paramètre effectif peut être une constante (litéral ou expression), une variable ou un signal.
- Pour un paramètre formel de classe *variable*, *signal* ou *fichier*, le paramètre effectif ne peut être qu'un objet de la même classe.

Pour un paramètre formel de classe *signal* et de mode **in** ou **inout**, l'objet effectif utilisé dans la procédure est une ***copie*** de l'objet passé en paramètre. Si une instruction **wait** est utilisée sur ce signal (local), la valeur du signal (local) peut changer avant que la procédure ne se termine. ceci peut être contradictoire avec l'intention initiale d'avoir un signal en lecture seule.

Il est important de noter que les déclarations locales sont détruites à la terminaison de la procédure et réélaborées à la prochaine exécution. Une procédure ne peut pas conserver un état (un processus le peut).

Procédures (2/4)

- **Exemple:**
  procédure sans paramètres et appel séquentiel

```
process (…)
    procedure report_max_and_sum is
        variable max: real := real'low;
        variable sum: real := 0.0;
    begin
        for i in samples'range loop
            sum := sum + samples(i);
            if samples(i) > max then
                max := samples(i);
            end if;
        end loop;
        report "Maximum value is: " & real'image(max);
        report "Total value is : " & real'image(sum);
    end procedure report_max_and_sum;
    type real_vector is array (natural range <>) of real;
    variable samples: real_vector(1 to 16);
begin
    … -- samples filled with values
    report_max_and_sum;
end process;
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 81

Procédures (3/4)

- **Exemple: procédure avec paramètres et appel concurrent**

```
entity dlatch is
    generic (Tsu_en_clk, Tsu_din_clk: time := 0 ns);
    port (signal clock, enable, din: in bit; signal dout: out bit);
end entity dlatch;

architecture dfl of dlatch is
    procedure check_setup (signal clk, data: in bit; constant Tsetup: in time);
    begin
        if clk = '1' then
            assert data'last_event >= Tsetup
                report "Setup time violation" severity error;
        end if;
    end procedure check_setup;
begin
    P1: check_setup(clk => clock, data => enable, Tsetup => Tsu_en_clk);
    P2: check_setup(clock, din, Tsu_din_clk);
    P3: dout <= din when enable = '1' and clock = '1';
end architecture dfl;
```

```
P1: process (clk, enable) is
begin
    check_setup(clock, enable, Tsu_en_clk);
end process P1;
```

```
P2: process (clk, din) is
begin
    check_setup(clock, din, Tsu_din_clk);
end process P2;
```


L'appel concurrent de procédure check_setup est équivalent à un processus sensible sur les paramètres de la procédure de classe *signal* et de mode **in** et dont le corps se résume à l'appel de la procédure.

Une procédure concurrente ne crée par de hiérarchie structurelle.

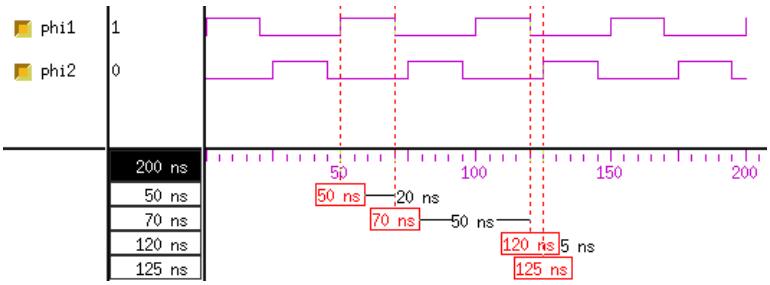
Procédures (4/4)

- **Exemple:**
  procédure concurrente pour génération d'horloge à phases asymétriques

```
procedure clkgen (
    signal clk: out bit;
    constant Tperiod, Tpulse, Tphase: in time) is
begin
    wait for Tphase;
    loop
        clk <= '1', '0' after Tpulse;
        wait for Tperiod;
    end loop;
end procedure clkgen;
```

- **Génération d'horloges non recouvrantes**

