Écrire & comprendre VHDL & AMS

Jacques Rouillard

Écrire & Comprendre VHDL & AMS

ISBN 978-1-4092-3689-4
Imprimé en Espagne ou aux USA
(USA si indiqué en dernière page)
Lulu éditeur

Illustration de couverture:
Codex sur le vol des oiseaux
Léonard de Vinci - 1485-1490

Table des Matières

1 Clés de ce manuel

1.1 Objectif

Ce petit manuel ambitionne de présenter les différentes techniques de modélisation en VHDL, avec des exemples paradigmatiques, en présentant les concepts et techniques liés à VHDL mais en faisant délibérément l'impasse sur la présentation en détail du langage : le lecteur aura donc besoin d'une connaissance préalable ou d'un autre manuel, par exemple le compagnon de celui-ci « Lire & Comprendre VHDL & AMS qui au contraire parle du langage et non des modèles, et qui est organisé par traits du langage et non par style d'application. Un aide mémoire de 12 pages est à la fin de ce manuel, que l'on pourra découper et agrafer. Il est aussi disponible en *pdf*. (Voir bibliographie [ROU1] et [ROU2], chapitre15 page 161)

1.2 L'organisation de ce manuel

Après une présentation des questions qui transcendent tous les styles (organisation générale du langage **chapitre 2**, bibliothèques **chapitre 3**, hiérarchie **chapitre 4**), les chapitres étudient sur des exemples ad-hoc les niveaux de modélisation qu'on peut attaquer en VHDL : depuis le niveau système, avec l'exemple d'un réseau comportant une centaine de transmetteurs, jusqu'au niveau analogique où l'on s'intéresse au composant lui-même.
Les niveaux décrits sont :

- La modélisation d'éléments de bibliothèque, utilisant le standard VITAL. C'est une modélisation qui est techniquement surtout structurelle, mais qui utilise des conventions très serrées permettant de rétro-annoter les modèles après synthèse, c'est-à-dire d'y insérer des délais de transmission très proches de la réalité, sans avoir à ouvrir les modèles qui se servent de ces briques. **Chapitre 5.**
- Système : niveau où l'on s'intéresse à des abstractions, le temps n'est intéressant qu'en termes de causalité. On simule des systèmes qui représenteront à la fin une très grande quantité de matériel et de logiciel, on fait des statistiques et on envoie des vecteurs de test basés sur des séquences aléatoires. La simulation n'est pas déterministe. **Chapitre 6.**
- Comportemental : on valide des algorithmes, dont on peut ne pas savoir encore s'ils seront implémentés en logiciel ou matériel. On peut aussi écrire des briques qu'on ne réalisera jamais –elles existent déjà ou sont l'objet d'un contrat-, simplement pour faire fonctionner le reste du modèle. **Chapitre 7.**
- Synchrone : le temps est mesuré en termes de coups d'horloge. Le circuit est un assemblage de blocs ayant la même horloge ; C'est un niveau extrêmement efficace en termes de simulation. **Chapitre 8.**
- Asynchrone : le circuit est décrit, *in fine*, en termes d'équations logiques et le plus généralement les objets manipulés sont de type binaire ou entier se résolvant en codage binaire. On y traite aussi des questions de protocole qui sont en général asynchrones et sans horloge. **Chapitre 9.**
- Switch : c'est un niveau un peu bâtard et anecdotique, qui permet de faire de la logique tout en s'intéressant aux questions de force électrique. Le lecteur intéressé par le cycle de simulation pourra y trouver d'intéressants sujets de réflexion. **Chapitre 10.**
- Analogique : le temps est continu et tout se résout en équations différentielles. VHDL-AMS est une extension de VHDL, et les modèles écrits en AMS sont mixtes. **Chapitre 11.**

1.3 Les exemples de ce manuel

Ils sont là seulement pour exemplifier des constructions et des méthodes, voire des difficultés du langage. Ils sont choisis délibérément très courts, sont en général beaucoup trop simples pour être directement utilisables dans un contexte réel ; et ils ne constituent en aucun cas des projets complets, par contre ils sont fonctionnels et compilables.

Ce manuel n'est donc pas un recueil de modèles, chose qu'on trouvera par ailleurs facilement et en particulier sur Internet. Par ailleurs les exemples donnés sont corrects à la connaissance de l'auteur, mais aucune garantie ni engagement ne sont donnés à ce sujet.

Une mine d'exemples et de modères « réels » seront particulièrement trouvés sur le site : http://tams-www.informatik.uni-hamburg.de/vhdl/ , voir [HAM] en bibliographie chapitre 15 page 161.

1.4 Les paquetages en annexe

Les paquetages VHDL donnés en annexe chapitre 12 à partir de la page 115 ne sont pas *tous* les paquetages standardisés, ni *tous* les paquetages libres qui sont dans la nature, mais ceux dont il est raisonnable de faire usage selon l'avis subjectif de l'auteur. Les questions de copyright étant complexes, les versions données ici sont, à la connaissance de l'auteur, ou bien des versions publiques, ou bien les dernières versions de travail publiques utilisées par les comités en principe identiques aux versions finales, aux entêtes près. Dans le cas des paquetages VITAL, les commentaires en ont été retirés car ils triplaient le volume de texte. Le lecteur intéressé par lesdits paquetages in extenso, ou par un paquetage public rare ou exotique le trouvera probablement sur Internet. Certains sont dans le livre compagnon « Lire & Comprendre VHDL & AMS » ; voir la bibliographie [ROU1] chapitre 15 page 161.

1.5 Les outils

Les outils utilisés pour vérifier les exemples de ce manuel sont, pour la partie digitale, les outils disponibles à cette date dans leur version « éducation » de *Mentor Graphics* (*ModelSim*), et pour les modèles mixtes et analogiques, par ordre alphabétique, ceux de *Ansoft* (*Simplorer*), *Dolphin* (*Smash*), et *Mentor-Graphics* (*SimpliVision*).

Quelques différences d'interprétation ou de couverture du langage peuvent nécessiter des adaptations s'ils sont utilisés sur d'autres plateformes que celles où ils ont été testés. À titre indicatif et sans prétendre à l'exhaustivité, on pourra avoir des différences

- sur les caractères accentués : ils sont autorisés mais pas depuis le début de VHDL, certains systèmes ont gardé l'interdiction.
- sur la disponibilité des paquetages non-standard. Ceux mentionnés ici sont tous libres d'accès sur Internet, mais ne sont pas tous des standards IEEE. La solution est simple s'il vient à manquer sur votre plateforme : il suffit de récupérer le paquetage et de le compiler dans la bonne bibliothèque.
- sur le nom des bibliothèques pour l'analogique : DISCIPLINES ou IEEE. Il suffit de modifier la bibliothèque appelée et la clause « use » qui va probablement avec dans le modèle.
- sur les traits coûteux du langage dans certains simulateurs analogiques (types composites de natures, **generate**, **procedural**, **break** dans toute sa fonctionnalité, agrégats en cible d'affectation, …). L'absence de tableaux de natures et d'instruction **generate** est une vraie limitation qui obligera le concepteur à dérouler « à la main »

les régularités pour chaque configuration possible. C'est surtout handicapant en conception mixte digital/analogique, car les modèles disponibles dans le monde digital qu'on peut avoir envie de réutiliser pour « analogiser » un bloc peuvent faire largement usage des structures et de la génération.

1.6 Méthodologies usuelles

1.6.1 Conception descendante (*Top-Down)*

C'est la méthodologie la plus employée dans le cadre commercial le plus courant : un client veut un circuit spécifique, un ASIC (*Application Specific IC*) pour une application, par exemple un DSP spécifique pour un téléphone. Il commande la chose à une maison de conception (*Design-House*) qui est en général une structure moyenne, ou un petit département d'une grosse entreprise, travaillant sur CAO.
La conception se fait alors des spécifications vers le circuit, la validation étant sous le contrôle du client. Les styles employés vont du niveau système au niveau flot-de-données. Dans ce cadre il faut un accord tripartite entre le client, la maison de conception et le fondeur : le *sign-off*. Ce contrat est signé entre un vendeur de CAO qui garantit la qualité de ses outils, le fondeur qui garantit que la puce qui sortira sera conforme aux simulations de l'outil de CAO et le client qui paie. S'il y a un souci, c'est le vendeur de CAO et le fondeur qui vont s'expliquer pour que la sortie de l'usine soit conforme aux prévisions de la CAO. Le contrat fait en sorte que « si ça ne marche pas », ce ne soit pas le client final qui paie la rectification. (Il n'en aurait pas les moyens le plus souvent, vu les coûts de production et la taille des maisons de conception).
Les métiers impliqués sont pour l'essentiel des métiers d'ingénieur de conception « logique », qui connaît le client, a une bonne idée de l'application et ignore la technologie.

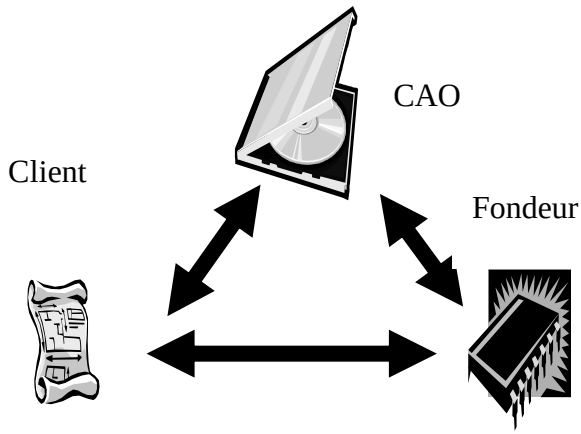


Figure 1 Le Sign-Off

1.6.2 Conception montante (*Bottom-Up)*

C'est le cas de figure où un fournisseur cherche à faire un produit « d'étagère »; le meilleur processeur possible pour tel créneau, un circuit spécialisé générique (ex contrôleur d'écran, de disque, décodeur Mpeg...), etc. Le développement se fait en interne avec des spécifications souples que l'on peut remettre en question plus facilement que si l'on avait un contrat à amender. C'est le marché qui valide ensuite.
Aujourd'hui de tels produits sont proposés sous la forme d'IPs (*Intellectual Property*) sous forme entièrement synthétisable multi-cible, ce sont donc des textes VHDL par exemple.
Les professionnels sont des ingénieurs de conception « logique » ou « analogique », qui ignorent le client final et ont une idée des applications finales. La technologie peut ou peut ne pas être un paramètre important de la question.

1.6.3 Conception montante-descendante (*Meet-In-The-Middle)*

Un fondeur doit développer pour chaque technologie une bibliothèque d'éléments de moyen ou bas-niveau, ou extrêmement génériques, sur laquelle les outils de CAO et en particulier les outils de synthèse vont replier les circuits conçus par les maisons de conception (*Design-*

Houses). Ces éléments, demandés par les outils de synthèse, sont conçus comme des briques mais sont eux-mêmes des assemblages de briques, d'où le qualificatif.
Les spécifications sont reconfigurables, « au mieux » de ce qu'on sait faire. Le fondeur maîtrise la technologie et peut donc violer en connaissance de cause les règles qu'il impose à ses clients (paradigme: la cellule de Ram). Il utilisera massivement les paquetages VITAL (chapitre 5, page 37).
La validation est interne ou soumise à un *sign-off* (voir ci-dessus) avec proposition du produit sur « catalogue».
Les ingénieurs impliqués sont des ingénieurs de conception plutôt « analogique » qui ignorent le client final et peuvent ignorer les applications finales ; ils maîtrisent bien la technologie.

1.7 Les versions des langages et de l'environnement

1.7.1 Les versions des langages

VHDL existe en plusieurs versions dans la nature : les règlements IEEE obligent à revisiter les standards tous les 5 ans –souvent un retard est observé mais il doit être dûment autorisé par les IEEE. Le numéro du standard est 1076. On a ainsi :

1.7.1.1 VHDL 7.2

C'est l'ancêtre du standard dans sa version 7.2 (il y a donc eu des versions antérieures) qui était classifié « défense » et qui ne doit plus guère exister. Ce langage était lui-même un sous-produit d'un énorme projet militaire « VHSIC » (*Very High Speed Integrated Circuits*) vers 1983, d'où le nom de VHDL (*VHSIC Hardware Description Langage*), le « langage de description de matériel du projet VHSIC », c'est donc un acronyme d'acronymes.
Le même DoD avait créé de toutes pièces le langage Ada, et à cette époque Ada influençait tous les développements logiciels militaires américains. La dernière implémentation de VHDL 7.2 (connue de l'auteur) était écrite en Ada et produisait de l'Ada qu'il fallait encore recompiler, ce qui n'était pas rapide –euphémisme-. Elle a servi de base au prototype du premier compilateur VHDL 1076.

1.7.1.2 VHDL 1076-1987

C'est le standard IEEE initial, standard créé de toutes pièces sur la base du précédent par décision du DoD (ministère de la défense américain) qui a mis VHDL7.2 dans le domaine public en 1985, a payé pendant plusieurs années des sociétés –principalement CLSI- pour envoyer des ingénieurs aux comités IEEE, transformer VHDL 7.2 en standard, faire développer un prototype et gérer le vote initial.

1.7.1.3 VHDL 1076-1991 (Interprétations)

Le standard sous la forme "papier" contenait quelques bogues bloquantes pour le développement des premiers compilateurs, par exemple la visibilité des arguments génériques dans la déclaration des ports qui suivait immédiatement n'était pas assurée. Ainsi qu'il est expliqué au §1.7.3 ci-dessous, un comité technique (ISAC) chargé de la maintenance du langage émettait des interprétations validées par le VASG. Celles-ci sont rassemblées dans ce document qui n'est pas une nouvelle version du standard, mais la « vue du comité », (*Sense of the VASG*) sur le traitement de ces anomalies.

1.7.1.4 VHDL 1076-1993

La première révision qui propose des modifications majeures. Cette révision a rassemblé toutes les corrections importantes nécessaires ou désirées par la première génération

d'utilisateurs et de développeurs, en particulier celles du document précédent. On y trouve aussi les variables partagées, une simplification de la gestion de fichiers, les opérateurs de rotation et de décalage, les identificateurs étendus, et bien d'autres nouvelles fonctionnalités ou réparations. C'est cette version qui fait office de référence pour la plupart des outils existants.

1.7.1.5 VHDL-AMS 1076.1-1999

Le standard strict sur-ensemble de VHDL 1993, permettant de traiter l'analogique. Son utilisation est traitée dans le chapitre 11 page 97. En principe, la seule source d'incompatibilité entre VHDL 1076-1993 et VHDL 1076.1-1999 (dans le sens 1076 vers 1076.1) est l'existence de mots réservés supplémentaires. En pratique, les sociétés fournissant des simulateurs analogiques « adaptés » à VHDL.1 font parfois l'impasse sur les traits les plus coûteux du langage.

1.7.1.6 VHDL 1076-2000

Version pratiquement dédiée aux variables partagées, dont l'introduction en 1993 avait fait l'objet d'une guerre de tranchées entre membres des comités et d'un emballement de passions dont seuls sont capables les concepteurs de langages[1]. On y trouve l'introduction des types protégés, les variables partagées doivent maintenant être de ces types. Ceci permettant une paix des braves et l'accès « propre » à de la mémoire partagée dans le cas de simulateurs multi-processeurs.

1.7.1.7 VHDL 1076-2002

Changement d'utilisations des ports de mode buffer et quelques révisions mineures.

1.7.1.8 VHDL-AMS 1076.1-2007

Révisions mineures de VHDL 1076.1-1999, pour ce qui est de la partie analogique du standard, prenant par contre en compte toutes les modifications faites à VHDL 1076 en 2000 et 2002.

1.7.1.9 VHDL 1076-2008

VHDL 2008 est le projet qui vient d'être voté à cette date, qui est une révision majeure et dont aucun outil n'existe encore, voir [ASH2] en bibliographie chapitre 15 page 161. Son existence va automatiquement entraîner une nouvelle version du standard analogique VHDL-AMS 1076.1 dans un avenir proche, puisque ce dernier est défini comme étant un sur-ensemble de VHDL 1076.
Il contient entre autres suppléments

- un langage d'assertions[2] (PSL),
- une méthode pour crypter des portions de la description, de façon à ce que les outils puissent la traiter mais sans que le client ne puisse la lire,
- une interface procédurale permettant d'écrire des outils qui s'interfacent avec les simulateurs,

[1] En effet, la variable partagée casse une propriété essentielle de VHDL: elle permet de rendre l'exécution dépendante de l'ordre d'exécution des processus. À son actif, elle est indispensable pour modéliser des phénomènes stochastiques.

[2] Un langage d'assertion permet d'exprimer des vérifications de propriétés qui doivent être toujours vraies. Par exemple « jamais tel signal et tel autre signal ne sont ensemble à un sur le front montant de l'horloge ». L'instruction « assert » de VHDL, les processus retardés (postponed) et la possibilité d'écrire des processus passifs (n'affectant aucun signal) dans la déclaration d'entité sont déjà, en VHDL 1993, un embryon de langage d'assertions.

- la possibilité de rendre un processus sensible sur tous les signaux qui sont dedans, pour être sûr de ne pas en oublier et aller gaiement à la synthèse: **`process(all)`**;
- la possibilité de mettre des expressions dans les **`port map`**, (s'il s'agit d'expressions de signaux);
- **`force`** et **`release`** de signaux, permettant de coller un signal à une valeur indépendamment du reste de la description;
- la possibilité d'utiliser des types comme arguments génériques (ceci est bien commode, par exemple décrire un multiplexeur ne devrait pas exiger qu'on connaisse le type des données qui circulent). Et la possibilité de faire des paquetages génériques (pour, par exemple, développer des paquetages arithmétiques indépendants du nombre de bits de l'instance). Pour illustration, cette limitation actuelle doit être contournée dans l'exemple décrit en §7.1.4.2.2 page 62;
- la possibilité d'avoir des tableaux et des enregistrements de types non-contraints. Les versions antérieures ne permettent pas de déclarer un tableau "**`array`** `(integer` **`range`** `1` **`to`** `5)` **`of`** `STRING`": il est possible avant 2008 de déclarer un tableau non contraint, mais pas ses éléments qui doivent être contraints;
- diverses simplifications comme la gestion du *don't care* dans les instructions **case**? (une variante du **case**). Ainsi une clause **`when`** `"01---"` couvre toutes les combinaisons de 5 bits commençant par 01;
- l'utilisation de types comme STD_LOGIC en place de booléen (**`if`** `S1` **`and`** `S2` à la place de **`if`** `S1='1'` **`and`** `S2='1'`);
- les expressions conditionnelles (affectation conditionnelle comme `S <= A` **`when`** `condition` **`else`** `B;` autorisée en contexte séquentiel);
- les opérateurs de réduction (`S <=` **`and`** `T;` où T est un tableau de bits, devient équivalent à `S <= T(0)` **`and`** `T(1)` **`and`** *`etc.`*);
- de nouveaux opérateurs, préfixés par un "?" permettant des comparaisons "logiques", par exemple dans le type STD_LOGIC la comparaison entre *un* fort et *un* faible `"'1' ?= 'H'"` rendra '1';
- différents paquetages mathématiques implémentant l'arithmétique sur des représentations normalisées de flottant à virgule fixe ou à précision relative;
- des primitives permettant d'avoir une vue textuelle de tous les types (fonctionnalité de l'attribut IMAGE existant);
- des paquetages mathématiques et la façon de les utiliser;
- …et bien d'autres choses.
- La liste des mots-réservés est de plus en plus importante, et le concepteur utilisant une version antérieure devrait, dans la limite du raisonnable, éviter d'employer des identificateurs qui entreront fatalement en collision avec des mots-clés des futures versions. Cette liste est augmentée de: **`assume assume_guarantee context cover fairness force parameter property release restrict restrict_guarantee sequence strong vmode vprop vunit`**

1.7.1.10 Portabilité entre versions

Ceci implique que l'on doit souvent se préoccuper de régler la version du langage utilisée. Rassurons nous, à part quelques traits rarement utilisés (types protégés), la question se résume le plus souvent, en attendant les implémentations de VHDL 2008, à choisir entre la version 87 et une autre, essentiellement à cause de la gestion de fichiers qui est incompatible. Parfois un mot-clé apparu dans une version ultérieure pourra entrer en collision avec un nom déclaré dans la version antérieure (**shared** par exemple).

On mentionnera donc ici, à l'occasion, des limitations de la version originale antérieure à VHDL'93. Même si 1993 peut sembler déjà bien loin pour qu'on se préoccupe encore de 1987, il ne faut pas perdre de vue que c'est simplement la date de la standardisation, les outils sont apparus plus tard et, notamment en synthèse où l'on travaille de toutes façons avec des sous-ensemble du langage, courir après les modifications souvent cosmétiques n'est pas la priorité des développeurs d'outils. Bien des outils contiennent encore des traits du langage antérieurs à VHDL'93.

1.7.2 Les types logiques

Parallèlement aux travaux sur le langage se tenaient des travaux sur la définition de types logiques. VHDL est le premier et le seul langage de modélisation qui permette, *dans le langage*, de définir une logique (0, 1, X, etc), ses opérateurs (**and**, **or**, etc.) et son comportement en cas de conflit (fonctions de résolution, notion de haute impédance, niveaux forts et faibles). Les autres langages proposent un type logique supposé suffisant pour décrire des circuits électroniques, fournissant grosso-modo les fonctionnalités de STD_LOGIC_1164.

Ceci a entraîné trois graves inconvénients :

- Chaque ingénieur ayant son idée sur ce qu'est le meilleur type logique, une profusion de paquetages se sont trouvés sur le marché, provenant d'universitaires ou suggérés par les vendeurs de CAO, chacun défendant mordicus son idée, ce qui tuait partiellement un des bénéfices de VHDL : sa portabilité.
- Les langages concurrents furent vus comme « plus simples » de ce fait.
- Les vendeurs de CAO VHDL ne pouvaient pas optimiser leurs outils pour un type particulier.

Ainsi, pour mémoire, a-t-on vu les paquetages logiques suivants, entre multiples autres.

1.7.2.1 Deux états

C'est le paquetage STD.STANDARD, qui propose la logique à deux états '0' et '1', avec le sens de logique positive, et où les conflits sont interdits.

1.7.2.2 Quatre états

'0', '1','Z','X' : les deux états '0' et '1' ont leur sens évident, 'Z' est la haute impédance et permet d'écrire des fonctions de résolution gérant des bus. 'X' représente un conflit entre '0' et '1', ou, selon les implémentations, entre deux valeurs quelconques autres que 'Z'. En effet, on peut décréter raisonnablement qu'il est anormal que deux portes logiques écrivent sur le même fil, y compris si c'est la même valeur.

1.7.2.3 Six états

Les valeurs 0 et 1 du précédent étant distinguées en 0 faible, 0 fort, 1 faible et 1 fort, ceci permettant la modélisation de *pull-ups* et de bus dits « à drain ouvert ».

1.7.2.4 Neuf états

L'ancêtre du standard STD_LOGIC_1164, alors appelé MVL-9 (*multi-valued-logic*) et qui fut « donné » aux IEEE par le vendeur –*Synopsys*- qui l'avait développé. Voir ci-dessous.

1.7.2.5 Quarante six états

Dans ce système défendu par *Coelho* (Voir [COE] en bibliographie chapitre 15 page 161) et promu par la feue société *Vantage*, les quarante six états n'étaient pas les valeurs mais des possibilités de valeurs : ils étaient l'ensemble des combinaisons contigües de neuf valeurs

possibles : (F) zéro forcé, (R) zéro fort, (W) zéro faible, (Z) zéro haute impédance[3], (D) déconnection, et de même un haute impédance, un faible, un fort, un forcé. Ainsi l'état WD0, valeur 0 à cheval sur W et D, représentait la possibilité pour le signal d'avoir une valeur zéro faible ou zéro haute impédance. Là-dessus toute la logique pouvait être définie, ainsi que les fonctions de résolution; ce système était particulièrement adapté à la modélisation en niveau switch (voir chapitre 10 page 95 pour une vue de la question utilisant STD_LOGIC_1164).

1.7.2.6 Le paquetage STD_LOGIC_ 1164

Pour mettre un terme à ce désordre, le paquetage STD_LOGIC_1164 a été normalisé. On en trouvera la déclaration dans les références, §12.2.1 page 119. C'est un paquetage à neuf valeurs, distinguant 0, 1 et conflits forts et faibles, un état de haute impédance, un état 'U' disant que l'objet n'a pas été initialisé, et un état '-' utilisé, avant VHDL 2008, uniquement en synthèse pour dire « don't care ». En VHDL 2008, le '-' peut être utilisé dans une instruction case pour couvrir tous les cas de figure correspondants, voir §1.7.1.9 ci-dessus.
Depuis l'arrivée de ce standard « libérateur », rarissimes sont les concepteurs qui ne s'en servent pas. La possibilité de définir soi-même sa propre logique et ses propres fonctions résolution est aujourd'hui réservée aux concepteurs systèmes (voir 6.2 page 50 pour une fonction de résolution), aux testeurs d'idées en logique floue et autres originaux... Néanmoins la fonctionnalité existe.

1.7.2.7 Interopérabilité

Certainement d'autres types logiques hantent encore des modèles à maintenir. La possibilité d'écrire ses propres types logiques et les questions d'interopérabilité avait été réglée dans le standard initial par la possibilité d'appeler des fonctions de conversion lors des « **port map** », au vol. Ainsi il est possible d'instancier un composant utilisant 46 états dans un modèle qui n'en a que 9, pour autant qu'on dispose des fonctions de conversions dans les deux sens, en écrivant quelque chose comme:

```
port map (to_46_values(p46) => to_std_logic(pstd))
```

Ici `to_46_values` est une fonction qui prend un élément de type STD_LOGIC et propose une valeur dans le type à 46 valeurs. `to_std_logic` fait l'inverse. P46 et PSTD sont respectivement des objets de type à 46 ou 9 valeurs.
Ceci est évidemment un pis-aller, puisque la conversion va forcément perdre de l'information. De plus, l'appel de ces fonctions par le simulateur doit être fait au gré des événements et en fonction des modes (in, out, inout), et cela devient rapidement une usine à gaz. Depuis l'arrivée du paquetage STD_LOGIC_1164, cette fonctionnalité est rarement utilisée. Hélas, c'est une aventure qui peut arriver au concepteur s'il doit récupérer un ancien modèle. Ce sujet est traité §4.2.5 page 34.

1.7.3 La maintenance du langage et des paquetages

VHDL comme bien d'autres standards est un standard vivant, certains comités ont la charge de corriger les défauts ou bogues du manuel de référence, et aussi de rassembler les demandes de modifications pour la révision suivante. Le résultat du travail de ces comités peut prendre plusieurs formes :

- en cas de bogue bloquante, une recommandation est émise et applicable immédiatement. C'est arrivé par exemple quand on s'est rendu compte de certains problèmes de visibilités incorrectement décrits dans le LRM et qui bloquaient directement le développement des compilateurs, voir le résultat au §1.7.1.3 ci-dessus. Ce n'est plus arrivé depuis longtemps.

[3] Cela signifie que la valeur est zéro par conservation capacitive de la valeur précédente.

- en cas d'erreur non bloquante, la recommandation est mise en conserve pour inclusion dans la prochaine révision. Par exemple le fait que la fonction ENDLINE du paquetage TEXTIO initial ait été illégale en VHDL (son argument était de classe **variable**) n'a été découvert qu'après la première standardisation[4]. Cette fonction a été supprimée. On peut donc la retrouver dans d'anciens modèles, les compilateurs de l'époque ayant dû faire une verrue pour cette fonction.
- en cas de trait inesthétique ou mal bâti, une discussion est engagée pour la résolution qui est incluse dans la prochaine révision. La notion de « bit string » a ainsi été étendue à des types non prédéfinis afin de permettre l'utilisation des notations octale et hexadécimale pour les littéraux de std_logic_vector. Ce n'était possible que pour bit_vector en VHDL'87.
- en cas de demande de nouvelle fonctionnalité, tous les cinq ans –à peu près- un appel à propositions est publié, une compilation en est faite, des propositions sont émises et un vote a lieu qui doit enregistrer une super-majorité. Ceux qui votent *non* doivent dire pourquoi, et dire ce qui les ferait voter *oui* ; le comité doit tenter de répondre à tous les votes négatifs et tenter de les faire changer d'avis. Ce système a donc assez peu à voir avec un scrutin politique.

Le comité « mère » de tous les autres, le VASG (*VHDL Analysis & Standardization Group*) et son organe technique ISAC (*Issue Screening & Analysis Committee*), qui s'occupent de VHDL en tant que langage et dont les travaux ont un effet sur tous les standards dérivés comme VHDL-AMS (l'analogique, voir chapitre 11 page 97) ; et même les standards « en VHDL » comme STD_LOGIC-1164 (§12.2.1 page 119), WAVES[5] ou VITAL (§12.3 page 125), etc. Ces comités peuvent être contacté par n'importe qui et les sites, à cette date, sont : http://www.vhdl.org/vasg/ et http://www.vhdl.org/isac/. Les industriels et les clients peuvent anticiper les modifications en consultant les documents de ces comités qui décrivent ce qui sera probablement soumis au vote de la prochaine révision.
Cette activité étant le fait de volontaires (soit à titre personnel, soit mandatés par leur société mais nominalement, il n'y a pas de siège appartenant à une société), il ne faut donc pas s'attendre à une réponse immédiatement efficace. Par contre n'importe qui peut se porter volontaire pour aller dans les réunions et prendre sa part de travail.
Pour pouvoir voter sur le standard, il faut être membre IEEE.

Le langage VHDL en tant que tel est vu en détail dans le manuel compagnon de celui-ci « Lire & Comprendre VHDL & AMS » (voir bibliographie [ROU1], chapitre15 page 161) où l'on tente de couvrir tous les traits du langage, car on est souvent amené à lire et tenter de comprendre ce qu'on n'écrirait jamais.

Le passionné ou concepteur d'outils aura besoin du manuel ultime, le LRM (*Language Reference Manual*, (voir la bibliographie [IE1] chapitre 15 page 161), mais insistons sur le fait que ce LRM est d'une absolue inutilité pour le concepteur normal : il contient la description du langage à l'usage des développeurs de compilateurs.

[4] Ce qui montre bien que, contrairement à la plupart des langages où c'est l'implémentation du prototype qui tient lieu de première spécification, les premiers compilateurs VHDL sont sortis *après* la publication du manuel de référence, cette erreur ayant été signalée tout à fait normalement par un des premiers compilateurs.
[5] WAVES est un standard « en VHDL » qui permet de spécifier des vecteurs de test pour un modèle, les résultats attendus et d'automatiser le test d'une façon qui est compatible avec le testeur qui sera utilisé pour tester le vrai circuit. Il n'est pas couvert par ce manuel. Voir Bibliographie [IE5] chapitre 15 page 163.

2 Quoi, où, zones déclaratives, zones d'instructions

D'une façon générale, la plupart des constructions comprennent une partie déclarative et une partie instructions. Il y a un jeu de déclarations qui sont admissibles dans toutes les parties déclaratives, et d'autres qui dépendent du domaine où l'on se trouve (concurrent, séquentiel).

Les déclarations qui sont universellement acceptées sont :

- Les déclarations qui contrôlent la visibilité : clauses **use**, **alias**
- Les déclarations de type, sous-types, constantes, fichiers.
- Les déclarations et spécifications d'attributs, de groupes (usage rare).
- Les spécifications (seulement) de sous-programmes – les corps de sous-programmes sont aussi acceptés partout sauf dans les déclarations de paquetage et les spécifications de types protégés.

Nous les appellerons ci-dessous *déclarations générales* pour éviter d'avoir à répéter ce paquet.

2.1 Les unités de compilation

2.1.1 La déclaration d'entité

C'est une des cinq unités (une chose compilable en tant que telle) ; on peut la préfixer par des appels de bibliothèques (clause **library**) et des clauses **use**. La déclaration d'entité peut être vide : tout ce qui suit est optionnel.

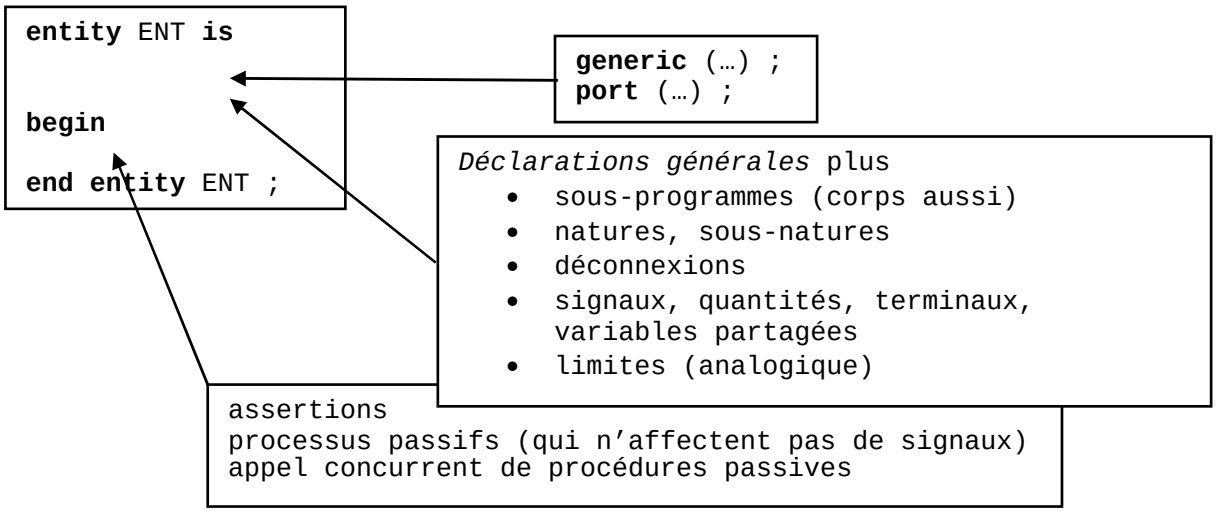


Figure 2 Contenu d'une entité

La déclaration d'entité contient le plus généralement ses spécifications d'interface : les arguments génériques, et les ports. Ce sont les moyens qu'on a de configurer l'entité et de l'appeler depuis l'extérieur.

Mais l'entité peut aussi contenir une quantité de déclarations qui seront vues par toutes ses architectures : des instructions de visibilité (clause use) ou des déclarations de types, sous-types, même des objets comme des signaux, des quantités ou des variables partagées.

Après le **begin** (optionnel donc), la déclaration d'entité peut contenir des instructions, pour autant que ces instructions ne créent pas d'événements et ne participent pas à la simulation : ce sont les assertions (plus utiles dans leur variante retardée - *postponed*) et les processus passifs –qui n'affectent as de signaux- dont le seul effet est de créer des traces à la console ou de remplir des fichiers.

2.1.2 Le corps d'architecture

C'est une des cinq unités (une chose compilable en tant que telle) ; on peut la préfixer par des appels de bibliothèques (clause **library**) et des clauses **use**. Le corps d'architecture a lui aussi une partie déclarative et une partie instructions, qui peuvent être vides mais séparées par le mot-clé **begin** qui n'est pas ici optionnel.

La partie déclarative peut contenir, en sus des déclarations générales, celles qui sont spécifiques au domaine concurrent : ce qui a trait au composant (déclaration, configuration), les objets « concurrents » : signaux, quantités, variables partagées et ce qui a trait aux signaux (déconnexion des signaux gardés).

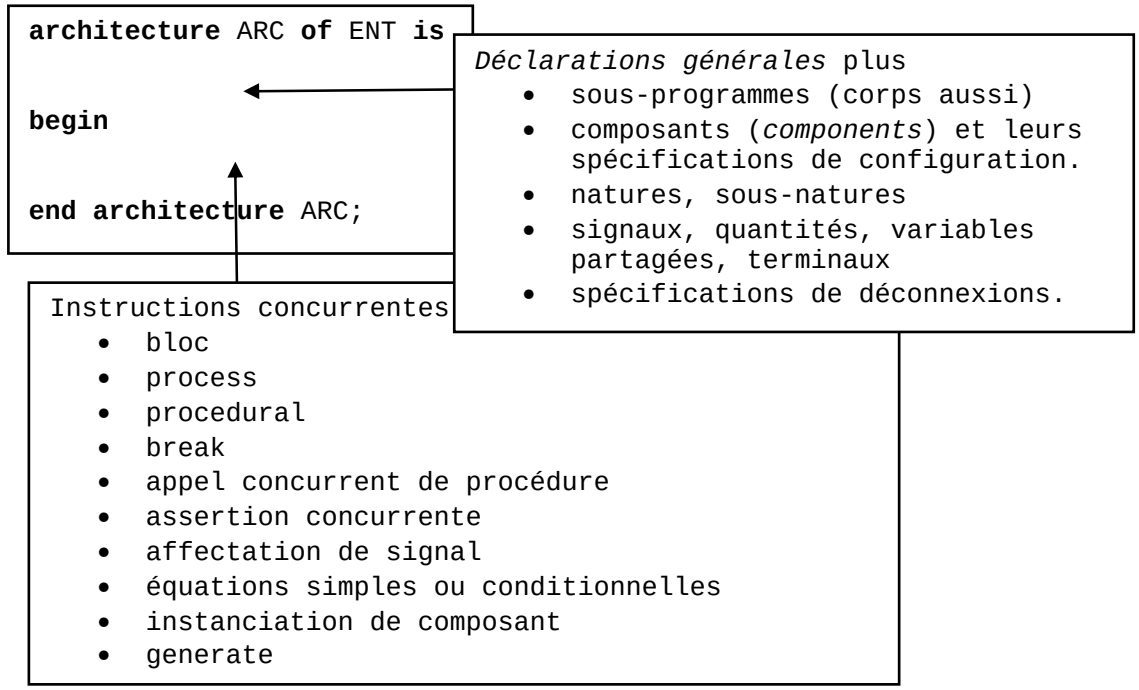


Figure 3 Contenu d'une architecture

La partie instruction ne contient que des instructions concurrentes (c'est-à-dire que leur ordre d'écriture respectif n'a pas de sens pour ce qui est de la description).

