Aide-Mémoire

VHDL !!

Jacques Rouillard

Aide-mémoire VHDL

Les types

Types numériques : integer, real, on se sert essentiellement des types prédéfinis.

Types énumérés

- `type enum1 is (un, deux);`
- `type enum2 is ('0', '1');`
- `type enum3 is ('B', bla, bli`

Types structurés

- Tableaux
  - `type tableau is array (1 to 10) of integer;`
  - `type tableau is array (character) of bit;`
  - `type tableau is array (10 downto 0, bit) of float;`
- Enregistrements
  - `type enreg is record a: integer; b: character; end record;`

Types physiques: utilisé pratiquement seulement avec TIME qui est prédéfini.

Types accès

- `type pointeur is access enreg;`

Types fichiers

- `type text is file of string;`

[AMS] Natures: voir page 6.

```
Attributs sur types scalaires ou tableaux
T'HIGH borne max – ou T'HIGH(dimension)
T'LOW borne min  - idem
T'LEFT borne de gauche -- idem
T'RIGHT borne de droite
T'RANGE  = X'LEFT [down]to X'RIGHT
T'REVERSE_RANGE X'RIGHT [down]to X'LEFT
```

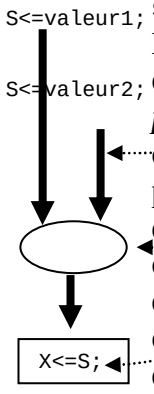
```
enum1'POS(un) = 0
enum2'VAL(1) = '1'
```

La surcharge sur types énumérés (voir aussi les sous-programmes page 5):

```
type enum1 is (un, deux);signal X1:enum1;
type enum2 is (bla, un); signal X2:enum2;
…
X1 <= un; X2 <= un; --mais pas X1<=X2 !
enum1'pos(un) vaut 0 et enum2'pos(un) = 1
```

Les objets

- **Constantes** : sémantique triviale : `constant C : integer := 3 ;`
- **Fichiers :** sémantique triviale sauf qu'on ne devrait ni les ouvrir (cela peut être fait à la déclaration) ni les fermer (pour interdire la communication inter-process) : `file F : text ;`
- **Variables** : déclarées et utilisées seulement dans les process et les sous-programmes (sauf variables partagées, à ne pas utiliser hors conception niveau système). Ont la même sémantique que dans un langage de programmation. Peuvent être de n'importe quel type. `variable V : le_type ;`
- **Signaux** : déclarés dans toute zone concurrente (architecture, déclaration de paquetage) et utilisés partout, y compris dans les process. Peuvent être de tout type sauf accès et fichier. Peuvent être des « port » s'ils servent à communiquer entre entités.



`signal S : le_type ;` ou `port ( P : in le_type )`

Leur sémantique est différente de la variable : l'affectation de signal (S <= valeur) n'a pas d'effet observable instantanément. La valeur est enfilée dans un ***driver*** (il y *a un driver par signal et par process ou process équivalent*) et prévue pour apparaître dans le futur, au plus tôt un delta (un cycle de simulation) plus tard. Quand la mise à jour se fait, tous les drivers proposant une valeur pour ce signal là à cette date là sont « résolus »[1] et c'est cette résolution qui devient la valeur observable[2]. Voir le cycle de simulation, page 5

```
S'EVENT , boolean, S a changé de valeur
S'ACTIVE boolean, S a été affecté
S'LAST_EVENT time, date du dernier événement
S'LAST_ACTIVE time, date dernière affectation
S'LAST_VALUE avant dernière valeur
S'DELAYED(t) copie de S décalée
S'STABLE(t)  boolean, S a été stable pendant t
S'QUIET(t) boolean, pas été affecté pendant t
S'TRANSACTION boolean, bascule à chaque affectation
```

En italique : rendent des signaux.

- ***[AMS] terminaux****: voir page 6.*
- ***[AMS] quantités****: voir page 6.*

[1] C'est là que les signaux forts l'emportent sur les faibles ou que les conflits sont détectés. Voir le paquetage STD_LOGIC_1164, section **paquetages standards.**

[2] Avec pour conséquence qu'on n'observe jamais dans le même cycle la valeur qu'on vient de proposer.

Blocs de base/Logique combinatoire/Dataflow : *le circuit peut se représenter comme Sorties = fonction (Entrées), et il n'y a pas de boucle entre entrées et sorties.*

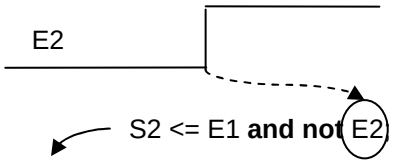
```
library IEEE ;use IEEE.std_logic_1164.all ;
entity decodeur is
      port (E1,E2 : in std_logic ; S1,S2,S3,S4 : out std_logic ) ;
end decodeur;

architecture dataflow of decodeur is
begin
  S1 <= not E1 and not E2;
  S2 <= E1 and not E2;
  S3 <= not E1 and E2;
  S4 <= E1 and E2;
end;
```

Affectation conditionnelle :

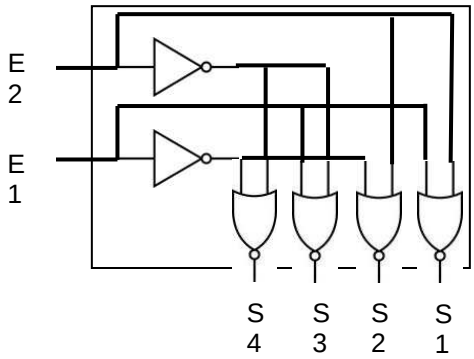
```
S <=  expression1 when condition1
 else expression2 when condition2
 ..... else expression3 ; [ou unaffected]
```

Affectation sélectée :

```
with sélecteur select s<= expr1 when valeur1 ,
                          expr2 when valeur2 ,
                          .... ;
```



Pour la simulation: chaque instruction a une liste de sensibilité (la liste des signaux qui sont dans sa partie droite). Si l'un d'eux change, l'instruction est évaluée et son résultat est déposé sur le signal affecté. Inversement : pas de changement, pas d'évaluation. C'est cette économie qui fait l'efficacité de la simulation « event-driven » (peu de signaux changent à un instant donné dans un circuit). Dans cet exemple simple, toutes les sorties sont sensibles sur toutes les entrées, mais ce n'est pas toujours le cas.