```
signal phi1, phi2: bit := '0';
…
gen_phi1: clkgen(phi1,
                 Tperiod => 50 ns,
                 Tpulse => 20 ns,
                 Tphase => 0 ns);
gen_phi2: clkgen(phi2,
                 Tperiod => 50 ns,
                 Tpulse => 20 ns,
                 Tphase => 25 ns);
```



A. Vachoux, v2.0, 2003 Le langage VHDL - 83

Fonctions (1/2)

- **Syntaxe: déclaration de fonction**

function *nom-fonction* [(*liste-paramètres-formels*)] **return** *sous-type* **is**
  { *déclaration-locale* }
begin
  { *instruction-séquentielle* }
end [**function**] [*nom-fonction*] ;

type, sous-type, constante, variable, déclaration de sous-programme

- **Classes de paramètres formels:**

[**constant**] *identificateur* { ,... } : [**in**] *sous-type* ;
[**signal**] *identificateur* { ,... } : [**in**] *sous-type* ;
[**file**] *identificateur* { ,... } : *sous-type* ;

- **Le corps de fonction doit contenir au moins une instruction return**

[*étiquette* :] **return** *expression* ;

- **Syntaxe: appel de fonction**

nom-fonction [(*liste-paramètres-actuels*)] ;

- **Associations paramètre formel - paramètre actuel par nom ou par position**

- **Les déclarations locales sont (re)créées à chaque début d'exécution et détruites en fin d'exécution de la fonction**

Les paramètres formels d'une fonction peuvent être de ***classe*** *constante*, *signal* ou *fichier*. Les deux premières classes de paramètres ne peuvent qu'être de ***mode*** **in**.

Les paramètres de classe *fichier* n'ont pas de mode. Le mode d'accès d'un fichier est défini dans sa déclaration.

Le corps d'une fonction peut inclure des instructions séquentielles avec les restrictions suivantes:

- Il doit inclure au moins une instruction **return**. Cette instruction a pour effet de retourner une valeur associée au nom de la fonction dans l'expression qui appelle cette fonction.
- L'instruction **wait** n'est pas admise.

L'association des paramètres peut se faire par nom ou par position et doit respecter les règles suivantes:

- Pour un paramètre formel de classe *constante*, le paramètre effectif peut être une constante (litéral ou expression), une variable ou un signal.
- Pour un paramètre formel de classe *variable*, *signal* ou *fichier*, le paramètre effectif ne peut être qu'un objet de la même classe.

Il est important de noter que les déclarations locales sont détruites à la terminaison de la fonction et réélaborées à la prochaine exécution. Une fonction ne peut pas conserver un état.

Fonctions (2/2)

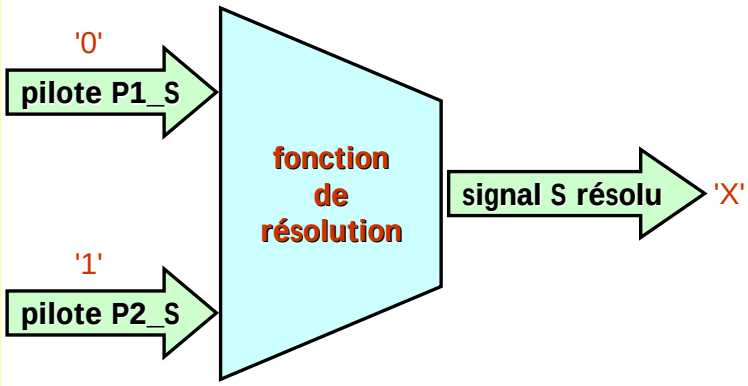
- **Exemple: conversion bit_vector -> entier**