2.1.3 La déclaration de paquetage

C'est une des cinq unités (une chose compilable en tant que telle) ; on peut la préfixer par des appels de bibliothèques (clause **library**) et des clauses **use**. C'est une zone purement déclarative, le seul endroit où l'on ait le droit d'écrire une déclaration de sous-programme sans pouvoir en écrire le corps. Comme le paquetage peut être appelé depuis une entité ou une architecture, c'est une zone déclarative concurrente, et on peut y mettre :

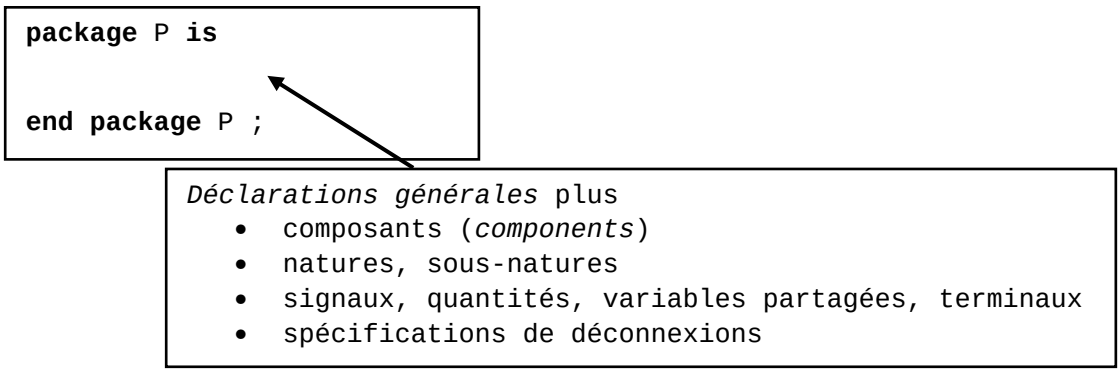


Figure 4 Contenu d'une déclaration de paquetage

2.1.4 Le corps de paquetage

C'est une des cinq unités (une chose compilable en tant que telle). Le corps de paquetage est simplement censé implémenter sa propre déclaration. On y trouve bien entendu toutes les déclarations générales plus les corps de sous-programme (qu'on ne pouvait pas mettre dans la déclaration). Des variables partagées peuvent être déclarées et utilisées par les sous-programmes du paquetage. Mais pas de signaux, de terminaux ou de quantités dont on ne saurait que faire.

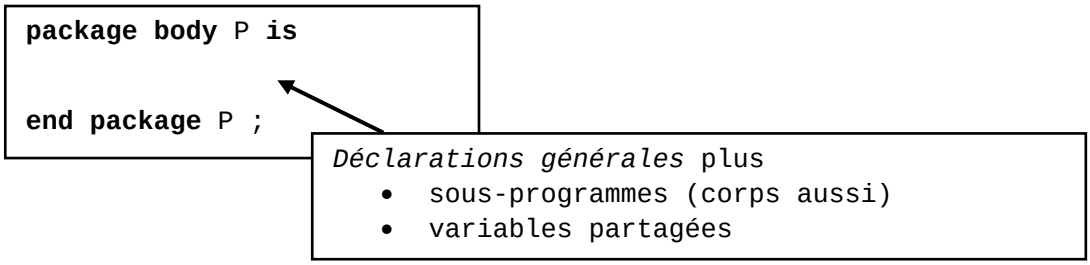


Figure 5 Contenu d'un corps de paquetage

2.1.5 La déclaration de configuration

C'est une des cinq unités (une chose compilable en tant que telle) ; on peut la préfixer par des appels de bibliothèques (clause **library**) et des clauses **use**. C'est une construction qui permet d'associer, de l'extérieur, chaque composant d'une architecture à un couple entité/architecture ou à une autre configuration, et ceci transitivement. À chaque étage, on ouvre la visibilité dans le bloc ou l'architecture ouverte, réglée éventuellement par des clauses use ; et soit on donne le nom (label) d'une architecture ou d'un bloc de niveau inférieur, et c'est reparti pour une configuration, soit on donne le nom (label) d'un composant et la transitivité se termine ici.

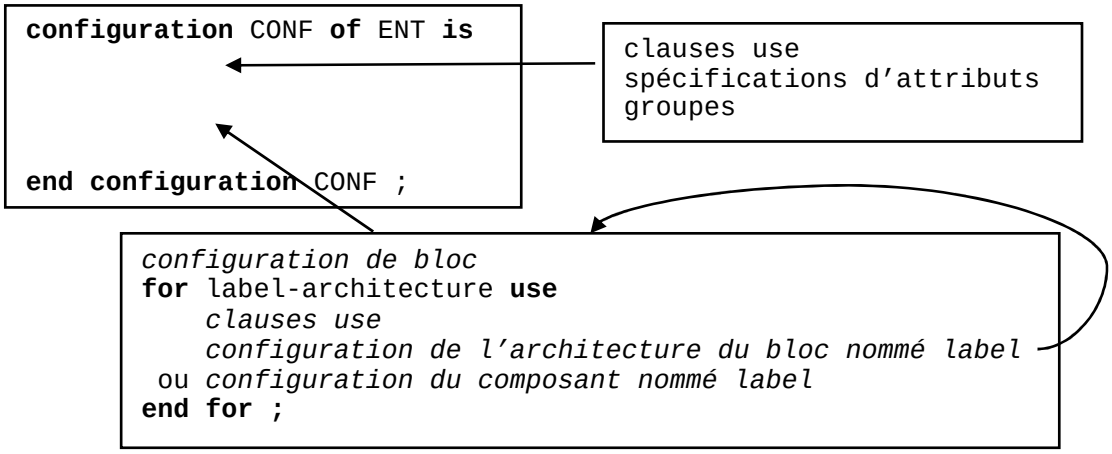


Figure 6 Contenu d'une déclaration de configuration

2.2 Les instructions à déclarations locales

2.2.1 Le bloc

Le bloc est une des instructions qui peut apparaître dans l'architecture ; il est un usage très restreint en VHDL, et il n'est mentionné ici que pour la complétude : il ne sert qu'à créer une zone déclarative quand on en a besoin dans une instruction **generate**, ou à utiliser « la garde », trait de VHDL qui tombe en désuétude (voir §8.4 page 83). Le bloc à lui tout seul récapitule les fonctionnalités de l'entité et de l'architecture, il fait une découpe arbitraire dans le code en nommant les fils de l'interface.
Il a des arguments génériques et des ports, que l'on configure sur le champ avec des **generic map** et **port map**. L'intérêt de cette fonctionnalité surprenante est que la sémantique de l'instanciation est entièrement décrite en blocs équivalents[6], et que cela permet de décrire de multiples instances de la même entité/architecture comme des duplications de code ; le passage aux signaux réels étant fait au fil de l'eau par les **port map**.

On voit sur la figure que le bloc peut contenir d'autres blocs. L'architecture elle-même n'est finalement qu'un bloc de premier niveau, les déclarations et instructions qu'on peut y mettre sont les mêmes.

[6] Ce qui permet de simplifier le manuel de référence, qui explicite seulement la sémantique du bloc; toutes les fonctions d'instanciation se ramenant à des constructions lexicalement équivalentes.

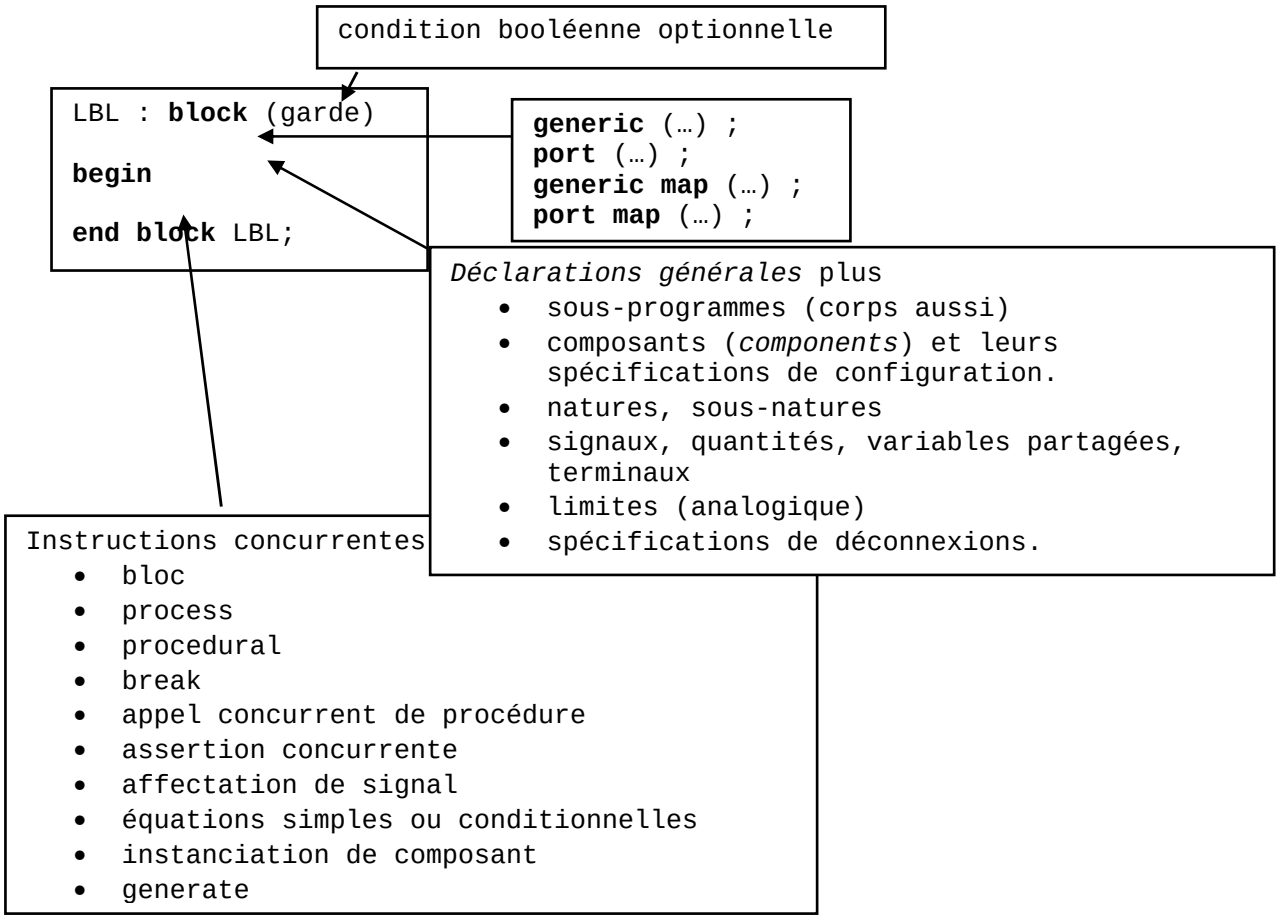


Figure 7 Contenu d'un bloc

2.2.2 Le processus

Le processus (*process*) est une des instructions concurrentes. À ce titre, sa position dans le code vis-à-vis des autres instructions concurrentes n'a pas d'importance.
Le processus contient des instructions séquentielles. Ces instructions là, à l'intérieur des zones séquentielles, sont exécutées dans l'ordre qui a donc de l'importance. Le processus peut être retardé (*postponed*) ce qui permet d'être assuré qu'il sera exécuté une fois et une seule par temps où il est réveillé, quand tous les signaux sont stables. Évidemment la condition de validité est qu'il n'affecte pas de signal avec un délai nul.
La liste de sensibilité est une liste de signaux dont le changement de valeur va réveiller le processus. Sa présence empêche de mettre des **wait** explicites dans le processus. Cette forme est préférée des outils de synthèse, bien que (ou car) plus contraignante.

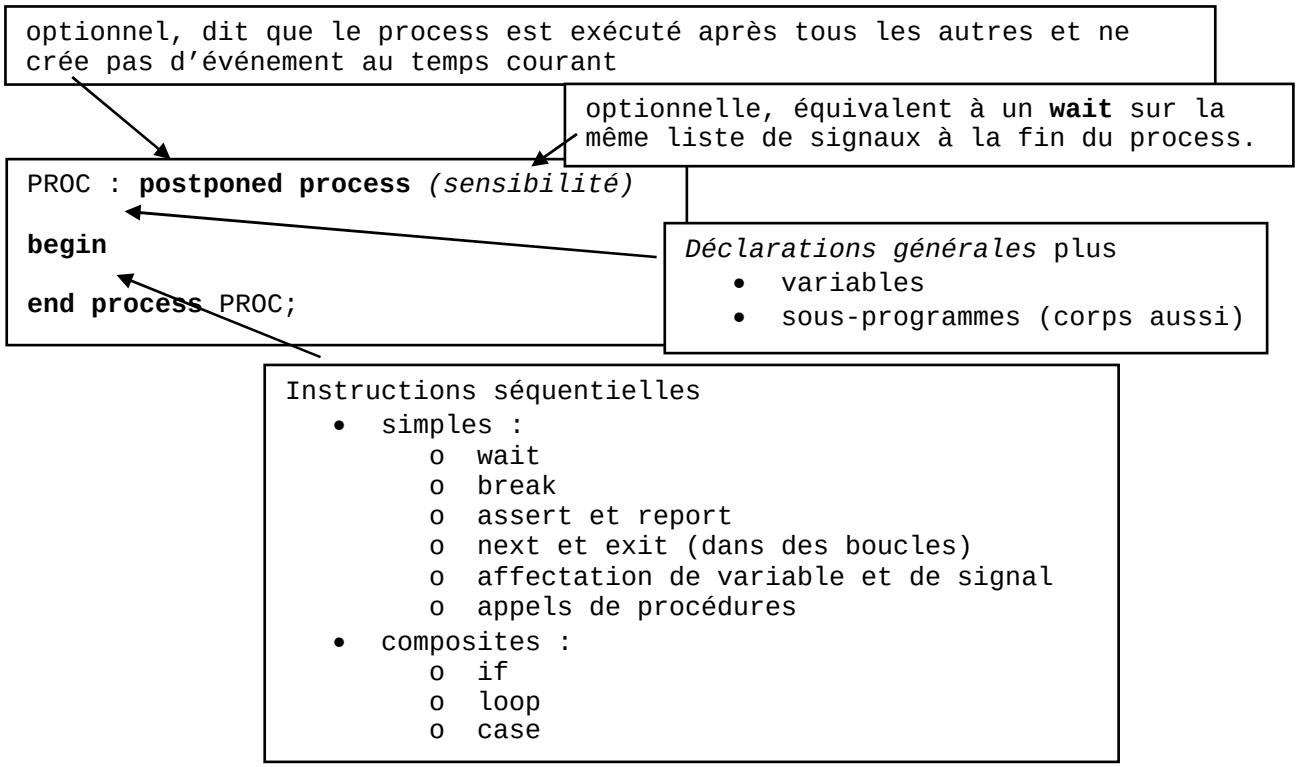


Figure 8 Contenu d'un processus

2.2.3 Le simultané procédural

Le *procedural* est une instruction simultanée et, comme le processus, sa position parmi les autres instructions concurrentes et simultanées n'a pas d'importance. Le *procedural* contient des instructions séquentielles. Ces instructions là, comme dans le cas du processus, sont exécutées dans l'ordre qui a de l'importance. Le *procedural* est appelé à la discrétion du noyau analogique, autant de fois qu'il veut et quand il veut, avec pour but de faire en sorte que les quantités qui y sont affectées ne changent pas (trop) entre deux exécutions successives du même instant de simulation.

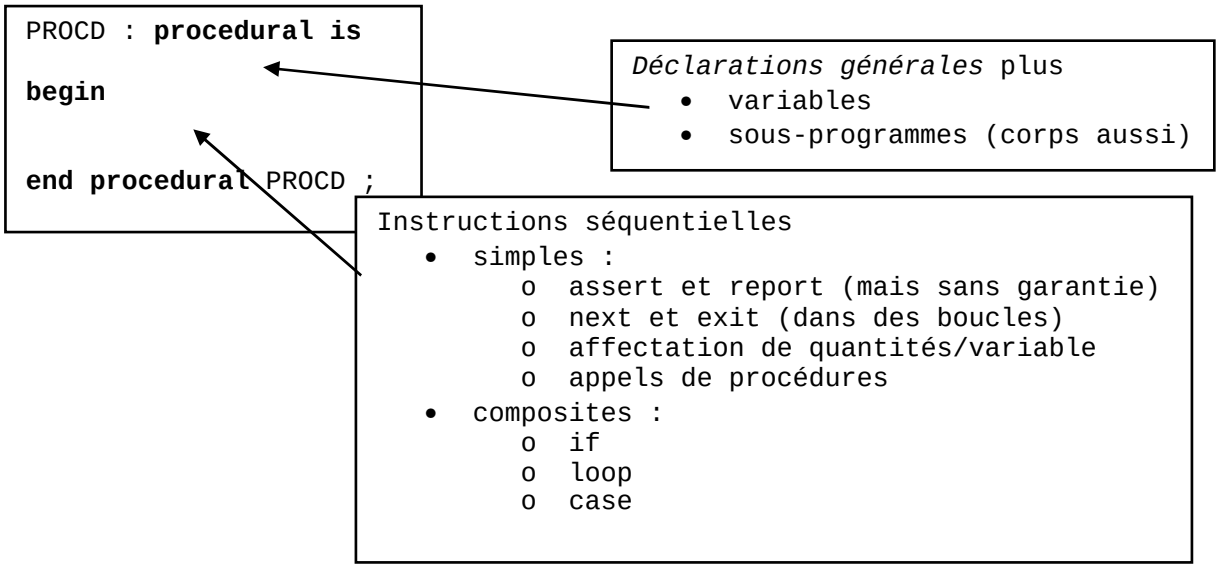


Figure 9 Contenu d'un procedural

Les quantités y sont vues comme des variables, et affectées comme telles avec le signe « := ».
Les instructions y sont tout le jeu des instructions séquentielles sauf celles qui interfèrent avec le temps et les événements sur signaux : **wait**, **break**, affectation de signal.
Les instructions **assert** et **report** y sont légales mais la norme ne dit pas si elles sont exécutées à chaque exécution du noyau, ou « mises en conserve » d'une façon ou d'une autre pour un effet unique.

2.2.4 Le type protégé (moniteur)

Il s'agit d'une fonctionnalité orientée objet, qui permet de déclarer une zone et des objets dont l'accès est contrôlé de façon qu'un processus au plus ait l'usage de ces objets à un instant donné. Attention : un type protégé est un type, et n'a aucune activité en lui-même ; ce sont les objets déclarés de ce type qui en auront.
En tant que type, il se déclare aux mêmes endroits que les autres types. Il serait baroque toutefois d'en déclarer dans une zone séquentielle (processus, simultané procédural, sous-programme) puisque les règles de visibilité garantissent déjà la propriété essentielle du type protégé. Les types protégés ont au contraire plutôt vocation à devenir les types de variables partagées. Cette "vocation" devient une obligation à partir des implémentations de VHDL 2000.
Les types protégés se déclarent en deux morceaux, la spécification et le corps. Ces deux morceaux peuvent être dans la même zone déclarative, leur séparation permet les déclarations incomplètes nécessaires pour faire des types récursifs. Bien entendu, le corps doit apparaître après la spécification et compléter les sous-programmes déclarés.

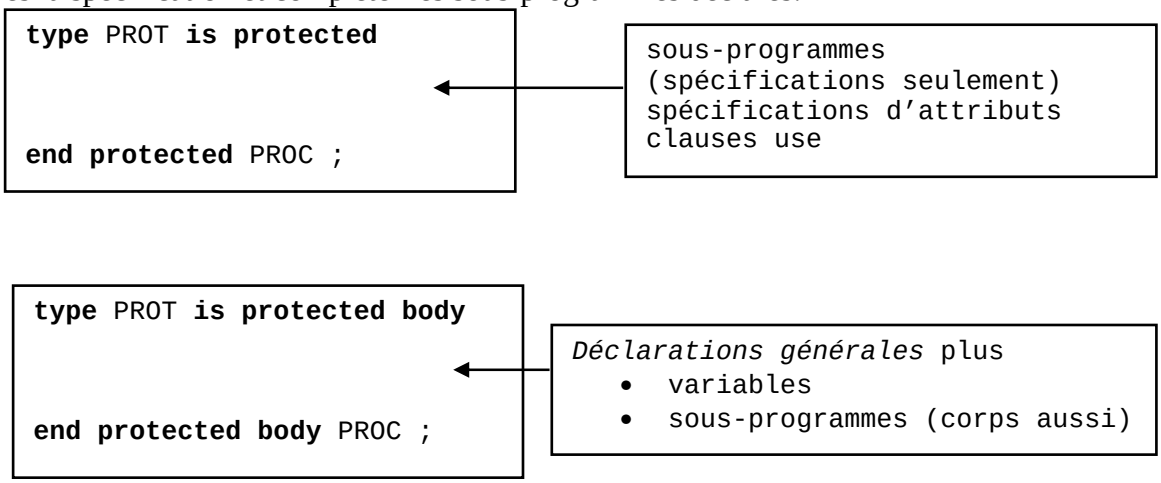


Figure 10 Contenu d'un type protégé

Plus tard, une variable partagée sera déclarée en contexte concurrent :
```
shared variable V : PROT ;
```
Et les sous-programmes du type protégé seront invoqués par la notation pointées :
```
V.sousprog(arg1,arg2) ;
```
La propriété de protection est assurée, entre autres, par le fait que les variables locales à cette structure ne peuvent être déclarées que dans le corps du type protégé, et de ce fait sont accessibles seulement par des sous-programmes du même type.

2.3 Les sous-programmes

Les sous-programmes sont des zones de code séquentiel, qui n'existent et ne sont actifs que si on les appelle (au contraire des composants instanciés et configurés qui ont une vie pendant toute la simulation, par exemple). Les sous-programmes peuvent être spécifiés dans une déclaration qui ne donne que leur nom et les noms et types des arguments sur lesquels ils

travaillent. Cette spécification peut être isolée, auquel cas il faudra la répéter plus bas en ajoutant le corps du sous-programme. Elle peut se poursuivre par le corps du sous-programme, auquel cas elle ne sera pas répétée. La séparation en deux « morceaux » permet d'une part de confiner l'implémentation aux corps de paquetage en publiant la spécification, d'autre part de permettre l'écriture d'algorithmes à récursivité croisée (A appelle B qui appelle A). Il y a deux catégories de sous-programmes, les procédures et les fonctions.

2.3.1 Les arguments

Chaque sous-programme peut avoir une liste d'arguments entre deux parenthèses (pas de parenthèse s'il y a zéro argument) et séparés par des points virgules. Chacun de ces arguments est présenté de la façon suivante :

```
……[genre] NOM : [mode] TYPE [:= valeur par défaut]…
```

Les zones entre crochet sont optionnelles et ont une valeur par défaut qui peut changer suivant les contextes. Le genre concerne signal/variable/constant/file/quantity/terminal, le mode est in/out/inout/buffer. Tous les genres et tous les modes ne sont pas autorisés partout.

Plusieurs arguments qui partagent exactement la même déclaration peuvent être factorisés et séparés par une virgule : la déclaration sera dupliquée pour chacun d'eux.
…Nom1, Nom2 : INTEGER := 3 ;…

Si la valeur par défaut est un appel de fonction, cette fonction sera appelée autant de fois qu'il y a d'arguments à initaliser.

2.3.2 La procédure

La procédure est une portion de code séquentiel qu'on invoque en tant qu'instruction : l'appel de procédure est une instruction séquentielle et aussi une instruction concurrente. La spécification donne son nom, et la liste des arguments espérés.

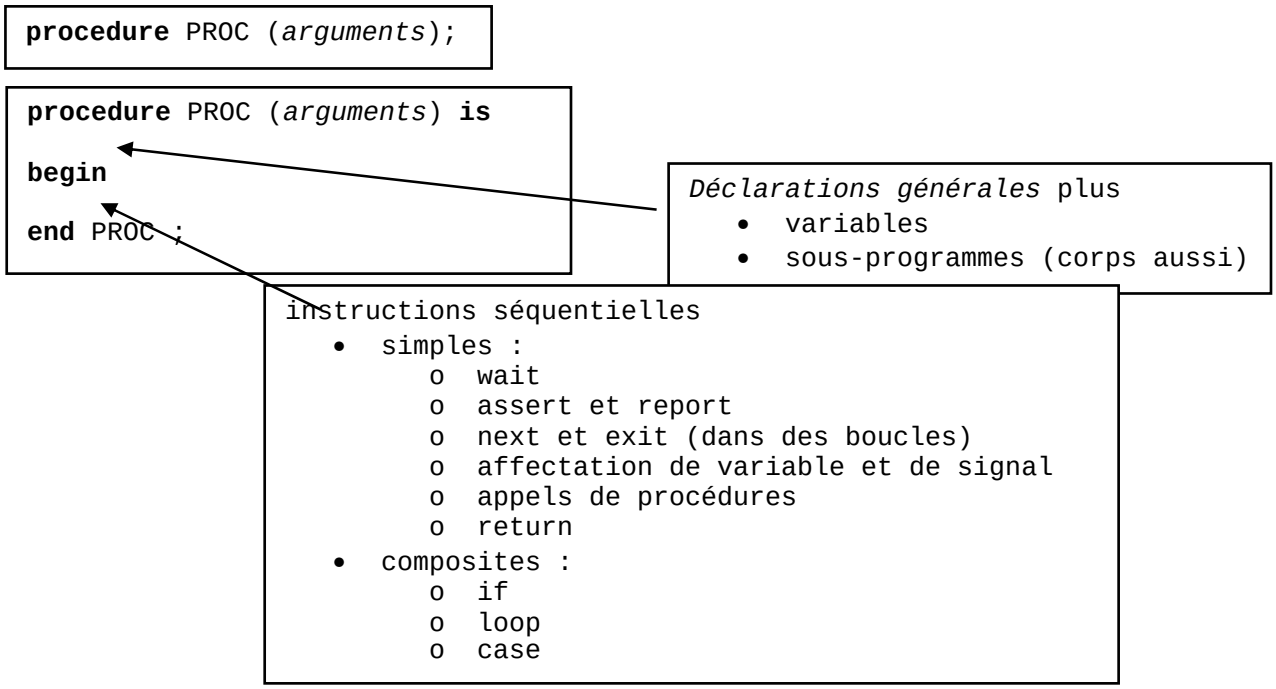


Figure 11 Contenu d'une procédure

2.3.3 La fonction

La fonction est une portion de code séquentiel qu'on invoque en tant que valeur dans une expression : l'appel de la fonction n'est pas une instruction. La spécification donne son nom, et la liste des arguments espérés avec pour chacun son genre (signal, constante, fichier (file) mais pas variable) son nom, son type, son mode (**in** forcément qui est par défaut), et sa valeur par défaut éventuelle.
Elle donne aussi le type de la valeur retournée par la fonction.

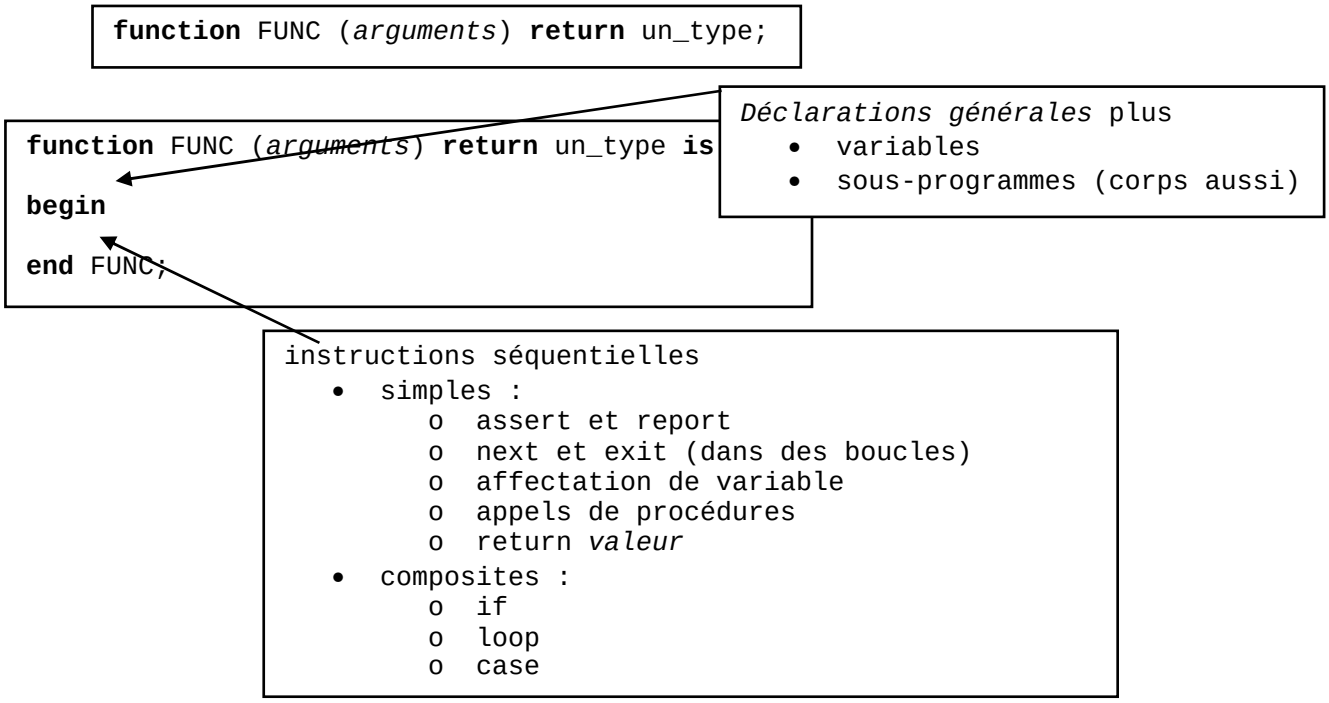


Figure 12 Contenu d'une fonction

Il y a deux catégories de fonctions : pure et impure.

- Une fonction est pure si elle n'a pas d'effets de bord, c'est-à-dire si ce qu'elle rend ne dépend que de ses arguments, et qu'elle n'écrit rien en dehors de ses variables locales.
- Une fonction est impure si elle a des effets de bord, par exemple une fonction qui compterait combien de fois elle est appelée, ceci nécessitant le maintien d'une variable extérieure à la fonction.

Une fonction se déroule à temps strictement nul (elle fait partie de l'évaluation des expressions). Elle ne peut donc pas contenir d'instruction **wait** ou **break**. Enfin, l'algorithme d'une fonction doit absolument tomber sur un **return**, au contraire de la procédure où cela est optionnel. Cela ferait une erreur à l'exécution si l'exécution passait sur son « **end** ».

3 Environnement, Bibliothèques

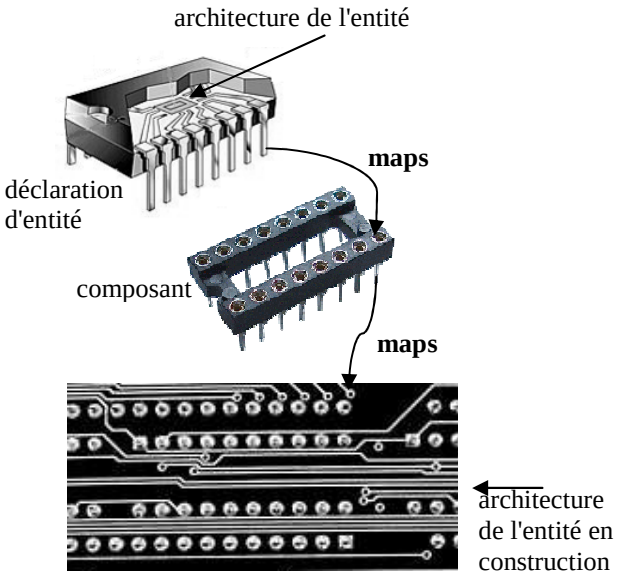
Les bibliothèques sont, pour le langage, de simples noms logiques : STD, IEEE, etc. ; C'est l'installation de l'outil qui redirige ces noms sur les emplacements physiques réels. Quelques unes de ces bibliothèques ont un statut spécial :

- STD est la bibliothèque mère indispensable pour faire quoi que ce soit, elle contient le nécessaire paquetage STANDARD. Il est inutile de la mentionner au début d'une unité, par défaut toutes sont préfixées de `library STD ; use STD.STANDARD.all ;`
  - STD.STANDARD c'est là que sont définis les types BOOLEAN, BIT et INTEGER. Or ces types sont utilisés dans le langage lui-même, par exemple BOOLEAN pour les instructions **if**, BIT pour l'attribut TRANSACTION, INTEGER pour l'élévation à la puissance. C'est pourquoi l'usage de STD est « par défaut », la clause « `library STD ;` » est implicitement mise au début de chaque unité, avec `use STANDARD.all`.
  - STD.TEXTIO contient des primitives rustiques de lecture et écriture sur fichier et sur console. La base du système d'entrées-sorties est qu'on lit ou on écrit ligne à ligne des chaînes de caractères. La conversion de ces chaînes depuis ou vers des types du langage est à faire par ailleurs, quelques primitives sont fournies dans le paquetage sur les types de STANDARD.
- WORK est la bibliothèque où l'on travaille. C'est le seul endroit où l'on a le droit de compiler. La bibliothèque WORK est vue, par ses clients, sous un autre nom (MARTIN par exemple, depuis chez Dupont). Chaque concepteur a donc « sa » bibliothèque WORK, et accède à celles des autres en donnant leur nom. Il est inutile de la mentionner au début des unités, comme pour STD par défaut toutes les unités sont préfixées de `library WORK` ; (attention, pas de **use** par défaut).
- IEEE est une bibliothèque qui contient les paquetages standardisés par les IEEE, ainsi que de façon consensuelle des paquetages « presque » standards, mis dans le domaine public par un vendeur par exemple.
  - IEEE.STD_LOGIC_1164 contient le type à 9 états universellement utilisé par les concepteurs. : '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. 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 (en attendant VHDL 2008, voir §1.7.1.9 page 11) 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).

  - IEEE.STD_LOGIC_TEXTIO n'est pas standard (il le devient avec VHDL 2008) mais son source est public: c'est un paquetage qui complète utilement TEXTIO en fournissant les primitives de conversion entre chaînes de caractères et les types de STD_LOGIC_1164
  - IEEE.STD_LOGIC_ARITH contient des primitives fournissant l'arithmétique signée et non signée sur des vecteurs de STD_LOGIC. C'est un paquetage assez long et le concepteur aura avantage à en inspecter le code pour se rendre compte de ses possibilités. À noter que les opérateurs travaillent sur deux types SIGNED et UNSIGNED, qui ne sont pas compatibles entre eux ni avec STD_LOGIC_VECTOR mais qui sont parents, c'est-à-dire qu'on peut toujours changer l'un en l'autre. C'est grâce à ce choix que le même paquetage fournit l'arithmétique entre un entier signé et un entier non signé.
  - IEEE.STD_LOGIC_SIGNED et IEEE.STD_LOGIC_UNSIGNED sont deux paquetages qui font le choix inverse du précédent : l'arithmétique est fournie directement sur le type STD_LOGIC_VECTOR mais du coup on ne peut plus avoir à la fois des nombres signés et non signés, sauf à utiliser la notation pointée ce qui est hostile quand il s'agit d'opérateurs comme « + » ou « - ». Par contre plus besoin de changements de types avant d'opérer.
  - IEEE.MATH_REAL et IEEE.MATH_COMPLEX deux jeux de primitives mathématiques, qui sont très utilisés en VHDL AMS.
  - [AMS]IEEE.FUNDAMENTAL_CONSTANTS
  - [AMS]IEEE.MATERIAL_CONSTANTS
- DISCIPLINES est la bibliothèque utilisée dans le monde analogique et qui contient les déclarations pertinentes de natures et de constantes pour les différents domaines physiques que l'on veut modéliser.
  - DISCIPLINES.ENERGY_SYSTEMS
  - DISCIPLINES.ELECTRICAL_SYSTEMS
  - DISCIPLINES.MECHANICAL_SYSTEMS
  - DISCIPLINES.THERMAL_SYSTEMS
  - DISCIPLINES.FLUIDIC_SYSTEMS
  - DISCIPLINES.RADIANT_SYSTEMS

  Les implémentations confondent parfois DISCIPLINES avec IEEE. C'est une des corrections que l'on peut être amené à faire quand on porte un modèle.

4 Hiérarchie et Structure



Il y a deux niveaux de branchement dans une architecture: D'une part le composant, sorte de support, est connecté aux signaux et aux ports de l'architecture dans laquelle il est instancié. D'autre part l'entité est connectée à son composant, soit par une spécification de configuration – cela se passe dans l'architecture elle-même - soit par une déclaration de configuration –ce qui se fait depuis l'extérieur, dans une unité de conception spécifique, la déclaration de configuration-. À chaque étage, on peut spécifier les génériques, et les branchements.

Figure 13 L'instance du composant et sa configuration avec une entité demandent deux « port map » et deux « generic map »

4.1 La généricité

Les entités et les composants peuvent être génériques ; la généricité consiste à déclarer une constante dans une zone spéciale, après le mot-clé **generic**, et à ne donner sa valeur que plus tard. Cela peut être au moment d'instancier un composant, au moment de le configurer ou carrément au moment de simuler ou de synthétiser.

Il y a plusieurs cas de figure.

4.1.1 L'entité est générique, le composant est pareillement générique

```
entity SHIFTN is
  generic (N : INTEGER ; delay: TIME);
  port (inp: BIT_VECTOR(N-1 downto 0); outp: out BIT_VECTOR(N-1 downto 0))
end entity ADDN;

component COMP is
  generic (X : INTEGER ; delay: TIME);
  port (E: BIT_VECTOR(N-1 downto 0); S: out BIT_VECTOR(N-1 downto 0))
end component ADDN;
```

On configure le composant avec cette entité là:

```
for CX: COMP use entity WORK.SHIFTN
              generic map (N=>X, delay=>delay)
              port map (inp=>E, outp=>S);
```

On ne donne pas de valeur, on attache un argument à l'autre. Noter ici que si les arguments de l'entité et du composant avaient le même nom et le même ordre, on pourrait compter sur les liens par défaut et ne pas écrire les branches **generic map** et **port map**. On verra par ailleurs qu'on n'est pas obligé de d'attacher les ports tels quels (voir §4.2 page 29).

L'instance de composant fixe enfin la généricité du composant générique:

```
CX: COMP generic map (X=>8, delay=> 2 ns) port map (E=>P1, S=>P2);
```

Ces valeurs (8 et 2 ns), fournies à la généricité du composant, sont transmises à celle de l'entité qui s'en sert comme d'une constante dans ses architectures.
Dans ce cas de figure, le composant et sa configuration restent « virtuels » jusqu'au moment où l'on instancie le composant en fournissant les valeurs qui manquent. On peut donc, immédiatement dessous, instancier un autre composant de la même entité avec d'autres arguments génériques :

```
CY: COMP generic map (X=>16, delay=> 4 ns) port map (E=>P3,S=>P4);
-- CY est sur 16 bits et 4 ns.
```

4.1.2 L'entité est générique, pas le composant

```
entity SHIFTN is
  generic (N : INTEGER ; delay: TIME);
  port (inp: BIT_VECTOR(N-1 downto 0); outp: out BIT_VECTOR(N-1 downto 0))
end entity ADDN;

component COMP is
  port (E: BIT_VECTOR(7 downto 0); S: out BIT_VECTOR(7 downto 0))
end component ADDN;
```

On configure le composant avec cette entité là, mais il nous faut dire combien valent *N* et *delay* pour que ça marche:

```
for CX: COMP use entity WORK.SHIFTN
              generic map (N=>8, delay=>3 ns)
              port map (inp=>E, outp=>S=>);
```

L'instance de composant voit un composant qui n'est pas générique:

```
CX: COMP port map (E=>P1, S=>P2);
```

Si l'on instancie un autre composant de ce type, ce sera forcément le même (8, 3 ns) contrairement à ce que l'on a fait au paragraphe précédent.

```
CY: COMP port map (E=>P3, S=>P4); -- CY est aussi sur 8 bits et 3 ns.
```

Ceci n'est pas le signe d'une infériorité de cette méthode. D'une part on n'a pas souvent le choix, les bibliothèques étant importées ou achetées telles quelles ; d'autre part si l'on ne veut se servir que de tel composant en multiples exemplaires identiques, il est inutile et pénible, voire peu optimisable pour le simulateur, de repousser au dernier moment la donnée de ses paramètres.