Pour la synthèse : l'outil va chercher à faire une logique équivalente à celle qui est décrite, en utilisant des éléments de bibliothèque. Dans un cas trivial comme celui-ci, on peut penser que l'outil utilisera une brique « décodeur 2 vers 4 ». Le résultat de synthèse donnera une architecture structurelle dont les sorties porteront les délais fournis par la bibliothèque, ici sous une forme outrageusement simplifiée :

```
library lib4212;use lib4212.delaypkg.all;
architecture structure of decodeur is
  component decodeur
      port (E1,E2 : in std_logic ; S1,S2,S3,S4 : out
std_logic ) ;
  end component;
  signal Ss1,Sz2,Ss3,Ss4: std_logic;
  for dec: decodeur use entity lib4212.deco2to4(bla666);
begin
dec :decodeur port map (Ee1,Ee2,Ss1,Ss2,Ss3,Ss4) ;
S1 <= Ss1 after delaynor;
S2 <= Ss2 after delayinv+delaynor;-- ici on pourrait raffiner et distinguer
S3 <= Ss3 after delayinv+delaynor;-- suivant les valeurs de E1 et E2.
S4 <= Ss4 after delayinv+delaynor;
end structure;
```

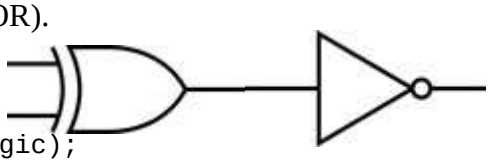
La simulation de ce bloc devra donner, aux délais près, les mêmes chronogrammes que ceux de la spécification (ici *dataflow*.) Si l'on est mécontent du résultat obtenu, il faut revenir à la spécification s'il y a matière à optimisation, et/ou revenir à la synthèse en changeant des paramètres, par exemple la bibliothèque utilisée. Voir cycle de conception page 9.

Une norme (Vital) permet aux fournisseurs de bibliothèque de fournir sous une forme standard ce genre de délais pour les résultats de synthèse. Voir Vital page 8.

Structure : *on représente le circuit comme une interconnexion de blocs de bibliothèque.*
La bibliothèque contient les éléments nécessaires :

```
entity XorGate is port (E1,E2: in std_logic;S: out std_logic); end XorGate;
entity NotGate is port (E : in std_logic; S :out std_logic); end NotGate;
```

Les architectures peuvent être en source ou en binaire, cela n'a pas d'importance pour ce qui est de l'utilisation. Faisons un XNOR (avec le NOT et le XOR).

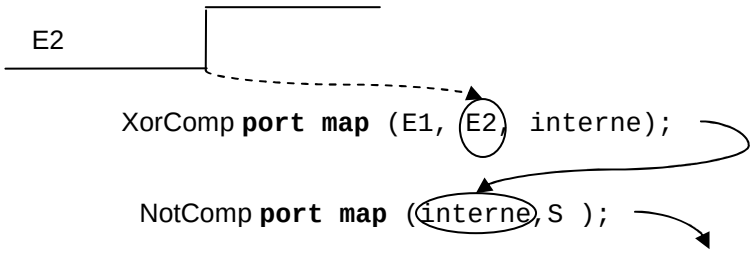


```
library IEEE ;use IEEE.std_logic_1164.all ;
entity XnorGate is
 port (E1,E2 : in std_logic ; S :out std_logic);
end XnorGate;
```

L'entité appelante contient plusieurs zones : la déclaration de composants (les supports destinés à recevoir les entités), la configuration (qui dit quelle entité va sur quel composant) et les instances (là où les composants sont effectivement posés et connectés.) On a besoin d'un signal interne pour connecter les deux boîtes.

```
library ressources_lib ;
architecture structure of XnorGate is
 signal interne: std_logic; -- un signal pour connecter les deux portes.
   -- déclaration des composants
    component XorComp port(E1,E2:in std_logic;S: out std_logic);end component;
    component NotComp port (E : in std_logic;S :out std_logic); end component;
   -- configurations
 for all: XorComp use entity ressources_lib.XorGate;[3]
 for all: NotComp use entity ressources_lib.NotGate;
begin
  -- instances
  XnorInstance: XorComp port map (E1, E2, interne);
  NotInstance : NotComp port map (interne, S) ;
end;
```

Pour la simulation: chaque changement sur un signal provoque l'activation du bloc concerné, et ceci de façon transitive s'il y a de nouveau changement de valeur. Chaque bloc se simule en fonction de sa propre description dans la bibliothèque appelée. Si un délai est spécifié dans une affectation, les « glitchs » inférieurs à ce délai sont gommés dans le mode normal (pas **`transport`**).



Pour la synthèse : on peut supposer que les blocs de la bibliothèque appelée sont déjà synthétisés ou synthétisables et que la synthèse de l'assemblage se borne à une recopie de la structure et de la netlist. Bien entendu, des optimisations sont possibles à travers la hiérarchie mais il ne faut pas trop compter dessus sur des outils simples.

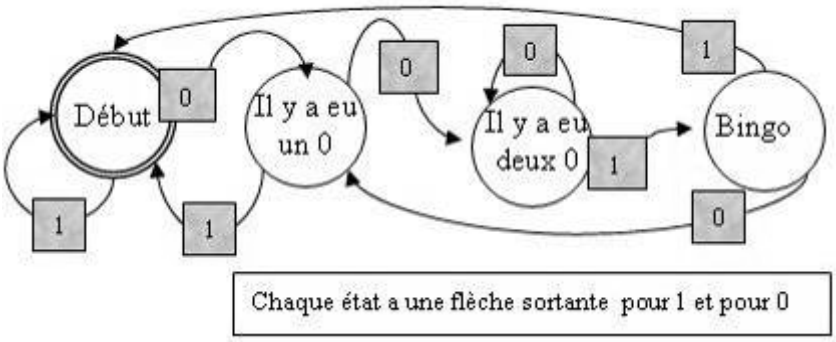
[3] La configuration peut être simple comme ici (simplement renvoyer à une entité existante) ou encore spécifier de quelle architecture on parle s'il y en a plusieurs, et enfin dire explicitement quel port du composant correspond à quel port de l'entité. Une forme plus complète (hors généricité) est :

```
for ref-des-composants: nom-du-composant
                        use entity ressources_lib.entité(architecture)
                        port map (port-du-composant => port-de-l'entité,…)
```

`ref-des-composants` peut être le nom du label, une liste, ou **`others`**, ou **`all`**.