```
architecture A of E is

    function bv_to_nat (bv: in bit_vector) return natural is
        variable result: natural := 0;
    begin
        for i in bv'range loop -- hyp.: MSB = bv'left
            result := result*2 + bit'pos(bv(i));
        end loop;
        return result;
    end function bv_to_nat;

    signal data_bv1, data_bv2: bit_vector(15 downto 0);
    signal data_nat1, data_nat2: natural;
begin
    …
    data_nat1 <= bv_to_nat(data_bv1);
    data_nat2 <= bv_to_nat(bv => data_bv2);
    …
end architecture A;
```


A. Vachoux, v2.0, 2003 Le langage VHDL - 85

Fonction de résolution (1/2)

```
architecture A of E is
    …
    signal S: logic4;
    …
begin
    P1: process begin
        wait for 10 ns;
        S <= '1';
        wait for 20 ns;
        S <= '0';
    end process P1;
    P2: process begin
        wait for 20 ns;
        S <= '0';
        wait for 20 ns;
        S <= '1';
    end process P2;
    …
end architecture A;
```



L'affectation multiple de signaux à partir de plusieurs instructions concurrentes (processus, composants) nécessite un mécanisme spécial appelé ***résolution***. Cette situation arrive notamment lorsque l'on modélise un bus sur lequel plusieurs unités peuvent potentiellement écrire une valeur ou à partir duquel elles peuvent lire une valeur. Normalement une seule source est active à un moment donné et les autres sont en état haute impédance. Une autre possibilité est de modéliser une logique cablée ET ou OU (*wired-AND*, *wired-OR*).

Fonction de résolution (2/2)

- **Types non résolus**

```
type ulogic4 is ('X', '0', '1', 'Z'); -- type non résolu
type ulogic4_vector is array (natural range <>) of ulogic4;
```

- **Fonction de résolution**

```
function resolve (sources: ulogic4_vector) return ulogic4 is
    type table is array (ulogic4, ulogic4) of ulogic4;
    constant resolution_table: table :=  -- 'X'  '0'  '1'  'Z'
                                          (( 'X', 'X', 'X', 'X' ), -- 'X'
                                           ( 'X', '0', 'X', '0' ),  -- '0'
                                           ( 'X', 'X', '1', '1' ),  -- '1'
                                           ( 'X', '0', '1', 'Z' )); -- 'Z'
    variable result: ulogic4 := 'Z';
begin
    for i in sources'range loop
        result := resolution_table(result,sources(i));
    end loop;
    return result;
end function resolve;
```

- **Types résolus**

```
subtype logic4 is resolve ulogic4;
type logic4_vector is array (natural range <>) of logic4;
```


Le type énuméré ulogic4 définit quatre états logiques: 'X' (état indéfini), '0', '1' et 'Z' (état haute-impédance). Ce type et le type ulogic4_vector sont dits ***non résolus*** car des signaux de ces types ne peuvent pas avoir plus d'une source (ceci est détecté à la compilation ou à l'élaboration).

La déclaration du sous-type ***résolu*** logic4 spécifie le nom d'une fonction de résolution, resolve, qui accepte pour paramètre un tableau non contraint de valeurs en provenance des multiples sources d'un signal. La fonction retourne une valeur unique calculée à partir d'une table de résolution qui réalise les règles suivantes:

- Si l'une des sources vaut 'X', ou si deux sources sont en conflit avec l'une valant '0' et l'autre valant '1', le signal résolu vaut 'X'.
- Si une ou plusieurs sources valent '0' et le reste vaut 'Z', le signal résolu vaut '0'.
- Si une ou plusieurs sources valent '1' et le reste vaut 'Z', le signal résolu vaut '1'.
- Si toutes les sources valent 'Z', le signal résolu vaut 'Z'.

Il est aisé de définir une fonction de résolution qui réalise un ET câblé ou un OU câblé. La fonction de résolution ne peut posséder qu'un seul argument (un tableau non contraint mono-dimensionnel). D'autre part le résultat résolu ne doit pas dépendre de l'ordre dans lequel on traite les sources: la fonction de résolution doit être commutative.

La fonction de résolution est automatiquement exécutée, même si le signal ne possède qu'une seule source.

Le type non résolu ulogic4 et le sous-type résolu logic4 sont compatibles, c'est-à-dire qu'il est légal d'affecter des signaux de ces deux types entre eux. Par contre, les types tableaux ulogic4_vector et logic4_vector sont deux types différents et ne sont pas compatibles. Il s'agit d'utiliser une conversion de type si l'on veut affecter un type de signal à l'autre:

```
signal S1: ulogic4_vector;
signal S2: logic4_vector;

S2 <= logic4_vector(S1);
S1 <= ulogic4_vector(S2);
```

Paquetages

- **Un paquetage permet de grouper des déclarations et des sous-programmes et de les stocker dans une bibliothèque**
- **Deux unités de conception**

package *nom-paquetage* **is**
{ *déclaration* [| *corps-sous-programme*] }
end [**package**] [*nom-paquetage*] ;

en-tête de paquetage

package body *nom-paquetage* **is**
{ *déclaration* [| *corps-sous-programme*] }
end [**package body**] [*nom-paquetage*] ;

corps de paquetage

- **Un corps de paquetage n'est pas toujours nécessaire**
  - Exemple: en-tête de paquetage ne contenant que des déclarations de constantes et de (sous-)types

A. Vachoux, v2.0, 2003 Le langage VHDL - 88

Paquetage STANDARD

```
package STANDARD is
    type boolean is (false, true);
    type bit is ('0', '1');
    type character is ( ... 256 caractères 8 bits... );
    type severity_level is (note, warning, error, failure);
    type integer is range depend de l'implantation;
    subtype natural is integer range 0 to integer'high;
    subtype positive is integer range 1 to integer'high;
    type real is range depend de l'implantation;
    type time is range depend de l'implantation
        units
            fs; ps = 1000 fs; ...; hr = 60 min;
        end units;
    subtype delay_length is time range 0 to time'high;
    impure function now return delay_length;
    type string is array (positive range <>) of character;
    type bit_vector is array (natural range <>) of bit;
    type file_open_kind is (read_mode, write_mode, append_mode);
    type file_open_status is (open_ok, status_error, name_error, mode_error);
    attribute foreign: string;
end package STANDARD;
```

- **Dans bibliothèque STD**

Le paquetage STANDARD déclare les types et sous-types prédéfinis du langage, la fonction now (qui retourne le temps simulé courant) et l'attribut foreign.

Les déclarations du paquetage STANDARD sont automatiquement visibles dans toute unité de conception.

Paquetage TEXTIO

```
package TEXTIO is
    type line is access string;
    type text is file of string;
    type side is (right, left);
    subtype width is natural;
    file input: text open read_mode is "std_input";
    file output: text open write_mode is "std_output";
    procedure readline (file f: text; l: out line);
    procedure writeline (file f: text; l: inout line);
    -- pour chaque valeur de type prédéfini:
    procedure read (l: inout line; value: out bit; good: out boolean);
    procedure read (l: inout line; value: out bit);
    ...
    procedure write (l: inout line; value: in bit; justified: in side := right; field: in width := 0);
    ...
    procedure write (l: inout line; value: in real; justified: in side := right;
                     field: in width := 0; digits: in natural := 0);
    procedure write (l: inout line; value: in time; justified: in side := right;
                     field: in width := 0; unit: in time := ns);
end package TEXTIO;
```

- **Dans bibliothèque STD**
- **Requiert la clause de contexte**

  use STD.TEXTIO.**all**;

Le paquetage prédéfini TEXTIO déclare des types et des sous-programmes pour la manipulation de fichiers textes. Il déclare aussi des procédures pour lire et écrire les valeurs d'objets de types prédéfinis.

La lecture d'un fichier texte est effectuée en deux étapes: 1) lire une ligne complète avec la procédure readline et stocker la ligne dans un tampon de type line, 2) lire chaque élément de la ligne lue avec l'une des version de la procédure read relative au type de valeur à lire. La procédure read peut retourner l'état de l'opération par le paramètre good.

L'écriture dans un fichier est aussi effectuée en deux étapes: 1) une ligne complète est construite dans un tampon de type line au moyen d'une des versions de la procédure write, en fonction des types de valeurs à écrire, 2) la ligne est écrite dans le fichier avec la procédure writeline.

Notes:

- Si L est le tampon stockant une ligne lue dans un fichier, chaque opération read sur L consomme un élément et a pour effet que la valeur retournée par L'length décroît. La fin de la ligne est ainsi atteinte lorsque L'length = 0.
- La fonction endfile qui teste la fin de fichier est implicitement déclarée lors de la déclaration du type fichier text.

L'utilisation du paquetage TEXTIO dans une unité de conception requiert la spécification d'une clause de contexte.

Paquetage STD_LOGIC_1164 (1/2)

- **Déclaration de paquetage**