4.1.3 L'entité est générique, le composant aussi mais moins

Voyons le cas où l'entité a deux arguments génériques, et le composant un seul.

```
entity SHIFTN is
  generic (N : INTEGER ; delay: TIME);
  port (inp: BIT_VECTOR(N-1 downto 0); outp: out BIT_VECTOR(N-1 downto 0))
end entity ADDN;

component COMP is
  generic (delay: TIME);
  port (E: BIT_VECTOR(7 downto 0); S: out BIT_VECTOR(7 downto 0))
end component ADDN;
```

On configure le composant avec cette entité là, mais il nous faut dire combien vaut *N* et passer *delay* comme argument générique restant:

```
for CX: COMP use entity WORK.SHIFTN
              generic map (N=>8, delay=>delay )
              port map (inp=>E, outp=>S=>);
```

L'instance de composant voit un composant qui n'est générique que sur *delay*:

```
CX: COMP generic map (delay=> 2 ns) port map (E=>P1,S=>P2);
```

D'autres instances du même composant pourront faire varier *delay*, mais pas N:

```
CY: COMP generic map (delay=> 4 ns) port map (E=>P3, S=>P4);
-- CY est sur 8 bits avec un délai de 4 ns.
```

On utilisera cette fixation partielle de la généricité lors de la configuration chaque fois que l'on voudra utiliser une entité très générique venant d'une bibliothèque achetée ou non maîtrisée, sur un composant dont la généricité « nous suffit ».

4.2 Les ports

De même qu'il y a double *mapping* pour la généricité, de même pour les ports. Les ports représentent la connexion des composants à l'architecture, et de l'entité au composant. VHDL fournit dans le langage tous les « bidouillages » que peut s'autoriser un concepteur de cartes imprimées :

- changer l'ordre des pattes,
- en laisser certaines « en l'air » en sachant qu'elles sont tirées à une valeur donnée
- en forcer d'autres à la masse ou à une valeur donnée,
- en changer le type (passer de la logique 5V à 3V)
- n'utiliser que la moitié d'un bus ou en reconstruire un avec deux moitiés
- et, qui plus est, aux deux niveaux de l'instance et de la configuration.

Attention : l'utilisation d'une de ces facilités n'est pas exclusive des autres. En poussant VHDL et ses facilités d'associations dans ses derniers retranchements, on peut arriver à des modèles complètement illisibles et qui sont de vraies gageures à exécuter pour les outils de CAO. À utiliser avec autant de modération que les breuvages à radicaux COOH, sinon le mal de tête subséquent est de même nature.

4.2.1 L'ordre des ports

Supposons que l'entité correspond parfaitement au composant, mais hélas l'ordre ou la numérotation des ports n'est pas le même.

```
entity E is
 port (A, B, C : BIT ; S : out BIT) ;
end entity E;

component C is
  port (T: out BIT; X, Y, Z: BIT)
end component;
```

Nous allons utiliser l'association par nom, qui permet de s'affranchir des questions de position:

```
for CX: C use entity LIB.E port map (A=>X, B=>Z, C=> Y, S => T);
```

La même syntaxe d'association fonctionne pour l'instance :

```
CX : C port map (X=> P1, Y=> P2, Z=> P3, S => P4) ;
```

De même, il peut arriver que la numérotation des ports tabulés ne corresponde pas, alors même que le nombre et le type des éléments scalaires correspond parfaitement :

```
entity E is
 port (A: BIT_VECTOR (1 to 10); … ) ;
        -- 10 éléments
end entity E;

component C is
  port (X :BIT_VECTOR (111 downto 102), …) ;
           -- 10 éléments aussi
end component;
```

Figure 14 Connexion en changeant la numérotation des ports

Ici, la bonne propriété de VHDL est que les associations sont faites élément par élément, de gauche à droite et sans regarder les directions:

```
for CX: C use entity E port map (A => X, …);
```

Il est possible toutefois qu'on ait envie d'associer dans l'autre sens ou d'ailleurs dans un ordre quelconque : A(1) à X(102), A(2) à X(103), etc. Dans ce cas, on peut écrire le détail des associations scalaires :

```
for CX: C use entity WORK.E
 port map (A(1) => X(102), A(2) => X(103), A(3) => X(104),
           A(4) => X(105), A(5) => X(106), A(6) => X(107),
           A(7) => X(108), A(8) => X(109), A(9) => X(110),
           A(10) => X(111),…);
```

On voit ici qu'on peut effectivement associer dans le désordre n'importe quel élément scalaire de l'un à n'importe quel élément scalaire de l'autre.

Là encore, la même syntaxe de **port map** fonctionne pour l'instance de composant, quand il faut relier ses ports aux signaux/ports de l'architecture où il est instancié.

4.2.2 Forçage

Les signaux de mode **in** peuvent être associés à une expression. On peut ainsi forcer une patte à la masse, par exemple.

Voyons l'exemple d'un composant :

```
component C is
  port (ChipSelect : in BIT, …….) ;
end component;

CX : C port map ('0', ….autres ports) ;
```

Ici, le signal *in fine* associé à ChipSelect via la configuration aura une valeur permanente à 0. Ceci ne marche pas avec les outils fonctionnant sur une version de VHDL antérieure à 93, auquel cas il suffit de déclarer un signal nommé, qu'on met à 0 et qu'on passe au **port map**.

```
signal masse : BIT := '0' ;

…

CX : C port map (masse, ….autres ports) ;
```

4.2.3 Laisser ouvert *(open)*

Les signaux de mode **out** peuvent être ignorés du simulateur. Il suffit de leur associer le mot-clé **open**. Ici par exemple, on a besoin d'une porte ou-exclusif (**xor**) mais on ne dispose que d'un additionneur dans la bibliothèque. On s'en sort en appelant l'additionneur et en ignorant ses retenues: celle d'entrée est collée à 0, celle de sortie est ouverte. Le reste…est un xor.

```
component XOR_GATE port (A,B: BIT; S: out BIT); end component;

for X1: XOR_GATE use entity LIB.ADDER
         port map(A=> X, B=> Y, Ret_In=>'0', S => Z, Ret_Out=>open);
```

4.2.4 Défaut

On peut donner une valeur par défaut lors de la déclaration d'un port de mode **in**.
L'existence de cette valeur par défaut permettra d'omettre ce port dans la liste d'association.
Nous nous retrouverons donc dans le cas du forçage explicite vu au 4.2.2 ci-dessus.

```
component C is
  port (A, B: BIT :='0'; S: out BIT) ;
end component;

CX : C port map (S => P1) ;
```
est équivalent à
```
CX : C port map (A=> '0', B=> '0', S => P1) ;
```

4.2.5 Changer le type

Il peut arriver que l'entité qui nous intéresse corresponde exactement au composant que l'on veut configurer, mais hélas l'un est écrit avec le type BIT, l'autre avec le type STD_LOGIC. Ou encore, l'un a un port de sortie du type INTEGER, l'autre un port de sortie de type BIT_VECTOR où l'entier est codé.

Cette situation est heureusement assez rare depuis l'arrivée du standard STD_LOGIC_1164, qui a unifié les types utilisés par les concepteurs VHDL, voir à ce sujet §1.7.2.7 page 13. Néanmoins on peut trouver des modèles antérieurs et avoir à les maintenir ou à les utiliser. On peut donc, dans une liste d'association, changer le type « au vol » des différents arguments formels et réels. Pour cela on emballe le port en question par une fonction qui prend l'autre en argument et rend son type.

Supposons que FA soit une fonction prenant le type de B en argument et rendant le type de A, et FB faisant l'inverse. L'exemple le plus simple de telles fonctions est la fonction « changement de type » que l'on a gratuitement en utilisant le nom du type quand les types sont apparentés, par exemples arithmétiques. Exemple :
Si I est entier et R réel, on peut écrire I := INTEGER(R) et INTEGER est ici vu comme une fonction de conversion. Nous avons donc dans le cas qui nous occupe:

```
function FA (Arg :TypeB) return TypeA ;
function FB (Arg :TypeA) return TypeB ;
```

Et nous voulons configurer ce composant ci avec cette entité là:
```
component C is
   port (…A1 : in TypeA; A2 : out TypeA; A3 : inout TypeA;..);
end component;

entity E is
   port (…B1 : in TypeB; B2 : out TypeB; B3 : inout TypeB;..);
end entity;
```

Il est bien sûr impossible d'associer A1 à B1 par un simple B1 => A1 dans la configuration, puisque les types sont différents.
Nous pouvons par contre écrire dans une liste d'association :

- …A1 => FB(B1) … : De mode **in**, le flux va aller du composant vers l'entité et le changement de type des valeurs qui circulent aura toujours lieu « vers B ».
- …FA(A2) => B2… De mode **out**, le flux va aller de l'entité vers le composant et le changement de type des valeurs qui circulent aura toujours lieu « vers A ».
- …FA(A3) => FB(B3)… De mode **inout**, le flux peut aller dans les deux sens et le simulateur appellera l'une ou l'autre des fonctions au gré du mouvement des événements sur les signaux.

Là encore, ce changement de type peut s'opérer dans la configuration, dans l'instance ou…dans les deux. Il est clair que l'abus de cette, *ahem*, « facilité » ne conduit pas à des modèles très lisibles ni très performants, sans compter que c'est une fonctionnalité rarement utilisée et donc sujette à « défauts » non signalés dans les outils de CAO. Elle fut définie dans le langage en un temps où il n'y avait pas de paquetage logique standard (STD_LOGIC_1164) et il était alors nécessaire de pouvoir convertir au vol les types des ports des entités instanciées.

4.3 Génériques et génération

```
Etiquette : if condition generate
[begin] optionnel dans
implementations récentes
…
end generate;
```

L'instruction **generate** existe sous deux formes : la conditionnelle (**`if…generate`**, qui n'a pas de branche **elsif** ou **else**) et l'itérative (**`for…generate`**) avec un schéma de boucle avec indice. Dans les deux cas, il s'agit de créer du code VHDL algorithmiquement, code qui sera ensuite exécuté. L'exécution de ces instructions se fait avant le temps zéro de la simulation, dans la phase dite d'élaboration.

Voyons par exemple l'utilisation d'une boucle de génération dans la construction d'un additionneur N bits à partir d'un demi additionneur un bit (deux entrées, une sortie et une retenue sortante) et d'additionneurs complets un bit (deux entrées, une retenue entrante, une sortie et une retenue sortante).

```
Etiquette : for I in 1 to 10 generate
[begin] optionnel dans implementations
récentes
…
end generate;
```

Voici l'entité, noter que la taille des ports dépend de la valeur de l'argument générique :

```
entity ADDN is
   generic (N: INTEGER:=8);
   port (A,B: BIT_VECTOR(N-1 downto 0); S: out BIT_VECTOR (N downto 0));
end entity ADDN;
```

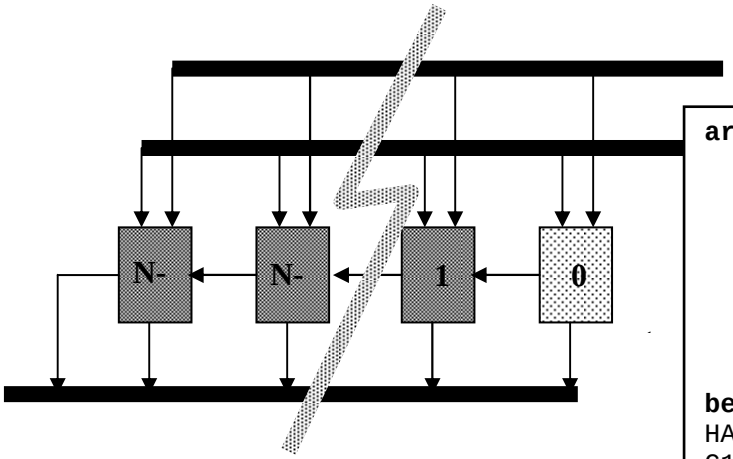


Figure 15 Un additionneur N bits

```
architecture STRUCT of ADDN is
  signal CARRY: BIT_VECTOR (N downto 1);
  component ADD1
     port (A,B,CIN: BIT; S,COUT: out BIT);
  end component;
  component HALF_ADD
     port (A,B: BIT; S,COUT: out BIT);
  end component;
  -- configurations par défaut
begin
HA: HALF ADD port map(A(0),B(0),S(0),CARRY(1));
G1: for I in N-1 downto 1 generate
    XI: ADD1(A(I),B(I), CARRY(I),S(I), CARRY(I+1));
end generate;
S(N)<=CARRY(N);
end architecture STRUCT;
```

```
HA: HALF ADD port map(A(0),B(0),S(0),CARRY(1));
G1(1).XI: ADD1 port map (A(1),B(1), CARRY(1),S(1), CARRY(2));
G1(2).XI: ADD1 port map (A(2),B(2), CARRY(2),S(2), CARRY(3));
Etc…
```

L'architecture utilise la boucle de génération pour la partie régulière du circuit : comme souvent, il y a une exception pour le premier élément qui est sorti de la boucle et géré séparément.

4.4 Amusant et inutile : la hiérarchie récursive

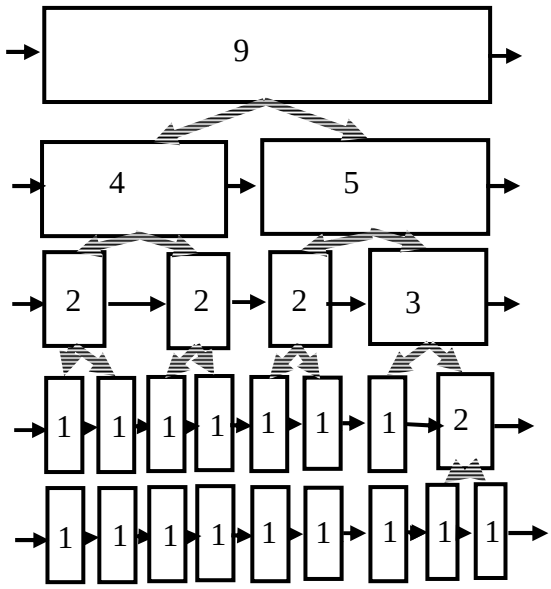
Il est possible en VHDL d'utiliser la récursivité lors de l'instanciation de modèles, autrement dit un modèle générique peut s'instancier lui-même (avec une autre généricité bien entendu !). Cette « facilité » sera surtout utilisée pour rendre un modèle obscur ou pour tester les limites d'un compilateur. Néanmoins elle démontre bien le fait qu'une déclaration d'entité existe indépendamment de son architecture.

Pour illustrer ceci, voyons le cas d'une ligne à retard de N coups d'horloge sur un bit : l'entrée (*inp*) est propagée sur la sortie (*outp*) au bout de N fronts montants d'horloge (*h*).

La déclaration, très classiquement, est générique sur N –le nombre de coups d'horloge pour le retard-, a une entrée, une sortie et une horloge.

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

entity décalage is
  generic (N: positive:=16);
  port (h, inp: in std_logic; outp:out std_logic);
end;
```

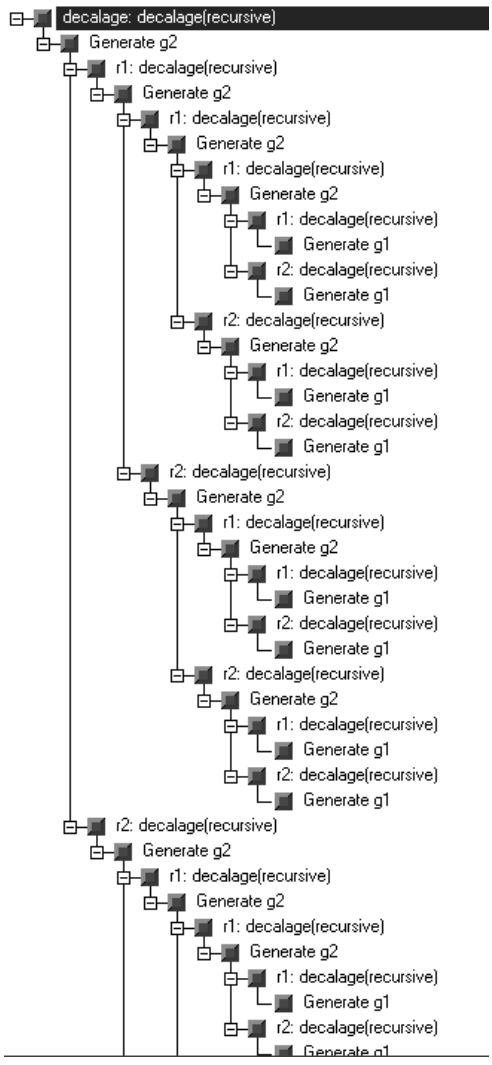
L'architecture, elle, est plus originale: elle déclare sa propre entité en tant que composant. Une instruction **generate** sert de condition d'arrêt : si N=1, alors on écrit **une** instruction flot de données qui réalise le décalage d'une période d'horloge, et une seule. Dans les autres cas de figure, la question est ramenée à la mise bout-à-bout d'un décalage de N/2 et d'un décalage de N-N/2 (il s'agit de divisions entières), en passant par un unique signal local, et le problème est repoussé à l'étage inférieur, ce qui fait que, *in fine,* tout le modèle sera composé de décaleurs d'une période d'horloge, instanciés en poupées russes.
Sur l'exemple ci-contre, on voit que l'appel d'un décaleur avec N=9 va instancier deux décaleurs avec N=4 et N=5, puis quatre avec N valant respectivement 2, 2, 2 et 3, puis huit avec les valeurs 1, 1, 1, 1, 1, 1, 1, 2, et enfin neuf avec duplication du dernier.
Chaque fois qu'on arrive à N=1, la récursivité est terminée, une unique instruction flot de données est générée.



L'architecture récursive est détaillée ci-après.

Figure 16 Instances récursives

```
architecture récursive of décalage is

component décalage
  generic (N: positive);
  port (h, inp: in std_logic; outp:out std_logic);
end component;
-- pas de configuration, le défaut fonctionne bien car la
-- déclaration de composant est identique à l'entité

signal local: std_logic; -- le signal qui va connecter les deux
                         -- instances appelées.

begin

  -- un décalage d'une période d'horloge.
  G1:if N=1 generate
    outp<=inp when h'event and h='1' else unaffected;
  end generate;

  -- le découpage en deux sous-problèmes, double appel récursif
  G2:if N>1 generate
    R1: décalage generic map(N/2) port map(h, inp, local);
    R2: décalage generic map(N-N/2) port map(h, local, outp);
  end generate;

end;
```



On peut voir ci-contre la hiérarchie simulée telle que déployée par l'élaboration du modèle. En particulier, on peut constater que toutes les feuilles de cette hiérarchie sont labélisée « G1 », le seul bloc du modèle où il se « passe quelque chose ».

Le concepteur coriace pourra essayer la récursivité croisée (A appelle B qui appelle A), si toutefois il arrive à imaginer une fonctionnalité qui s'accommode de ce genre de description.

5 Modélisation de bibliothèques - VITAL

5.1 Pourquoi, comment ?

Dans le cycle de conception classique,

- il est des modèles qui n'ont pas vocation à être synthétisés (par exemple un modèle de RAM qui n'est là que pour faire marcher le reste du modèle) ; dans ce genre de synthèse tous les coups sont permis « pourvu que ça marche ». Voir par exemple §7.1 page 55.
- il est des modèles qui ont vocation à être synthétisés et alors on fait attention à éviter la sur-spécification, pour permettre à l'outil de synthèse de faire toutes les optimisations qu'il jugera utiles. Voir par exemple §7.3 page 70.
- Enfin il est des modèles qui sont l'image exacte du circuit tel qu'il a été synthétisé, et qu'on simule afin de vérifier les propriétés espérées, notamment temporelles, ou qu'on donne à un client qui veut une image exacte de ce qu'il achète. Or la synthèse consiste à faire l'interconnexion d'éléments de bibliothèque, éléments qui sont fournis par le fondeur. Dans ce cadre là, la spécification de ces éléments et des éventuels délais dus à leur interconnexion doit être extrêmement précise. Les délais en question peuvent être simples (la sortie bouge après un temps T si l'entrée bouge), plus complexe (on peut spécifier des temps différents suivant les fronts) ou très complexe (suivant les différentes combinaisons, temps d'aléas, temps de propagation sur les fils.). C'est ce style là qui nous intéresse dans ce chapitre.

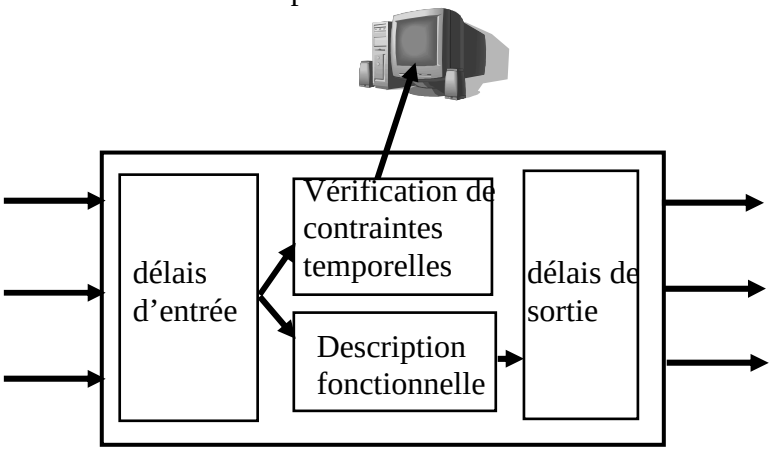
VITAL (VHDL Initiative Towards ASIC Libraries) a été développé, comme STD_LOGIC, dans un souci de standardisation de *l'usage* de VHDL.

C'est un standard de "bon usage" de VHDL, permettant aux fondeurs de développer des modèles de leurs bibliothèques, et aux clients de construire leurs propres briques avec les mêmes outils, tout en garantissant aux clients un jeu de bonnes propriétés dont voici les essentielles :

- La rétro-annotation permet d'injecter dans un modèle déjà simulé « sans les temps » tous les délais fournis par la synthèse. On peut ainsi valider séparément la fonctionnalité et la performance. Cette injection se fait sans nécessiter l'accès au source du modèle, par la définition de génériques et via un outil spécifique (utilisant le standard SDF[7]) ou par des déclarations de configuration en VHDL crées automatiquement.
- L'interface des modèles utilise uniquement le type STD_LOGIC_1164 et même souvent un sous-ensemble de celui-ci.
- Les informations temporelles sont injectées grâce à des conventions de noms (la syntaxe des identificateurs des ports et génériques se trouve donc faire partie de ce langage), et peut définir entre autres des informations sur les délais ou des temps « pin to pin ».

[7] Standard **IEEE 1497-2001** (SDF). Voir bibliographie [IE4] chapitre 15 page 163.

- Les simulateurs pourront optimiser ces primitives.
- Les accélérateurs hardware –ce sont des simulateurs utilisant des FPGA programmés automatiquement de façon à reproduire le circuit simulé et fournissant donc une sorte de prototype- pourront contenir « en dur » les primitives de bibliothèque et donc seront d'autant plus efficaces.



Un modèle VITAL se décompose donc en plusieurs blocs, dont le plus simple est très souvent le bloc fonctionnel (celui qui dit « ce que ça fait »), ce qui peut surprendre. Il contient aussi la description des différents délais d'entrée et de sortie représentant par exemple les capacités des fils de connexion ; et un bloc de vérification de différentes contraintes, dont l'effet essentiel est de faire des messages d'erreur à la console.

Figure 17 Schéma d'un modèle VITAL

L'apprentissage et l'utilisation de VITAL nécessiteraient un volume à eux seuls et sont clairement en dehors du cadre de ce manuel. On trouvera en bibliographie chapitre 15 page 161 les liens nécessaires, et en références §12.3 page 125 les déclarations de paquetages correspondantes. Néanmoins nous présentons ici les grands traits de cette norme, à titre d'aide-mémoire.

On peut écrire un modèle VITAL en respectant deux niveaux de la norme : le niveau 0 et le niveau 1. Tout ce qui est conforme au niveau 1 est aussi conforme au niveau 0. Le niveau 1 est assez contraignant, beaucoup de modèles se contentent du niveau 0.
L'attribut VITAL_LEVEL0 ou VITAL_LEVEL1 doit être défini dans l'entité et l'architecture, de façon à « prévenir » les outils. :

```
attribute VITAL_LEVEL0 of ·Entité : entity is TRUE;
```

- **Niveau 0 : pour la portabilité et la rétro-annotation**. Les blancs-soulignés (*underscores*) sont interdits dans les noms des ports. Les seuls types utilisés pour les ports sont les types STD_LOGIC et STD_LOGIC_VECTOR. Les noms des arguments génériques sont soumis à des conventions permettant la rétro-annotation et l'introduction de délais sans avoir à éditer le modèle. Par exemple, si un port s'appelle TOTO, un des génériques qui lui sont dédiés pourra s'appeler `tipd_TOTO` ou encore tpw_TOTO_negedge, permettant, après synthèse, de spécifier un délai de propagation ou un temps de largeur de pulse après front négatif. Le type de ces arguments permet en sus de spécifier des délais différents pour chaque passe de telle valeur à telle autre valeur. Voir les divers *VitalDelayType* du paquetage VITAL_Timings §5.2 ci-dessous.
  Les blancs soulignés servent donc à délimiter des champs significatifs et c'est pourquoi ils sont interdits par ailleurs.

- **Niveau 1: pour la performance**. En sus d'être niveau 0, les conflits sont interdits sur les signaux (deux processus ne peuvent pas écrire sur le même signal, il n'y a donc jamais d'appel de fonctions de résolution), les sous-programmes appelés ne peuvent

pas être écrits par le concepteur (ils doivent venir des paquetages standards). Tous les signaux –et pas seulement les ports- doivent être des types STD_LOGIC et STD_LOGIC_VECTOR. Aucune déclaration, à part des alias, n'est autorisée. Avec de telles contraintes, chaque bloc du code peut être optimisé puisqu'ils viennent tous de paquetages prédéfinis et que le travail du concepteur ne consiste qu'à les assembler entre eux.

On trouvera par ailleurs, dans le chapitre 12 page 115 :

- La syntaxe VITAL, définie comme une contrainte sur la syntaxe VHDL ; c'est du BNF.
- Les trois déclarations de paquetages nécessaires : timing, primitives, mémoire. Les corps de ces paquetages sont définis en VHDL dans la norme à titre de documentation, mais ils sont probablement codés « en dur » dans les simulateurs commerciaux. Leur longueur interdit de les mettre ici, le lecteur intéressé les trouvera facilement sur le Web.

5.2 Le paquetage VITAL_Timings

On en trouvera la déclaration §12.3.2 page 128, expurgée de ses commentaires qui sont trop volumineux pour être mis ici. Ce paquetage est utilisé par les autres et définit types et primitives concernant les délais du modèle.

Les types remarquables sont:

- **VitalDelayType** est un simple sous-type de TIME, pour les situations où un temps unique suffit.
- **VitalDelayType01** est un tableau de deux valeurs de TIME. Il contient des temps pour front montant 0 vers 1 et descendant 1 vers 0, quand il y a deux valeurs logiques.
- **VitalDelayType01Z** est un tableau de six valeurs de TIME. Il peut définir les temps de passage dans les logiques à trois états (0 1 et Z), pour toute les combinaisons, dans l'ordre tr01, tr10, tr0Z, trZ1, tr1Z . trZ0.
- **VitalDelayType01ZX** est un tableau de douze valeurs de TIME, et définit les temps de passage de toutes les combinaisons entre 0, 1, X et Z pour les cas de figure où il y a un état inconnu (X), l'ordre est : tr01, tr10, tr0Z, trZ1, tr1Z, trZ0, tr0X, trX1, tr1X, trX0, trXZ, trZX.
- **VitalResultMapType** est un tableau (indexé par 'U' 'X' '0' '1') de STD_ULOGIC[8], permettant de convertir les sorties des modèles VITAL à d'autres systèmes de force ; par exemple convertir 1 en H dans le cas d'une sortie drain-ouvert.
- **VitalTableSymbolType** est un type énuméré de caractères littéraux[9], une liste de symboles utilisés pour représenter les transitions ou les conditions de stabilité. Il est utilisé par les primitives VITAL. Attention : les caractères littéraux (entre apostrophes) ont une casse qui est significative, ne pas confondre majuscules et minuscules.

[8] STD_ULOGIC est le type racine dont STD_LOGIC est le sous-type résolu. Ils sont compatibles, l'usage du premier permet des optimisations dans les cas où l'on est certain de ne pas avoir besoin de résolution.

[9] Il s'agit d'un type énuméré « de genre caractère », c'est-à-dire que les symboles contenus s'écrivent graphiquement comme des caractères mais n'ont rien à voir, et sont incompatibles, avec les autres types du même genre comme BIT (pour les valeurs '0' et '1'), CHARACTER (pour toutes les valeurs) ou STD_LOGIC (pour '0', '1', 'X', '-', 'Z'), même si le sens attribué est le même que pour BIT et STD_LOGIC.

- '/' : 0 vers 1
- '\' : 1 vers 0
- 'P' : Union de '/' et '^' (tout front allant vers 1)
- 'N' : Union de '\' et 'v' (tout front allant vers 0)
- 'r' : 0 vers X
- 'f' : 1 vers X
- 'p' : Union de '/' et 'r' (tout front venant de 0)
- 'n' : Union de '\' et 'f' (tout front venant de 1)
- 'R' : Union de f '^' et 'p' (tout front montant)
- 'F' : Union de 'v' et 'n' (tout front descendant)
- '^' : X vers 1
- 'v' : X vers 0
- 'E' : Union de 'v' et '^' (tout front venant de X)
- 'A' : Union de 'r' et '^' (front montant de ou vers X)
- 'D' : Union de 'f' et 'v' (front descendant de ou vers X)
- '*' : Union de 'R' et 'F' (tout front)
- 'X' : Inconnu
- '0' : zéro, niveau bas
- '1' : un, niveau haut
- '-' : "don't care" – indifférent.
- 'B' : 0 ou 1
- 'Z' : haute impédance
- 'S' : état stable

Le paquetage VITAL_Timing contient aussi des primitives permettant de définir des contraintes de temps et divers délais.

- **VitalPathDelay01** et **VitalPathDelay01Z** permettent de définir une valeur décalée dans le temps à un signal de sortie. La première si le signal ne peut avoir que deux valeurs, la seconde ajoute 'Z'.
- **VitalWireDelay** permet de spécifier un délai de connexion (dû à un fil entre deux composants, par exemple).
- **VitalSignalDelay** est utilisé quand le modèle a des contraintes temporelles négatives, c'est-à-dire qu'on est conduit à dire que le signal change « avant » l'horloge qui a pris du retard pour une raison ou une autre.
- **VitalSetupHoldCheck** pour détecter les violations de setup ou de hold.
- **VitalRecoveryRemovalCheck** s'assure qu'un signal est actif dans une fenêtre contrôlée autour d'un front d'horloge, par exemple.
- **VitalPeriodPulseCheck** permet de vérifier les bonnes propriétés d'une horloge ou d'un signal comme reset: taille minimum du pulse, maximum de la période, etc.
- **VitalInPhaseSkewCheck** et **VitalOutPhaseSkewCheck** vérifient la phase de deux signaux.

5.3 Le paquetage VITAL_Primitives

Ce paquetage contient les primitives logiques qu'on va assembler pour construire des ASIC. Elles sont de très bas niveau et leur nom est auto-documenté :

```
VitalBUF, VitalBufIf0, VitalBufIf1
VitalINV, VitalInvIf0, VitalInvIf1
VitalAND, VitalAND2, VitalAND3, VitalAND4
VitalNAND, VitalNAND2, VitalNAND3, VitalNAND4
VitalOR, VitalOR2, VitalOR3, VitalOR4
VitalNOR, VitalNOR2, VitalNOR3, VitalNOR4
```

```
VitalXOR, VitalXOR2, VitalXOR3, VitalXOR4
VitalXNOR, VitalXNOR2, VitalXNOR3, VitalXNOR4
VitalMux, VitalMux2, VitalMux3, VitalMux4
VitalDecoder, VitalDecoder2, VitalDecoder4, VitalDecoder8
```

On y trouve aussi les types permettant de définir des composants par une table de vérité ou une table d'états :

`VitalTruthTables`
et
`VitalStateTables`.

5.4 Le paquetage VITAL_Memory

Ce paquetage défini plus récemment (révision 2000) et permet de définir les fonctionnalités et contraintes des éléments de mémorisation. Nous ne nous étendrons pas dessus dans ce manuel, le code de sa déclaration est néanmoins fourni au §12.3.4 page 141, expurgé de ses commentaires qui sont extrêmement volumineux dans le source original.

5.5 Un petit exemple

5.5.1 Sans les conventions VITAL

5.5.1.1 Fonctionnalité

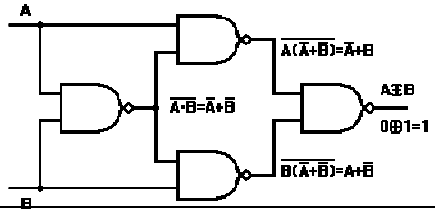
Pour illustrer l'intérêt de VITAL, reprenons l'éternel **xor** :

```
library IEEE; use IEEE.std_logic_1164.all;
entity xorent is
  port (a, b : in std_logic; s : out std_logic);
end xorent;

architecture synth of xorent is
begin
  s <= a xor b;
end;
```

5.5.1.2 Délai simple et unique

Le modèle ci-dessus est un modèle adapté à la synthèse: on dit ce qu'on veut, on laisse l'outil travailler. Mais après la synthèse, il faudra simuler avec les délais induits par la construction de cette fonctionnalité avec les éléments de bibliothèque disponibles. Pour cela deux solutions, la mauvaise consiste à rééditer le texte et à mettre « en dur » une clause **after** dedans, avec le temps observé après synthèse. C'est une mauvaise solution car elle implique une recompilation, donc la disponibilité du source et aussi elle ne peut pas être automatisée !

```
architecture post_synth of xorent is
begin
  s <= a xor b after 3.2 ns;
end;
```



5.5.1.3 Délai dépendant des valeurs proposées

Notons ici que, si le **xor** est implémenté à base de portes plus élémentaires, le délai entre les entrées et la sortie dépend non seulement du chemin électrique

mais des valeurs proposées à ce moment là. Le lecteur pourra vérifier sur le schéma ci-contre –qui implémente le **xor** à base de portes **nand**- en imaginant un délai associé à chaque porte, que si on passe de la configuration 01 à 00, la première porte ne change pas d'état et donc seules deux portes intervenant dans le délai sont sur le chemin entre l'entrée qui bouge et la sortie. Par contre, le passage de 10 à 11 fait intervenir trois portes dans le chemin. Une bonne modélisation nous conduirait à écrire une affectation compliquée dépendant des valeurs, ou, plus lisible, un processus:

```
process (a,b)
  Variable résultat: bit;
begin
  résultat:=a xor b;
  if (a = '0') or ( b='0' )
    then s <= résultat after 2 ns;
    else s <= résultat after 3 ns;
  end if;
end process;
```

5.5.1.4 Délais dépendant des sens de variation

De même que les délais peuvent dépendre des valeurs proposées, ils peuvent aussi dépendre du sens du front: dans une logique "à drain ouvert", le passage à zéro se fait par un transistor "fort" alors que le passage à un se fait par une résistance. La capacité attaquée étant la même dans les deux cas, les délais seront donc différents selon le sens de variation de la sortie considérée. Tout ceci complique sérieusement la question de l'injection des délais dans un modèle fonctionnel, on voit ici que même sur un exemple simplissime comme le **xor**, la spécification des délais est bien plus complexe que la spécification de la fonctionnalité.

5.5.1.5 Délais génériques

Indépendamment de la question des délais dépendant des valeurs, une meilleure solution bien adaptée à VHDL consiste à passer par un argument générique: la compilation n'est pas nécessaire pour changer la valeur du générique, une simple configuration "de l'extérieur" suffit à fournir les valeurs manquantes. Nous reprenons ici l'exemple simple du délai unique. On pourrait facilement passer plusieurs génériques correspondant aux différents chemins, mais on ne peut pas passer les conditions décidant du choix de ces valeurs : elles devraient être codées « en dur ».