Séquentiel synchrone, machines d'état : *il s'agit ici de faire de la logique à état.* Un état peut être représenté comme un élément de type énuméré. Par exemple si on veut déterminer si une entrée contient la séquence « 001 »:

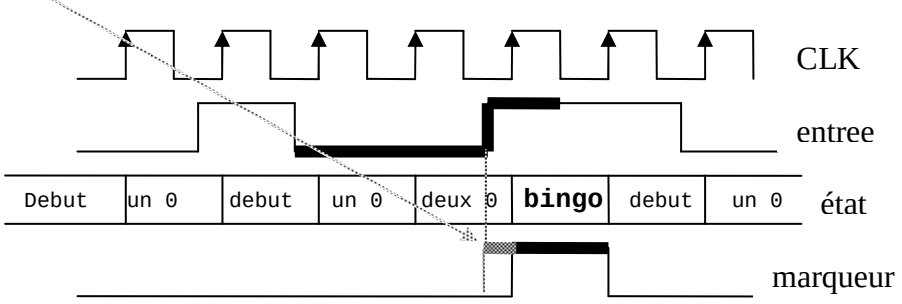
```
type type_etat is (debut,
il_y_a_eu_un_0,
il_y_a_eu_deux_0, bingo) ;
signal etat : type_etat :=
debut ;
```



On implémente la machine d'états comme un *process* (ou deux suivant les styles, ce qui permet d'isoler le combinatoire et le synchrone) contenant un *case* dont chaque branche est activée par une valeur de l'état, prend les actions nécessaires -dont le changement d'état. On suppose ici que «marqueur» est un signal qui marque la reconnaissance de la séquence. Il est mis à 0 sur tous les états sauf le dernier.

```
comb: process(etat,entree)
begin
 case etat is
  when debut => marqueur<='0' ;
   if entrée='0' then etat_suivant<= il_y_a_eu_un_0;end if ;
  when il_y_a_eu_un_0=> marqueur<='0' ;
   if entree ='0' then etat_suivant<=l_y_a_eu_deux_0;
   else etat_suivant <= debut ; end if ;
  when il_y_a_eu_deux_0=> marqueur<='0' ;
   if entree ='1' then etat_suivant <= bingo ;
   marqueur<='1' (si Mealy) ; end if ;
  when bingo => marqueur<='1';
   if entree = '0' then
     etat_suivant <=
       il_y_a_eu_un_0;
   else etat_suivant <=
      debut; end if;
  end case;
end process;
```

```
sync: process(ck, reset)
begin
if reset='1' then
  etat<=debut;
elsif ck'event and ck='1'
then
  etat<=etat_suivant;
end if;
end process ;
```

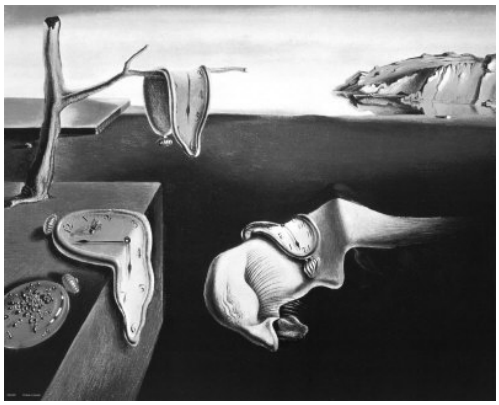


Pour la Simulation:
À chaque front d'horloge, l'entrée est lue par la branche correspondante du case; le marqueur est mis à 1 ou 0, et l'état suivant est calculé.

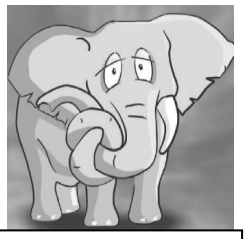
Pour la Synthèse : ici une synthèse « à la main » et en VHDL, sans aucune optimisation. L'état va être codé (ici sur 2 bits, le codage est optimisable). 00=début, 01= etc.

```
etat <= "00" when reset='1' else etat_suivant when
clk'event and clk='1';
marqueur<='1' when etat="11" else '0';
with etat & entree select
etat_suivant <= "01" when "000", "01" when "010",
"11" when "101",  "01" when "110", "00" when "111",
unaffected when others ;
```

On voit qu'on calcule deux bits en fonction de trois par une table. Cela fera un PLA.



Comportemental : *VHDL est ici un langage de programmation.* L'instruction hôte est le *process*. Un nouvel objet est la *variable*, au sens de C qui ne peut se déclarer que dans un *process* ou un sous-programme : **variable** V : *le_type* ;
Le process est une instruction qui a sa place dans l'architecture. Il peut avoir une liste de sensibilité qui sera strictement équivalente à un **wait on** sur la même liste placé à la fin. **process** (S1,S2,..) **begin** *séquence* **end process** ;
process begin *séquence* **wait on** S1,S2…; **end process**;



-{} = zéro fois ou plus. [] = zéro fois ou une.
séquence= séquence d'instructions

Instructions :

- V := *valeur* ; S<= *valeur* ;
- **if** *condition* **then** *séquence* {**elsif** *séquence*} [**else** *séquence*] **end if** ;
- **case** *sel* **is** {**when** valeur=>*séquence*}[**when others** =>*séquence*] **end case**;

D'un type discret ou tableau de discrets (bit, entier) ou vecteur de discrets (bit_vector)

- [label:] **loop** *séquence* **end loop**;
  [label:] **for** I **in** 1 **to** 10 **loop** *séquence* **end loop**;
  [label:] **while** condition **loop** *séquence* **end loop**;

Sortir d'une boucle:
exit [label][**when** *condition*];
(pas de label = la plus interne)

- **wait**; **wait on** *liste_de_signaux*; **wait until** *condition*; **wait for** 10 ns;
- procedure(*liste d'arguments*) ; X = fonction(*liste d'arguments*) ;
- Dans les sous programmes seulement : **return** [valeur] ;
- Déclaration et définition de sous-programme :
  - **procedure** P (arg1 : [**in**][**out**]type1 ; …)[**is begin** séquence **end**];
  - **function** F (arg1 : [**in**]type1 ; …) **return** type3 [**is** *…idem.*

La fonction doit sortir par un **return**. La procédure peut.