```
package STD_LOGIC_1164 is
    type std_ulogic is ('U',    -- un-initialised
                        'X',    -- forcing unknown
                        '0',    -- forcing 0
                        '1',    -- forcing 1
                        'Z',    -- high impedance
                        'W',    -- weak unknown
                        'L',    -- weak 0
                        'H',    -- weak 1
                        '-')    -- don't care);
    type std_ulogic_vector is array (natural range <>) of std_ulogic;
    function resolved (s: std_ulogic_vector) return std_ulogic;
    subtype std_logic is resolved std_ulogic;
    type std_logic_vector is array (natural range <>) of std_logic;
    -- opérateurs logiques surchargés and, nand, or, nor, xor, xnor, not
    -- fonctions de conversion: to_bit, to_bitvector, to_stdulogic, to_stdlogicvector, to_stdulogicvector
    -- plus autres fonctions...
end package STD_LOGIC_1164;
```

- **Dans bibliothèque IEEE**
- **Requiert la clause de contexte**

```
library IEEE;
use IEEE.std_logic_1164.all;
```


Le paquetage STD_LOGIC_1164 ne fait pas partie de la définition du langage VHDL, mais est défini comme un standard séparé.

Le paquetage déclare deux types logiques à neuf états std_ulogic (non résolu) et std_logic (résolu):

- Les états '0', '1' et 'X' représentent respectivement l'état faux, vrai et inconnu (ou indéterminé).
- Les états 'L', 'H' et 'W' représentent des états résistifs du type *pull-down* ou *pull-up*.
- L'état 'Z' représentent un état haute-impédance.
- L'état 'U' est l'état initial par défaut et identifie les signaux qui n'ont pas été affectés en simulation.
- L'état '-' est parfois utilisé pour des vecteurs de test ou en synthèse pour spécifier que certains bits ne sont pas importants.

Le paquetage déclare des versions des opérateurs logiques pour ces nouveaux types. Il n'est par contre pas possible d'utiliser sans autre des opérateurs arithmétiques sur des objets appartenant à ces types.

L'utilisation du paquetage STD_LOGIC_1164 dans une unité de conception requiert la spécification d'une clause de contexte.

Paquetage STD_LOGIC_1164 (2/2)

- **Corps du paquetage (extrait)**

```
package body STD_LOGIC_1164 is
    ...
    type stdlogic_table is array (std_ulogic, std_ulogic) of std_ulogic;
    constant resolution_table : stdlogic_table := (
        --| U   X   0   1   Z   W   L   H   - |
          ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'),  -- | U |
          ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'),  -- | X |
          ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'),  -- | 0 |
          ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'),  -- | 1 |
                                    'Z', 'W', 'L', 'H', 'X'),  -- | Z |
                                    'W', 'W', 'W', 'W', 'X'), -- | W |
                                    'L', 'W', 'L', 'W', 'X'),  -- | L |
                                    'H', 'W', 'W', 'H', 'X'),  -- | H |
                                    'X', 'X', 'X', 'X', 'X')   -- | - |);
                                      ...
```