```
library IEEE; use IEEE.std_logic_1164.all;
entity xorent is
  generic ( délai : TIME := 3.2 ns);
  port (a, b : in std_logic; s : out std_logic);
end xorent;

architecture simple_générique of xorent is
begin
  s <= a xor b after délai;
end;
```

5.5.2 Avec les conventions VITAL

Se pose maintenant la question d'automatiser le passage de cette valeur générique. Il est clair que le nom du générique ne peut plus être quelconque, il faut un moyen d'associer ce nom au

délai qui nous intéresse, une convention qui puisse être reconnue par des outils automatiques. La solution VITAL consiste à donner à l'argument générique un nom calculé qui contient toutes les informations utiles. Le nom du port peut ainsi être préfixé et postfixé par des indications pertinentes, et séparées par des blancs-soulignés.

5.5.2.1 Fonctionnalité

Pour l'architecture de notre **xor**, nous allons appeler une des primitives de VITAL_primitives: le XOR2 qui a le bon goût d'exister. On peut lire dans le paquetage §12.3.3 page 133 que cette procédure est ainsi déclarée :

```
procedure VitalXOR2  (
    signal          q : out std_ulogic;
    signal          a, b :  in std_ulogic;
    constant        tpd_a_q :  in VitalDelayType01    := VitalDefDelay01;
    constant        tpd_b_q :  I in N VitalDelayType01    := VitalDefDelay01;
    constant        ResultMap :  in VitalResultMapType := VitalDefaultResultMap );
```

5.5.2.2 Délais dépendant des fronts

Remarquons que les délais sont des VitalDelayType01 (voir §5.2 ci-dessus), c'est à dire des tableaux de deux valeurs spécifiant un délai différent par front (montant ou descendant). Nous n'avons pas de délai dépendant des valeurs d'entrées, probablement parce que le **xor** modélisé est supposé réalisé de façon native et pas avec la combinaison d'autre portes.

Ce type sera donc propagé dans les arguments génériques de notre entité :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.VITAL_timing.all;
use IEEE.VITAL_primitives.all;
entity xorent is
  generic (
  tpd_a_s : VitalDelayType01 ;
  tpd_b_s : VitalDelayType01   );
  port (a, b : in std_logic; s : out std_logic);
  attribute VITAL_LEVEL0 of xorent: entity is TRUE;
end xorent ;
```

Nous allons donc invoquer la procédure du paquetage de primitives *de manière concurrente* (voir §9.1.4 page 87) en lui passant les arguments attendus. Nous ignorons ici le dernier argument, qui nous permettrait de changer les valeurs de sortie selon une table prédéfinie, par exemple « transformer » tous les 1 en H si nous voulions faire une porte avec sortie *drain-ouvert*.

```
architecture vital1 of xorent is
attribute VITAL_LEVEL1 of vital1: architecture is TRUE;
begin

  VitalXOR2 (
  q => s,
  a => a,
  b => b,
  tpd_a_q => tpd_a_s,
  tpd_b_q => tpd_b_s
  );

end;
```

5.5.2.3 Processus VITAL

Si maintenant nous voulons contrôler plus finement les différents délais, il nous faut écrire un processus VITAL. Ce processus contiendra, dans l'ordre :

- Une section optionnelle sur les vérifications des contraintes de temps
- Une section « fonctionnalité »
- Une section optionnelle sur les délais par chemin

```
library IEEE; use IEEE.std_logic_1164.all;
use IEEE.VITAL_timing. all ;
use IEEE.VITAL_primitives. all;
entity xorent is
  generic (
  tpd_a_s : VitalDelayType01 ;
  tpd_b_s : VitalDelayType01   );
  port (a, b : in std_logic; s : out std_logic);
attribute VITAL_LEVEL0 of xorent: entity is TRUE;
end xorent;


architecture vital1_bis of xorent is
attribute VITAL_LEVEL1 of vital1_bis: architecture is TRUE;

  begin

  VITAL_xor : process (a,b)

    variable s_zd: std_ulogic :='U' ;
    -– contient le résultat avant intervention des délais.
    -- le type n'a pas besoin d'être résolu (d'où le 'u' de ulogic)
    -- la convention usuelle est de prendre le nom de la sortie et de
    --  postfixer par « zd » de zero-delay.

    variable s_GlitchData : VitalGlitchDataType;
    -- contient une variable nécessaire pour gérer les glitchs, même
    -- si comme ici on ne s'en sert pas.

begin

    -- ceci est l'unique instruction parlant de comportement.
    -- on appelle une fonction prédéfinie et optimisée. On ne se sert
    -- pas ici de son dernier argument qui permettrait de transformer
    -- les valeurs de sortie par une table de conversion (par exemple
    -- '1' en 'H' pour représenter la technologie drain-ouvert)
    s_zd := VitalXOR2 (
            a => a,
            b => b);

    -- ici l'appel à la primitive VITAL qui va gérer les délais
    -- on voit qu'elle prend en entrée le signal "zéro délai" et
    -- qu'elle affecte la « vraie » sortie s.
    -- la chaîne de caractères est là uniquement pour les traces
    -- et les messages à la console.
    -- On spécifie deux chemins possibles pour les délais,
    -- l'un si a change, et l'autre si b change. Les conditions
    -- sont inhibées (toujours vraies).
```

```
          VitalPathDelay01 (
                  OutSignal => s,
                  OutSignalName => "s",
                  OutTemp => s_zd,
                    Paths => (
                      0 => (InPutChangeTime => a'LAST_EVENT,
                            PathDelay => tpd_a_s,
                            PathCondition => TRUE ),
                      1 => (InPutChangeTime => b'LAST_EVENT,
                            PathDelay => tpd_a_s,
                            PathCondition => TRUE)
                       ),
                  GlitchData => s_GlitchData );
      end process ;
end;
```

Faisons un zoom sur le champ PATH: si nous voulions que des délais différents soient associés à des conditions dynamiques différentes, il suffirait d'étendre les champs élémentaires; pour reprendre l'exemple du délai qui change si l'une des entrées est à zéro (voir §5.5.1.3 ci-dessus la solution sans VITAL) :

```
                    Paths => (
                      0 => (InPutChangeTime => a'LAST_EVENT,
                            PathDelay => un_délai_approprié,
                            PathCondition => (b='0') ),
                      1 => (InPutChangeTime => b'LAST_EVENT,
                            PathDelay => un_délai_approprié,
                            PathCondition => (a='0')),
                      2 => (InPutChangeTime => a'LAST_EVENT,
                            PathDelay => un_délai_approprié,
                            PathCondition => TRUE ),
                      3 => (InPutChangeTime => b'LAST_EVENT,
                            PathDelay => un_délai_approprié,
                            PathCondition => TRUE)
                       ),
```

Les items du PATH sont évalués dans l'ordre et "c'est le premier qui gagne". C'est pourquoi les dernières conditions peuvent être attachées à TRUE, si les autres conditions sont valides elles seront exécutées avant et le délai correspondant mis en œuvre.

5.5.2.4 Etc…

Nous pourrions complexifier cet exemple *ad nauseam*, en permettant de gérer les *glitchs*, en introduisant les délais dus aux pistes de connexion, etc. Ce que nous avons voulu montrer ici, c'est que grâce à ces conventions de noms, un outil peut automatiquement forcer les bonnes valeurs génériques sur un modèle, valeurs qu'il a extraites du résultat de la synthèse ; ceci sera fait en général par des outils travaillant avec SDF (un standard de description de délais, voir note 7 page 39). Nous avons aussi voulu montrer que les conventions VITAL permettent toutes sortes de spécifications de délais collant au plus près avec la réalité des circuits représentés.
Pour forcer les valeurs des génériques à l'exécution, la déclaration de configuration permet de faire cela entièrement en VHDL, en utilisant une propriété *ad-hoc* : on peut « écraser » une valeur générique donnée par une configuration, en utilisant une autre configuration « par-dessus ». Ainsi les résultats de la synthèse, exacts, peuvent remplacer les valeurs estimées avant la synthèse, sans avoir à toucher au code source ni à le recompiler.

6 Système

La modélisation système permet de valider des idées, des protocoles ou des architectures sans entrer dans les détails de l'implémentation. Elle utilise des types abstraits, des variables partagées, des générateurs aléatoires et ses résultats peuvent être simplement statistiques. Elle peut amener à écrire des fonctions de résolution. Pour illustrer cela, nous allons prendre l'exemple de la gestion de collision sur un câble de genre Ethernet, dite CSMA (Carrier Sense Multiple Access : accès multiple avec test de la porteuse)

6.1 Illustration

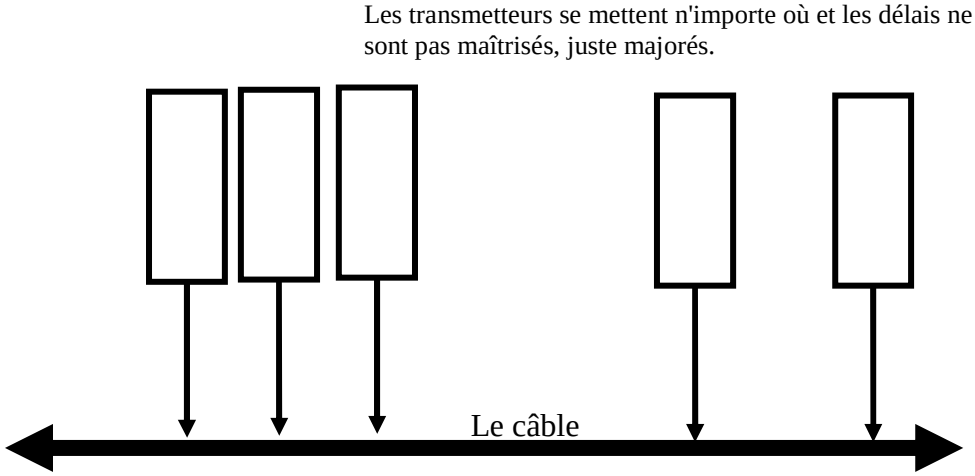


Figure 18 Un réseau CSMA

En simplifiant beaucoup, nous allons considérer que le câble est un médium commun à N transmetteurs. Chacun peut « le prendre » pour envoyer son message. Les messages, même s'ils portent une adresse de destination, peuvent être lus par tout le monde, mais nous ne nous intéressons pas aux récepteurs. Il est clair que, plus il y a de transmetteurs, moins il y a de place pour les messages de chacun. Il est clair aussi que, chaque transmetteur ignorant les autres, aucun protocole de diffusion ne peut éviter les collisions. Voyons celui qui est le plus couramment utilisé :

1. Un transmetteur attend d'avoir un message à transmettre
2. Il teste le câble pour savoir s'il est libre.
   - Si non, il attend un temps convenu et recommence en 2
   - Si oui, il prend le câble, il va en 3 en envoyant son message
3. Si, pendant la transmission, le câble est en conflit
   - Il force un long conflit pour que tout le monde soit au courant indépendamment des délais de transmission
   - Il libère le câble
   - Il attend un temps aléatoire avant de retourner en 2
4. Sinon, c'est fini et retour en 1

Ceci appelle plusieurs commentaires :

- en 3, pourquoi le câble peut-il être en conflit alors que chaque transmetteur teste avant de prendre la main? Simplement parce que deux, ou plus, transmetteurs peuvent « en même temps » (à un temps de transmission près) déterminer que le câble est libre et en même temps le prendre. Et se retrouver en conflit.
- Pourquoi forcer un conflit long, comme si on n'en avait pas assez ? Parce que, vus les délais de transmission, un transmetteur peut rater le fait qu'il y a eu conflit et croire son message correctement envoyé alors qu'il a été en collision à l'autre bout. Il faut forcer l'état de conflit plus longtemps que le plus long délai de transmission.
- Pourquoi attendre un temps aléatoire après le conflit ? Parce que tous les transmetteurs (en général 2) qui se retirent en même temps doivent revenir à des temps différents, sinon un nouveau conflit est certain.

Qu'est-ce qui **ne** nous intéresse **pas** à ce niveau ? La nature du conflit, comment on le repère, le contenu des trames des messages et leur codage. On veut rester aux notions de message, collision, état libre ou occupé du câble.

6.2 Fonction de résolution

Notre câble a trois états logiques pertinents: libre, occupé, en conflit. Nous allons faire un type énuméré abstrait pour détailler ces états.

```
type type_câble is (libre, occupé, conflit);
```

La fonction de résolution a pour objet de « dire » quel est le « vrai » état du câble quand plusieurs valeurs sont « proposées ». Dans le jargon VHDL, les valeurs proposées sont dans les pilotes (*drivers*), le vrai état est la valeur réelle (*actual value*). La fonction de résolution est capable de prendre un tableau de pilotes pour calculer la valeur réelle.

Pour définir une fonction de résolution, il nous faut un type capable de recevoir plusieurs valeurs du même type dont le nombre n'est pas dit : ce sera un tableau non contraint.

```
type ad_hoc is array (integer range <>) of type_câble;
```

La fonction aura le profil suivant:

```
function resol (arg: ad_hoc) return type_câble ;
```

On voit qu'effectivement la fonction prend en entrée un nombre quelconque de valeurs possibles, et calcule une et une seule valeur réelle.

Enfin il faut déclarer le sous-type du câble qui est le type initial contraint par la fonction de résolution:

```
subtype pour_câble is resol type_câble;
```

6.3 Générateurs pseudo-aléatoires, variables partagées

Pour tester en vraie grandeur notre système, il nous faudra des dizaines ou des centaines d'instances du même transmetteur. Mais pour que la simulation veuille dire quelque chose, il faut évidemment que chaque transmetteur ait sa vie propre, il nous faut un germe de hasard dans la simulation ; cela n'aurait aucun sens de lancer un système où tous les transmetteurs voudraient toujours faire la même chose au même instant.
Il y a dans le paquetage IEEE.MATH_REAL une procédure UNIFORM qui rend un nombre réel distribué dans [0.0 1.0[; comme tous les générateurs pseudo-aléatoires, elle a besoin d'une mémoire statique qu'elle utilise pour calculer ses séquences, laquelle doit être initialisée à une valeur quelconque au début, c'est la ou les graine(s) (*seed*). La procédure en question a besoin de deux *seeds*.

En VHDL, les signaux ont une propriété qui ne nous convient pas ici : leur valeur ne change que quand tous les processus sont sur un **wait**. Utiliser un signal pour cette mémoire statique nous donnerait le même nombre « aléatoire » à chaque appel entre deux **wait**. Il nous faut des variables. Hélas, les variables « normales » de contexte séquentiel ne peuvent être déclarées que dans les processus (ça ne va pas ici, on aura plusieurs processus et utiliser les variables de l'un empêcherait d'appeler la fonction UNIFORM avec ses *seeds* dans un autre) et dans les sous-programmes, où elles ne sont pas statiques.
Il nous faut donc utiliser une possibilité sulfureuse du langage, à ne jamais utiliser hors des simulations indéterministes : les variables partagées. Dans ce cadre simple nous n'irons pas jusqu'à utiliser les types protégés, trait du langage relativement récent[10] et qui permet de garantir l'atomicité de l'accès à la mémoire partagée en garantissant qu'au plus un processus (*process*) a accès à la mémoire en question –par exemple dans le cas, rare, où le simulateur est effectivement multi-processeurs- . En effet, même dans le cas d'école où deux processus appelleraient en même temps la fonction UNIFORM qui utilise ces *seeds*, le seul effet serait d'avoir une erreur dans le calcul de la séquence aléatoire, ce qui nous importe peu.
On déclare donc les variables partagées en zone concurrente.

```
shared variable seed1, seed2: positive := 3; -- n'importe quelle valeur
```

Ces variables seront utilisées plus tard en **inout** par la procédure UNIFORM (de IEEE.MATH_REAL) qui, à chaque appel, met à jour sa variable de sortie (mode **out**) laquelle suivra cette fameuse séquence aléatoire.

```
procedure uniform(variable seed1, seed2 : inout positive ;x: out real);
```

Tout ceci nous donne la déclaration de paquetage que voici:

```
package pkg_réseau is

 type type_câble is (libre, occupé, conflit);
 type ad_hoc is array (integer range <>) of type_câble;
 function resol (arg: ad_hoc) return type_câble ;
 subtype pour_câble is resol type_câble;

 shared variable seed1, seed2: positive := 3;
                 -- pourquoi 3? Et pourquoi pas?

end pkg_réseau;
```

[10] L'utilisation des types protégés serait obligatoire avec les implémentations de VHDL 2000.

Le code de la fonction de résolution doit être dans le corps de paquetage. Rappelons qu'il s'agit de faire quelque chose de pertinent lorsque le câble est attaqué par 0, 1, 2, ou plus, transmetteurs.

L'algorithme est simple : on parcourt tous les éléments du tableau, et on se souvient dans MEM de l'état précédemment calculé. Si le câble est (encore) *libre* et qu'on a une demande d'occupation, MEM passe à *occupé*. Si le câble est *occupé* et qu'il y a une seconde demande d'occupation, MEM passe à *conflit* et c'est fini, on sort de la boucle devenue inutile. Si le câble est déjà en conflit, on ne change rien et on sort aussi. À la fin on rend MEM.

```
package body pkg_réseau is

 function resol (arg: ad_hoc) return type_câble is
   variable mem:type_câble:=libre;
 begin
   for i in arg'range loop
               -- on balaye toutes les contributions
     case  mem is
        when libre => mem:=arg(i);
        when occupé => if arg(i)/=libre then
                          mem:=conflit;
                          exit;
                       end if;
        when conflit => exit;
     end case;
   end loop;
   return mem;
 end function resol;

end pkg_réseau;
```

6.4 Protocole

L'entité du transmetteur va être générique sur tous les paramètres que l'on aura envie de faire bouger pour valider notre système: tous les temps dont nous avons parlé plus haut; nous lui passons aussi un numéro qui permettra de l'identifier dans les messages console, chaque transmetteur étant capable de faire un message signé.

Cette entité a un seul port : le câble de transmission.

```
use work.pkg_réseau.all;
entity transmetteur is

 generic (mon_numéro: integer;
          temps_moyen_entre_tentatives,
          temps_moyen_pour_réessai,
          temps_moyen_d_une_trame,
          temps_de_réaction_pour_prendre_le_câble,
          temps_d_attente_si_câble_occupé,
          temps_de_forçage:time);

 port (câble: inout pour_câble);

end entity transmetteur;
```

Nous voici maintenant dans le cœur du problème. En fait il nous suffit de coder en VHDL les étapes détaillées en 6.1 ci-dessus.
Cela nécessite une petite cuisine dont voici les secrets : Les conversions de types sont là pour pouvoir utiliser un réel (le nombre aléatoire) pour définir un temps. La multiplication par 1000 et la division par 1000 dans la même expression parce que le passage par des entiers tronque les réels, on garde donc l'équivalent de 1000 positions discrètes possibles pour nos temps aléatoires. La multiplication par 2 (qui fait les 2000) parce que les temps spécifiés sont supposés être des temps moyens, et que le nombre aléatoire a, lui, une moyenne de 0.5.

Dans le code ci-après, des étiquettes sont mises en marge pour faire la correspondance avec l'algorithme décrit en 6.1 ci-dessus.

On se rappellera que l'expression **`wait until`** `condition` **`for`** `temps` a pour effet d'arrêter le processus jusqu'à ce que la condition soit vraie (il faut au moins un signal dedans sinon elle n'est jamais évaluée) mais il y a un temps maximum au bout duquel le processus repartira, condition vraie ou pas.
On voit aussi que, chaque fois qu'un nombre aléatoire est invoqué, la procédure UNIFORM est appelée juste avant.

```
library ieee; use ieee.math_real.all;
architecture arch of transmetteur is
begin
émission:  process
    variable rnd: real;
  begin
    uniform(seed1, seed2, rnd);
    wait for (integer(rnd*2000.0)*
              temps_moyen_entre_tentatives)/1000;
    loop
      while câble /= libre loop
        wait until câble=conflit
             for temps_d_attente_si_câble_occupé;
      end loop;
      wait for temps_de_réaction_pour_prendre_le_câble;
      câble <= occupé;
      uniform(seed1, seed2, rnd);
      wait until câble=conflit for (integer(rnd*2000.0)*
                                    temps_moyen_d_une_trame)/1000;
      if câble=conflit then
        report "conflit dans transmetteur numéro "
                & integer'image(mon_numéro);
        câble<=conflit;
        wait for temps_de_forçage;
        câble<=libre;
        uniform(seed1, seed2, rnd);
        wait for (integer(rnd * 2000.0) *
                    temps_moyen_pour_réessai)/1000 ;
      else
        câble <= libre;
        exit;
      end if;
    end loop;
  end process émission;
end architecture arch;
```

1

2

3

6.5 Tester

Pour tester tout ça, nous allons faire classiquement une boîte noire (entité sans ports ni génériques), …

```
entity réseau is
end entity réseau;
```

…dont l'architecture instanciera des transmetteurs *ad libitum*. En changeant les valeurs des constantes, voire en en donnant de différentes à chaque transmetteur ce qui n'est pas fait ici, on pourra observer quand le câble est saturé par les collisions, mesure qui n'est pratiquement accessible qu'à l'expérience.

L'architecture contient la déclaration du câble Ethernet. Nous allons nous passer de la déclaration de composant avec configuration, et utiliser l'instanciation directe ce qui est légitime dans un modèle de test.

```
use work.pkg_réseau.all;
architecture arch of réseau is
signal câble: pour_câble;

begin
G: for i in 1 to 100 generate   -- 100 transmetteurs
 T: entity work.transmetteur(arch)
    generic map (i,
                 temps_moyen_entre_tentatives => 500 us,
                 temps_moyen_pour_réessai => 500 us,
                 temps_moyen_d_une_trame => 10 us,
                 temps_de_réaction_pour_prendre_le_câble => 1us,
                 temps_d_attente_si_câble_occupé => 50 us,
                 temps_de_forçage => 30 us)
    port map (câble);
end generate;

assert câble /=conflit report "câble en conflit" severity note;

end architecture arch;
```

Les messages console seront de deux sortes : chaque transmetteur contient une instruction **report** qui marque le conflit et dit son numéro. Par surcroît, l'architecture de test contient une assertion.

On peut maintenant simuler le modèle, et rassembler dans des variables (partagées) ou des fichiers des informations statistiques en lançant la simulation pendant un temps qui peut être très long. On s'aperçoit que le début de la simulation contient énormément de collisions entre de nombreux transmetteurs, en effet dans ce modèle simple les transmetteurs ne se désynchronisent qu'au bout d'un certain temps. À ce moment là, l'essentiel des conflits concernera 2 transmetteurs. Si l'on exploite les traces, il faudra tenir compte de cette simplification, ou alors changer un peu l'algorithme pour qu'il fonctionne correctement dès le début.

L'objectif est évidemment de déterminer pour quels paramètres on a le risque d'une coagulation générale du réseau, chaque transmetteur n'observant le protocole que pour mieux

revenir contribuer à l'embouteillage. Dans les « vrais » systèmes, pour éviter la coagulation du système, on s'en tire en faisant patienter les utilisateurs finaux et en augmentant les délais de réessais de l'étape 3 chaque fois qu'il y a échec. Ce qui ne fait que repousser l'attente aux clients du système, mais il faut bien que quelqu'un attende s'il y a trop de candidats.

Une fois le concept validé, le concepteur intrépide pourra attaquer le niveau logique, où les messages sont des trames codées en NRZ (voir §9.2.2 page 88) avec des drapeaux HDLC et une insertion de zéros (§7.2.2 page 67), où le conflit se repère au fait que la valeur moyenne du signal n'est pas 0.5 (comptage sur un temps donné) et où le forçage de conflit est simplement la mise à 0 du câble pendant un temps suffisamment long, lequel fonctionne selon le mode collecteur ouvert.

7 Comportemental

7.1 Comportemental non synthétisable

Il est fréquent d'avoir à écrire des blocs qu'on veut simuler mais ne pas synthétiser. C'est le cas général pour les structures régulières, dont on veut une instance dans le modèle pour faire marcher le reste, mais qui, *in fine*, seront des blocs préexistants fournis par le fondeur.

7.1.1 RAM

Nous voulons une RAM classique qui comporte un ordre de lecture/écriture : *rw* pour read/write, et un ordre de validation : *cs* pour chip select. Nous allons rendre cette RAM générique sur le nombre de mots et le nombre de bits par mots.

```
library ieee;
use ieee.std_logic_1164.all;
entity RAM is
  generic (nb_bits_adresse: positive:=16;
           nb_bits_données: positive := 8);
  port(
       ADDRESS: in STD_LOGIC_VECTOR(nb_bits_adresse-1 downto 0);
       DATA: inout STD_LOGIC_VECTOR(nb_bits_données-1 downto 0);
       rw,cs: in STD_LOGIC
       );
end entity RAM;
```

L'architecture est simple: sur le front montant de *cs*, on teste l'état de *rw* (qui, juste testé, ne déclenche donc pas le processus) pour, selon le cas, mettre le contenu du bus d'entrée dans la mémoire, ou l'inverse. Quand *cs* retombe, le bus est mis à "Z" ce qui signifie que la RAM le "lâche". On utilise les fonctions de conversion du paquetage *std_logic_arith*, que l'on trouvera en §12.2.2 page 123.

```
library ieee; use ieee.std_logic_arith.all; -- pour “conv_integer”
architecture BEH of RAM is
begin
  process (cs)
    type RAM_MEM is array(0 to 2**nb_bits_adresse-1)
                 of STD_LOGIC_VECTOR (nb_bits_données-1 downto 0);
    variable MEM: RAM_MEM:=(others=>(others=>'0'));
  begin
      if cs='1' then
        if rw='1' then – écriture
              MEM(conv_integer(unsigned((ADDRESS)))):=DATA;
        else        -- lecture
              DATA<=MEM(conv_integer(unsigned((ADDRESS))));
        end if;
      else
        DATA<=(others=>'Z');
      end if;
  end process;
end architecture BEH;
```

7.1.2 Grande RAM : optimisation

Si la taille du tableau est un problème, on peut le compacter en le transformant en tableau de *bit_vectors*. En effet le contenu d'une RAM est entièrement composé de 0 et de 1 ! Pas de Z ou de L. Le prix à payer est la performance : il faudra deux conversions de type pour venir et aller au bus : *to_stdlogicvector* et *to_bitvector*, fonctions qui sont dans le paquetage STD_LOGIC_1164.

Ci-dessous les portions de code à changer : la déclaration du type, et l'utilisation de la variable du type.

```
    type RAM_MEM is array(0 to 2**nb_bits_adresse-1)
                 of BIT_VECTOR (nb_bits_données downto 0);
…
    if rw='1' then – écriture
         MEM(conv_integer(unsigned((ADDRESS)))):= to_bitvector(DATA);
    else        -- lecture
          DATA<= to_stdlogicvector(MEM(conv_integer(unsigned((ADDRESS)))));
    end if;
```

7.1.3 Très grande RAM : liste chaînée

La question est ici que l'on veut modéliser une RAM très, très grande, qui pulvériserait la mémoire du simulateur si l'on devait adopter la solution « tableau ». On peut de même modéliser un disque dur ou toute autre système ayant une grande quantité de mémoire adressable.
Il n'y a pas de miracle toutefois : cela n'est possible que parce que l'on va n'utiliser qu'une très petite partie de la mémoire adressable. Au lieu de réserver une fois pour toutes un tableau immense, on mettra en liste des granules contenant à la fois l'adresse de l'objet, et sa valeur, et l'adressage consistera à aller courir sur cette liste pour trouver la cellule qui porte la bonne adresse.

Un type récursif en VHDL : liste chaînée

```
type cellule;
 type pointeur is access cellule;
 type cellule is record
   …champs pertinents…
   suivant: pointeur;
 end record;
```

Au final, le paradoxe est qu'effectivement on a un espace d'adressage immense, mais on peut stocker moins d'objets que dans les solutions précédentes, et avec beaucoup moins d'efficacité. En effet chaque valeur porte, en sus, son adresse et le pointeur vers la suivante.

```
library ieee;
use ieee.std_logic_arith.all;
architecture IMMENSE of RAM is
begin
  process (cs)
  --------------- déclaration de type récursif en VHDL
  type cellule;
  type pointeur is access cellule;
  type cellule is record
    adresse: integer;
    donnée: bit_vector(nb_bits_données-1 downto 0);
    suivant: pointeur;
  end record;
  ---------------
```

```
  variable MEM, UTIL: pointeur;
  variable adresse_integer: integer;
  -- si on voulait plus de (32 ?) bits d'adresse, il suffirait
  -- de ne pas convertir en entier et de stocker l'adresse
  -- sous forme de tableau de bits. Ça irait encore moins vite.
  begin
  adresse_integer:=conv_integer(unsigned((ADDRESS)));

  --------------- traitement de la liste; trois cas de figure
  if MEM = null then
   -- liste vide, on crée une cellule et ce sera la bonne.
   MEM := new cellule;
   MEM.adresse:=adresse_integer;
   UTIL:=MEM; -- on a le pointeur vers la zone à lire ou écrire
   MEM.suivant:=null;
  elsif MEM.adresse=adresse_integer then
   -- liste non vide, le premier élément est le bon.
   UTIL := MEM;
  else
   --  MEM n'est pas nul et le premier élément n'est pas le bon.
   UTIL:=MEM;
   while UTIL/=null loop – on parcourt la liste pour chercher l'adresse
     if UTIL.adresse=adresse_integer then
       exit; -- on sort de la boucle, UTIL est à la bonne valeur
     end if;
     UTIL:=UTIL.suivant; -- on n'a pas trouvé, on va tester le suivant
   end loop;
   if UTIL=null then    -- on est arrivé au bout de la liste sans trouver
     UTIL:=new cellule; -- alors il faut créer une nouvelle cellule
     UTIL.adresse:=adresse_integer;
     UTIL.suivant:=MEM; -- on l'attache en tête, pourquoi pas.
     MEM:=UTIL;         -- on a mis un nouveau maillon en tête de liste.
   end if;
  -- ici UTIL vise forcément la case qui nous intéresse, et qui existe,
  -- pour laquelle le champ adresse est correct.
  end if;

-- traitement classique de lecture/écriture:
  if cs='1' then
    if rw='1' then -- ecriture
       UTIL.donnée:=to_bitvector(DATA);
    else        -- lecture
       DATA<=to_stdlogicvector(UTIL.donnée);
    end if;
  else
    DATA<=(others=>'Z');
  end if;
  end process;
end architecture IMMENSE;
```

Ce traitement « par liste » peut évidemment être compliqué à loisir si la performance est un problème : on pourrait trier la liste (voir les algorithmes d'insertion dans une liste déjà triée) ; on pourrait aussi gérer N listes qui seraient donc N fois plus courtes, N étant calculé à partir de l'adresse désignée et servant à indexer un tableau de N pointeurs débuts de listes. Ceci est de l'algorithmique et sort du cadre de ce manuel.

7.1.4 ROM

À première vue, la ROM pourrait sembler être plus simple que la RAM, une sorte de RAM sans commande d'écriture. En fait, cela est exact pour ce qui est de la partie fonctionnelle,

après le temps 0. Mais hélas la ROM pose la question de son initialisation. Il y a plusieurs façons de procéder, nous allons voir les deux extrêmes : dans le code, hors du code.

7.1.4.1 Rom à initialisation par agrégat

Dans le code, VHDL fournit la possibilité d'écrire des constantes de types structurés sous la forme d'agrégats. C'est très simple mais hélas une mauvaise idée en général. Si la ROM s'initialise en VHDL, il devient impossible de changer son contenu sans recompiler du VHDL : cela suppose qu'on ne peut pas donner le modèle à quelqu'un qui n'a pas le moyen de recompiler en contexte. Et il faut donner le code source, on ne peut pas se contenter de donner du binaire.

Néanmoins, dans le cas d'usage interne, on pourra trouver une utilité à cette méthode. Voici un exemple non générique (on aurait du mal à écrire un agrégat générique).

```
library ieee;
use ieee.std_logic_1164.all;
entity ROM is
  port(
       ADDRESS: in STD_LOGIC_VECTOR(15 downto 0);
       DATA: inout STD_LOGIC_VECTOR(15 downto 0);
       cs: in STD_LOGIC
       );
end entity ROM;

library ieee;
use ieee.numeric_std.all; -- pour les fonctions “to_integer”
architecture BEH of ROM is
begin
  process (cs)
    type ROM_MEM is array(0 to 2**15)
                 of STD_LOGIC_VECTOR (15 downto 0);
    variable MEM: ROM_MEM:=
       (0 => "1110001110001111",[11]
        1 => "0000000000000000",
        2 | 3 => "1111000011110000", -- utilisation de la conjonction
        4 to 10 => "1111111111111111", -- utilisation de l'étendue
        others=> "0000000000000000"); -- utilisation de la clause others
  begin
    if cs='1' then
      DATA<=MEM(to_integer(unsigned((ADDRESS))));
    else
      DATA<=(others=>'Z');
    end if;
  end process;
end architecture BEH;
```

7.1.4.2 Rom à initialisation par fichier

Supposons maintenant que nous voulons lire les données de la ROM dans un fichier: voilà un algorithme qui doit se dérouler une fois et une seule, avant le temps 0 de la simulation. Le processus décrivant la ROM aura donc, en premier lieu, une phase de lecture de fichier, sans

[11] On peut ben sûr utiliser la notation hexadécimale: x"ABC0" ou octale: o"74523", on peut aussi aérer les constantes avec des blancs-soulignés, dans toutes les bases.

wait de façon à s'exécuter au début de la simulation. Puis il y aura une boucle infinie, contenant le « **wait on** cs » qui réglera le fonctionnement de la ROM au moment de simuler.

7.1.4.2.1 Fichier simple binaire

Voyons d'abord le cas simplissime du fichier contenant des 0 et des 1 en rangs d'oignon, avec une ligne par mot. Ce cas est simple à traiter mais l'écriture du fichier sera une vraie punition (aucun outil ne le fait automatiquement, et c'est particulièrement illisible).

La déclaration d'entité est générique, et contient le nom du fichier à lire.

```
library ieee;
use ieee.std_logic_1164.all;
entity ROM is
  generic (nb_bits_adresse: positive:=16;
           nb_bits_données: positive := 8;
           init_file: STRING:="C:\bla.txt");
  port( ADDRESS: in STD_LOGIC_VECTOR(nb_bits_adresse-1 downto 0);
        DATA: out STD_LOGIC_VECTOR(nb_bits_données-1 downto 0) ;
        cs: in STD_LOGIC);
end entity ROM;


use std.textio.all;
library ieee;
use ieee.std_logic_textio.all;
use ieee.numeric_std.all;
architecture BEH of ROM is
begin
  process
    type ROM_MEM is array(2**nb_bits_adresse -1 downto 0)
        of STD_LOGIC_VECTOR (nb_bits_données-1 downto 0);
    file data_file:text open read_mode is init_file;
    variable MEM: ROM_MEM;
    variable L:line;
    variable index:integer:=-1;
    begin
                          while not endfile(data_file) loop
Code sans wait              index:=index+1;
                            readline(data_file,L); -- de textio
                            read(L, MEM(index)); -- de std_logic_textio
                          end loop;

                      -- début de la boucle infinie qui contient le wait
                          loop
Boucle                      wait on cs;
infinie                     if cs='1' then
avec wait                     DATA<=MEM(to_integer(unsigned((ADDRESS))));
                            else
                               DATA<=(others=>'Z');
                            end if;
                          end loop;
  end process;
end architecture BEH;
```

Le format du fichier texte est donc aussi simple que rustique : une ligne de 0 et de 1, en quantité nécessaire et suffisante, par mot de la mémoire. On peut compliquer à loisir le format d'entrée pour, par exemple, le rendre compatible avec un standard (voir ci-dessous §7.1.4.2.2) ou lui faire accepter des commentaires.

On utilise ici le paquetage STD_LOGIC_TEXTIO qui n'est pas standard (il le sera avec les implémentations VHDL 2008) mais très largement diffusé en code source (§12.1.3 page 118). Il nous permet ici de lire une chaîne de caractères sous la forme de std_logic_vector. Les préfixes *textio* et *std_logic_textio* ont été laissés dans le code pour illustrer cela, bien qu'ils soient inutiles par la grâce des clauses **use** qui sont en début de l'architecture.

7.1.4.2.2 Lecture d'un fichier au format INTEL

Il s'agit ici d'un exercice consistant à lire le format INTEL qui est largement utilisé, et qui va nous permettre de survoler quelques constructions algorithmiques ainsi que l'utilisation des instructions **assert**:

- Le format en question est formé de lignes commençant toujours par ":".
- Il y a ensuite un octet (deux caractères) qui donne la longueur de la donnée utile (le nombre d'octets à mettre dans la ROM depuis cette ligne, maximum 256 donc),
- puis l'adresse de début sur deux octets (quatre caractères, maximum 65536 en version non étendue),
- puis un indicateur disant si la ligne est une ligne de données (00), un marqueur de fin de fichier (01) ou une ligne avec adresse étendue (02) pour aller au-delà de 2 octets d'adresse (non traité ici), et d'autres extensions que nous ne traiterons pas non plus ici.
- les données éventuelles
- et enfin un checksum calculé en faisant le complément à 256 de la somme modulo 256 des octets qui sont passés sauf le premier (le ":").

Voici un exemple d'un tel fichier:

```
:10000000C29FD29EE587D2E7F587758920758DFD61
:10001000758BFDD28E753061753162753263753 3C3
:10002000007830E6600EA2D092E7F599083099FD8D
:06003000C29980EF80FE82
:00000001FF
```

Figure 19 Exemple de fichier au format INTEL

La première ligne découpée comme dit ci-dessus

```
: 10 0000 00 C29FD29EE587D2E7F587758920758DFD 61
```

…nous dit qu'il y a 16 octets ("10" hexa), qu'on va écrire à l'adresse 0 et suivantes ("0000"), que c'est une ligne de données (le marqueur "00"), que les données sont C2 9F D2 9E E5 87 D2 E7 F5 87 75 89 20 75 8D FD, et qu'il manque 61 pour que la somme de tous les octets fasse 256.

Nous ne nous intéresserons donc pas dans ce petit exemple aux adresses étendues et autres extensions qui n'ont rien de difficile mais surchargeraient ce manuel:

Afin d'isoler la question, faisons un paquetage dédié à la lecture des formats INTEL et que nous appellerons *pour_INTEL*. La procédure devra charger un tableau découpé en mots de 8, 16, 32 ou plus bits alors que le format Intel spécifie des octets et des adresses d'octets. S'il se trouve que le tableau-mémoire que l'on veut charger est constitué de mots de 8 bits, le code peut être élagué de bien des assertions (modulos) et de quelques calculs. Cela étant, ce code est assez général pour pouvoir charger n'importe quel type de tableau ; il faut quand même que le mot-mémoire soit un multiple de 8 bits !