Surcharge : les sous-programmes peuvent partager le même nom (avec les éléments de types énumérés). Ils sont distingués par le contexte (essentiellement nombre et type des arguments)

Pour la simulation

Le cycle

- Tous les *process* à lancer sont lancés dans un ordre quelconque (et inobservable)
- Les valeurs affectées aux signaux sont conservées dans des *drivers*. Les variables par contre sont affectées immédiatement.
- Tous les *process* doivent finir par tomber sur un **wait** (sinon erreur).
- Alors et seulement, les valeurs sont sorties des *drivers* pour calculer les valeurs effectives des signaux, cf page 3. C'est cela qui permet d'émuler la concurrence et qui fait que l'ordre d'exécution est inobservable.

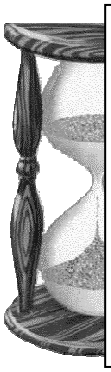
```
S <= '1' ;
if S='1' then …
-– pas forcément vrai (pas de wait entre)
```

- Si la valeur d'un signal change (événement), et si un *process* est sensible sur ce signal, ce *process* devient candidat pour le prochain cycle au même temps de simulation.
- Quand il n'y a plus d'événement à un temps donné, le temps avance : soit au prochain temps marqué par **wait for**, soit au prochain changement de signal marqué par une clause **after,** soit [analogique] au prochain événement qui sera éventuellement créé par le noyau analogique. Directement au plus proche de ces temps [digital] ou par pas [analogique]. Et on recommence.

Pour la synthèse : celle-ci est extrêmement dépendante des outils. Quelques règles usuelles :

- Beaucoup de constructions sont par essence non synthétisables : fichiers, accès, procédures récursives…La première passe du synthétiseur sert surtout à corriger le VHDL pour le rendre synthétisable.
- Il ne faut pas chercher à synthétiser des blocs réguliers (RAM, etc.) ; il faut appeler des blocs existants ou compter sur le synthétiseur pour « découvrir » un PLA.
- Un signal affecté dans toutes les branches d'un **if** ou d'un **case** sera construit avec de la logique combinatoire. S'il en manque une ou plusieurs, il y aura un registre puisqu'il faut sauver la valeur d'un cycle à l'autre.
- Il est souvent de bon goût de rendre le *process* sensible à tous les signaux qui y sont lus (par un **wait** explicite ou implicite). On distinguera alors les choses à faire sur niveau par un test de la valeur (**if** reset='1') et les choses à faire sur événement par un test sur les attributs *event* et *stable*.(**if** clk'event **and** clk='1').
- Ne pas initialiser les objets lors de leur déclaration : le faire dans une étape « reset ».
- Il **faut** lire la doc de l'outil de synthèse. Puis il **faut** la relire.

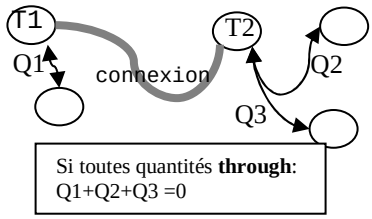
Simultané : *VHDL-AMS permet d'écrire des équations différentielles et un solveur s'arrange pour qu'elles soient (à peu près) vérifiées à certains points du temps.*



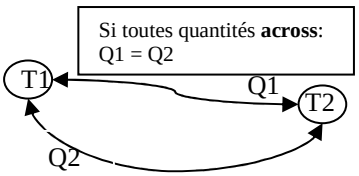
```
Principaux attributs usuels:
S'RAMP S'SLEW rendent des rampes définies par temps ou par pente.
Q'DOT : derivée / temps
Q'INTEG: intégrale depuis 0
Q'DELAYED(T): Q décalé de T
T'REFERENCE : quantité across entre T et sa référence
T'CONTRIBUTION: quantité through somme des quantité through incidentes à T
Q'ABOVE(E): signal TRUE si Q > E
```

- **discipline**: les domaines physiques que l'on simule. Ils sont définis dans des paquetages de la bibliothèque DISCIPLINES.
- **nature**: un ensemble de concepts obéissant à l'équivalent des lois de Kirchhoff pour le domaine électrique, servent de "types" pour les terminaux. Contiennent un champ **across**, un champ **through** et une **reference**. Dans le cas du domaine électrique, il s'agit de représenter la tension, le courant et la masse. Se déclare là où on déclare les types.

```
subtype voltage is real;
subtype current is real;
nature electrical is
      voltage across
      current through
      masse reference;
```



- **terminal**: un objet appartenant à une nature. Dans le domaine électrique, c'est une équipotentielle, il n'a pas de « valeur ». Les lois de Kirchhoff disent qu'entre deux terminaux, il y a une quantité **across** et une seule (la tension) et que la somme de l'ensemble des quantités **through** qui partent et arrivent sur une équipotentielle (un terminal ou des terminaux interconnectés) est algébriquement nulle. Un terminal se déclare aux mêmes endroits que les signaux. Il peut être un port de composant et d'entité.




```
terminal T1,T2: electrical;
```

- **quantity**: Les quantités peuvent être libres, sources ou attachées:

```
quantity Q: real;
```

1. *libres* l'équivalent d'une variable au sens mathématique du terme. Le solveur va "essayer" de rendre vraies les équations en ajustant les valeurs des quantités
2. *sources* de bruit ou de spectre dans le domaine fréquentiel.
3. ***through*** *ou* ***across*** , attachées à un terminal.

```
quantity SRC: real spectrum exp-magn, exp-phase
quantity NSE: real noise exp-puissance;
```

```
quantity V across I1,I2 through T1 to T2;
-- V est la tension entre T1 et T2
-- I1&I2 deux branches de courant parallèles.
```

Les quantités se déclarent aux mêmes endroits que les signaux. Elles peuvent être des ports de composants et d'entités.