```
    ...
    function resolved (s : std_ulogic_vector) return std_ulogic is
        variable result : std_ulogic := 'Z'; -- default state
    begin
        if s'length = 1 then return s(s'low); -- single driver case
        else
            for i in s'range loop
                result := resolution_table(result, s(i));
            end loop;
        end if;
        return result;
    end function resolved;
    ...
end package body STD_LOGIC_1164;
```


Le corps de paquetage inclut les corps des sous-programmes déclarés dans la déclaration de paquetage correspondante.

A titre d'exemple, la fonction de résolution resolve utilise une table constante resolution_table pour définir les valeurs résolues de toutes les paires de sources possibles. Le corps de la fonction traite d'abord le cas pour lequel le signal résolu n'a qu'une seule source de manière à optimiser le temps de calcul. Dans le cas contraire, une boucle est utilisée pour appliquer la table de résolution à chaque source du signal.

Paquetages mathématiques

```
package MATH_REAL is
    -- constantes e, pi et dérivées
    -- fonctions sign, ceiling, floor, round, min, max
    -- procédure random
    -- racine carrée, racine cubique, exponentiation
    -- fonction exponentielle et logarithmes
    -- fonctions trigonométriques (sin, cos, tan, etc.)
    -- fonctions hyperboliques (sinh, cosh, tanh, etc.)
end package MATH_REAL;
```

- **Dans bibliothèque IEEE**
- **Requierent la clause de contexte**

```
library IEEE;
use IEEE.math_real.all;
use IEEE.math_complex.all;
```

```
package MATH_COMPLEX is
    -- type complexe (formes cartésienne et polaire)
    -- constantes 0, 1 et j
    -- fonctions abs, argument, negation, conjugate
    -- racine carrée, exponentiation
    -- fonctions arithmétiques avec opérandes cartésiens et polaires
    -- fonctions de conversion
end package MATH_COMPLEX;
```


Seuls les opérateurs arithmétiques de base sont a priori disponibles pour un objet du type prédéfini real. Les paquetages MATH_REAL et MATH_COMPLEX ne font pas partie du standard VHDL de base, mais sont définis comme des standard séparés.

Paquetage NUMERIC_BIT

```
package NUMERIC_BIT is
    type unsigned is array (natural range <>) of bit;   -- équiv. à type entier non signé
    type signed is array (natural range <>) of bit;     -- équiv. à type entier signé
    -- opérateurs abs et "-" unaire
    -- opérateurs arithmétiques: "+", "-", "*", "/", rem, mod
    -- opérateurs relationnels: "<", ">", "<=", ">=", "=", "/="
    -- opérateurs de décalage et rotation: sll, srl, rol, ror
    -- opérateurs logiques: not, and, or, nand, nor, xor, xnor
    -- fonctions de conversion:
    --      to_integer(arg)
    --      to_unsigned(arg, size)
    --      to_signed(arg, size)
end package NUMERIC_BIT;
```

- **Requiert la clause de contexte**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_bit.all;
```

- **Utile pour disposer d'opérateurs arithmétiques sur des opérandes de types similaires (mais pas équivalents) au type bit_vector**

Un objet de type (un)signed n'est pas directement compatible avec un objet de type bit_vector. Une conversion de type est requise:

```
signal S1: unsigned(7 downto 0);
signal S2: bit_vector(7 downto 0);

S1 <= unsigned(S2);
S2 <= bit_vector(S1);
```

Paquetage NUMERIC_STD

```
package NUMERIC_STD is
    type unsigned is array (natural range <>) of std_logic;   -- équiv. à type entier non signé
    type signed is array (natural range <>) of std_logic;     -- équiv. à type entier signé
    -- opérateurs abs et "-" unaire
    -- opérateurs arithmétiques: "+", "-", "*", "/", rem, mod
    -- opérateurs relationnels: "<", ">", "<=", ">=", "=", "/="
    -- opérateurs de décalage et rotation: sll, srl, rol, ror
    -- opérateurs logiques: not, and, or, nand, nor, xor, xnor
    -- fonctions de conversion:
    --       to_integer(arg)
    --       to_unsigned(arg, size)
    --       to_signed(arg, size)
end package NUMERIC_STD;
```