Comme VHDL ne connaît pas les paquetages génériques (ce qui changera avec l'arrivée des implémentations de VHDL 2008, voir §1.7.1.9 page 11), le paramétrage se fera par une constante unique déclarée dans la spécification. Le changement de cette constante implique donc une recompilation.

```
library ieee;
use ieee.std_logic_1164.all;
package pour_INTEL is
    constant largeur_mot : integer:=32;
               -- c'est ici que l'on decide de la taille du mot,
               -- tout le reste est calculé
    type ROM_MEM is array(integer range <>)
                 of STD_LOGIC_VECTOR (largeur_mot-1 downto 0);
    procedure lire_intel(ou: out ROM_MEM; fichier: in string) ;
end;
```

Le corps de ce paquetage contient uniquement le corps de la procédure déclarée et quelques fonctions de conversion utiles :

```
library ieee;
use ieee.std_logic_arith.all;
use std.textio.all;
package body pour_INTEL is

--=============================================================
--Trois fonctions de service convertissant les caractères hexa
--en valeurs entières, dans les deux sens, et convertissant un entier
--en std_logic_vector de 8 bits

    function to_int(c:character) return integer is
          -- convertir un caractère hexa en valeur entière
    begin
     if (c>='0') and (c<='9') then
       return character'pos(c)-character'pos('0');
     elsif (c>='A') and (c<='F') then
       return character'pos(c)-character'pos('A') +10;
     elsif (c>='a') and (c<='f') then
       return character'pos(c)-character'pos('a') +10;
     else report "erreur valeur hexa incorrecte:"& c severity ERROR;
       return 0;
     end if;
    end;

    function to_char(i:integer) return character is
          -- convertir un entier en sa représentation "caractère hexa"
    begin
      if (i>=0) and (i<=9) then
        return character'val(i+character'pos('0'));
```

```
      elsif i>=10 and i<=15 then
        return character'val(i-10+character'pos('A'));
      else report "erreur valeur hexa incorrecte:"& integer'image(i)
                  severity ERROR;
        return 'X';
      end if;
    end;

    function to_std8 (arg: integer) return std_logic_vector is
     -- convertit un entier en STD_LOGIC_VECTOR(7 downto 0)
     -- le choix est d'accepter un entier et de vérifier ensuite
     -- explicitement ses bornes (0-255), on aurait pu aussi
     -- déclarer un sous-type d'entier et laisser VHDL vérifier
     -- les bornes.

      variable res : std_logic_vector(7 downto 0):=(others=>'0');
      variable arg1, ix : integer :=0;
    begin
      assert (arg>=0) and (arg<256)
        report "conversion impossible " & integer'image(arg)
        severity error;
        arg1:=arg;
        while arg1>0 loop
          if arg1 mod 2 =0
            then res(ix):='0';
            else res(ix):='1';
          end if;
        arg1 := arg1/2;
        ix:=ix+1;
        end loop;
      return res;
    end;

--=====================================================================
  procedure lire_intel(ou: out ROM_MEM; fichier: in string) is
    -- lit « fichier » au format Intel dans « OU ».

    file data_file:text open read_mode is fichier;
    variable L:line;
    variable num_ligne,nb_bytes,adresse_debut,sum,mot,
             nb_oct_par_mot, taille_mem:integer;
    variable tmp_res: std_logic_vector(OU(1)'LENGTH -1  downto 0);
             -- tmp_res est un vecteur de la taille du mot de la mémoire.

  begin
    num_ligne:=0;
    taille_mem:=OU'LENGTH;
          -- la taille du tableau passé en argument.

    assert OU(1)'LENGTH mod 8 = 0
      report "la taille du mot mémoire n'est pas un multiple de 8"
      severity ERROR;

    nb_oct_par_mot:=OU(1)'LENGTH/8;
          -- OU(1)'LENGTH est la taille du mot du tableau passé
          -- en argument, égale à «largeur_mot» . En divisant par 8
          -- on a le nombre d'octets mot.

    report "taille mémoire:" & integer'image(taille_mem)
           & " nb octets par mot:" & integer'image(nb_oct_par_mot)
      severity note ; -- un simple message informatif
```

```
while not endfile(data_file) loop  -- on lit le fichier ligne à ligne
      num_ligne:=num_ligne+1;

      readline(data_file,L); -- de textio

      report L.all severity note; -- on fait l'écho à titre d'information.

        --  le premier caractère doit être un ':'
      assert L(1)=':'
        report
        "fichier ROM pas conforme au standard INTEL; pas de ':' ligne "
              &integer'image(num_ligne)
        severity ERROR;

        -- Les deuxième et troisième caractères contiennent
        -- en hexa le nombre d'octets, qui doit être ici multiple
        -- du nombre d'octets par mot mémoire. C'est « seulement » un
        -- warning, les derniers octets d'une ligne seront ignorés s'ils
        -- sont en nombre insuffisant. L'autre branche de l'alternative
        -- est de bourrer la ligne en question dans le fichier avec des
        -- « 00 » en nombre suffisant devant le checksum, ou alors de
        -- modifier le code pour allonger L avec des « 00 », toujours
        -- devant l'octet de checksum.
        nb_bytes := 16*to_int(L(2)) + to_int(L(3));
        assert (nb_bytes mod nb_oct_par_mot) =0
          report "Nombre d'octets pas conforme avec la taille du mot"
                 & " dans le fichier ROM ligne "
                 & integer'image(num_ligne)&", il y aura troncature"
          severity WARNING;

        -- Les caractères 4 à 7 contiennent l'adresse de
        -- début de la séquence en hexa. On vérifie que l'adresse spécifiée
        -- (qui vise des octets dans le format Intel) est un multiple du
        -- nombre d'octets par mot-mémoire. Ensuite on divise l'adresse
        -- par ce nombre.
        adresse_debut:= 4096*to_int(L(4))+
                        256*to_int(L(5))+
                        16*to_int(L(6))+
                        to_int(L(7));
        assert (adresse_debut mod nb_oct_par_mot) =0
          report
          "Adresse de début pas conforme avec la taille du mot "
          &" dans le fichier ROM ligne "&integer'image(num_ligne)
          severity ERROR;
        -- nos adresses visent des mots de X x 8 bits,
        -- le standard INTEL des mots de 8 bits, il faut diviser.
        adresse_debut := adresse_debut/nb_oct_par_mot;


        --  le huitième caractère doit être un '0'
        assert L(8)='0'
          report "Ligne fichier ROM pas conforme au standard INTEL; "
                 &"pas de '0' en position 8 ligne "
                 &integer'image(num_ligne)
          severity ERROR;
```

```
-- le neuvième caractère nous dit de quel type est
-- la ligne d'octets
case L(9) is
  when '0' =>  -- ligne de données

   -- la boucle extérieure parcourt le tableau mémoire mot par mot
    for iy in 0 to nb_bytes/nb_oct_par_mot-1 loop

      -- la boucle intérieure parcourt la ligne octet par octet.
      for ix in 0 to nb_oct_par_mot-1 loop
        mot :=  16 * to_int(L(10+2*iy*nb_oct_par_mot+2*ix));
        mot := mot+ to_int(L(10+2*iy*nb_oct_par_mot+2*ix+1));
        -- mot contient la valeur de l'octet courant
        -- on va le transformer en vecteur de 8 bits et l'affecter
        -- à la tranche idoine du mot tampon
        tmp_res(tmp_res'high - 8*ix downto tmp_res'high - 8*ix -7)
              := to_std8(mot);
      end loop;
      -- mot tampon est affecté à l'adresse concernée du
      -- tableau mémoire.
      ou(adresse_debut+iy) := tmp_res;
      -- les variables mot et tmp_res ne sont là que pour
      -- la lisibilité qui en a bien besoin. Tout ce code pourrait
      -- être remplacé par
          -- for iy in 0 to nb_bytes/nb_oct_par_mot-1 loop
          --   for ix in 0 to nb_oct_par_mot-1 loop
          --    ou(adresse_debut+iy)
          --        (tmp_res'high-8*ix downto tmp_res'high-8*ix-7)
          --  := to_std8(16* to_int(L(10+2*iy*nb_oct_par_mot+2*ix))
          --           + to_int(L(10+2*iy*nb_oct_par_mot+2*ix+1)));
          --   end loop;
          -- end loop;
    end loop;

    -- Vérification du checksum: la somme de tous les octets
    -- y compris les premiers et le checksum doit faire
    -- 0 modulo 256.
    sum:=0;
    for ix in 2 to 2*(nb_bytes)+9 loop
      if ((ix+2) mod 2)=0 then
        sum:= (sum+(16*to_int(L(ix)))) mod 256;
      else
        sum:= (sum+to_int(L(ix))) mod 256;
      end if;
    end loop;
    -- sum est la somme mod 256 de tous les octets sauf le dernier.
    sum:=256-sum;  -- ceci doit donc être la valeur du dernier.

    assert 16*to_int(L(2*nb_bytes+10))
                    +to_int(L(2*nb_bytes+11))
                    = sum
      report "Checksum faux ligne "&integer'image(num_ligne) &
             " trouvé "&L(2*nb_bytes+10 to 2*nb_bytes+11) &
             " attendu " & to_char(sum/16)& to_char(sum mod 16)
      severity ERROR;
```

```
          when '1' =>  -- fin de fichier
            assert (nb_bytes=0)
              report "Fin de fichier avec longueur non nulle ligne "
                     &integer'image(num_ligne)
              severity ERROR;
            assert L(10 to 11) ="FF" – le checksum est forcément FF.
              report "Erreur checksum sur fin de fichier ligne "
                      &integer'image(num_ligne)
              severity ERROR;
            exit;

          when others => -- extension non gérée ici
            report
                "type de ligne non géré dans le fichier ROM, ligne "
                &integer'image(num_ligne)
              severity ERROR;
          exit;
        end case;
      end loop;
    end;
  end;
--============================================================================
```

L'entité sera non générique, pour simplifier un peu le code:

```
library ieee;
use ieee.std_logic_1164.all;
entity ROM is
  generic (
     init_file: STRING:="intel.txt"); -- le nom du fichier
  port(
       ADDRESS: in STD_LOGIC_VECTOR(15 downto 0);
       DATA: inout STD_LOGIC_VECTOR(WORK.pour_INTEL.largeur_mot downto 0);
       CS: in STD_LOGIC
       );
end entity ROM;
```

L'architecture appelle simplement la procédure du paquetage:

```
library ieee;
use ieee.std_logic_arith.all;
use work.pour_INTEL.all;
architecture BEH of ROM is
begin
  process
    variable MEM: ROM_MEM(0 to 4095);
    begin

    lire_intel(MEM, init_file); -- lecture du fichier au format INTEL

--=========================================
-- début de la boucle infinie qui contient le wait
      loop
        wait on cs;
        if cs='1' then
          DATA<=MEM(conv_integer(unsigned((ADDRESS))));
        else
           DATA<=(others=>'Z');
        end if;
```

```
      end loop;
  end process;
end architecture BEH;
```

7.2 Comportemental synthétisable

7.2.1 Conversion parallèle série

On veut un système à qui l'on entre un mot de N bits en parallèle (disons 8) et qui sorte, au rythme d'une horloge, ces bits un par un.
L'entité aura donc : une *entrée* sur 8 bits et sa *validation* sur front, et une *sortie* sur un bit. Une entrée d'*horloge* et une de *reset* donc on a toujours besoin. Un signal *occupé* indique côté parallèle que la conversion n'est pas terminée.

```
library ieee; use ieee.std_logic_1164.all;
entity sérialisateur is
 port (entrée:  std_logic_vector(7 downto 0);
       reset: in std_logic;
       validation: std_logic;
       horloge: std_logic;
       occupé: out std_logic;
       sortie: out std_logic);
end entity sérialisateur;
```

La réalisation se fera en stockant l'entrée dans un registre interne (un signal) et en l'explorant bit à bit avec un indice de boucle (i) C'est un cas de figure synthétisable parce que les bornes de la boucle sont des constantes statiques, ce serait plus délicat s'il s'agissait de variables.

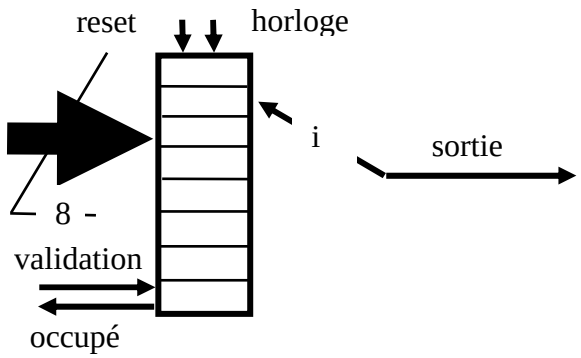


Figure 20 Sérialisateur

On notera que le signal occupé compte sur la bonne discipline du côté « gauche » : si l'on se mêle de recharger le registre avant que la conversion ne soit terminée, l'affaire recommence à zéro. On pourrait rendre le bloc « aveugle » à un nouveau rechargement pendant ce temps, c'est une affaire de choix.

```
architecture beh of sérialisateur is
  signal registre: std_logic_vector(7 downto 0);
begin
 process (horloge, validation, reset)
   variable i : integer := 0;
   variable trame_en_cours : boolean := false;
 begin
  if reset='1' then
      sortie<='Z';
      occupé<='0';
      i := 0;
     trame_en_cours := false;
  elsif validation'event and validation='1' then
    registre <= entrée;
    occupé <= '1';
    i:=0;
    trame_en_cours := true;
  elsif horloge'event and horloge = '1' and trame_en_cours then
   sortie<=registre(i);
   i:=i+1;
   if i=8 then
     i := 0;
     occupé <= '0';
     trame_en_cours := false;
     end if;
  end if;
 end process;
end architecture beh;
```

7.2.2 Drapeau HDLC

Quand les données se promènent par paquets et en série sur un câble, il est important de pouvoir signaler le début et la fin d'une trame. Une convention classique est ce qu'on appelle le drapeau HDLC (***H**igh-Level **D**ata **L**ink **C**ontrol)*, la séquence 01111110, c'est-à-dire six un encadrés par deux zéros.
La question qui se pose alors, c'est que les données encadrées par ces drapeaux peuvent contenir, par hasard, la fameuse séquence. La solution est simple; entre deux drapeaux, on insère un zéro chaque fois qu'il y a une séquence de 5 uns consécutifs.
Côté réception, chaque fois qu'il y a cinq uns consécutifs, soit le bit suivant est un 0 et il faut l'enlever, soit c'est un 1 et alors on a un début de drapeau, le bit suivant est 0 (ou 1 pour d'autres situations non documentées ici.). Du coup il y a plus de bits qui passent sur le canal de transmission que de bits d'information, ce qui suppose une synchronisation.

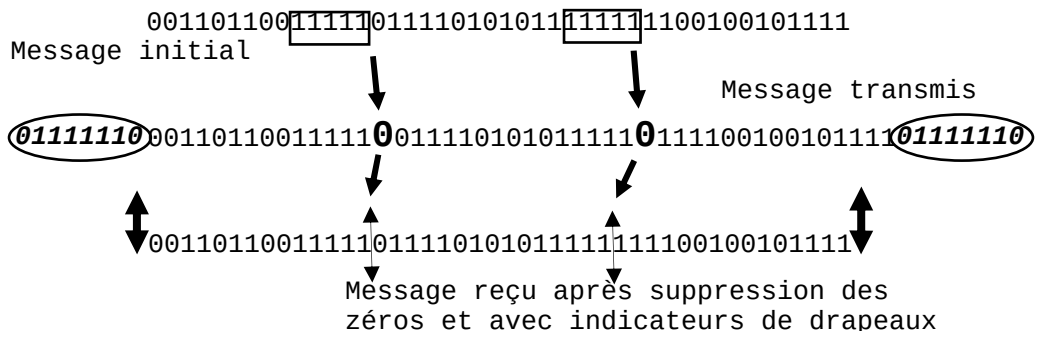


Figure 21 Insertion et destruction de zéros

Pour insérer les zéros, nous allons prendre une approche qui ne demande aucune mémoire interne; cela va se payer par le fait que c'est le bloc « insertion » qui va demander ses données et qui, donc, ne demandera rien quand il insérera un zéro. Autrement dit, les bits entreront d'une façon non régulière, à la demande. L'autre branche de l'alternative serait de mettre une mémoire tampon dans le bloc.

```
library ieee; use ieee.std_logic_1164.all;
entity insertion_zéros is
  port(entrée: in std_logic;
       horloge: in std_logic;
       entrée_suivante: out std_logic;
       sortie: out std_logic);
end entity insertion_zéros;
```

L'architecture contient un processus sensible sur l'horloge: comme souvent, on va utiliser un front pour demander la donnée, l'autre pour la lire.

- Sur le front montant de l'horloge, on envoie un front sur « entrée_suivante » pour demander le bit suivant. Sauf s'il y a eu 5 uns sur l'entrée.
- Sur le front descendant, on recopie l'entrée sur la sortie sauf s'il y a eu 5 uns sur l'entrée. Auquel cas on met zéro sur la sortie.

```
architecture beh of insertion_zéros is
begin
  process(horloge)
    variable cpt: integer:=0;
  begin
    if horloge = '1' then
       if cpt=6
           then cpt:=0;
           else entrée_suivante<='1';
       end if;
    else
      entrée_suivante<='0';
      if entrée = '1' then
           cpt := cpt+1;
      end if;
      if cpt<=5 then
         sortie<=entrée;
      else
         sortie <= '0';
      end if;
    end if;
  end process;
end architecture beh;
```

Pour décoder, faisons l'opération inverse. L'entité prend le signal codé, l'horloge, et produit une sortie et un front indiquant quand la donnée est valide.

```
library ieee; use ieee.std_logic_1164.all;
entity destruction_zéros is
  port(entrée: in std_logic;
       horloge: in std_logic;
       sortie_valide: out std_logic;
       sortie: out std_logic);
end entity destruction_zéros;
```

L'architecture est sensible aux fronts de l'horloge; sur le front descendant, l'entrée est copiée sur la sortie. Sur le front montant, le signal « sortie_valide » est mis à un, sauf s'il y a eu 5 uns au coup d'horloge précédent (d'où l'utilisation d'un booléen « inhibé » pour en garder mémoire.

```
architecture beh of destruction_zéros is
begin
  process(horloge)
    variable cpt: integer:=0;
    variable inhibé:boolean:=false;
  begin
    if horloge = '0' then
       if cpt<=5 then
         if not inhibé then
            sortie_valide<='1';
         end if;
         inhibé:= (cpt=5);
       end if;
    else
      sortie_valide<='0';
      if entrée = '1' then
          cpt := cpt+1;
       else
          cpt:=0;
       end if;
      sortie<=entrée;
    end if;
  end process;
end architecture beh;
```

Pour tester, il suffit d'instancier les deux entités dans une architecture de test:

```
entity testserie is end entity testserie;

library ieee; use ieee.std_logic_1164.all;
architecture arch of testserie is
signal entrée, horloge, entrée_suivante,fil,
       sortie_valide, sortie: std_logic :='0';
begin
horloge <= not horloge after 50 ns;
entrée <=  (le motif de test) ;
C1: entity work.insertion_zéros
    port map (entrée, horloge, entrée_suivante,fil);
C2: entity work.destruction_zéros
    port map (fil, horloge, sortie_valide, sortie);
end architecture arch;
```

En examinant les chronogrammes, il faudra bien tenir compte que la sortie n'est pas « régulière », il faut la considérer seulement quand elle est validée par un front de sortie_valide.

7.3 Machines d'états

Une machine à états finis est un système qui a un **ensemble fini d'états**, un **état initial** dans l'ensemble susdit, des **états dits terminaux** toujours dans cet ensemble, **un ensemble fini d'entrées**, et **une fonction** permettant de passer d'un état à un autre à partir du couple {*état_courant, entrée*}. On dit que l'automate « reconnaît » les séquences d'entrées qui l'amènent de son état initial à l'un de ses états terminaux.

Dans les exemples ci-dessous, et pour ne pas surcharger le code, nous allons prendre comme exemple la machine d'état très simple qui consiste à ouvrir une porte automatique avec deux boutons. Il y a deux états *: porte_ouverte, porte_fermée* qui sont aussi des états terminaux. L'état initial est *porte_fermée*. Il y a deux entrées qu'on va supposer exclusives: *ouvrir_la_porte, fermer_la_porte*. La sortie est un voyant *libre*/*occupé*. On dira qu'essayer d'ouvrir une porte ouverte la laisse ouverte, et inversement.

En VHDL l'abstraction gagne à ce qu'on décrive l'état comme un élément d'un type énuméré. Nous aurons deux signaux de ce type, l'état courant et l'état à venir calculé par la machine :

```
type type_état_porte is (porte_fermée, porte_ouverte) ;
signal état, état_suivant : type_état_porte := porte_fermée;
```

Les deux entrées seront des signaux de type BIT, à 1 si on appuie sur le bouton. Rappelons qu'on les suppose exclusives, c'est-à-dire que, par exemple, quelque dispositif mécanique empêche physiquement qu'on ait à la fois l'une et l'autre.

```
signal ouvrir_la_porte, fermer_la_porte : BIT ;
```

On va aussi décrire la sortie comme un élément d'un type énuméré :

```
type type_sortie is (libre, occupé);
signal sortie : type_sortie :=occupé;
```

Dans un exemple aussi simple, on ne perdrait pas grand-chose en lisibilité en prenant simplement le type BIT. On note aussi qu'ici nous allons supposer que les entrées et le signal de sortie ne sont pas des ports, ce qui n'a aucune influence sur la suite des événements.

7.3.1 Machine de Moore

Pour décrire une machine de Moore, on met les valeurs de sortie (libre/occupé) dans les places, et on obtient le diagramme suivant :

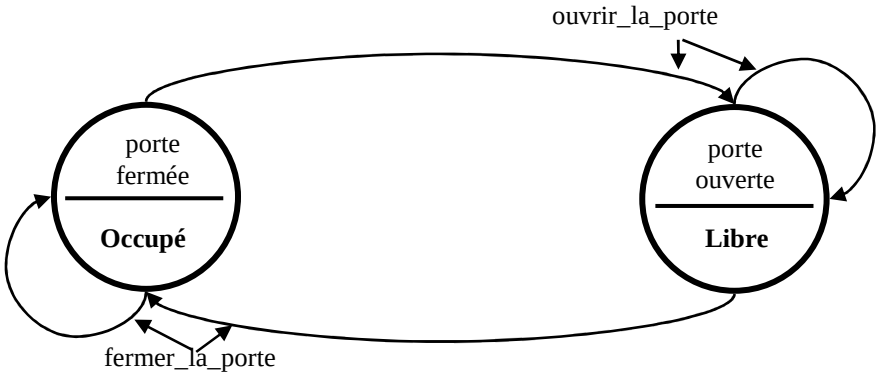
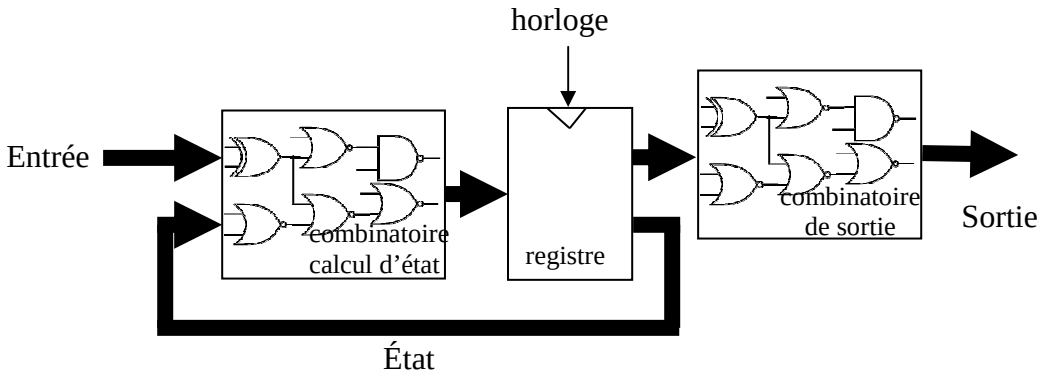


Figure 22 Machine de Moore à deux états

L'implémentation électronique est la suivante : un bloc combinatoire (c'est-à-dire qui peut se ramener à un jeu d'équations logiques) attaque un registre commandé par un front d'horloge.

Une partie de la sortie de ce registre est la sortie du circuit, une autre partie est le codage de l'état de la machine, qui revient sur l'entrée du bloc combinatoire.

Figure 23 Machine de Moore



La combinatoire d'entrée peut être écrite comme un processus ou comme du flot-de-données. Comme le flot de données se ramène toujours à un processus équivalent pour ce qui est de la simulation, nous allons ici écrire ce dernier :

```
combinatoire_états : process (ouvrir_la_porte, fermer_la_porte, état)
begin
  case état is
    when porte_ouverte => if fermer_la_porte = '1'
                             then état_suivant <= porte_fermée ;
                          end if ;
    when porte_fermée =>  if ouvrir_la_porte = '1'
                             then état_suivant <= porte_ouverte ;
                          end if ;
  end case ;
end process ;
```

Ce qui est équivalent à l'écriture flot-de-données:

```
combinatoire_états : état_suivant <=
     porte_fermée when (état = porte_ouverte)  and (fermer_la_porte = '1')
else porte_ouverte when (état = porte_fermée)  and (ouvrir_la_porte = '1')
else unaffected ;
```

Le registre sera décrit également par un processus sensible sur le front montant de l'horloge:

```
registre : process (horloge)
begin
  if horloge='1' then -- on ne peut être ici que s'il y a événement sur
                      -- l'horloge, donc ce test garantit un front montant.
    état <= état_suivant ;
  end if;
end process ;
```

Enfin la combinatoire de sortie sera ici très simple dont voici la version « processus », la version « flot-de-données » étant trivialement déduite :

```
combinatoire_sortie : process (état)
begin
   case état is
     when porte_ouverte => sortie <= libre ;
     when porte_fermée => sortie <= occupé ;
   end case ;
end process ;
```

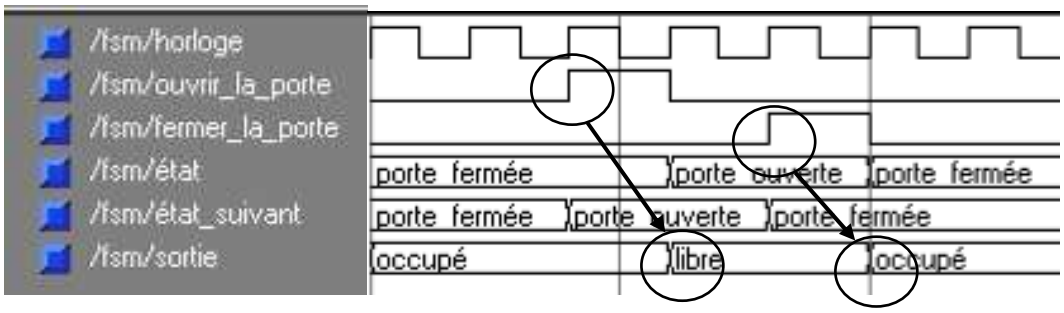


Figure 24 Chronogrammes de la machine de Moore

Au bilan, on voit sur cet exemple très simple que l'état de sortie se met à jour une période d'horloge après l'action causale.

7.3.2 Machine de Medvedev

C'est un cas particulier de la machine de Moore où l'état est aussi la sortie, par exemple un compteur. La combinatoire de sortie est nulle, et le vecteur de sortie est identique au vecteur d'état, ce qui ramène la question à deux processus.

7.3.3 Machine de Mealy

Pour décrire une machine de Mealy, on met les valeurs de sortie sur les transitions. Ainsi le signal est-il « occupé » dès qu'on entreprend de fermer la porte, plutôt qu'au moment où elle est fermée. On gagne ainsi un coup d'horloge pour une décision qui apparaît pertinente.

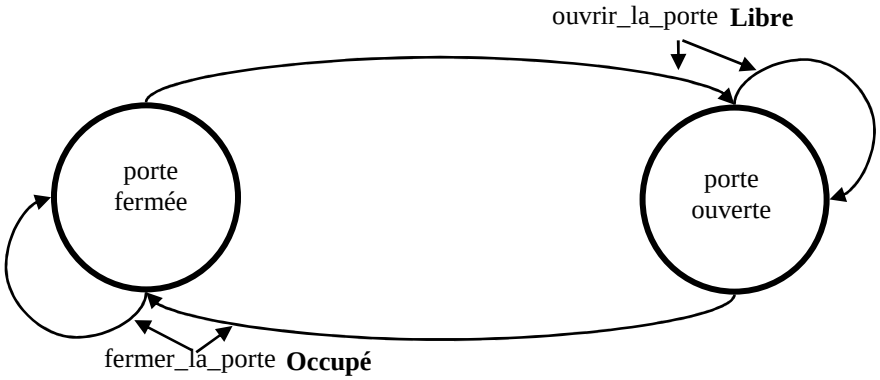


Figure 25 Machine de Mealy à deux états

Pour son implémentation électronique, la machine de Mealy ressemble à la machine de Moore mais la sortie est calculée à partir du registre et des entrées. Cela permet de calculer la sortie avec un coup d'horloge d'avance, au prix d'un chemin combinatoire entre l'entrée et la sortie donc il y a une sensibilité de la sortie à d'éventuels aléas sur les entrées.

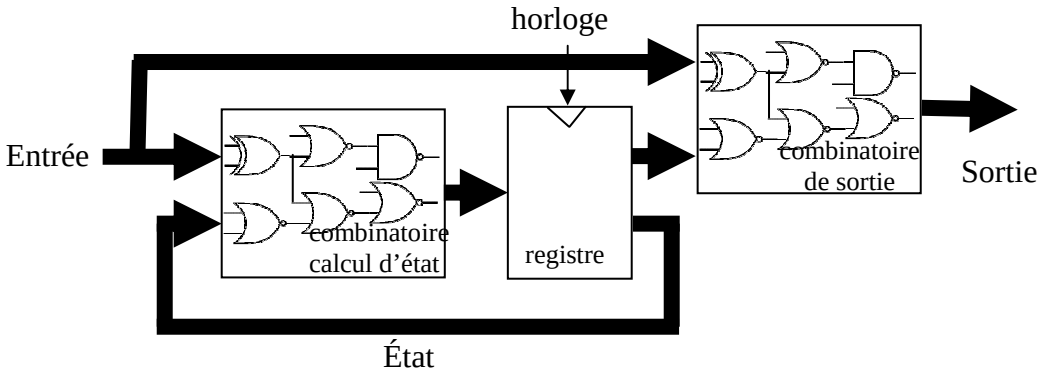


Figure 26 Machine de Mealy

Une cascade de telles machines risque de conduire à un long chemin combinatoire aux délais imprévisibles. Pire, si de telles machines se parlent les unes aux autres avec des boucles possibles, on peut trouver des boucles combinatoires et le circuit, censé être logique, se met à osciller et se transforme en émetteur ondes courtes avant de partir en chaleur et fumées.

Pour implémenter une machine de Mealy, nous n'avons donc qu'à changer le bloc de sortie de notre machine de Moore, les autres restent identiques :

combinatoire_sortie : **process** (état, ouvrir_la_porte, fermer_la_porte)

```
begin
   case état is
     when porte_ouverte => if fermer_la_porte='1'
                             then sortie <= occupé ;
                             else sortie <= libre ;
                           end if;
     when porte_fermée =>  if ouvrir_la_porte='1'
                              then sortie <= libre ;
                              else sortie <= occupé ;
                           end if;
   end case ;
end process ;
```

Cette fois-ci, les sorties changent aussitôt que la condition d'entrée change :

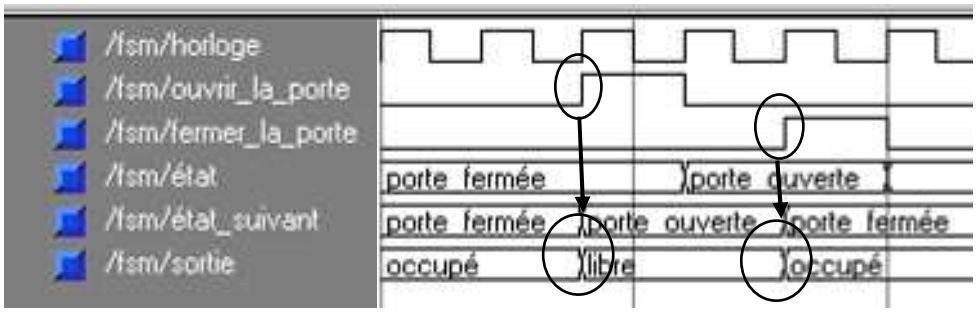


Figure 27 Chronogrammes de la machine de Mealy

En envoyant des aléas sur une entrée (ici *ouvrir_la_porte*) on constate qu'ils se propagent instantanément sur la sortie, chose qui n'arrive pas dans une machine de Moore et qui illustre le souci dû au fait qu'il y a un chemin combinatoire entre l'entrée et la sortie.

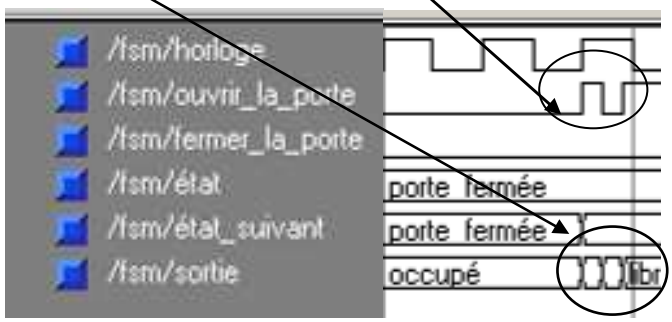


Figure 28 Aléas sur la machine de Mealy

7.3.4 Machine de Mealy synchronisée

Pour éviter les questions de chemin combinatoire non maîtrisé, on peut synchroniser la sortie de la machine de Mealy avec la même horloge que celle qui sert au changement d'état. Les deux blocs combinatoires peuvent alors être réunis.

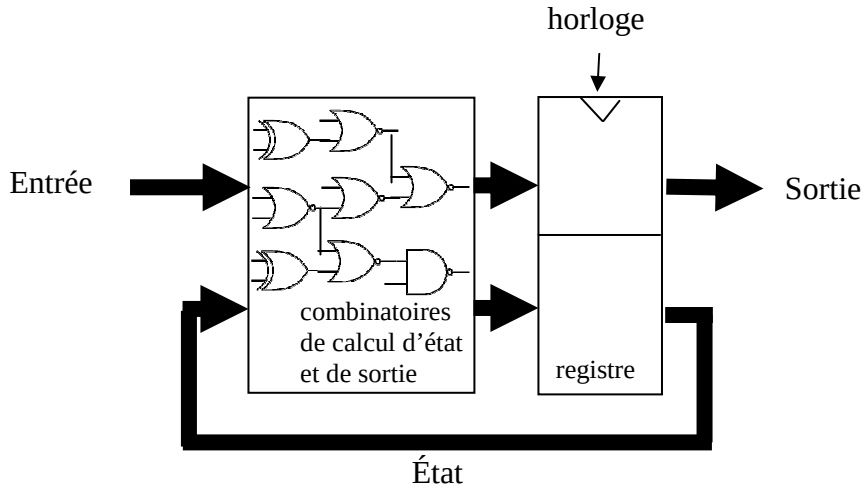


Figure 29 Machine de Mealy Synchronisée

Nous n'avons plus que deux blocs (deux processus donc):

Il nous faut un signal intermédiaire pour la sortie:

```
signal sortie_intermédiaire: type_sortie;
```

Nous aurons

- Un processus décrivant à la fois la combinatoire du changement d'états et de calcul de la sortie
- Un processus sensible sur l'horloge et activant le registre.

```
combinatoire : process (ouvrir_la_porte, fermer_la_porte, état)
begin
  case état is
    when porte_ouverte=>if fermer_la_porte = '1'
                        then état_suivant <= porte_fermée ;
                             sortie_intermédiaire <= occupé;
                         end if ;
    when porte_fermée=> if ouvrir_la_porte = '1'
                         then état_suivant <= porte_ouverte ;
                               sortie_intermédiaire <= libre;
                         end if ;
  end case ;
end process ;

registre : process (horloge)
begin
  if horloge='1' then -- on ne peut être ici que s'il y a événement sur
                      -- l'horloge, donc ce test garantit un front montant.
    état <= état_suivant ;
    sortie <= sortie_intermédiaire;
  end if;
end process ;
```

Dans ce montage, nous obtenons les chronogrammes de la machine de Mealy, mais sans les aléas si une entrée se met à bouger plus vite que l'horloge.

7.3.5 Codage

Les types énumérés employés permettent une bonne abstraction, mais il vient un temps où il faut coder ces valeurs sur un nombre de bits donnés, et tous les codages ne sont pas équivalents. On se souviendra d'une bonne propriété de VHDL, qui permet de faire des tests y compris des **case**, sur des vecteurs de bits. Ainsi, étant donnés le type et le signal:

```
type type_états is (bleu, blanc, rouge);
signal état: type_états;
```

Il est possible de ne changer que très légèrement le code VHDL au moment de donner des codes à nos valeurs:

```
subtype type_état is bit_vector (1 downto 0);
signal état: type_état;
constant bleu: type_état:= "00";
constant blanc: type_état:= "01";
constant rouge: type_état:= "10";
```

Ici nous prenons le choix d'avoir une combinaison par état, ce qui demandera un décodage à la synthèse. Une solution plus rapide mais moins économique est d'avoir un bit par état: on aurait besoin de 3 bits codés "001", "010", "100", avec la nécessité de gérer toutes les combinaisons interdites par un attrape-tout dans la logique. Autrement dit l'introduction d'une

clause « **when others =>** » dans les instructions **cas**e. Il arrive aussi que l'on ait besoin d'utiliser le code de Gray, qui a la bonne idée de ne changer que d'un bit entre deux états successifs : 000, 001, 011, 010, 110, 100, 101, 111, ce qui permet de se prémunir contre l'arrivée d'états baroques –combinaisons interdites- à cause de délais imprévus.

Après cela, la beauté de VHDL fait que les instructions **if** et **case** ne demandent aucune modification, c'est-à-dire que l'essentiel du modèle ne change pas. Peut-être faudra-t-il ajouter une branche **others** dans les instructions **case**, si toutes les combinaisons ne sont pas couvertes. Ce qui sera forcément le cas si l'on utilise le type STD_LOGIC qui n'a pas que des valeurs 0 et 1, lesquelles n'ont pas de sens pour la synthèse mais ont un sens pour le compilateur.

Ainsi l'instruction **if** ou la boucle **while** ne changent pas du tout, et l'instruction **case** ne change pratiquement pas de syntaxe en passant du type énuméré au codage binaire:

```
case état is
 when bleu =>…
 when blanc =>…
 when rouge =>…
 when others => report "erreur!";
              -- nécessaire pour couvrir même les cas qui n'arrivent pas
end case;
```

8 Synchrone

Le monde synchrone est un monde où plusieurs choses se passent au même instant. Cet instant est déterminé en général par une horloge.

8.1 Bascules

8.1.1 Bascule D

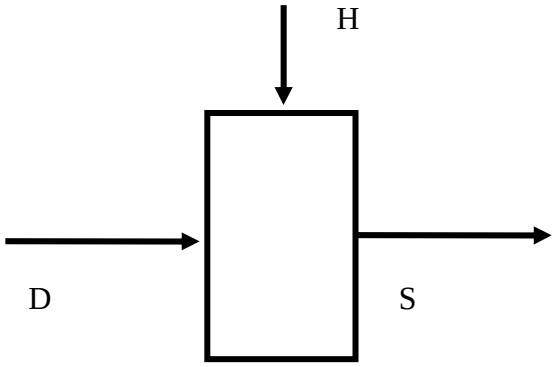


Figure 30 Bascule D

À chaque front montant de H, l'entrée D est recopiée sur la sortie S. Cette valeur est maintenue jusqu'au prochain front ontant de H.

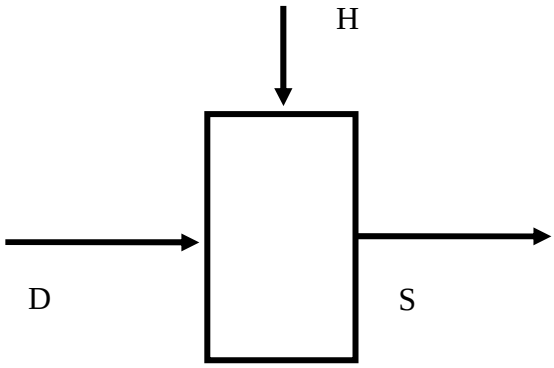
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity basculeD is
  port (H,D: in STD_LOGIC;
        S: out STD_LOGIC);
end entity basculeD;
```

```
architecture dataflow of basculeD is
begin
  S <= D when H'EVENT and H = '1' else unaffected;
end architecture dataflow;
```

Le modèle flot de données a l'inconvénient, pour la simulation, d'être sensible aux événements sur D alors qu'ils seront inefficaces : seuls les transitions sur H ont un effet sur la sortie. Il vaut mieux éviter ce désagrément en utilisant un processus, explicitement sensible uniquement sur H :

```
architecture beh of basculeD is
begin
  process(H)
  begin
    if H = '1' then -- front montant
      S <= D;
    end if;
  end process;
end architecture beh;
```

8.1.2 Latch



La sortie est égale à l'entrée quand H vaut 1, elle reste à sa dernière valeur quand H vaut 0. Cette « transparence » fait qu'on n'est pas sensible sur front, mais sur niveau de H.

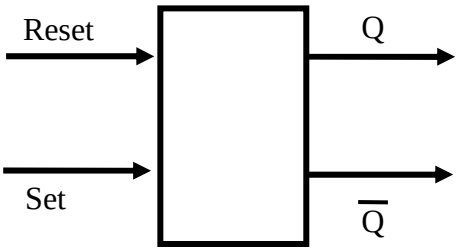
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity latch is
  port (H,D: in STD_LOGIC;
        S: out STD_LOGIC);
end entity latch;
```

Figure 31 Latch

```
architecture dataflow of latch is
begin
  S <= D when H = '1' else unaffected;
end architecture dataflow;
```

Ici nous n'avons aucun intérêt à passer par un processus sensible seulement sur H, puisque le registre est « transparent » et doit rester sensible aux événements sur D.