Les instructions simultanées*: se mettent en zone concurrente (comme les process)*

- Équation simple: `[label:] expression == expression;` -- doit contenir au moins une quantité.
- Instruction conditionnelle:

```
[label:]  if condition use {instructions simultanées}
          {elsif condition use {instructions simultanées}}
          [else {instructions simultanées}]end use [label];
```

- Instruction sélective:

```
[label:] case sélecteur use
                {when choixx =>{instructions simultanées}}
         end case [label];
```

- Procédural:

```
[label:] procedural begin
                {instructions séquentielles sauf wait et break
                les quantités y sont vues comme des variables}
         end procedural [label];
```

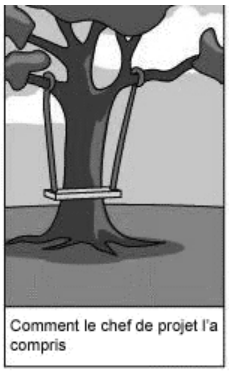
break on instruction concurrente avec équivalence séquentielle, indispensable pour dire au noyau qu'une discontinuité nécessite un recalcul des conditions "initiales".

```
break for Q1 use Q2 => expression
on S when condition;
```

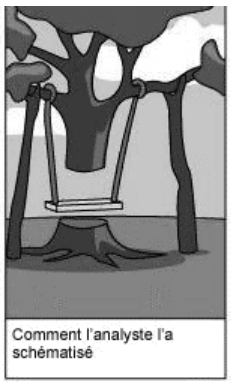
Condition de solvabilité: La somme des quantités libres, des quantités de mode **out** et des quantités **through** d'une unité doit être égale au nombre d'équations (ou équations équivalentes). Ceci est une condition nécessaire, mais ne garantit évidemment pas la convergence.

Gestion de projet : *VHDL a des bibliothèques, permet des configurations, gère plusieurs architectures par entité, permet la généricité et la génération de code.*

- **Les unités de compilation** (design units) sont les portions de texte que l'on peut compiler et stocker en bibliothèque. Les entités –**entity**- (spécifications), les **architecture**s (implémentations), déclarations de paquetages –**package**- (jeu de ressources partageables) et implémentation de paquetage, les **configuration**s (isolement de toute l'information de configuration d'un modèle).
- **Clause with** : les bibliothèques sont, pour le programmeur, des noms. Le lien avec le système de fichiers n'est pas dans le langage. On appelle une bibliothèque par la clause `with lib ;` à placer en tête d'unité (avant les mots **`entity`**, **`architecture`**, etc.). Dès qu'une bibliothèque est référencée, tous les noms qui sont dedans sont visibles par la notation pointée : `lib.entite` ou `lib.pack.objet`.

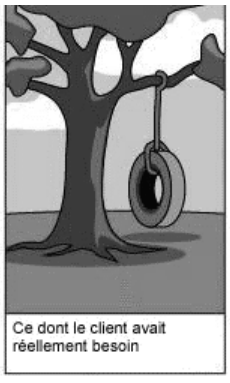


- **Clause use** : on peut « factoriser » un préfixe trop encombrant à écrire, sous réserve que cela ne crée pas d'ambiguïtés. La clause « **`use`** `prefixe.`**`all`** `;` » peut être mise n'importe où dans une zone déclarative. En dessous, les objets déjà visibles (et seulement ceux-là) par la notation `prefixe.objet` deviennent visibles directement (sauf conflit). « **`use`** `prefixe.objet ;` » moins utile, ne rend directement visible que l'objet.



- **Bibliothèques par défaut et standardisées** : il y a deux bibliothèques par défaut (STD, WORK) et beaucoup de bibliothèques très utiles et standardisées : ex. IEEE qui contient IEEE.STD_LOGIC_1164. Voir page 8. Chaque unité est préfixée d'office et par défaut avec : **`with`** `STD ;` **`use`** `STD.STANDARD.`**`all`** `;` **`with`** `WORK ;` Pour VHDL-AMS, la bibliothèque DISCIPLINES contient tous les paquetages des différents domaines physiques*: electrical, mecanical, thermal*, etc.
- **Chaque entité peut avoir plusieurs architectures**. C'est pourquoi celles-ci ont un nom. Quand on veut simuler ou synthétiser, il faut dire quelle couple entité/architecture ; au niveau de l'outil (simulateur par ex.) cela dépend de l'interface. Dans le texte lui-même, lors des configurations, la notation est : `entite(archi)`.
- **Les configurations :** c'est le moyen, dans une description structurelle, de dire à chaque niveau de hiérarchie quelle couple entité/architecture on instancie. On utilise pour cela une clause « **`for`** `X : composant` **`use`** `entity ent(arch)` » qui permet de passer des arguments génériques ou une association des ports (voir description structurelle.) Une unité de configuration, compilable telle quelle, permet de rassembler toutes les clauses de configuration d'un modèle en un seul fichier.

- **La généricité :** paramètrise un modèle. On l'annonce lors des déclarations d'entité ou de composant : «**`entity E is generic (X:integer) ;port(…); end;`**». On indique sa valeur avant la simulation ou synthèse, par un **`generic map`**. « **`for`** `X :composant` **`use entity`** `ent(arch)`**`generic map`**`(3) …` »



- **La génération :** permet de créer du code répétitif ou conditionnel, par programme. C'est du code qui sera exécuté avant la simulation. Uniquement en zone concurrente (pas dans un *process*).

```
if condition generate (instructions) end generate ;--pas de else.
for I in 1 to 10  generate (instructions) end generate;
```

Dans ce dernier cas, l'indice I peut être utilisé dans les instructions pour indexer des tableaux et construire des circuits complexes ; les bornes (ici 1 et 10) peuvent venir d'un argument générique. Attention ! on a vite fait d'avoir tellement de combinaisons possibles que le circuit effectivement construit a de bonnes chances de n'avoir jamais été testé.