- **Requiert la clause de contexte**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
```

- **Utile pour disposer d'opérateurs arithmétiques sur des opérandes de types similaires (mais pas équivalents) au type std_logic_vector**

Un objet de type (un)signed n'est pas directement compatible avec un objet de type std_logic_vector. Une conversion de type est requise:

```
signal S1: unsigned(7 downto 0);
signal S2: std_logic_vector(7 downto 0);

S1 <= unsigned(S2);
S2 <= std_logic_vector(S1);
```

Table des matières

A. Vachoux, v2.0, 2003 Le langage VHDL - 96

Livres (1/2)

- **IEEE Standard VHDL Language Reference Manual**
  ISBN 0-7381-3247-0; Product No.: SS94983-TBR; IEEE Standard No.: 1076-2002
  - *LA référence officielle du langage pour les puristes et les développeurs d'outils*

- **IEEE Standard Multivalue Logic System for VHDL Model Interoperability**
  IEEE Standard 1164-1993, Ref.: SH16097
  - *Définition officielle du paquetage STD_LOGIC_1164*

- **IEEE Standard VITAL ASIC Modeling Specification**
  IEEE 2001; ISBN 0-7381-2691-0; Product No.: SH94959-TBR; IEEE Standard No.: 1076.4-2000
  - *Définition officielle de la méthode de modélisation VHDL de composants de bibliothèques ASIC*

- **Peter J. Ashenden, The Designer's Guide to VHDL, 2nd Edition**
  Morgan Kaufmann Publishers; ISBN: 1558606742; 2nd Bk&cdr edition (May 29, 2001)
  - *Présentation complète du langage VHDL avec des études de cas réalistes*

A. Vachoux, v2.0, 2003 Le langage VHDL - 97

Livres (2/2)

- **Mark Zwolinski, Digital System Design and VHDL, Prentice Hall; ISBN: 0201360632; 1 edition (October 18, 2000)**
  - *Présentation plus axée sur les aspects de modélisation, simulation et synthèse de circuits logiques et numériques*

- **Roland Airiau, Jean-Michel Bergé, Vincent Olive et Jacques Rouillard, VHDL: Du langage à la modélisation, Presses Polytechniques Romandes, ISBN: 2-88074-361-3, 1998**
  - *Couverture complète du langage avec beaucoup d'exemples*

- **Thierry Schneider, VHDL: méthodologie de design et techniques avancées, Dunod, ISBN : 210005371X - Code : 45371, 2001**
  - *Approche pratique de VHDL*

- **Janick Bergeron, Writing Testbenches: Functional Verification of HDL Models, Kluwer Academic Publishers, ISBN: 0306476878, 2000**
  - *Présentation axée sur le développement de modèles de test*

A. Vachoux, v2.0, 2003 Le langage VHDL - 98

Web

- **http://lsmwww.epfl.ch/design_languages**
  - *Le site du cours + informations sur d'autres langages (VHDL-AMS, Verilog(-AMS), SystemC(-AMS)*
- **http://standards.ieee.org/catalog/index.html**
  - *Catalogue des standards IEEE*
- **http://www.eda.org/**
  - *Site généraliste pour les outils EDA avec accès à des ressources VHDL (modèles, outils, FAQs)*
- **http://tech-www.informatik.uni-hamburg.de/vhdl/**
  - *L'un des premiers répertoires de ressources VHDL*
- **http://www.eda.ei.tum.de/forschung/vhdl/**
  - *Un autre répertoire de ressources VHDL*
- **http://opensource.ethz.ch/emacs/vhdl-mode.html**
  - *Mode VHDL pour l'éditeur Unix Emacs*

A. Vachoux, v2.0, 2003 Le langage VHDL - 99