8.1.3 Bascule RS



La sortie Q prend la valeur 1 quand S vaut 1 et R vaut 0. Elle prend la valeur 0 quand S vaut 0 et R vaut 1. Elle garde son ancienne valeur quand S = R = 0. . QB vaut « **not** Q » sauf si S=R=1 qui est un état interdit.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity basculeRS is
  port (R,S: in STD_LOGIC;
        Q,QB: out STD_LOGIC);
end entity basculeRS;
```

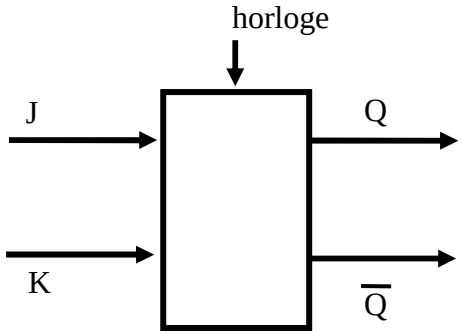
Figure 32 Bascule RS

On peut construire cette fonctionnalité avec des **nor** ou des **nand**.

```
architecture dataflow of basculeRS is
  signal Q1,QB1 : STD_LOGIC := '0';
begin
  QB1 <= S nor Q1;
  Q1 <= R nor QB1;
  Q <= Q1;
  QB <= QB1;
end architecture dataflow;
```

Nous ne présentons pas d'architecture comportementale, en effet elle nécessiterait un processus sensible sur deux signaux et demanderait un test pour savoir lequel a change; le bénéfice serait nul.

8.1.4 Bascule JK



C'est la bascule mère en ce sens qu'on peut construire toutes les autres (sur front) à partir de celle-ci. Elle est définie par l'équation :
Au front d'horloge: $Q_{n+1} = J\overline{Q}_n + \overline{K}Q_n$

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity basculeJK is
  port (J,K,H: in STD_LOGIC;
        Q,QB: out STD_LOGIC);
end entity basculeJK;
```

Figure 33 Bascule JK

```
architecture dataflow of basculeJK is
  signal Q1 : STD_LOGIC := '0';
begin
  Q1 <= (J and (not Q1)) or ((not K) and Q1) when H'EVENT and H='1'
                                             else unaffected;
  Q <= Q1;
  QB <= not Q1;
end architecture dataflow;
```

Comme dans le cas de la bascule D, il y a une économie à faire en utilisant un processus sensible uniquement sur H: en effet le modèle flot-de-données est inutilement sensible aux événements sur J et K.

```
architecture beh of basculeJK is
begin
  process(H)
    variable Q1 : STD_LOGIC := '0';
  begin
    if H = '1' then – front montant
      Q1:= (J and (not Q1)) or ((not K) and Q1);
      Q <= Q1;
      QB <= not Q1;
    end if;
  end process;
end architecture beh;
```

8.2 Combinatoire synchronisé

Considérons l'entité et l'architecture ci-après : il s'agit d'un compteur asynchrone sur 3 bits. L'horloge d'entrée est testée pour son front montant et provoque le changement d'état du bit de poids faible. Chaque fois que ce bit passe à 0, le bit de poids supérieur change d'état (c'est la retenue en base 2), et de même à l'étage au dessus.

```
entity e is
  port (horloge:in bit; sortie: out bit_vector(2 downto 0));
end entity e;
architecture a1 of e is
```

```
 signal tmp:bit_vector(2 downto 0);[12]
begin
  tmp(0)<= not tmp(0) when horloge 'event and horloge ='1' else unaffected;
  tmp(1)<= not tmp(1) when tmp(0)'event and tmp(0)='0' else unaffected;
  tmp(2)<= not tmp(2) when tmp(1)'event and tmp(1)='0' else unaffected;
  sortie<= tmp;
end architecture a1;
```

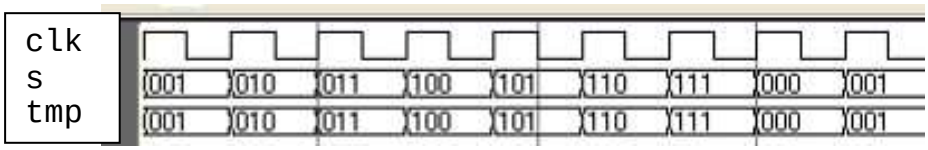


Figure 34 Chronogrammes d'un compteur asynchrone idéal

Le résultat est clairement un comptage sur 3 bits, la valeur de S parcourant le cycle 0,1,2..7, 0,1,2, etc. Le problème est que compteur est asynchrone, défaut que nous mettons facilement en évidence en forçant des délais observables à la simulation dans chaque instruction : ajoutons **after** 3 ns à chaque affectation.

```
architecture a2 of e is
 signal tmp:bit_vector(2 downto 0);
begin
    tmp(0) <= not tmp(0) after 3 ns when horloge 'event and horloge ='1'
              else unaffected ;
    tmp(1) <= not tmp(1) after 3 ns when tmp(0)'event and tmp(0)='0'
              else unaffected;
    tmp(2) <= not tmp(2) after 3 ns when tmp(1)'event and tmp(1)='0'
              else unaffected;
    sortie <= tmp;
end architecture a2;
```

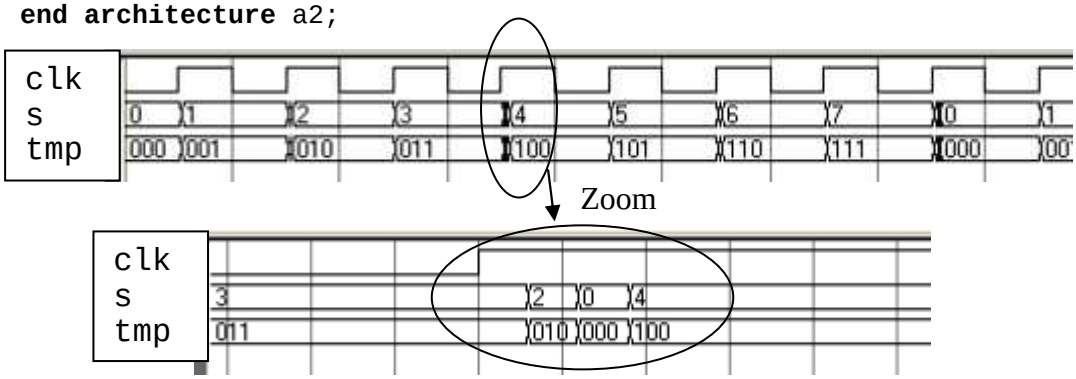


Figure 35 Chronogrammes d'un compteur asynchrone réel

Tous ces états intermédiaires finissent par se stabiliser mais hélas après quantité de “cheveux” qui font passer la sortie par quantité de valeurs incohérentes, ce qu’on voit sur le zoom ci-dessus. Pire, le nombre de cheveux dépend de la donnée qui passe, le temps de stabilisation n’est pas constant.

Synchronisation : la solution classique consiste à décréter que les étapes de calcul, sujettes à transitoires et états instables, se font sur un front d’horloge, et que la lecture du résultat se fait sur l’autre. Ainsi nous obtenons :

[12] Tmp est recopié sur S. Il sert de signal interne qu’on peut lire et écrire, au contraire de S qu’on ne peut pas lire (mode **out**) et qui ne pourrait pas être à droite d’une affectation.

```
architecture a3 of e is
 signal Tmp:bit_vector(2 downto 0);
begin
    tmp(0) <= not tmp(0) after 3 ns when horloge 'event and horloge ='0'
              else unaffected ;
    tmp(1) <= not tmp(1) after 3 ns when tmp(0)'event and tmp(0)='0'
              else unaffected;
    tmp(2) <= not tmp(2) after 3 ns when tmp(1)'event and tmp(1)='0'
              else unaffected;
    sortie <= tmp when horloge 'event and horloge ='1'
              else unaffected;
end architecture a3;
```

Les transitoires apparaissent toujours sur *tmp*. Mais le signal de sortie est lu sur l'autre front de l'horloge, et est, lui, propre. Le circuit compte maintenant les fronts descendants, mais donne la valeur sur front montant.

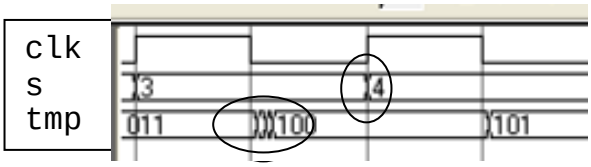


Figure 36 Chronogrammes après synchronisation

Par quelques calculs simples, ceci nous permet aussi de calculer la valeur maximale de l'horloge pour notre circuit : on calcule le cas pire de la stabilisation sur *tmp*, et le temps obtenu doit être inférieur à la demi-période de l'horloge.

8.3 Générateur de séquences pseudo-aléatoires

Il s'agit de produire une série de valeurs binaires présentant les caractéristiques statistiques d'une série aléatoire. Évidemment, le circuit est déterministe et il présente une période ce qui marque la limite de la méthode : c'est le « *pseudo* » de *pseudo-aléatoire*. La technique consiste à cascader des bascules et à calculer l'entrée à partir d'un ou plusieurs *ou-exclusif* prenant les sorties intermédiaires comme entrées.

Pour illustrer avec un exemple simple, nous allons utiliser 16 bits et une seule boucle *ou-exclusif*. Chaque étage sera simplement un bascule D telle que décrite §8.1.1 page 77.

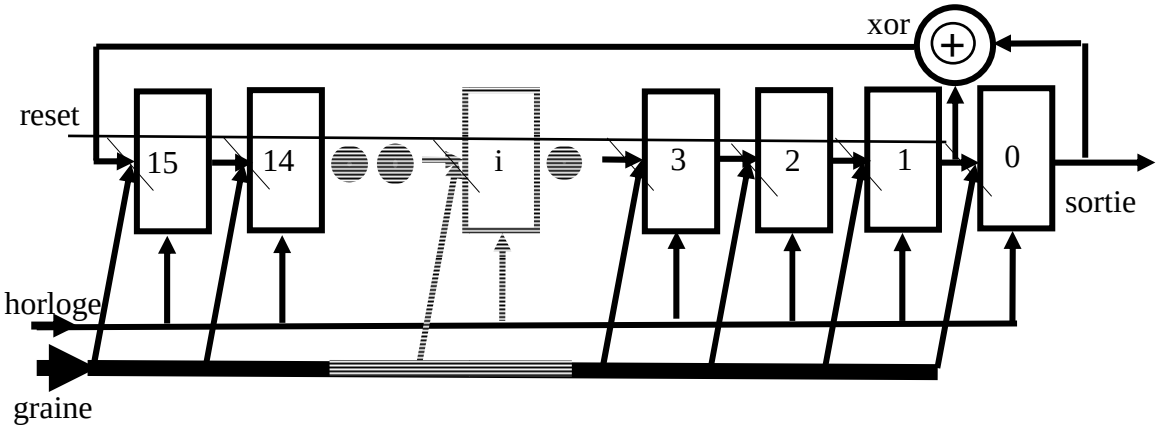


Figure 37 Schéma bloc d'un générateur de séquence pseudo-aléatoire

Pour initialiser le générateur, nous avons besoin d'une graine (la configuration initiale). Cela nécessite d'avoir un port d'entrée de 16 bits et un signal de *reset* que nous mettons sur niveau. Notons que la graine «0000 » (hexa) ne convient pas, elle est absorbante pour cet algorithme.

```
library ieee; use ieee.std_logic_1164.all;
entity alea is
  port (horloge: in STD_LOGIC;
        reset: in STD_LOGIC;
        graine: in STD_LOGIC_VECTOR(15 downto 0);
        sortie: out STD_LOGIC);
end entity alea;
```

Pour l'architecture, nous allons instancier les 16 bascules D par une boucle **generate**. Chacune sera reliée à deux signaux, en entrée et en sortie, se façon à pouvoir, pour l'entrée, choisir entre la graine et la sortie de l'étage précédent.

Attention : Le composant est, pour les besoins de la cause, défini avec un nom différent de l'entité basculeD, il faut donc faire une configuration : cette configuration ne peut pas se faire dans la zone déclarative de l'architecture, elle ne serait pas « vue » par l'instance qui est dans le **generate** ; la solution consiste

- Dans les implémentations très récentes, à utiliser la zone déclarative du **generate**
- Dans les implémentations plus anciennes –la plupart-, à déclarer un bloc *ad hoc*, ici celui qui a le label BB, dans le seul but d'ouvrir une zone déclarative à l'intérieur de la boucle **generate**, qui sera visible de l'instanciation.

Pour illustrer ce problème, dans le code de l'architecture la configuration « inutile » a été mise en commentaire Cette obligation est souvent une cause de soucis pour les concepteurs qui ne comprennent pas pourquoi le composant est « vu » mais pas sa configuration qui est juste à côté. Il y a pourtant une excellente raison à cet état de fait, hors du sujet de ce manuel.

```
architecture arch of alea is
  component bD is
    port (H,D: in STD_LOGIC;
          S: out STD_LOGIC);
  end component bD;

      -- for all : bD use entity work.basculeD;
        -- ceci n'est pas visible à l'intérieur de la boucle
        -- generate, donc serait inutile puisqu'il n'y a aucune
      -- instance dehors (voir commentaire)

signal intermédiaire_entrée, intermédiaire_sortie:
                            std_logic_vector(15 downto 0);
begin

    GG: for I in 0 to 15 generate
      BB: block -- nécessaire[13] pour caser la
                -- spécification de configuration
          for all : bD use entity work.basculeD;
             -- pas de port map, les noms et positions
             -- sont les mêmes.
          begin
          étage: bD port map(horloge,
                             intermédiaire_entrée(i),
                             intermédiaire_sortie(i));
      end block BB ;

    end generate;

    intermédiaire_entrée(15) <=
           graine(15) when reset='1'
           else intermédiaire_sortie(0) xor intermédiaire_sortie(1);

    intermédiaire_entrée(14 downto 0) <=
                 graine(14 downto 0) when reset='1'
                 else intermédiaire_sortie (15 downto 1) ;

    sortie <= intermédiaire_sortie(0);

end architecture arch;
```

Ne marche pas

oui

8.4 Blocs gardés

Le bloc est une instruction concurrente, qui contient d'autres instructions concurrentes dont, éventuellement, des blocs. Il possède une zone de déclaration de généricité et de ports dont nous ne parlerons pas ici. Le bloc est une instruction qui fait partie des capacités de VHDL en matière de hiérarchie et de structure. Il est néanmoins très peu utilisé pour cette fonctionnalité, à la seule petite exception de l'intérieur des blocs **generate** quand on a besoin d'une zone déclarative pour caser une spécification de configuration. Son autre capacité est de permettre la mise en facteur d'une condition sur un signal booléen.

[13] Dans les implémentations récentes, on a le droit d'ouvrir une zone déclarative dans une instruction **generate**, le code serait exactement le même mais en enlevant les deux lignes «`BB: block` » et « `end block BB;` »

```
architecture a of e is
…
begin

-- instructions concurrentes

      label : block (condition)
      begin
      -- instructions concurrentes
      end block label ;

-- instructions concurrentes


end architecture a;
```

La magie de cette condition c'est qu'elle devient condition de toutes les instructions d'affectation de signaux qui comptent le mot-clé **guarded**.

Exemple de synchronisation :

```
      label : block (not clk'stable and clk='1') -- front montant
      begin
        SOMME <= guarded ARG1 + ARG2;
        MOYENNE <= guarded SOMME /2;
      end block label ;
```

Dans cet exemple, rien ne se passe *que* sur front montant de l'horloge (utilisation de S'STABLE qui rend un signal, S'EVENT ferait un système marchant sur niveaux). Si nous proposons deux valeurs à ARG1 et ARG2, au prochain front montant SOMME est calculé, et MOYENNE sera calculé au front montant suivant. Ceci permet de cascader proprement les résultats des opérations en laissant aux circuits le temps de se stabiliser. Voir ci-dessous les timings de la situation où l'on propose sur ARG1 et ARG2 d'abord 10 et 16, puis 10 et 6. On voit bien que SOMME y est disponible au front montant d'horloge suivant, et MOYENNE un coup d'horloge après.

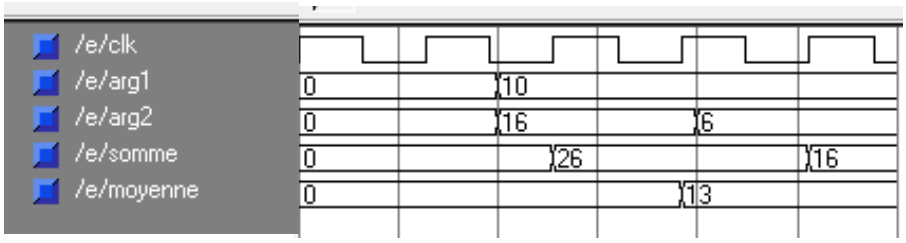


Figure 38 Chronogrammes d'un bloc gardé avec deux affectations gardées

Le bloc gardé a d'autres usages dans le langage, si on le marie avec les signaux gardés (*guarded signal*). Leur intérêt est toutefois mince et leur usage anecdotique.

9 Asynchrone

Dans le monde asynchrone, les événements arrivent à des temps différents. S'il se trouve que tel événement arrive en même temps que tel autre, c'est au hasard de la course des événements et cela n'a pas d'incidence sur le fonctionnement. Du moins, cela ne doit pas en avoir. À la limite, en utilisant un « zoom » suffisamment grossissant, on doit toujours pouvoir distinguer un ordre d'arrivée entre deux événements qui peut, ou peut ne pas, être pertinent. Ceci est vrai sur le matériel réel, évidemment pas sur un simulateur logique.

9.1 Asynchrone inondant, le flot-de-données (data-flow)

Il s'agit d'instruction dont l'exécution est conditionnée par les valeurs des données, et non par leur ordre d'écriture. La synthèse donnera probablement une « mer de portes » (*sea of gates*). Ce sont des instructions essentiellement asynchrones puisqu'il n'y a pas de signal « magique » déclenchant l'activité, comme la condition de garde des blocs gardés (§8.4 page 83.) Néanmoins, la logique synchrone étant faite de portes asynchrones, il est évidemment possible d'écrire du *flot-de-données* synchronisé explicitement, voir par exemple ci-dessous §9.2.
Rappelons rapidement les quatre instructions concourant principalement, c'est le mot, à la conception *flot-de-données* : l'affectation simple, l'affectation conditionnelle (qui comprend la première quand il n'y a pas de condition), l'affectation sélectée et l'appel concurrent de procédure. Non mentionné ici car participant de tous les styles de modélisation, le processus qui est de toutes façon la transformation ultime de toute instruction de VHDL digital.

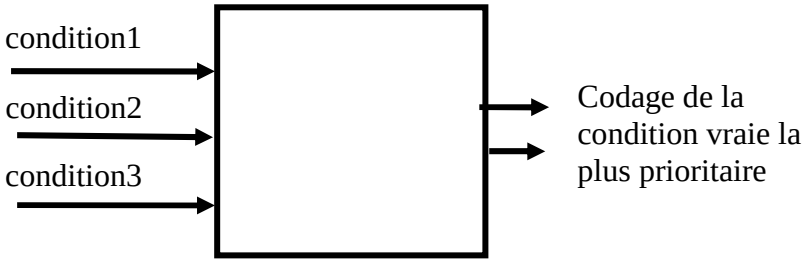
9.1.1 L'affectation simple : un fil ou un registre

Il s'agit d'affecter un signal avec une expression qui elle-même contient généralement des signaux.
Exemple :

```
s <= a or b;
```

9.1.2 L'affectation conditionnelle: un encodeur de priorité

Figure 39 Encodeur de priorité



Il s'agit d'envoyer à un signal une valeur choisie parmi plusieurs selon un jeu de conditions évaluées dans un ordre de priorité. Cette instruction, dans son utilisation asynchrone, est bien illustrée ici par un circuit très utilisé dans les circuits qui gèrent les interruptions, et leur priorité. Il s'agit de proposer à un circuit quelques (ici 3) conditions et de propager sur la sortie (ici sur 2 bits) le numéro de la condition la plus prioritaire. Nous

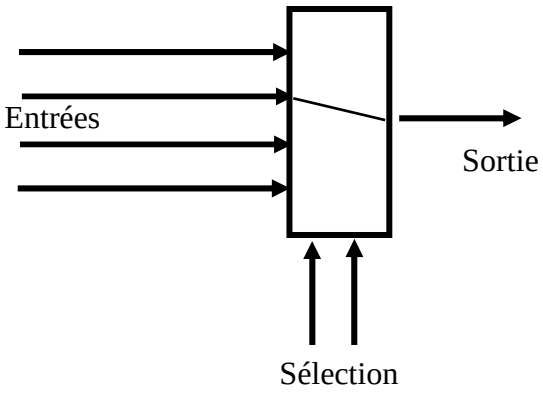
allons utiliser l'instruction d'affectation conditionnelle, puisque sa première propriété est effectivement de tester ses conditions dans l'ordre de leur écriture, et de ne retenir que la première qui convient.

```
sortie <= "00" when condition1
     else "01" when condition2
     else "10" ;
```

Très probablement, dans cet exemple, les conditions seront toutes de la forme (entréeN = '1').

9.1.3 L'affectation sélectée: un multiplexeur

Figure 40 Multiplexeur



Il s'agit d'envoyer à un signal une valeur choisie parmi plusieurs selon la valeur d'un sélecteur qu'on compare à un jeu de constantes. Il n'y a pas de priorité, toutes les constantes sont différentes et doivent couvrir toutes les valeurs possibles du sélecteur. Ceci, dans sa forme asynchrone, est bien illustré par le multiplexeur :
Supposons N canaux d'entrée (ici 4), un canal de sortie, et on veut choisir quel canal d'entrée sera celui qui va sur le canal de sortie, voir schéma ci-contre.
Ceci s'exprime en VHDL de la façon suivante, en utilisant l'instruction d'affectation de signal sélectée.
Dans une telle affectation, il faut fournir une valeur et une seule pour chaque combinaison de la sélection. On a le droit d'utiliser la clause « **when others** » qui sert d'attrape-tout. Si le sélecteur était un type scalaire, on aurait le droit d'utiliser des étendues : 1 **to** 10, ou des choix avec la barre verticale : 3 | 7.

```
with Sélection select sortie <= entrée0 when "00",
                                 entrée1 when "01",
                                 entrée2 when "10",
                                 entrée3 when "11";
```

Dans ce cas particulier, on pourrait même écrire plus simplement si les entrées sont organisées en tableau:

```
sortie <= Entrées(TO_INTEGER(Sélection));
```

Où TO_INTEGER est une fonction qui convertit le tableau de bits de la sélection en entier, de telles fonctions avec toutes les variantes possibles existent en abondance dans les paquetages IEEE.

9.1.4 L'appel concurrent de procédure : un processus avec des arguments

Il est possible d'appeler une procédure en milieu concurrent. La sémantique associée est alors que son appel est déclenché par tout événement sur un signal de sa liste de sensibilité, laquelle est simplement la liste de tous les signaux apparaissant dans ses arguments.

```
LBL : Proc (S1,S2,S3) ;
```

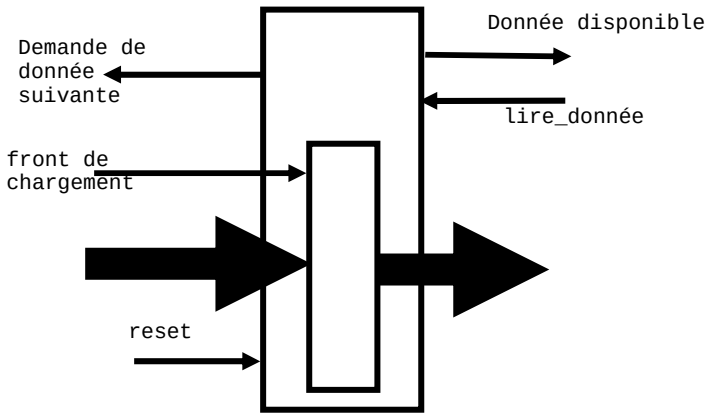
Cela permet d'écrire du code comportemental et séquentiel réutilisable, auquel on peut passer des arguments de tous modes.

9.2 Asynchrone protocolaire

Un protocole (de communication par exemple) est l'exemple type de situation asynchrone séquentielle : quand bien même parfois une horloge est transmise, elle fait partie du problème plus que de la solution, à cause des délais de transmission qui rendent son exploitation compliquée. Il s'agit le plus souvent de machines ayant des « états » identifiables, mais pas d'horloge commune. Les machines à états synchrones sont vues §7.3 page 70.

9.2.1 Handshake

Figure 41 Handshake



Un registre chargé ou lu de façon synchrone par un système à horloge, est déchargé ou écrit par un système asynchrone, par exemple dépendant d'une action d'un opérateur humain. Pour cela chaque côté du dispositif gère deux signaux. Côté synchrone, il y a un signal pour charger le registre (disons front montant), et un signal permettant de faire savoir que ce registre a été lu (niveau). Nous avons aussi probablement besoin d'un signal de remise à zéro, un reset (niveau). Côté asynchrone, il y a un signal disant qu'une nouvelle donnée est disponible (niveau), et un signal entrant signifiant qu'on veut la lire (front montant)

```
library ieee ; use ieee.std_logic_1164.all ;
entity hds is
 port(port_synch: in std_logic_vector(7 downto 0);
      port_asynch: out std_logic_vector(7 downto 0);
      chargement,
      lire_donnée,
      reset : in std_logic ;
      demande_donnée_suivante,
      donnée_disponible : out std_logic) ;
end entity hds ;
```

L'architecture sera un processus sensible sur tous les signaux de mode *in* ; ce processus sera un *if/elsif* à trois branches :

- le reset sur niveau, qui est prioritaire,

- la lecture du registre interne depuis l'extérieur asynchrone par un front montant sur « lire_donnée », le front descendant indiquant que l'affaire est faite.
- l'écriture du registre interne par l'intérieur synchrone, sur un front montant de « chargement ».

Les signaux de sortie indiquant que le bloc est libre ou occupé d'un côté ou de l'autre « donnée_disponible » et « demande_donnée_suivante » sont gérés dans chaque branche.

Un bloc symétrique se chargeant de lire depuis le monde asynchrone et d'écrire vers le monde synchrone serait extrêmement semblable.

```
architecture BEH of hds is
  signal registre: std_logic_vector(7 downto 0);
begin
process(chargement,lire_donnée, reset)
begin
  if reset='1' then
    donnée_disponible<='0';
    demande_donnée_suivante<='0';
  elsif lire_donnée 'event then
    if lire_donnée ='1' then
       donnée_disponible <='0';
       port_asynch<=registre ;
     else
       demande_donnée_suivante <='1';
     end if;
  elsif chargement'event and (chargement ='1') then
    registre <= port_synch;
    donnée_disponible <='1';
    demande_donnée_suivante <='0';
  end if;
end process;
end architecture BEH ;
```

9.2.2 Reconstitution d'un signal codé NRZ et de son horloge

Supposons un signal série créé sur les fronts montants d'une horloge : le codage NRZ consiste à pratiquer un *ou exclusif* entre le signal synchronisé sur l'horloge, et l'horloge elle-même. Le signal synchronisé n'existe pas sur le circuit, il est dessiné ici juste pour éclaircir le procédé ; on produit directement le signal de sortie par un *ou exclusif*.

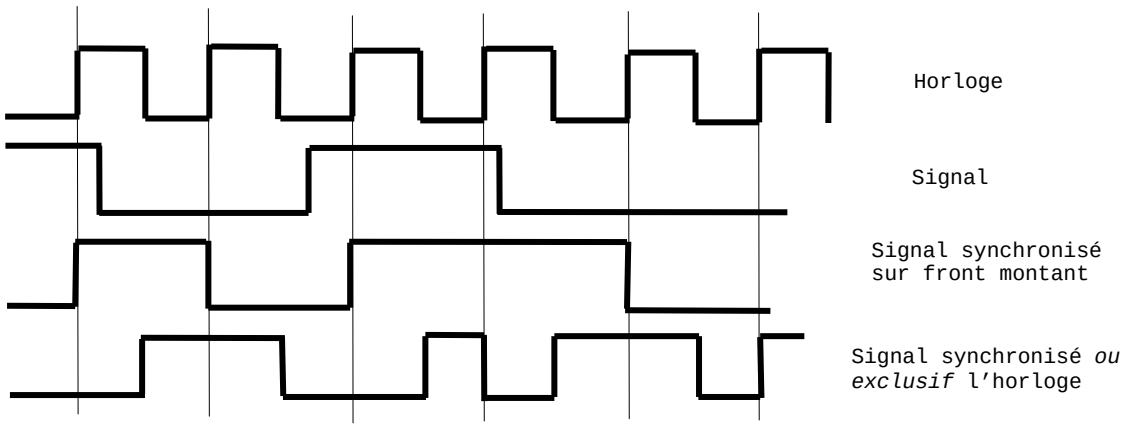


Figure 42 Codage NRZ

C'est ce dernier signal qui sera transmis sur la ligne : il a la bonne propriété d'être forcément à moyenne ½ sur une courte période (la période d'horloge), c'est-à-dire qu'il supporte la perte de la composante continue par le passage par des éléments capacitifs ou passage par

transformateurs. Pour le reconstituer, il suffit de refaire un *ou exclusif* avec la même horloge, l'opération est symétrique.

```
library ieee ; use ieee.std_logic_1164.all ;
entity codeur is
  port (entrée, horloge : in std_logic ; sortie : out std_logic ) ;
end entity codeur ;
```

L'architecture aura deux processus : l'un assurant la synchronisation de l'entrée avec l'horloge, l'autre assurant le codage *ou exclusif*. Ce dernier pourrait être remplacé par la simple ligne :

```
sortie <= entrée_synchronisée xor horloge;
```

Mais le processus équivalent serait alors sensible sur l'horloge et sur l'entrée synchronisée qui dépend aussi de l'horloge, avec pour conséquences des *glitchs* de un delta sur le signal transmis. En rendant le processus de codage sensible uniquement sur les transactions de l'entrée synchronisée, on évite ce petit désagrément mais on court le risque d'un refus du synthétiseur. Il est facile d'explorer d'autres solutions en fonctions des contraintes du synthétiseur utilisé, par exemple la sensibilité sur l'horloge seulement.

```
architecture beh of codeur is
  signal entrée_synchronisée : std_logic;
begin
  process (horloge) – un process sensible sur horloge et pas sur entrée
  begin
    entrée_synchronisée <= entrée;
  end process;
  process
  begin
    sortie <= entrée_synchronisée xor horloge;
    wait on entrée_synchronisée'transaction;
  end process;
end architecture beh;
```

Le signal transmis a aussi la bonne propriété de porter sa propre horloge, puisqu'il suffit d'observer le signal un certain temps et de déterminer la transition la plus courte : c'est forcément la demi-période. De toute façon, les horloges étant déclenchées par quartz et leur fréquence faisant partie du protocole, la question est plus celle de la synchronisation que celle du calcul de la période.
Reste à déterminer la phase. Pour cela, il est facile de remarquer qu'il y a toujours une transition sur le signal transmis quand il y a un front descendant de l'horloge, et qu'inversement ce n'est pas toujours le cas sur le front montant de l'horloge. Si l'on suppose acquise la fréquence, il y a deux horloges candidates, déphasées d'une demi-période. Une seule est telle que tous ses fronts descendants correspondent à une transition du signal transmis. C'est celle-là qu'il faut reconnaître et utiliser pour recombiner par *ou exclusif* avec le signal.

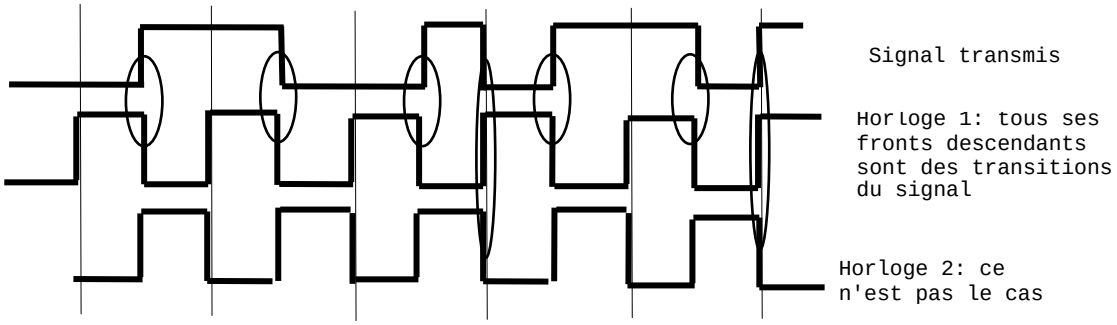


Figure 43 Reconstitution de l'horloge

Pour reconstituer l'horloge, nous allons avoir besoin d'une horloge de fréquence supérieure, le double étant un minimum (plus le multiple est grand, moins le risque de dérapage en cas d'aléa est grand).

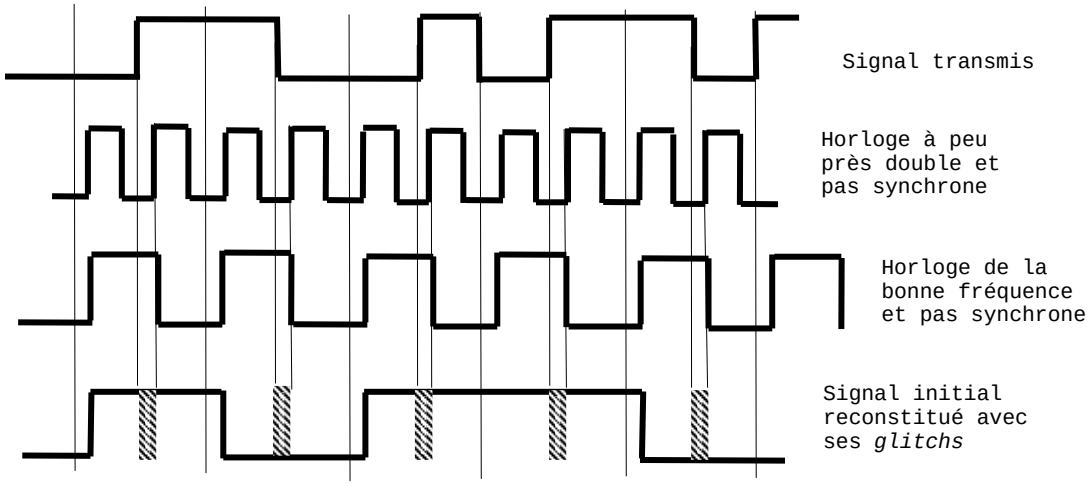


Figure 44 Décodage NRZ

Dans le procédé consistant à reconstituer une horloge à partir d'une horloge existante non synchronisée, nous avons un déphasage qui va créer des aléas. Pour minimiser ce déphasage, on peut augmenter la fréquence de l'horloge interne, mais cela ne fera que le minimiser et ne résoudra pas notre problème. On peut aussi mettre un asservissement à boucle de phase sur l'horloge, ce qui nous ramène à de la conception analogique. On peut enfin remarquer que ces aléas ne se produisent jamais quand le signal doit changer de valeur, et on pourrait compter sur une propriété des affectations de signaux : la clause **reject** ou, ici, son invocation par défaut avec la clause **after** qui a le bon goût de fonctionner sur les simulateurs n'implémentant pas VHDL'93. Elle permet de dire au simulateur d'ignorer les mouvements plus rapides qu'un temps qu'on spécifie et gommerait facilement ces *glitchs* au prix de l'inscription d'un délai dans le modèle, ce que n'aiment pas les outils de synthèse.
Dans le cas qui nous occupe, nous allons plutôt synchroniser l'entrée du décodeur (le signal transmis, donc) avec l'horloge reconstituée, avant de faire le *ou exclusif*.

```
library ieee ; use ieee.std_logic_1164.all ;
entity décodeur is
  port (entrée, double_horloge : in std_logic ;
        sortie, horloge : out std_logic ) ;
end entity décodeur ;
```

L'architecture comporte trois instructions concurrentes: le calcul de l'horloge locale, le calcul de la sortie avec délai et réjection, et la propagation de l'horloge locale vers le port horloge.

On ne peut pas utiliser ce dernier directement dans le modèle, car on a besoin de lire l'horloge et celle-ci est de mode **out**.

```
architecture beh of décodeur is
  signal horloge_locale, entrée_synchronisée: std_logic;
begin

  reconstitution_horloge: process
  begin
    wait on entrée;
    wait until double_horloge  ='1';
    horloge_locale <= '1';
    entrée_synchronisée <= entrée;
    wait until double_horloge  ='1';
    entrée_synchronisée <= entrée;
    horloge_locale <= '0';
  end process reconstitution_horloge;

  sortie <= entrée_synchronisée xor horloge_locale;
  horloge <= horloge_locale;

end architecture beh;
```

Le cœur du modèle est dans le processus de reconstitution d'horloge. Il va « pédaler dans la semoule » tant qu'il ne sera pas dans une situation où chaque front descendant de l'horloge locale correspond à une transition du signal entrant. Quand il tombe sur cette situation qui doit forcément arriver parce qu'on envoie en tête des motifs qui garantissent que ça va arriver (le drapeau HDLC suffit), il se verrouille dessus et l'horloge est correcte.

On testera facilement tout cela en assemblant dans une même architecture le codeur, le décodeur, les constructions d'horloge et une forme d'onde de test sur l'entrée. On pourra observer que, dans des limites décentes, le système est résistant à un déphasage de l'horloge double et aussi à une imprécision sur sa fréquence. Mais attention : comme dit ci-dessus, on verra aussi que, pour se caler sur la bonne phase, le système a besoin d'un certain temps, le temps de voir apparaître la configuration qui permet de différencier les deux horloges candidates. C'est pourquoi les systèmes qui utilisent ce codage envoient des entêtes porteurs de configurations qui assurent, à coup sûr, la synchronisation.

10 Switch

Le *switch* ou « interrupteur » est simplement un transistor MOS. Toute la logique classique est évidemment basée sur de tels interrupteurs, mais pour celle-ci les tensions et courants de sortie ne viennent pas des entrées, dont l'impédance est grande ; et enfin les deux réseaux de transistors sont duaux, on dit que ce sont des portes complémentaires. Voici pour mémoire une porte *nor* : on y voit que le courant de sortie n'est pas fourni par les entrées. Les impédances d'entrées sont grandes, celle de sortie est faible.
Nous allons voir ici qu'il est possible de construire de la logique non complémentaire, où il n'y a pas de réseaux de portes duaux. Cela se payera par le fait que, dans certains états, les sorties seront connectées directement sur une ou l'autre entrée avec contamination des impédances.