Les paquetages standard : *l'environnement de base permettant de compiler est dans un jeu de paquetages dont la visibilité est assurée par défaut (pas besoin de clause with).*

- **STD.STANDARD :** tous les types indispensables, comme INTEGER, BIT etc.
- **STD.TEXTIO :** contient un jeu de primitives assez rustique, permettant de faire des entrées sorties textuelles sur fichier. Il faut lire ligne à ligne des chaînes de caractères (READLINE), puis on exploite cette chaîne par d'autres primitives de traitement de chaîne appelées READ. À noter un STD_LOGIC_TEXTIO qui n'est pas standard mais bien commode.

Les paquetages IEEE : *pour rendre VHDL plus utilisable, un jeu de paquetages ont été standardisés.* Ils sont distribués avec toutes les implémentations et sont compris par les outils de synthèse. L'organisme normalisateur est IEEE ; certains paquetages d'usage courant ne sont toutefois pas standardisés bien qu'étant souvent placés dans la bibliothèque IEEE..

- **IEEE.STD_LOGIC_1164** : c'est le plus important de tous : bien des concepteurs ne se servent *jamais* du type BIT défini dans STD.STANDARD. Il fournit un type « logique » à 9 valeurs : 'U' 'X' '0' '1' 'Z' 'W' 'L' 'H' ' -' 'U' pour *uninitialized*, valeur par défaut déposée au début des simulations et permettant de repérer les signaux qui n'ont pas été initialisés au bout d'un temps raisonnable. Deux systèmes logiques (fort : 01X et faible :LHW –low, high, weak conflict) permettent de gérer les « forces » : en cas de conflit entre un faible et un fort, c'est le fort qui gagne, voir « signaux » page 3. On peut ainsi modéliser des systèmes « drain ouvert » par exemple. 0/L et 1/H ont le sens logique usuel. X/W signifient « conflit ». Z est la haute impédance, pour un bus que personne ne prend par exemple. Le '-' veut dire « don't care » et ne sert qu'en synthèse pour permettre de ne pas sur-spécifier et laisser le synthétiseur optimiser (inversement, les valeurs UXW ne servent qu'au simulateur, on ne va pas « spécifier » un conflit).

	UX01ZWLH-
U	UUUUUUUUU
X	UXXXXXXXX
0	UX0X0000X
1	UXX11111X
Z	UX01ZWLHX
W	UX01WWWWX
L	UX01LWLWX
H	UX01HWWHX
-	UXXXXXXXX

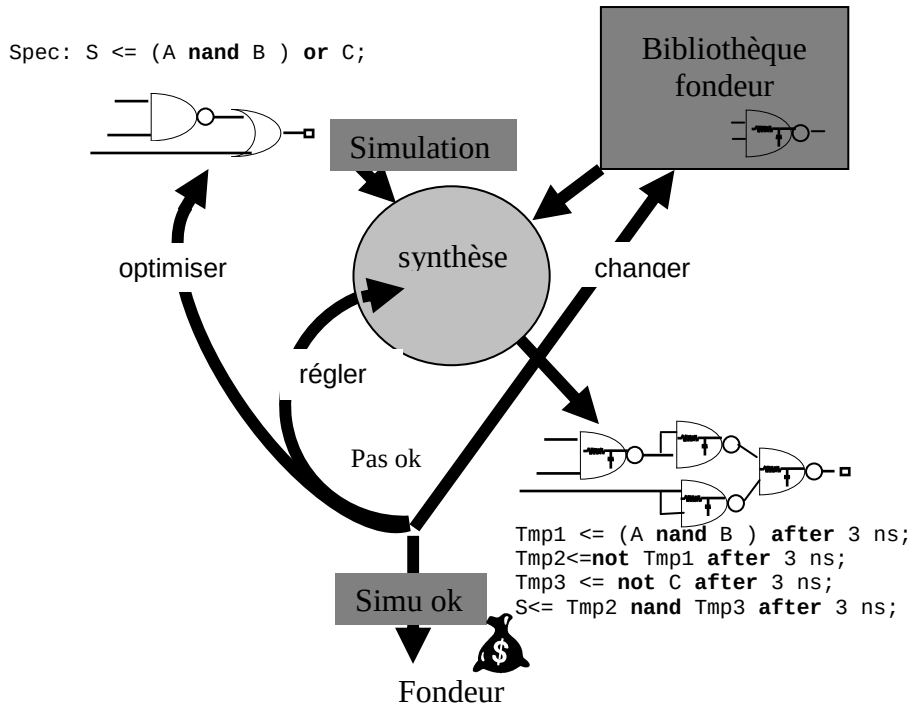
- **IEEE.STD_LOGIC_ARITH :** permet de faire de l'arithmétique sur `std_logic_vector` Pour cela il fournit deux types `signed` et `unsigned` qui ont la même définition que `std_logic_vector` : « `array (natural range <>) of std_logic` ». Suivant la définition des compatibilités de types en VHDL, tout `std_logic_vector` peut être converti explicitement en `signed` ou en `unsigned`. Les opérateurs arithmétiques sont ensuite définis dessus (et entre eux). On peut ainsi écrire des expressions comme `signed (V1) + unsigned (V2)`. Attention à ce que ceci sera synthétisé (les outils reconnaissent les paquetages standard) et que l'addition n'est pas gratuite. Un paquetage **NUMERIC_BIT**, moins utilisé, fait la même chose sur bit_vector. Si on ne veut faire *que* de l'arithmétique signée ou non-signée, deux autres paquetages non standardisés existent aussi qui n'obligent pas à changer le type avant les opération : **STD_LOGIC_SIGNED** et **STD_LOGIC_UNSIGNED**. On ne peut utiliser directement (avec une clause **use**) qu'un des deux à la fois.
- **IEEE.MATH_REAL** et **MATH_COMPLEX** : permettent d'avoir les fonctions mathématiques sur entiers et réels.
- **IEEE.VITAL** : (Vhdl Initiative Towards Asic Libraries). C'est un ensemble de paquetages – VITAL_TIMING,VITAL_PRIMITIVES et autres- permettant de décrire les fonctions logiques de bibliothèques avec des primitives standard, en repoussant à l'extérieur du modèle les questions temporelles. Ainsi le même modèle peut être simulé logiquement et validé sans les délais, puis synthétisé de telle sorte que les délais obtenus par la synthèse puissent être réinjectés dans le modèle initial sans avoir à l'éditer. Un format (SDF) permet cette rétro-annotation qui peut aussi être faite par *configuration.*