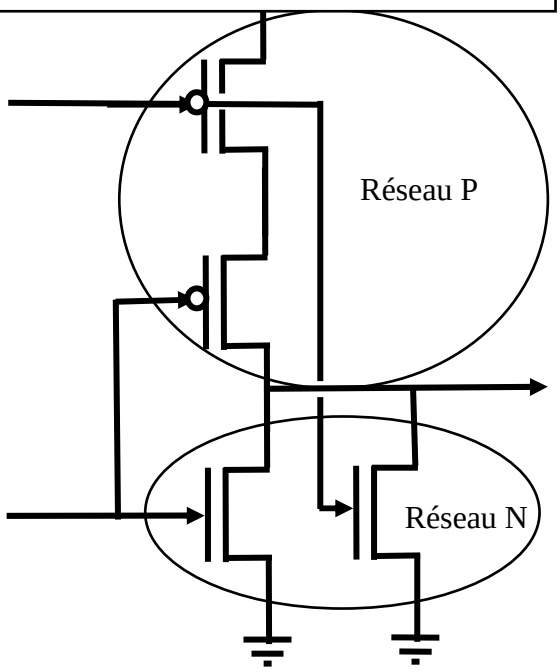


Figure 45 Une porte NAND en MOS

10.1 Un modèle de switch

Pour décrire notre *switch*, nous allons commencer par définir un paquetage qui n'est pas indispensable mais permet d'avoir un joli type énuméré pour les types N et P des MOS, et qui permet aussi de déclarer une fois pour toutes le composant susceptible de recevoir l'entité *switch*.

```
library ieee;use ieee.std_logic_1164.all;
package switch_util is

type genre_mos is (P, N);

component switch
  generic (genre: genre_mos);
  port (source, drain: inout std_logic; grille: std_logic);
end component;

end package switch_util;
```

Voici l'entité qui est générique sur le type de MOS et qui a deux ports de mode **inout**, plus la commande de mode **in**.

```
library ieee;use ieee.std_logic_1164.all;
use work.switch_util.all;
entity switch is
  generic (genre: genre_mos);
  port (source, drain: inout std_logic; grille: std_logic);
end entity switch;
```

L'architecture, qui contient un processus. Le « truc » du modèle consiste à débrancher fugitivement le *switch* (en mettant Z sur le drain et la source) pour observer si l'extérieur force une valeur ou non. Après quoi on le rebranche si la commande est à la bonne valeur.

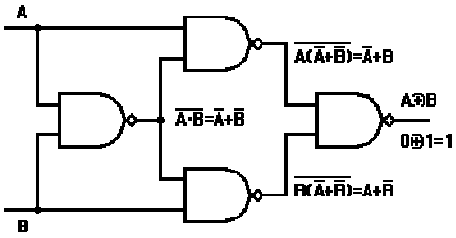
```
architecture arch of switch is
begin
  process

     impure function passant return boolean is
     -- Une fonction pour factoriser la condition "transistor passant"
     -- On pourrait optimiser tout ça en dupliquant l'architecture
     -- et en faisant un Mos P et un Mos N, et deux entités non génériques.
      begin
         return (grille ='1' and genre=N ) or (grille ='0' and genre=P);
      end;

    variable lasttime:time:=-1 ns;

  begin
      source <='Z'; -- on libère source et drain "de l'intérieur"
      drain <='Z';  -- pour connaître les contributions extérieures

      wait for 0 ns;
         -- wait pour que source et drain contiennent
         -- les valeurs "extérieures"

      if passant then
          -- ici le switch est passant, il faut propager

          source <= drain; -- deux affectations croisées qui prendront
          drain <= source; -- effet simultanément sur le wait suivant
                           -- (et pas avant, ce sont des signaux)
      end if;

      lasttime:=now;
      wait on source 'transaction, drain 'transaction, grille
                  until now/=lasttime or (passant and (source /= drain));

  end process;
end;
```

La difficulté de ce modèle réside dans sa condition de réveil: il faut évidemment être sensible sur la grille, mais aussi sur les simples transactions sur la source et le drain : en effet il est possible que, le *switch* étant passant et forçant 1 « de l'intérieur » sur la source par exemple, voit cette même source ensuite forcée « de l'extérieur » avec la même valeur 1 : cela ne créera pas d'événement, juste une transaction mais il faut la prendre en compte car ensuite, si le drain venait à être lâché « de l'extérieur » il faudrait aussitôt le forcer « de l'intérieur ».

Or la sensibilité sur transaction apporte un risque de boucle infinie à délai nul dans un modèle VHDL : il faut limiter le réveil à une transaction par temps simulé (branche **until**) à moins que source et drain ne viennent à être différents alors que le transistor est passant. C'est comme cela que se lit la condition **until** de l'instruction **wait**.

À la simulation, certains simulateurs vont être perturbés par toutes ces valeurs différentes apparaissant au même temps de simulation mais à des deltas différents, et cela peut se traduire par des couleurs…créatives sur les transitions. Il suffit de les ignorer.

10.2 Un « ou exclusif » en switch



Voyons l'exemple typique de la porte *ou exclusif*. Sa synthèse en portes élémentaires de la logique classique doit implémenter d'une façon ou d'une autre l'équation `S = (A and not B) or (B and not A)`
Ceci suppose deux transistors par inverseur et quatre par porte, ce qui nous amène facilement à dépasser la dizaine de transistors, seize dans le cas ci-contre qui fait l'effort d'utiliser un seul type de porte, puisqu'une porte *nand* en demande quatre. La même porte implémentée en logique à interrupteurs va utiliser les entrées comme générateurs de courant, et donc d'information, pour la sortie. Voici ci-contre une implémentation en transistors, et ci-dessous le modèle VHDL correspondant. On remarque qu'il s'agit d'une modification de la porte *nand* dans laquelle les sources de courant sont aussi les deux entrées. Si a=b=1, c'est un *nand* alimenté par a et b, dont les deux entrées sont aussi a et b et à 1, donc la sortie vaut 0. Si a=b=0, tout le circuit est coagulé à 0, et la sortie donc aussi. Si a ou b vaut 1 et l'autre 0, un des deux transistors P tire la sortie à 1 et un des deux transistors N coupe le chemin vers 0, la sortie vaut 1. C'est bien la fonction « ou exclusif ».

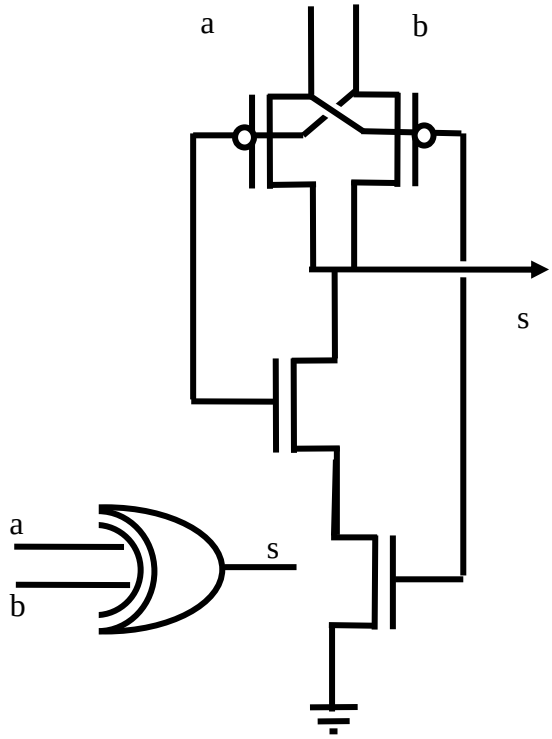


Figure 46 Une porte XOR en MOS

```
library ieee;use ieee.std_logic_1164.all;
entity switch_xor is
  port (A, B: in std_logic; S: out std_logic);
end entity switch_xor;

use work.switch_util.all;
architecture struct of switch_xor is
  signal interne: std_logic:='Z';
  signal copie_de_a, copie_de_b,source_de_s: std_logic;
         -- signaux locaux qui seront passes en inout
         -- alors que A,B,S sont en in et en out.
begin
 copie_de_a<=A ; -- A est en inout,
 copie_de_b<=B ; -- idem
 S<=source_de_s; -- S est en out
 C1: switch generic map (N) port map ('0', interne, copie_de_a);
 C2: switch generic map (N) port map (interne, source_de_s, copie_de_b);
 C3: switch generic map (P) port map (source_de_s, copie_de_a, copie_de_b);
 C4: switch generic map (P) port map (source_de_s, copie_de_b, copie_de_a);
end architecture struct;
```

11 Analogique

Le niveau de description analogique est utilisable aux deux bouts de la chaîne de développement : soit au niveau système (description de gabarits de filtres), soit au niveau électrique, ce qui est le plus fréquent.
Une description analogique demande en général l'utilisation de paquetages spécifiques, définissant le domaine physique dans lequel on se place lequel emporte ses propres lois de conservation. Dans la vaste majorité des cas qui nous intéressent, le paquetage appelé sera `disciplines.electrical_systems.` Attention : certaines implémentations confondent la bibliothèque DISCIPLINES avec la bibliothèque IEEE, certaines ont deux bibliothèques distinctes.

11.1 Objets, instructions, contraintes, domaines

11.1.1 Concepts et Objets : nature, quantité, terminal

Tous les objets analogiques valués (porteurs de valeurs qui changent au cours de la simulation) dérivent de la notion de **nature** qui est au monde analogique ce que le type est au monde informatique. La **nature** est attachée à une « discipline », c'est-à-dire à la partie du monde physique qui l'on veut représenter, et qui correspond en général à un paquetage spécifique contenant aussi les constantes et types utiles au domaine. Cette discipline peut être électrique, mécanique, thermique, etc. Le concepteur électronique utilisera principalement le paquetage *electrical_systems* (§12.4.2 page 152). On déclare des terminaux (**terminal**) comme appartenant à une **nature**. On déclare aussi des quantités (**quantity**) qui ont le statut de variables dans le jeu d'équations à résoudre et qui peuvent subir des équations différentielles implicites selon la forme de leur déclaration. Quantités et terminaux peuvent être membres de structures, on peut faire des tableaux ou des structures incluant des natures.

- La **nature** représente sous un nom unique la vue de la discipline qui obéit à des lois de conservation comme celles de Kirchhoff en électricité. Elle déclare un terminal référence (voir ci-dessous la description des quantités et références). Elle définit aussi que les terminaux de cette nature auront deux quantités implicites : une des quantités (ACROSS) est de la catégorie *potentiel*, l'autre (THROUGH) est de la catégorie *flux*. Les lois de conservation disent que le potentiel entre deux terminaux est unique, et que la somme des flux est nulle. La nature définit aussi une *référence* qui sera la masse en électricité, point zéro des mesures de potentiel. La nature électrique est ainsi définie :

```
nature electrical is
      voltage across
      current through
      electrical_ref reference ;
```

- **La quantité** est une variable du système au sens mathématique du terme, le solveur cherche à modifier sa valeur de façon que toutes les équations de la description soient vraies (aux tolérances près). La quantité est toujours d'un type ou sous-type de genre réel. Les quantités peuvent être

- **libres** : simples variables à la discrétion du concepteur. Elles sont simplement déclarées en tant que telles : `quantity Q : real ;`
- **de branches** : les quantités associées à un terminal –**through, across**- subissent les lois de conservation sans qu'on ait à les écrire. Elles sont déclarées avec la mention de leur genre et des terminaux concerné :
  `quantity V across term_plus to term_moins ;`
  (V est la mesure de la tension entre plus et moins ; si on en déclare d'autres entre les mêmes terminaux, leurs valeurs seront identiques)
  `quantity I through term_plus to term_moins ;`
  (I est un des courants passant entre plus et moins. Si on en déclare d'autres entre les mêmes terminaux, leur effet sera additif). On peut aussi factoriser les déclarations, ici on déclare une tension et deux courants :
  ```
  quantity V across
           I1, I2 through
           term_plus to term_moins ;
  ```
  On peut encore omettre le second terminal, c'est la référence de la nature qui sera utilisée.
  Enfin, si on utilise des vecteurs de terminaux (Vect1 **to** Vect2), la correspondance se fera terme à terme si les deux côtés sont de même structure, ou *un vers tous* si un des côtés est scalaire et l'autre est structuré. Ici T1 est scalaire, T2 est un tableau de terminaux ; la déclaration T1 to T2 va créer des vecteurs de quantités, comme si l'on avait déclaré trois fois T1 **to** T2(x)

- **sources** : ce sont des quantités utilisables seulement dans le domaine fréquentiel, qui produisent au choix du bruit (**noise**) ou un spectre de fréquences (**spectrum**).
  quantity Q : real **spectrum** *magnitude, phase* ;
  quantity Q : real **noise** *puissance* ;

- **Le terminal** est, dans le monde électrique, l'équipotentielle qui porte les lois de conservation. Ainsi dans le monde électrique on ne peut parler de tension qu'entre deux terminaux (un terminal particulier déclaré dans la définition de la nature, la référence, fait office de masse). On ne peut parler de courant qu'en termes de « passage » à travers le terminal, la loi étant que la somme algébrique de tous les passages est nulle. Déclaration : `terminal T : electrical ;`
  Le terminal est défini par une **nature**, laquelle, comme vu ci-dessus, à son tour définit
  - deux quantités (**across**, **through**), accessibles depuis le terminal par les attributs éponymes : Term'THROUGH et T'ACROSS.
  - une référence sous la forme d'un terminal dédié (~la masse) accessible par un attribut Nat'REFERENCE. Attention, le même attribut appliqué à un terminal rend une quantité **across** entre ce terminal et la référence de sa nature.
  - le jeu d'équations de la loi de conservation correspondante (**across** ~ tension est unique entre deux terminaux, **through** ~ courant est tel que la somme de tous les **through** d'un jeu de terminaux interconnectés est nulle).

Quantités et terminaux peuvent apparaître en place de ports dans les interfaces des entités et des composants.

```
entity E is
port (signal S : out BIT ; quantity Q : in real ; terminal T : electrical)
end entity E ;
```

Contrairement aux autres objets du langage, les valeurs associées sont toujours de type réel ou d'un sous-type de genre réel, et sont initialisées à 0.0. Rappelons que dans le monde séquentiel, la valeur initiale est la plus petite du type, et donc pour le type réel est un réel négatif de très grande valeur absolue :

```
signal S : real ;
    -- initialisé à un nombre négatif très petit (de grande valeur absolue)
quantity Q : real ; -- initialisée à 0.0.
```

Attention: les valeurs initiales, explicites ou implicites, ne sont que les valeurs proposées au simulateur pour commencer ses heuristiques de résolution.

11.1.2 Instructions

Les quelques constructions spécifiques sont:

- **L'équation** : qui a deux branches d'ordre indifférent, et que le solveur cherche à « rendre vraie » :
  - la version simple : `A==B ;`
  - l'instruction conditionnelle:
    ```
    if condition use A==B ;
    elsif condition use C==D ; elsif…else…end use;
    ```
  - l'instruction sélectée:
    ```
    case expression use
        when X => A==B ;
        when Y => C==D ;…end case;
    ```

    Les conditions sont booléennes et donc changent de valeur instantanément : dans la plupart des cas, cela se traduira par une discontinuité dans le modèle. Cette discontinuité doit être signalée au simulateur par l'instruction break, voir ci-dessous § 11.1.3.
- **Le procedural:** sorte de processus permettant d'écrire du code séquentiel.

  ```
  procedural is
    déclarations
  begin
    instructions
  end procedural ;
  ```

  Cette instruction est définie par une équivalence de la forme équation simple, comprenant d'un côté l'agrégat de toutes les quantités affectées dans le **procedural**, et de l'autre côté un appel de fonction reprenant le code du procédural, à qui l'on passe en argument toutes les quantités et terminaux mentionnés.
  `(Q1, Q2, Q3) == Appel_de_F(Q1, Q2, Q3 , Q4, T1,…) ;`
  Le **procedural** est appelé par le noyau un nombre indéterminé de fois (contrairement au processus). Ces appels visent à rendre vraie l'équation équivalente. Les variables locales sont volatiles et réinitialisées à chaque appel (contrairement au processus).

11.1.3 Conditions d'exécution et de solvabilité :

- **Le nombre d'équations et d'inconnues :** il s'agit pour le solveur de résoudre un jeu de N équations. Comme chacun sait, il faut pour cela N inconnues. Cette contrainte en VHDL-AMS n'est pas vérifiée globalement mais localement à chaque couple entité/architecture. La règle est que la somme des quantités libres (les quantités déclarées), des quantités de mode **out** et des quantités **through** des terminaux doit

être égale au nombre d'équations dans chaque architecture. Cela est vérifié à la compilation ou au plus tard à l'élaboration si l'on utilise la génération (**generate**).

- **Les discontinuités** : chaque fois qu'il y a une discontinuité dans la représentation, le solveur doit en être averti au cours de la simulation, explicitement, par une instruction **break** qui est lancée depuis le monde logique, lui-même éventuellement prévenu depuis le monde analogique par l'utilisation de l'attribut ABOVE. Attention : rien ne signale au concepteur qu'il a oublié de mettre cette instruction, mais un bon indice est qu'il utilise des constructions conditionnelles dans le jeu d'équations. Le cas échéant, en cas d'oubli les traces de simulation vont simplement gommer la discontinuité et créer des situations impossibles (dans le cas de la balle qui rebondit, elle va passer à travers le mur avant de revenir, puisque le point de discontinuité n'a guère de chance d'être exactement un point de calcul). Cette instruction signale au solveur qu'il doit reprendre au temps spécifié qui devient donc un point de calcul obligatoire, avec de nouvelles conditions « pseudo-initiales » qu'il doit donc chercher.
- **La convergence** : aucune méthode automatique ne permet au compilateur de garantir la convergence du jeu d'équations. La mise au point des modèles se fait donc essentiellement sur cette question, avec la détermination des « bonnes » valeurs initiales.

11.1.4 Les domaines d'exécution

Les simulateurs peuvent fonctionner dans l'un des trois modes : temporel, fréquentiel, et recherche de stabilité.

- Dans le domaine temporel, le signal prédéfini DOMAIN vaut TIME_DOMAIN, le temps avance et les descriptions peuvent être mixtes (analogique/digital). Les quantités « sources », c'est-à-dire provoquant du bruit l'émission d'un spectre de fréquences sont collées à zéro.
- Dans le domaine fréquentiel, le signal prédéfini DOMAIN vaut FREQUENCY_DOMAIN. La partie digitale du modèle est ignorée et les quantités « sources », c'est-à-dire provoquant du bruit l'émission d'un spectre de fréquences sont fonctionnelles.
- Dans le domaine « stabilité », le signal prédéfini DOMAIN vaut QUIESCENT_DOMAIN. Il s'agit là pour le simulateur de trouver un point stable avant de partir pour de nouvelles aventures.

Le signal DOMAIN est un signal « normal » déclaré dans le paquetage STD.STANDARD. Un processus peut donc parfaitement être sensible sur ce signal et changer de comportement quand le domaine de travail change.

11.2 Éléments de base

11.2.1 Résistance

La résistance est simplement définie par sa valeur et ses bornes qui sont deux terminaux. La valeur peut être donnée en générique (ce sera une constante) ou comme une quantité, auquel cas elle pourra changer en cours de simulation.

```
library ieee;
use
disciplines.electrical_systems.all;
entity resist is
  port (quantity R : RESISTANCE;
        terminal p, m : ELECTRICAL);
end entity resist;
```

```
library ieee;
use
disciplines.electrical_systems.all;
entity resist is
  generic (R : RESISTANCE);
  port (terminal p, m : ELECTRICAL);
end entity resist;
```

```
architecture analog of resist is
    quantity v across i through p to m;
begin
    v == i*R;
end architecture analog;
```

11.2.2 Résistance thermique

On voit ici que le modèle de résistance thermique dans un autre domaine conservatif est tout à fait semblable.

```
library ieee;
use ieee.thermal_systems.all;
entity resistance_th is
  port (quantity k : IN THERMAL_RESISTANCE;
    terminal th1, th2 : THERMAL);
end entity resistance_thermique;

architecture analog of resistance_thermique is
    quantity t across h through th1 to th2;
begin
  t == h*k;
end architecture analog;
```

11.2.3 Capacité

La capacité est simplement définie par sa valeur et ses bornes qui sont deux terminaux. La valeur peut être donnée en générique (ce sera une constante) ou comme une quantité, auquel cas elle pourra changer en cours de simulation.

On teste par la valeur du signal DOMAIN si le modèle est en train de calculer son état DC en chargeant la capacité de façon que sa tension soit égale à `Vinit`; ou s'il est en train de simuler dans le temps.

```
library ieee;
use
disciplines.electrical_systems.all;
entity capacité is
 generic (Vinit : VOLTAGE := 0.0);
 port (quantity C : CAPACITANCE;
       terminal p, m  : ELECTRICAL);
end entity capacité;
```

```
library ieee;
use
disciplines.electrical_systems.all;
entity capacité is
 generic (Vinit: VOLTAGE := 0.0 ;
          C : CAPACITANCE) ;
 port (terminal p, m  : ELECTRICAL);
end entity capacité;
```

```
architecture analog of capacité is
    quantity v across i through p to m;
    quantity Q : CHARGE;
BEGIN
    if (domain = quiescent_domain) use
      Q == C* Vinit;
      v == Vinit;
    elsif (domain = time_domain) use
      Q == C * v ;
      i == Q'dot;
    end use;
end architecture analog;
```

11.2.4 Self

La self est simplement définie par sa valeur et ses bornes qui sont deux terminaux. La valeur peut être donnée en générique (ce sera une constante) ou comme une quantité, auquel cas elle pourra changer en cours de simulation.

On teste par la valeur du signal DOMAIN si le modèle est en train de calculer son état DC en « chargeant » la self de façon que son courant soit égal à `Iinit`; ou s'il est en train de simuler dans le temps.

```
library ieee;
use
disciplines.electrical_systems.all;
entity self is
  generic (Iinit: CURRENT := 0.0);
  port (quantity L : INDUCTANCE ;
        terminal p, m : ELECTRICAL);
end entity self;
```

```
library ieee;
use
disciplines.electrical_systems.all;
entity self is
  generic (Iinit : CURRENT := 0.0;
                   L : INDUCTANCE);
  port (terminal p, m : ELECTRICAL);
end entity self;
```

```
architecture analog of self is
    quantity v across i through p to m;
    quantity F : FLUX;
BEGIN
    if (domain = quiescent_domain) use
      F == L * Iinit;
      i == IInit;
    elsif (domain = time_domain) use
      F == L * i;
      v == F'dot;
    end use;
end architecture analog;
```

11.2.5 Source de tension

Cette source de tension fonctionne dans les domaines de simulation temporelle et fréquentielle. De même que pour les exemples précédents, certaines des quantités d'entrée pourraient être passées en générique. Comme il y en a trois, nous n'allons pas passer en revue toutes les combinaisons possibles.

```
library ieee;
use disciplines.electrical_systems.all;
entity source_tension is
    port (quantity consigne : in VOLTAGE;
          quantity MAG : in VOLTAGE;
          quantity PHASE : in REAL;
          terminal p, m: ELECTRICAL);
end entity source_tension;
```

L'architecture, dans les domaines « quiescent » et « time », fait simplement en sorte que la tension aux bornes des terminaux soit égale, au signe près, à la tension de référence. Dans le domaine « frequency », la tension porte un bruit défini par son spectre (magnitude, phase).

```
library ieee;
use ieee.math_real.all;
architecture analog of source_tension is
   quantity v across i through p to m;
   quantity ac_spec : real spectrum MAG, MATH_2_PI * PHASE/360.0;
begin
  if (domain = quiescent_domain) or (domain = time_domain) use
         v == - consigne;
  else
         v == - ac_spec;
  end use;
end architecture analog;
```

11.2.6 Source de courant

De même que précédemment, cette source de courant fonctionne dans les domaines de simulation temporelle et fréquentielle. Nous ne représenterons pas non plus toutes les combinaisons possibles de permutations entre la généricité et le passage de quantités.

```
library ieee;
use disciplines.electrical_systems.all;
entity source_courant is
    port (quantity consigne : IN CURRENT;
          quantity MAG : in CURRENT;
          quantity PHASE : in REAL;
          terminal p, m: ELECTRICAL);
end entity source_courant;
```

```
library ieee;
use ieee.math_real.all;
architecture analog of source_courant is
  quantity v across i through p to m;
  quantity ac_spec : real spectrum MAG, MATH_2_PI * PHASE/360.0;
begin
  if (domain = quiescent_domain) or (domain = time_domain) use
    i == consigne;
  else
    i == ac_spec;
  end use;
end architecture analog;
```

11.2.7 Interrupteur

Un interrupteur parfait a pour caractéristique que, en position fermé, la tension à ses bornes est nulle, et qu'en position ouverte, le courant qui le traverse est nul. L'entité comporte donc deux terminaux et une quantité de commande.

```
library ieee;
use disciplines.electrical_systems.all;
entity interrupteur is
    port (quantity commande : in REAL;
          terminal t1,t2  : ELECTRICAL);
end entity interrupteur;
```

L'architecture va simplement déclarer les deux égalités i=0 et v=0 selon la valeur de la commande. À noter l'utilisation de l'instruction break sur le signal créé par la commande, puisqu'il y a une discontinuité qu'il faut signaler.

```
architecture analog of interrupteur is
    quantity v across i through t1 to t2;
    signal local : BOOLEAN;
begin
    local <= commande (0.0);
    break on local;
    if (local and (commande > 0.0)) use
        v == 0.0;
    else
        i == 0.0;
    end use;
end architecture analog;
```

11.2.8 Diode

(Exemple repris de [TUT] page 161)

```
library IEEE, Disciplines;
  use disciplines.electrical_system.all;
entity Diode is
  generic (iss: REAL := 1.0e-14;
           n, af: REAL := 1.0;
           tt, cj0, vj, rs, kf: REAL := 0.0);
  port (terminal anode, cathode: electrical);
end entity Diode;
```

```
library IEEE ; use IEEE.math_real.all;
architecture Level0 of Diode is
  quantity vd across id, ic through anode to cathode;
  quantity qc: charge;
  constant vt: REAL := 0.0258;
       -- thermal voltage
begin
 id==iss * (exp((vd-rs*id)/(n*vt)) - 1.0);
 qc==tt*id - 2.0*cj0 * sqrt(vj**2 - vj*vd);
 ic == qc'dot;
end architecture Level0;
```

$$i_d = is \cdot \left(e^{(v_d - rs \cdot i_d)/n \cdot vt} - 1\right)$$

$$i_c = \frac{d}{dt}\left(tt \cdot i_d - 2 \cdot cj0 \cdot \sqrt{vj^2 - vj \cdot v_d}\right)$$

11.2.9 Amplificateur opérationnel idéal

L'amplificateur opérationnel idéal a deux entrées et une sortie, ses équations en mode linéaire sont simples: la différence de potentiel entre les entrées est nulle, l'impédance d'entrée est infinie donc le courant d'entrée est nul.

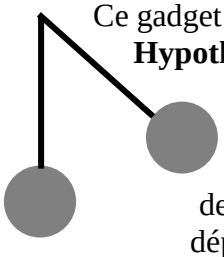
Ce modèle très simple ne traite donc, ni des amplificateurs à gain non infini, ni du mode saturé.

```
library DISCIPLINES;
use DISCIPLINES.electrical_systems.all;

entity ampli_op is

  port (terminal e_plus , e_moins, sortie : electrical);

end entity ampli_op;
```

On notera que le modèle demande deux équations, il faut que la somme des quantités libres, **through** et de mode **out** soit égale à 2. Nous voyons donc ici la nécessité de déclarer une quantité qui n'intervient pas dans les équations explicites : courant_sortie. Elle est évidemment rattachée au système par les équations implicites de Kirchhoff liées à ses terminaux de référence.

```
architecture ideal of ampli_op is

  quantity ddp across courant_entrée through e_plus to e_moins;
  quantity courant_sortie through sortie to ELECTRICAL_REF;

begin

  ddp == 0.0;
  courant_entrée == 0.0;

end architecture ideal;
```

11.2.10 Un modèle mécanique: Tac-Tac



Ce gadget très simple est constitué de deux boules attachées à un point fixe par une ficelle.

Hypothèses : Nous allons supposer ici que

- le choc est parfaitement élastique
- seule la résistance de l'air a donc un effet ralentisseur (la force due à cette résistance est de module $K.S.v^2$ où S est la surface de la projection de l'objet sur un plan perpendiculaire au mouvement, et K une constante dépendant de la forme s'exprimant en kg/m^3 ; on prendra ici K.S = 0.3 kg/m)
- les deux boules sont de masse égale et de diamète négligeable (quasi-ponctuelles.)
- Un peu de trigonométrie nous apprend que la valeur absolue de la composante horizontale de l'accélération due à l'action de la pesanteur est $\boldsymbol{\gamma} = g\, x^2/l^2$

Comme chacun sait, dans ce cas lors du choc les deux vitesses « s'échangent », par exemple si une boule était immobile la boule en mouvement s'arrête et la boule immobile part avec la vitesse de la boule qui s'arrête. C'est l'objet de la deuxième instruction **break**. Ceci se démontre facilement en posant la conservation de la quantité de mouvement (m1v1+m2v2) et la conservation de l'énergie ($m1v1^2/2+m2v2^2/2$).
Avec les hypothèses prises, la masse n'intervient pas dans les équations.

```
library DISCIPLINES; use DISCIPLINES.mechanical_systems.all;
entity tactac is
end entity tactac;

architecture a of tactac is

  quantity v1,v2: velocity;
  quantity x1,x2: displacement;
  constant g:real := 9.81;
  constant longueur_fil:real := 1.0;
  constant KS: real := 0.3;

-- Une fonction qui calcule l'accélération d'une boule en prenant en compte
-- les signes dus aux signes des x et des vitesses.

function calculg(x,v:real) return real is
begin
  if x>0.0 then
    if v > 0.0 then
      return  - g * (x**2/longueur_fil**2) - (v**2)*KS;
    else
      return  -g * (x**2/longueur_fil**2) + (v**2)*KS;
    end if;
  else
    if v1 > 0.0 then
      return  + g * (x**2/longueur_fil**2) - (v**2)*KS;
    else
      return  + g * (x**2/longueur_fil**2) + (v**2)*KS;
    end if;
  end if;
end;


begin
-- les conditions initiales:
  break v1 => 0.0, x1 => 0.1, v2 => 0.0, x2 => -0.0;
-- la discontinuité lors du choc
```

```
 break v1=>v2, v2=>v1 on x1'above(x2), x2'above(x1);

-- definition des vitesses
  x1'DOT == v1;
  x2'DOT == v2;

-- definition des accélérations
  v1'dot==calculg(x1,v1);
  v2'dot==calculg(x2,v2);

end architecture a;
```

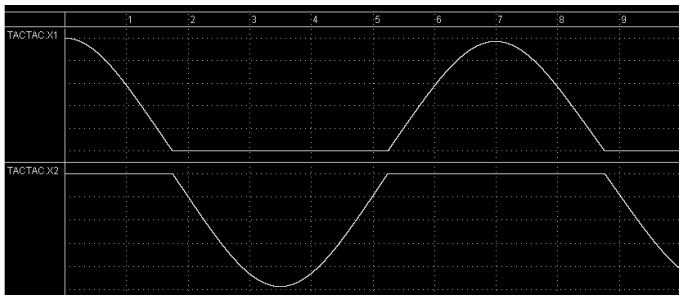


Figure 47 Chronogrammes de l'abscisse des boules d'un "Tac-Tac"

On vérifiera facilement qu'en donnant des conditions initiales symétriques (les deux boules séparées de leur position de repos avec la même quantité), on obtient bien le rebond caractéristique de ce jeu.
Ce modèle est facile à compliquer pour, par exemple, simuler l'excitation que donne le joueur au système en remuant verticalement l'ensemble à la bonne fréquence. On peut aussi simuler un choc plus ou moins mou en calculant la vitesse après le choc d'une façon qui tienne compte de la perte d'énergie (dans le deuxième **break**).

11.3 Modélisation mixte

11.3.1 Convertisseur digital/analogique

11.3.1.1 Réseau R/2R, structurel

Un convertisseur digital analogique sous sa forme la plus simple est un réseau dit « R/2R », c'est-à-dire un réseau dans lequel chaque bit contribue autant que la somme de tous les bits de poids inférieur, par la grâce d'un réseau maillé de ce type :

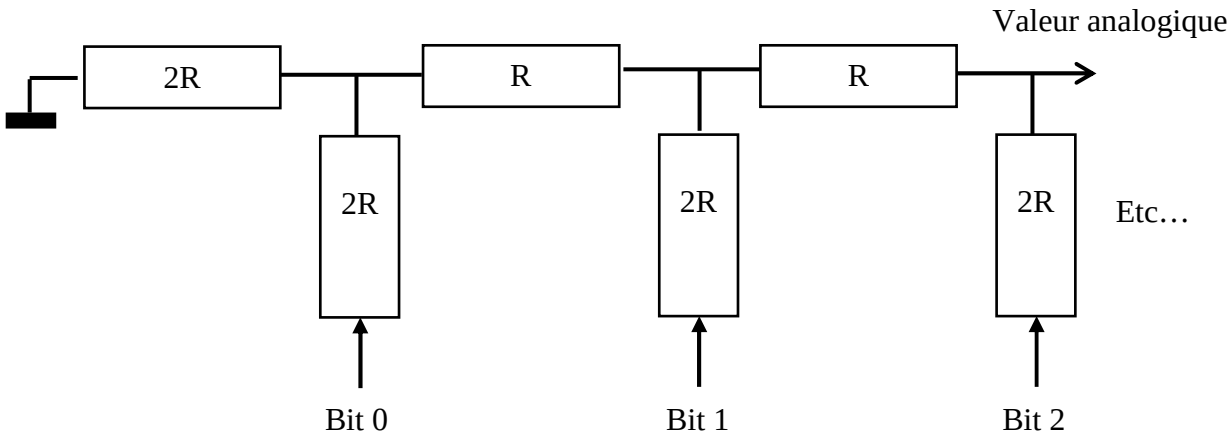


Figure 48 Réseau R/2R

On voit ainsi que, si les entrées numériques sont telles que 0 logique fait 0 volt et 1 logique fait 5 volts, la sortie analogique aura une tension de 5* K * la_valeur_représentée / 2^n volts, où K est un nombre inférieur à 1 qui dépend de N (K=1/2 pour N=1, 3/4 pour N=2, 7/8 pour

N=3, etc.) La démonstration est triviale et relève de l'application de la loi d'Ohm avec un raisonnement par récurrence.

Le plus souvent, les circuits de ce genre contiennent des amplificateurs opérationnels (pour ne pas être contraints par la tension d'alimentation, ou s'affranchir des impédances). Ils peuvent aussi avoir des réseaux de résistances plus simples mais nécessitant des résistances de valeurs R, 2R, 4R, 8R. Or il est infiniment plus facile de faire des résistances identiques dont la valeur exacte importe peu pour autant qu'elle soit commune, que des résistances exactement multiples les unes des autres dans des proportions qui peuvent aller de 1 à 2^{32}. C'est pourquoi le réseau R/2R, servi le plus souvent par des amplificateurs opérationnels, est très répandu.

L'entité d'un tel bloc sera :

```
library ieee; use ieee.std_logic_1164.all;
use disciplines.electrical_systems.all ;
entity R2R is
 generic (N : positive;
          tension_alimentation: voltage := 5.0 ;
          R :resistance :=1000.0) ;
 port (signal entree : std_logic_vector( N-1 downto 0) ;
       terminal sortie : electrical ) ;
end R2R ;
```

L'architecture peut être décrite de façon structurelle, en instanciant les résistances telles que définies au §11.2.1 page 100 :

```
architecture mixte of R2R is
component resist  -- en prenant la copie exacte de l'entité, on s'évite la
        -- configuration car la configuration par défaut va fonctionner.
  generic (R : RESISTANCE);
  port (terminal p, m : ELECTRICAL);
end component;
nature tableau is array (integer range <>) of electrical;
terminal Intermediaires : tableau(N-1 downto 0) ;
terminal Entrees_Analogiques: tableau(N-1 downto 0);
quantity V_EA across I_EA through Entrees_Analogiques to ELECTRICAL_REF;
begin

boucle1: for I in N-1 downto 0 generate
-- ici on convertit pour chaque bit les 0 et 1 logiques
-- en 0.0 et 5.0 volts.
  if Entree(I)='0'
     use V_EA(I) == 0.0 ;
  elsif Entree(I)='1'
     use V_EA(I) == tension_alimentation ;
  end use ;
  break on Entree(I);
end generate;

boucle2: for I in N-2 downto 1 generate
-- on instancie le réseau R/2R excepté à ses deux extrémités qui
-- ne sont pas régulières.
  Res_2R: resist generic map (2.0*R)
          port map (Entrees_analogiques(I), Intermediaires(I)) ;
  Res_R:  resist generic map (R)
          port map (Intermediaires(I-1), Intermediaires(I)) ;
end generate;
```

```
-- la première maille est spécifique, elle est chargée par 2R à la masse.
Res_2R_premier:  resist generic map (2.0*R)
        port map (Entrees_analogiques(0), Intermediaires(0)) ;
Res_charge:  resist generic map (2.0*R)
        port map (ELECTRICAL_REF, Intermediaires(0)) ;

-- la dernière maille sort sur le terminal de sortie et non pas
-- sur le vecteur interne.
Res_2R_dernier : resist generic map (2.0*R)
          port map (Entrees_analogiques(N-1), Sortie) ;
Res_R_dernier :  resist generic map (R)
          port map (Intermediaires(N-2), Sortie) ;
end mixte;
```

11.3.1.2 Convertisseur Digital-Analogique, comportemental

Un convertisseur peut être purement comportemental, c'est-à-dire défini par des équations qui ne préjugent en rien de l'implémentation future.

```
library ieee; use ieee.std_logic_1164.all;
use disciplines.electrical_systems.all ;
entity CNA is
 generic (N : positive;
          tension_reference: voltage := 5.0) ;
 port (signal entree : std_logic_vector( N-1 downto 0) ;
       terminal sortie : electrical ) ;
end CNA;
```

Une architecture comportementale pourrait être :

```
library ieee;
use ieee.std_logic_arith.all;
architecture beh of CNA is
quantity V_Sortie across I_Sortie through sortie  to ELECTRICAL_REF;
begin
  V_Sortie  ==
    tension_reference *(2.0**N-1.0)
       *real(conv_integer(unsigned(entree)))/2.0**(2*N);
  break on entree;
end;
```

On voit qu'ici, le modèle recopie le comportement d'un réseau R/2R et on calcule simplement la tension de sortie en lui appliquant la formule vue §11.3.1.1 ci-dessus. La seule différence est que l'impédance de sortie de ce modèle comportemental est nulle, ce modèle représente donc un système actif à amplificateur opérationnel, utilisant un réseau R/2R.

11.3.2 Convertisseur Analogique-Digital

Un tel convertisseur demande classiquement un convertisseur digital/analogique, un comparateur et un système digital proposant des valeurs sur la base du comparateur. Ce système peut être très lent et rustique (un compteur qui s'incrémente jusqu'à trouver la valeur par dépassement du seuil) ou un peu plus sophistiqué (essais par interpolations successives).