Les paquetages DISCIPLINES : pour VHDL-AMS, définissent les natures, constantes et attributs des différents domaines physiques: *electrical, mechanical, thermal*, etc. Ils utilisent deux paquetages de ressources: IEEE.FUNDAMENTAL_CONSTANTS et IEEE.MATERIAL_CONSTANTS.

Autres ressources : voir surtout *http://tams-www.informatik.uni-hamburg.de/research/vlsi/vhdl/*

Généralités : *la raison d'être et les principales spécificités de VHDL.*

Complexité : Les systèmes, même grand public, peuvent comprendre des centaines de millions de transistors interconnectés ; leur complexité est du même ordre que le dessin de la carte d'un pays avec toutes les pièces de tous les immeubles, toutes les voies d'accès jusqu'aux couloirs et ascenseurs. Il faut avoir des descriptions formelles, avec réutilisation de code ancien ou externe. VHDL a un système de bibliothèques. Il faut avoir le moyen de passer des contrats sur des spécifications et de faire les recettes de ces contrats. VHDL est construit de façon à ce qu'un modèle soit simulé de façon identique quelle que soit la plate-forme.
Maintenance : beaucoup de systèmes complexes sont obsolètes par construction : les systèmes stratégiques militaires par exemple. Il est important que les documents informatiques associés à ces systèmes ne périssent pas avec les outils de CAO qui les utilisent. VHDL est un standard indépendant des outils. Et d'origine militaire, accessoirement.
Re-conception : la technologie évoluant, il est fréquent de vouloir reconstruire un système. Par exemple intégrer un circuit à base de composants en un circuit intégré. Pour cela il est indispensable de pouvoir séparer les spécifications fonctionnelles de la description logique, et de les faire cohabiter. VHDL fournit les multiples architectures et les configurations, une architecture pouvant contenir la spécification de haut niveau et une autre la description en portes.
Le processus de conception : La plupart des circuits sont conçus par un cycle *écriture-validation par simulation- synthèse- validation du résultat- retour au début jusqu'à satisfaction*. Comme moyens d'actions, on peut modifier soit les spécifications s'il y a matière, soit les réglages du synthétiseur, soit les bibliothèques utilisées. C'est ainsi qu'on peut partir d'une spécification de haut niveau et la raffiner jusqu'à obtenir une synthèse correcte.



Difficultés:

L'association : chaque fois qu'il faut « passer » des arguments réels à des arguments formels. Si la déclaration a la forme :

…**port** (A,B,C : STD_LOGIC) ; -- (même chose avec les arguments de sous-programmes)

L'association peut se faire

- par position : …**port map** (X,Y,Z) – X va sur A, Y sur B et Z sur C
- par nom : …**port map** (A=>X, C=>Z, B=>Y) où l'ordre n'a pas d'importance.
- mixte : …**port map** (X, C=>Z, B=>Y) passer à l'association par nom est irréversible.

On peut associer des éléments de tableaux ou de record :

…**port** (A : STD_LOGIC_VECTOR(2 **downto** 0)) donnera

- …**port map** (A(0)=>X, A(1)=>Y, A(2)=>Z)
- ou …**port map** (A(1 **downto** 0)=>T,A(2)=>Z) si T est un vecteur de 2 bits.

On peut aussi, mais c'est du vice car tout cela concours à la résolution de la surcharge, changer le type et/ou appeler une fonction lors d'une association entre ports et signal.

ex : …**port map**(Fonc(A) => TypeDeA(X)…) où les changement de type et appels de fonctions sur les arguments réel/formel sont pertinents suivant la direction des ports (**in/out/inout**). Ici cas **inout**, il y a les deux qui sont appelées selon le sens du flot. En mode **in**, seul l'appel « de gauche » serait pertinent.

Enfin dans le cas des ports, on peut laisser un port ouvert (associé à **open**). Certains outils admettent qu'on associe un port à une constante ('0' pour mise à la masse) mais ce n'est pas standard : mieux vaut associer un signal qu'on affectera avec la constante.

Dans le cas des sous-programmes, si un argument est du genre **signal**, on ne peut lui passer que des signaux. S'il est du genre **variable**, on peut lui passer des variables ou des signaux dont seule la valeur passera au sous-programme qui le verra comme **variable**.

La spécification de configuration : for all :composant **use** work.entity E(A) ;

On est souvent surpris de voir que la configuration « ne marche plus» à l'intérieur des blocs **generate**. Pour de très bonnes raisons impossibles à expliquer rapidement, la portée de cette spécification ne « descend » pas dans les blocs hiérarchiques, il faut la répéter en utilisant ou bien la zone déclarative du **generate** (versions récentes de VHDL) ou l'instruction **block** ici illustrée:

```
for all :composant use work.entity E(A) ;
begin
  X: composant port map… -- sera configuré correctement
  G :for I in 1 to 10 generate
    B:block
      for all :composant use work.entity E(A);
      begin
         Y: composant port map…
      end block;
  end generate;
```

block *ad hoc* pour avoir une zone déclarative et pouvoir répéter la configuration. Y sera configuré alors que sans le **block** il ne l'aurait pas été.
Si l'implémentation autorise la zone déclarative dans le **generate**, on peut enlever les deux lignes où apparaît le mot **block**.

Contrôle de la sensibilité: l'écriture naïve conduit souvent à un gaspillage de temps lors de la simulation alors que le fonctionnement est correct :

registre <= bus **when** cs'event **and** cs='1' ; sera activé pour rien, sauf optimisations, chaque fois que bus change et même si cs ne passe à 1 qu'une fois par siècle!

Solution : le process sensible seulement sur cs:

process(cs) **begin if** cs='1' **then** registre <= bus ; **end if ; end process** ;

Un driver par signal et par process: si un signal est affecté dans un **process**, même au bout d'un quart d'heure, même dans un test qui est toujours faux, ce **process** crée une contribution pour ce signal dès le temps zéro de la simulation avec la valeur par défaut ('U' pour STD_LOGIC). Par ailleurs, quand un process contribue pour un signal, cette contribution est fournie **tant que** le process ne tombe pas sur une nouvelle affectation. Cela est vrai pour l'initialisation. Il est donc prudent d'initialiser au début du process (par exemple à 'Z') tous les signaux affectés ici et là dans un process complexe

VHDL-AMS: dès que l'on suppute qu'il y aura une discontinuité sur une quantité ou sa dérivée, il faut faire un événement à cette occasion (attribut ABOVE par exemple) et un **break** sur cet événement. Sinon le simulateur, qui travaille par pas, va dépasser la discontinuité. Le **break** l'oblige à revenir au pas précédent et à recalculer une solution pour le point de la discontinuité qu'il peut interpoler, puis à recalculer des conditions "initiales".

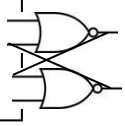
Exemples de code en vrac : (portions significatives du code seulement.)

Bascule D, comportemental

```
process(cs)
begin
if cs='1' then Q <= D ; end if;
end process;
```

Bascule RS, dataflow

```
Q <= R or not QB;
QB <= S or not Q;
```



Ligne à retard avec génération de 8 bascules D, structurel

```
component BASD port (CS,D,Q :std_logic_vector)
end component ; -- configuration à faire.
signal inter : std_logic_vector(6 downto 0 );
begin
  premier : BASD port map (cs, entree, inter(0) ) ;
  dernier : BASD port map (cs, inter(6),sortie);
  for i in 6 downto 1 generate
    elmt : BASD port map (cs, inter(i-1),inter(i));
  end generate;
```

Waveform :

```
S <= '0',
'1' after 10 ns,
'0' after 20 ns,
'1' after 100 ns,
'0' after 200 ns;
```

Horloge (attention à l'initialiser !)

```
clk <= not clk after 10 ns ;
```

Bascule JK, dataflow (formule: $Q_{+1} = J.\overline{Q} + \overline{K}.Q$)

```
Q<=(J and (not Q)) or (not K) and Q)  when clk'event and clk='1';
```

Ram, comportemental: ram est une variable, vecteur de std_logic_vector

```
process(cs) begin
if cs='1' then
   if rw='1' then ram(adresse convertie en integer):=data;
             else data<=ram(adresse convertie en integer) ;
   else data<=(others=>'Z') ;
end if;
end process;
```

Calcul par table de vérité (comportemental)

```
process (entree) begin
   case entree is
      when "000" => sortie <= "111"
      when "001" => sortie <= "101";
    etc.
      when others => null ;
end process;
```

Assertions sur conditions statiques (n) et dynamiques dans une entité

```
entity E is
generic( n :integer);
port(clk,e : bit ;s :out bit);
begin
  assert n>0 and n<10 report "n hors limites" severity fatal;
  assert not clk'stable(10 ns) report "clk trop lente";
  assert s'last_active – e'last_active > 3 ns report….
end;
```

Bloc gardé

```
B: block(clk'event and clk='1')
begin
   S<= guarded X;--équivalent à "X when clk'event etc"
   Q <= S and T; -- pas gardé
   R <= guarded V;-- voir S
end block;
```

Lire un fichier: `use std.textio.all;` puis dans un `process`:

```
file data_file:text open read_mode is "c:\bla.txt";
variable L:line;
begin
while not endfile(data_file) loop
  readline(data_file,L);
   -- ici L.all est une string. On peut s'en servir.
  read(L, x);-- suivant le type de x, lit x dans L
  …          -- voir dans TEXTIO l'existant.
```

Les deux façons d'instancier

```
component comp
  port(…)
end component ;
for all : comp use  entity work.E(A) ;
begin
instance_composant : comp port map (…)
```

```
instance_directe : use entity work.E(A) port map(…)
```

AMS: Générateur de tension

```
library Disciplines;
use Disciplines.electrical_system.all;
entity GeneTension is
  generic (valeur: REAL);
  port (terminal plus, moins: electrical);
end;
architecture A of GeneTension is
  quantity v across i through plus to moins;
  -- mention de i nécessaire pour la condition de solvabilité
begin
  v == valeur;
end;
```

AMS: Comparateur de tension, sortie signal

```
library Disciplines;
use Disciplines.electrical_system.all;
entity Comparateur is
  generic (limite: REAL);
  port (terminal T1, ref: electrical;
        signal sortie: out BOOLEAN);
end;
architecture A of Comparateur is
  quantity tension across T1 to ref;
begin
  sortie <= tension'above(limite);
end;
```

AMS: Limiteur de tension, avec break pour marquer les discontinuités

```
library Disciplines;
use Disciplines.electrical_system.all;
entity limiteur is
  generic (limite: REAL);
  port (terminal entrée_plus, entrée_moins, sortie_plus, sortie_moins: electrical);
end;
architecture A of limiteur is
  quantity tension_entree across entrée_plus to entrée_moins;
  quantity tension_sortie across courant_sortie through sortie_plus to sortie_moins;
begin
  if tension_entree'Above(limite) use
    tension_sortie == limite;
  elsif not tension_entree'Above(-limite) use
    tension_sortie == - limite;
  else
    tension_sortie == tension_entree;
  end use;
  break on tension_entree'Above(limite), tension_entree'Above(-limite);
end;
```

Solvabilité : *une* équation active à la fois, donc il faut *une* quantité libre ou de mode **out**. ou, ici, **through** même si elle n'est pas dans le code (elle est dans les équations de conservation implicites)