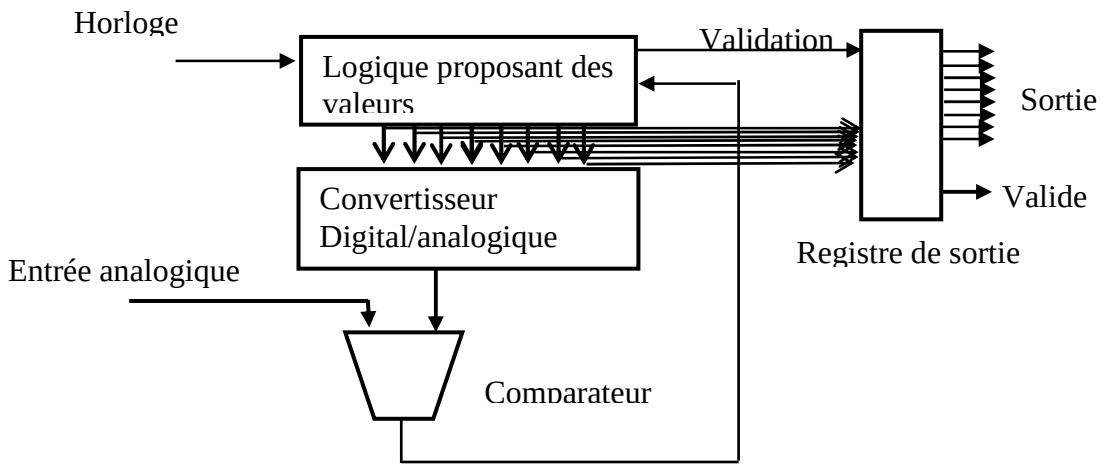


Figure 49 Convertisseur Analogique/Digital

Le convertisseur D/A est de la forme décrite plus haut, par exemple un R/2R.

11.3.2.1 Comparateur

Le comparateur prend deux valeurs analogiques en entrée et soumet 1 ou 0 sur son unique sortie logique selon la comparaison des deux valeurs.
L'attribut ABOVE va faire tout le travail, comme il rend un booléen et que nous avons besoin d'un STD_LOGIC nous utilisons une affectation de signal conditionnelle.

```
library ieee;
use ieee.std_logic_1164.all;
use disciplines.electrical_systems.all ;
entity comp is
 port (terminal A,B: electrical; resultat: out std_logic);
end;

architecture beh of comp is
 quantity IA across A;
 quantity IB across B;
begin
  resultat <= '1' when IA'ABOVE(IB) else '0';
end;
```

11.3.2.2 Proposer des valeurs (modèle purement digital)

Nous allons utiliser ici une solution rapide pour proposer des valeurs au comparateur, en procédant par essais et décisions sur chaque bit individuellement en commençant par le bit de poids fort. Par exemple, supposons une évaluation sur 4 bits et que la valeur analogique à déterminer soit entre la $10^{ème}$ position « 1001 » et la $11^{ème}$ position « 1010 » (sur 16, en commençant à 0)

- On part de 0000
  - ❖ On propose 0000, le résultat de la comparaison après CN/A est 0 (trop petit)
  - ❖ On change le premier bit
  - ❖ On propose 1000. Le résultat de la comparaison est 0 (trop petit) ; on laisse.
- Le premier bit est fixé, passons au second :
  - ❖ 1000 est trop petit, on propose 1100, le résultat du comparateur est 1 (trop grand)

- ❖ On revient à 1000.

- Le second bit est fixé, passons au troisième :
  - ❖ 1000 est trop petit, on propose 1010, le résultat est 1 (trop grand)
  - ❖ On revient à 1000.
- Le troisième bit est fixé, passons au quatrième :
  - ❖ 1000 est trop petit, on propose 1001.
  - ❖ Le résultat de la comparaison est 0 (trop petit) ; on laisse.

Le résultat est donc « 1001 »
Pour des raisons de simplicité, nous ne gérerons pas d'indicateurs de dépassement.

L'entité aura en entrée une horloge, dont nous utiliserons les deux fronts : l'un pour proposer la nouvelle valeur au convertisseur N /A, l'autre pour lire le résultat de la comparaison. Il y a deux ports sortants de N bits, l'un pour proposer la valeur d'essai au convertisseur, l'autre pour fournir en sortie une valeur stable entre deux évaluations. Une entrée venant du comparateur, et une sortie indiquant que la sortie est valide.

```
library ieee;
use ieee.std_logic_1164.all;
entity proposeur is
  generic (N:integer:=16);
  port (    horloge:in std_logic;
            reset: in std_logic;
            resultat_comparaison: in std_logic;
            valide : out std_logic ;
            proposition, sortie: out std_logic_vector(N-1 downto 0));
end entity proposeur;
```

L'architecture est organisée autour d'un processus qu'on rend sensible sur l'horloge et sur le reset.

```
library ieee;
use ieee.std_logic_1164.all;
architecture simple of proposeur is

begin
  process(horloge, reset)
    variable registre: std_logic_vector(N-1 downto 0);
    variable I:integer:=N-1;  -- pour indexer les bits
  begin
    if reset='1' then               -- zone de reset
      registre:=(others=>'0');
      proposition<=registre;
      sortie<=registre;
      valide<='0';
      I :=N-1 ;
    elsif horloge'event then        -- zone activée par l'horloge
      -- on suppose que la valeur proposée ici est toujours trop petite
      -- et que le bit courant vaut 0 (voir algorithme détaillé plus haut)
      valide<='1';           -- la sortie est ici stable et valide
                             -- (cf ci-dessous)
      if horloge='1' then    -- front montant
        registre(I):='1';    -- on passe à 1 le bit courant
      else                   -- front descendant
        if resultat_comparaison='1' then
                          -- si la valeur est devenue trop grande,
                          -- on revient en arrière.
```

```
          registre(I):='0';
        end if;
        I := I-1;
      end if;
      proposition<=registre;
      if I=0 then
        sortie<=registre;
        registre:=(others=>'0);
        valide<='0'; -- !!! la sortie sera stable et valide au passage à 1
                     -- de ce signal (car attention au fait que les
                     -- signaux ne changent que sur un wait, sortie sera
                     -- proposée au prochain wait et sa stabilité
                     -- demandera encore un demi-cycle d'horloge, cf supra)
        I:=N-1;
      end if;
    end if; -- reset/horloge
  end process;
end simple;
```

11.3.2.3 Assemblage du convertisseur : modèle structurel mixe analogique/digital

Pour assembler notre CAN, nous allons utiliser une architecture structurelle.

L'entité sera calquée sur le schéma du paragraphe 11.3.2:

```
library ieee;
use ieee.std_logic_1164.all;
use disciplines.electrical_systems.all ;
entity CAN is
generic(N:positive);
port (horloge : in std_logic;
      reset: in std_logic;
      terminal entree : electrical;
      sortie: out std_logic_vector(N-1 downto 0);
      valide: out std_logic);
end CAN;
```

L'architecture va appeler nos trois blocs que sont le proposeur, le comparateur et le CAN:

```
architecture struct of CAN is

component proposeur
  generic (N:integer);
  port (    horloge:in std_logic;
            reset: in std_logic;
            resultat_comparaison: in std_logic;
            valide : out std_logic ;
            proposition, sortie: out std_logic_vector(N-1 downto 0));
end component;
```

```
component comp
 port (terminal A,B: electrical; resultat: out std_logic);
end component;

component CNA
 generic (N : positive;
          tension_reference: voltage ) ;
 port (signal entree : std_logic_vector( N-1 downto 0) ;
       terminal sortie : electrical ) ;
end component;

terminal local: electrical;

signal local_prop: std_logic_vector(N-1 downto 0);

signal result_comp: std_logic;

begin

inst_CNA: CNA generic map(N, 5.0) port map (local_prop, local);
inst_comp : comp port map (entree , local, result_comp) ;
inst_proposeur: proposeur generic map(N)
                          port map(horloge,
                                   reset,
                                   result_comp,
                                   valide,
                                   local_prop,sortie );
end ;
```

Au test, nous pourrons observer la convergence de la valeur analogique essayée (après conversion CNA) vers la valeur de référence comme ci-dessous :

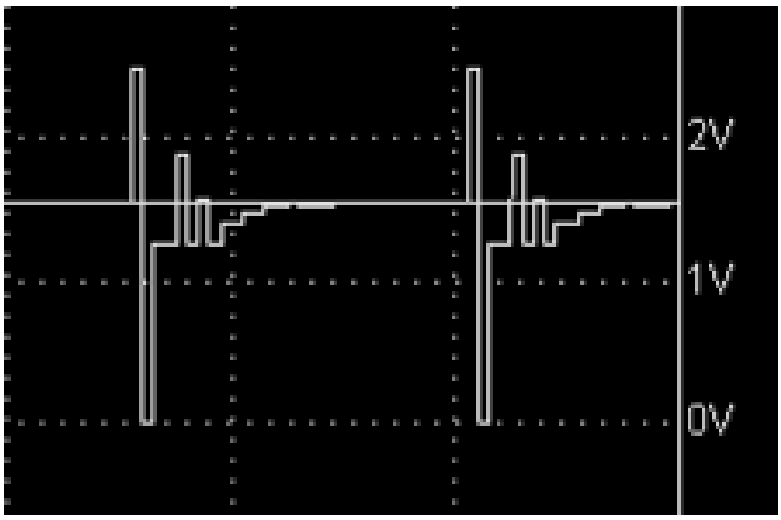


Figure 50 Convergence de la conversion A/D par essais successifs

12 Références

12.1 Standard et utiles

12.1.1 STD.STANDARD

En italique les lignes spécifiques AMS.

```
package standard is
      type boolean is (false,true);
      type bit is ('0', '1');

      type character is (
             nul, soh, stx, etx, eot, enq, ack, bel, bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
            dle, dc1, dc2, dc3, dc4, nak, syn, etb, can, em,  sub, esc, fsp, gsp, rsp, usp,
            ' ', '!', '"', '#', '$', '%', '&', ''', '(', ')', '*', '+', ',', '-', '.', '/',
            '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?',
            '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
            'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\', ']', '^', '_',
            '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
            'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}', '~', del,
            c128, c129, ……c143,
            c144, c145, ……c159,
            ' ', '¡', '¢', '£', '¤', '¥', '¦', '§', '¨', '©', 'ª', '«', '¬', '-', '®', '¯',
            '°', '±', '²', '³', '´', 'µ', '¶', '·', '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',
            'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç', 'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
            'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×', 'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
            'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç', 'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
            'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷', 'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ'
                        );

      type severity_level is (note, warning, error, failure);

      type integer is range -2147483648 to 2147483647;
      type real is range -1.0E308 to 1.0E308;
      type time is range -2147483647 to 2147483647
            units
                  fs;
                  ps = 1000 fs;
                  ns = 1000 ps;
                  us = 1000 ns;
                  ms = 1000 us;
                  sec = 1000 ms;
                  min = 60 sec;
                  hr = 60 min;
            end units;
        type DOMAIN_TYPE is (QUIESCENT_DOMAIN,
                             TIME_DOMAIN,
                             FREQUENCY_DOMAIN);
        signal DOMAIN: DOMAIN_TYPE:=QUIESCENT_DOMAIN;
      subtype delay_length is time range 0 fs to time'high;
      impure function now return delay_length;
```

```
      impure function now return real;
            -- deviendra pure dans la prochaine révision
      function frequency return real;
     subtype natural is integer range 0 to integer'high;
     subtype positive is integer range 1 to integer'high;
      type real_vector is array(natural range <>) of real;
     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 standard;
```

12.1.2 STD.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 READ(L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out bit);
    procedure READ(L:inout LINE; VALUE: out bit_vector;
                   GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out bit_vector);
    procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out character);
    procedure READ(L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out integer);
    procedure READ(L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out real);
    procedure READ(L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out string);
    procedure READ(L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);
    procedure READ(L:inout LINE; VALUE: out time);
    procedure WRITELINE(file f : TEXT; L : inout LINE);
    procedure WRITE(L : inout LINE; VALUE : in bit;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in bit_vector;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in character;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in integer;
            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 string;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0);
    procedure WRITE(L : inout LINE; VALUE : in time;
            JUSTIFIED: in SIDE := right;
            FIELD: in WIDTH := 0;
            UNIT: in TIME := ns);
end;
```

12.1.3 STD_LOGIC_TEXTIO

Une copie de TEXTIO mais le type BIT y est remplacé par le type STD_LOGIC. Paquetage non standard.

```
use STD.textio.all;
library IEEE;
use IEEE.std_logic_1164.all;
package STD_LOGIC_TEXTIO is

      -- Read and Write procedures for STD_ULOGIC and STD_ULOGIC_VECTOR
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC);
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC;
                     GOOD: out BOOLEAN);
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
      procedure READ(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;
                     GOOD: out BOOLEAN);
      procedure WRITE(L:inout LINE; VALUE:in STD_ULOGIC;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      procedure WRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      -- Read and Write procedures for STD_LOGIC_VECTOR
      procedure READ(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
      procedure READ(L:inout LINE; VALUE:out STD_LOGIC_VECTOR;
                     GOOD: out BOOLEAN);
      procedure WRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      --
      -- Read and Write procedures for Hex and Octal values.
      -- The values appear in the file as a series of characters
      -- between 0-F (Hex), or 0-7 (Octal) respectively.
      --
      -- Hex
      procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
      procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure HWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
      procedure HREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure HWRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      -- Octal
      procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
      procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure OWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
      procedure OREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR);
      procedure OREAD(L:inout LINE; VALUE:out STD_LOGIC_VECTOR;
                      GOOD: out BOOLEAN);
      procedure OWRITE(L:inout LINE; VALUE:in STD_LOGIC_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
end STD_LOGIC_TEXTIO;
```

12.2 IEEE

12.2.1 IEEE.STD_LOGIC_1164

```
PACKAGE std_logic_1164 IS
    ------------------------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------------------------
    TYPE std_ulogic IS ( 'U',  -- Uninitialized
                         'X',  -- Forcing  Unknown
                         '0',  -- Forcing  0
                         '1',  -- Forcing  1
                         'Z',  -- High Impedance
                         'W',  -- Weak     Unknown
                         'L',  -- Weak     0
                         'H',  -- Weak     1
                         '-'   -- Don't care
                       );
    ------------------------------------------------------------------
    -- unconstrained array of std_ulogic for use with
    -- the resolution function
    ------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;
    ------------------------------------------------------------------
    -- resolution function
    ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;
    ------------------------------------------------------------------
    -- *** industry standard logic type ***
    ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;
    ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
    ------------------------------------------------------------------
    -- common subtypes
    ------------------------------------------------------------------
    SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1';
                    -- ('X','0','1')
    SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z';
                    -- ('X','0','1','Z')
    SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1';
                    -- ('U','X','0','1')
    SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z';
                    -- ('U','X','0','1','Z')
    ------------------------------------------------------------------
    -- overloaded logical operators
    ------------------------------------------------------------------
    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
    ------------------------------------------------------------------
    -- vectorized overloaded logical operators
    ------------------------------------------------------------------
    FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
```

```
    FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
    FUNCTION "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;
    FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;
    ---------------------------------------------------------------------
    -- conversion functions
    ---------------------------------------------------------------------
    FUNCTION To_bit       ( s : std_ulogic; xmap : BIT := '0') RETURN BIT;
    FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0')
                           RETURN BIT_VECTOR;
    FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0')
                           RETURN BIT_VECTOR;
    FUNCTION To_StdULogic  ( b : BIT ) RETURN std_ulogic;
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR) RETURN std_logic_vector;
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector )
                               RETURN std_logic_vector;
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR) RETURN std_ulogic_vector;
    FUNCTION To_StdULogicVector ( s : std_logic_vector)
                               RETURN std_ulogic_vector;
    ---------------------------------------------------------------------
    -- strength strippers and type convertors
    ---------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_X01  ( b : BIT               ) RETURN  X01;
    FUNCTION To_X01Z ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( s : std_ulogic        ) RETURN  X01Z;
    FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_X01Z ( b : BIT               ) RETURN  X01Z;
    FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( s : std_ulogic        ) RETURN  UX01;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_logic_vector;
    FUNCTION To_UX01  ( b : BIT_VECTOR        ) RETURN  std_ulogic_vector;
    FUNCTION To_UX01  ( b : BIT               ) RETURN  UX01;
    ---------------------------------------------------------------------
    -- edge detection
    ---------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    ---------------------------------------------------------------------
    -- object contains an unknown
    ---------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_ulogic) RETURN  BOOLEAN;
END std_logic_1164;
```

12.2.2 IEEE.STD_LOGIC_ARITH

```
library IEEE;
use IEEE.std_logic_1164.all;
package std_logic_arith is
    type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
    type SIGNED is array (NATURAL range <>) of STD_LOGIC;
    subtype SMALL_INT is INTEGER range 0 to 1;
    function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: INTEGER) return SIGNED;
    function "+"(L: INTEGER; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
    function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
    function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
    function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
    function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
    function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: INTEGER) return SIGNED;
    function "-"(L: INTEGER; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
    function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
    function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
    function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
    function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
    function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
    function "+"(L: UNSIGNED) return UNSIGNED;
    function "+"(L: SIGNED) return SIGNED;
    function "-"(L: SIGNED) return SIGNED;
```

```
function "ABS"(L: SIGNED) return SIGNED;
function "+"(L: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED) return STD_LOGIC_VECTOR;
function "ABS"(L: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "*"(L: SIGNED; R: SIGNED) return SIGNED;
function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;
```

```
    function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
    function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
    function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
    function SHR(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
    function CONV_INTEGER(ARG: INTEGER) return INTEGER;
    function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
    function CONV_INTEGER(ARG: SIGNED) return INTEGER;
    function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;
    function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: SIGNED; SIZE: INTEGER) return UNSIGNED;
    function CONV_UNSIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return UNSIGNED;
    function CONV_SIGNED(ARG: INTEGER; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: UNSIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER) return SIGNED;
    function CONV_SIGNED(ARG: STD_ULOGIC; SIZE: INTEGER) return SIGNED;
    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
-- zero extend STD_LOGIC_VECTOR (ARG) to SIZE, SIZE < 0 is same as SIZE = 0
    -- returns STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function EXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER)
                 return STD_LOGIC_VECTOR;
-- sign extend STD_LOGIC_VECTOR (ARG) to SIZE, SIZE < 0 is same as SIZE = 0
    -- return STD_LOGIC_VECTOR(SIZE-1 downto 0)
    function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return
STD_LOGIC_VECTOR;
end Std_logic_arith ;
```

12.2.3 IEEE.MATH_REAL

```
package MATH_real is

constant MATH_E: real:= 2.71828_18284_59045_23536; -- Value of e
constant MATH_1_OVER_E: real:= 0.36787_94411_71442_32160; -- Value of 1/e
constant MATH_PI: real:= 3.14159_26535_89793_23846; -- Value of pi
constant MATH_2_PI: real := 6.28318_53071_79586_47693; -- Value of 2*pi
constant MATH_1_OVER_PI: real := 0.31830_98861_83790_67154;
constant MATH_PI_OVER_2: real := 1.57079_63267_94896_61923;
constant MATH_PI_OVER_3: real := 1.04719_75511_96597_74615;
constant MATH_PI_OVER_4: real := 0.78539_81633_97448_30962;
constant MATH_3_PI_OVER_2: real := 4.71238_89803_84689_85769;
constant MATH_LOG_OF_2: real := 0.69314_71805_59945_30942;
constant MATH_LOG_OF_10: real := 2.30258_50929_94045_68402;
constant MATH_LOG2_OF_E: real := 1.44269_50408_88963_4074;
constant MATH_LOG10_OF_E: real 0.43429_44819_03251_82765;
constant MATH_SQRT_2: real := := 1.41421_35623_73095_04880;
constant MATH_1_OVER_SQRT_2: real := 0.70710_67811_86547_52440;
constant MATH_SQRT_PI: real := 1.77245_38509_05516_02730;
constant MATH_DEG_TO_RAD: real:= 0.01745_32925_19943_29577;
constant MATH_RAD_TO_DEG: real:= 57.29577_95130_82320_87680;
function SIGN (X: in real) return real;
function CEIL (X: in real) return real;
function FLOOR (X: in real) return real;
function ROUND (X: in real) return real;
function TRUNC (X: in real) return real;
function "MOD" (X, Y: in real) return real;
function REALMAX (X, Y : in real) return real;
function REALMIN (X, Y : in real) return real;
procedure UNIFORM (
variable SEED1, SEED2: inout positive;
variable X: out real);
function SQRT (X: in real) return real;
function CBRT (X: in real) return real;
function "**" (X: in integer; Y: in real) return real;
function "**" (X: in real; Y: in real) return real;
function EXP (X: in real) return real;
function LOG (X: in real) return real;
function LOG2 (X: in real) return real;
function LOG10 (X: in real) return real;
function LOG (X: in real; BASE: in real) return real;
function SIN (X: in real) return real;
function COS ( X: in real) return real;
function TAN (X: in real) return real;
function ARCSIN (X: in real) return real;
function ARCCOS (X: in real) return real;
function ARCTAN (Y : in real) return real;
function ARCTAN (Y : in real; X: in real) return real;
function SINH (X: in real) return real;
function COSH (X: in real) return real;
function TANH (X: in real) return real;
function ARCSINH (X: in real) return real;
function ARCCOSH (X: in real) return real;
function ARCTANH (X: in real) return real;
end MATH_REAL;
```

12.3 IEEE-VITAL

12.3.1 BNF Vital

- VITAL_control_**generic**_declaration ::= [**constant**] identifier_list : [**in**] **type**_mark [index_constraint] [:= static_expression] ;
- VITAL_design_**file** ::=VITAL_design_unit { VITAL_design_unit }
- VITAL_design_unit ::= context_clause **library**_unit | context_clause VITAL_**library**_unit
- VITAL_**entity**_declarative_part ::= VITAL_Level0_**attribute**_specification
- VITAL_**entity_generic**_clause ::= **generic** (VITAL_**entity**_interface_list) ;
- VITAL_**entity**_header ::= [VITAL_**entity_generic**_clause] [VITAL_**entity_port**_clause]
- VITAL_**entity**_interface_declaration ::= interface_**constant**_declaration | VITAL_timing_**generic**_declaration | VITAL_control_**generic**_declaration
- | VITAL_**entity_port**_declaration
- VITAL_**entity**_interface_list ::= VITAL_**entity**_interface_declaration { ; VITAL_**entity**_interface_declaration }
- VITAL_**entity_port**_clause ::= **port** (VITAL_**entity**_interface_list) ;
- VITAL_**entity_port**_declaration ::= [**signal**] identifier_list : [mode] **type**_mark [index_constraint] [:= static_expression] ;
- VITAL_functionality_section ::= { VITAL_**Variable**_assignment_statement | **procedure**_call_statement }
- VITAL_internal_**signal**_declaration ::= **signal** identifier_list : **type**_mark [index_constraint] [:= expression] ;
- VITAL_Level_0_**architecture_body** ::= **architecture** identifier **of entity**_name **is** VITAL_Level_0_ **architecture**_declarative_part **begin architecture**_statement_part **end** [**architecture**] [**architecture**_simple_name] ;
- VITAL_Level_0_**architecture**_declarative_part ::= VITAL_Level0_**attribute**_specification { **block**_declarative_item }
- VITAL_Level_0_**entity**_declaration ::= **entity** identifier **is** VITAL_**entity**_header VITAL_**entity**_declarative_part **end** [**entity**] [**entity**_simple_name] ;
- VITAL_Level_1_**architecture_body** ::= **architecture** identifier **of entity**_name **is** VITAL_Level_1_**architecture**_declarative_part **begin** VITAL_Level_1_**architecture**_statement_part **end** [**architecture**] [**architecture**_simple_name] ;
- VITAL_Level_1_**architecture**_declarative_part ::= VITAL_Level1_**attribute**_specification { VITAL_Level_1_**block**_declarative_item }
- VITAL_Level_1_**architecture**_statement_part ::= VITAL_Level_1_concurrent_statement { VITAL_Level_1_concurrent_statement }
- VITAL_Level_1_**block**_declarative_item ::= **constant**_declaration | **alias**_declaration
- | **attribute**_declaration | **attribute**_specification
- VITAL_internal_**signal**_declaration
- VITAL_Level_1_ concurrent_statement ::= VITAL_wire_delay_**block**_statement | VITAL_negative_constraint_**block**_statement | VITAL_**process**_statement | VITAL_primitive_concurrent_**procedure**_call
- VITAL_Level1_Memory_**architecture_body** ::= **architecture** identifier **of entity**_name **is** VITAL_Level1_Memory_**architecture**_declarative_part **begin** VITAL_Level1_Memory_**architecture**_statement_part **end** [**architecture**] [**architecture**_simple_name] ;
- VITAL_Level1_Memory_**architecture**_declarative_part ::= VITAL_Level1_Memory_**attribute**_specification { VITAL_Level1_Memory_**block**_declarative_item }
- VITAL_Level1_Memory_**architecture**_statement_part ::= VITAL_Level1_Memory_concurrent_statement { VITAL_Level1_Memory_concurrent_statement }
- VITAL_Level1_Memory_**block**_declarative_item ::= **constant**_declaration | **alias**_declaration | **attribute**_declaration | **attribute**_specification | VITAL_memory_internal_**signal**_declaration
- VITAL_Level1_Memory_concurrent_statement ::= VITAL_wire_delay_**block**_statement | VITAL_negative_constraint_**block**_statement | VITAL_memory_**process**_statement
- | VITAL_memory_output_drive_**block**_statement
- VITAL_Level0_**attribute**_specification ::= **attribute**_specification
- VITAL_Level1_**attribute**_specification ::= **attribute**_specification
- VITAL_Level1_Memory_**attribute**_specification ::= **attribute**_specification

- VITAL_**library**_unit ::= VITAL_Level_0_**entity**_declaration | VITAL_Level_0_**architecture_body** | VITAL_Level_1_**architecture_body** | VITAL_Level_1_memory_**architecture_body**
- VITAL_memory_functionality_section ::= { VITAL_**Variable**_assignment_statement | VITAL_memory_**procedure**_call_statement }
- VITAL_memory_internal_**signal**_declaration ::= **signal** identifier_list : **type**_mark [index_constraint] [:= expression] ;
- VITAL_memory_**process**_declarative_item ::= **constant**_declaration | **alias**_declaration | **attribute**_declaration | **attribute**_specification | VITAL_**Variable**_declaration | VITAL_memory_**Variable**_declaration
- VITAL_memory_**process**_declarative_part ::= { VITAL_memory_**process**_declarative_item }
- VITAL_memory_**process**_statement ::= **process_label** : **process** (sensitivity_list) VITAL_memory_**process**_declarative_part **begin** VITAL_memory_**process**_statement_part **end process** [**process_label**] ;
- VITAL_memory_**process**_statement_part ::= [VITAL_memory_timing_check_section] [VITAL_memory_functionality_section] [VITAL_memory_path_delay_section]
- VITAL_memory_timing_check_condition ::= **generic**_simple_name
- VITAL_memory_timing_check_section ::= **if** VITAL_memory_timing_check_condition **then** { VITAL_memory_timing_check_statement } **end if** ;
- VITAL_memory_timing_check_statement ::= **procedure**_call_statement
- VITAL_memory_**Variable**_declaration ::= **Variable** identifier_list : **type**_mark [index_constraint] [:= expression] ;
- VITAL_negative_constraint_**block**_statement ::= **block_label** : **block begin** VITAL_negative_constraint_**block**_statement_part **end block** [**block_label**] ;
- VITAL_negative_constraint_**block**_statement_part ::= { VITAL_negative_constraint_concurrent_**procedure**_call | VITAL_negative_constraint_**generate**_statement _part}
- VITAL_negative_constraint_concurrent_**procedure**_call ::= concurrent_**procedure**_call
- VITAL_negative_constraint_**generate**_statement _part ::= VITAL_negative_constraint_**generate**_statement { VITAL_negative_constraint_**generate**_statement}
- VITAL_negative_constraint_**generate**_parameter_specification ::= identifier **in range_attribute**_name
- VITAL_negative_constraint_**generate**_statement ::= **generate_label** : **for** VITAL_negative_constraint_**generate**_parameter_specification **generate** { VITAL_negative_constraint_concurrent_**procedure**_call } **end generate** {**generate_label** }
- VITAL_output_drive_**block**_statement ::= **block_label** : **block begin** VITAL_output_drive_**block**_statement_part **end block** [**block_label**] ;
- VITAL_output_drive_**block**_statement_part ::= { VITAL_primitive_concurrent_**procedure**_call | concurrent_**signal**_assignment_statement }
- VITAL_primitive_concurrent_**procedure**_call ::= VITAL_primitive_concurrent_**procedure**_call
- VITAL_**process**_declarative_item ::= **constant**_declaration | **alias**_declaration | **attribute**_declaration | **attribute**_specification | VITAL_**Variable**_declaration
- VITAL_**process**_declarative_part ::= { VITAL_**process**_declarative_item }
- VITAL_**process**_statement ::= [**process_label** :] **process** (sensitivity_list) VITAL_**process**_declarative_part **begin** VITAL_**process**_statement_part **end process** [**process_label**] ;
- VITAL_**process**_statement_part ::= [VITAL_timing_check_section] [VITAL_functionality_section] [VITAL_path_delay_section]
- VITAL_target ::= unrestricted_**Variable**_name | memory_unrestricted_**Variable**_name
- VITAL_timing_check_condition ::= **generic**_simple_name
- VITAL_timing_check_section ::= **if** VITAL_timing_check_condition **then** { VITAL_timing_check_statement } **end if** ;
- VITAL_timing_check_statement ::= **procedure**_call_statement
- VITAL_timing_**generic**_declaration ::= [**constant**] identifier_list : [**in**] **type**_mark [index_constraint] [:= static_expression] ;
- VITAL_**Variable**_assignment_statement ::= VITAL_target := expression ;
- VITAL_**Variable**_declaration ::= **Variable** identifier_list : **type**_mark [index_constraint] [:= expression] ;
- VITAL_wire_delay_**block**_statement ::= **block_label** : **block begin** VITAL_wire_delay_**block**_statement_part **end block** [**block_label**] ;

- VITAL_wire_delay_**block**_statement_part ::= {
  VITAL_wire_delay_concurrent_**procedure**_call | VITAL_wire_delay_**generate**_statement
  }
- VITAL_wire_delay_concurrent_**procedure**_call ::= concurrent_**procedure**_call
- VITAL_wire_delay_**generate**_parameter_specification ::= identifier **in**
  range_attribute_name
- VITAL_wire_delay_**generate**_statement ::= **generate_label** : **for**
  VITAL_wire_delay_**generate**_parameter_specification **generate** {
  VITAL_wire_delay_concurrent_**procedure**_call } **end generate** [**generate_label**] ;

12.3.2 VITAL_TIMING

```
LIBRARY IEEE;
USE     IEEE.Std_Logic_1164.ALL;
PACKAGE VITAL_Timing IS
    TYPE VitalTransitionType IS ( tr01, tr10, tr0z, trz1, tr1z, trz0,
                                  tr0X, trx1, tr1x, trx0, trxz, trzx);
    SUBTYPE VitalDelayType     IS TIME;
    TYPE VitalDelayType01   IS ARRAY (VitalTransitionType   RANGE tr01 to tr10)
         OF TIME;
    TYPE VitalDelayType01Z  IS ARRAY (VitalTransitionType   RANGE tr01 to trz0)
         OF TIME;
    TYPE VitalDelayType01ZX IS ARRAY (VitalTransitionType   RANGE tr01 to trzx)
         OF TIME;
    TYPE VitalDelayArrayType     IS ARRAY (NATURAL RANGE <>) OF VitalDelayType;
    TYPE VitalDelayArrayType01   IS ARRAY (NATURAL RANGE <>) OF VitalDelayType01;
    TYPE VitalDelayArrayType01Z  IS ARRAY (NATURAL RANGE <>) OF VitalDelayType01Z;
    TYPE VitalDelayArrayType01ZX IS ARRAY (NATURAL RANGE <>) OF VitalDelayType01ZX;
    CONSTANT VitalZeroDelay     : VitalDelayType     :=   0 ns;
    CONSTANT VitalZeroDelay01   : VitalDelayType01   := ( 0 ns, 0 ns );
    CONSTANT VitalZeroDelay01Z  : VitalDelayType01Z  := ( OTHERS => 0 ns );
    CONSTANT VitalZeroDelay01ZX : VitalDelayType01ZX := ( OTHERS => 0 ns );
    ATTRIBUTE VITAL_Level0 : BOOLEAN;
    ATTRIBUTE VITAL_Level1 : BOOLEAN;
    SUBTYPE std_logic_vector2 IS std_logic_vector(1 DOWNTO 0);
    SUBTYPE std_logic_vector3 IS std_logic_vector(2 DOWNTO 0);
    SUBTYPE std_logic_vector4 IS std_logic_vector(3 DOWNTO 0);
    SUBTYPE std_logic_vector8 IS std_logic_vector(7 DOWNTO 0);
    TYPE VitalOutputMapType  IS ARRAY ( std_ulogic ) OF std_ulogic;
    TYPE VitalResultMapType  IS ARRAY ( UX01       ) OF std_ulogic;
    TYPE VitalResultZMapType IS ARRAY ( UX01Z      ) OF std_ulogic;
    CONSTANT VitalDefaultOutputMap  : VitalOutputMapType
                                    := "UX01ZWLH-";
    CONSTANT VitalDefaultResultMap  : VitalResultMapType
                                    := ( 'U', 'X', '0', '1' );
    CONSTANT VitalDefaultResultZMap : VitalResultZMapType
                                    := ( 'U', 'X', '0', '1', 'Z' );
    TYPE VitalTimeArrayT  IS ARRAY (INTEGER RANGE <>) OF TIME;
    TYPE VitalTimeArrayPT IS ACCESS VitalTimeArrayT;
    TYPE VitalBoolArrayT  IS ARRAY (INTEGER RANGE <>) OF BOOLEAN;
    TYPE VitalBoolArrayPT IS ACCESS VitalBoolArrayT;
    TYPE VitalLogicArrayPT IS ACCESS std_logic_vector;
    TYPE VitalTimingDataType IS RECORD
        NotFirstFlag : BOOLEAN;
        RefLast   : X01;
        RefTime   : TIME;
        HoldEn    : BOOLEAN;
        TestLast  : std_ulogic;
        TestTime  : TIME;
        SetupEn   : BOOLEAN;
        TestLastA : VitalLogicArrayPT;
        TestTimeA : VitalTimeArrayPT;
        HoldEnA   : VitalBoolArrayPT;
        SetupEnA  : VitalBoolArrayPT;
    END RECORD;
    FUNCTION VitalTimingDataInit RETURN VitalTimingDataType;
    TYPE VitalPeriodDataType IS RECORD
        Last : X01;
        Rise : TIME;
        Fall : TIME;
        NotFirstFlag : BOOLEAN;
    END RECORD;
    CONSTANT VitalPeriodDataInit : VitalPeriodDataType
                                 := ('X', 0 ns, 0 ns, FALSE );
    TYPE VitalGlitchKindType IS (OnEvent, OnDetect, VitalInertial, VitalTransport);
```

```
TYPE VitalGlitchDataType IS
  RECORD
    SchedTime    : TIME;
    GlitchTime   : TIME;
    SchedValue   : std_ulogic;
    LastValue    : std_ulogic;
  END RECORD;
TYPE VitalGlitchDataArrayType IS ARRAY (NATURAL RANGE <>)
     OF VitalGlitchDataType;
TYPE VitalPathType IS RECORD
    InputChangeTime : TIME;             -- timestamp for path input signal
    PathDelay       : VitalDelayType;   -- delay for this path
    PathCondition   : BOOLEAN;          -- path sensitize condition
END RECORD;
TYPE VitalPath01Type IS RECORD
    InputChangeTime : TIME;             -- timestamp for path input signal
    PathDelay       : VitalDelayType01; -- delay for this path
    PathCondition   : BOOLEAN;          -- path sensitize condition
END RECORD;
TYPE VitalPath01ZType IS RECORD
    InputChangeTime : TIME;             -- timestamp for path input signal
    PathDelay       : VitalDelayType01Z;-- delay for this path
    PathCondition   : BOOLEAN;          -- path sensitize condition
END RECORD;
-- For representing multiple paths to an output
TYPE VitalPathArrayType    IS ARRAY (NATURAL RANGE <> ) OF VitalPathType;
TYPE VitalPathArray01Type  IS ARRAY (NATURAL RANGE <> ) OF VitalPath01Type;
TYPE VitalPathArray01ZType IS ARRAY (NATURAL RANGE <> ) OF VitalPath01ZType;
TYPE VitalTableSymbolType IS (
   '/',      -- 0 -> 1
   '\',      -- 1 -> 0
   'P',      -- Union of '/' and '^' (any edge to 1)
   'N',      -- Union of '\' and 'v' (any edge to 0)
   'r',      -- 0 -> X
   'f',      -- 1 -> X
   'p',      -- Union of '/' and 'r' (any edge from 0)
   'n',      -- Union of '\' and 'f' (any edge from 1)
   'R',      -- Union of '^' and 'p' (any possible rising edge)
   'F',      -- Union of 'v' and 'n' (any possible falling edge)
   '^',      -- X -> 1
   'v',      -- X -> 0
   'E',      -- Union of 'v' and '^' (any edge from X)
   'A',      -- Union of 'r' and '^' (rising edge to or from 'X')
   'D',      -- Union of 'f' and 'v' (falling edge to or from 'X')
   '*',      -- Union of 'R' and 'F' (any edge)
   'X',      -- Unknown level
   '0',      -- low level
   '1',      -- high level
   '-',      -- don't care
   'B',      -- 0 or 1
   'Z',      -- High Impedance
   'S'       -- steady value
);
SUBTYPE VitalEdgeSymbolType IS VitalTableSymbolType RANGE '/' TO '*';
TYPE VitalSkewExpectedT
TYPE VitalSkewDataType IS RECORD
    ExpectedType : VitalSkewExpectedType;
    Signal1Old1 : TIME;
    Signal2Old1 : TIME;
    Signal1Old2 : TIME;
    Signal2Old2 : TIME;
END RECORD;
CONSTANT VitalSkewDataInit : VitalSkewDataType := (none,0 ns,0 ns,0 ns,0 ns );
FUNCTION VitalExtendToFillDelay (
        CONSTANT Delay : IN VitalDelayType
      ) RETURN VitalDelayType01Z;
FUNCTION VitalExtendToFillDelay (
        CONSTANT Delay : IN VitalDelayType01
```

```
        ) RETURN VitalDelayType01Z;
    FUNCTION VitalExtendToFillDelay (
            CONSTANT Delay : IN VitalDelayType01Z
        ) RETURN VitalDelayType01Z;
    FUNCTION VitalCalcDelay (
            CONSTANT NewVal : IN std_ulogic   := 'X';
            CONSTANT OldVal : IN std_ulogic   := 'X';
            CONSTANT Delay  : IN VitalDelayType
        ) RETURN TIME;
    FUNCTION VitalCalcDelay (
            CONSTANT NewVal : IN std_ulogic   := 'X';
            CONSTANT OldVal : IN std_ulogic   := 'X';
            CONSTANT Delay  : IN VitalDelayType01
        ) RETURN TIME;
    FUNCTION VitalCalcDelay (
            CONSTANT NewVal : IN std_ulogic   := 'X';
            CONSTANT OldVal : IN std_ulogic   := 'X';
            CONSTANT Delay  : IN VitalDelayType01Z
        ) RETURN TIME;
    PROCEDURE VitalPathDelay (
        SIGNAL   OutSignal          : OUT   std_logic;
        VARIABLE GlitchData         : INOUT VitalGlitchDataType;
        CONSTANT OutSignalName      : IN    string;
        CONSTANT OutTemp            : IN    std_logic;
        CONSTANT Paths              : IN    VitalPathArrayType;
        CONSTANT DefaultDelay       : IN    VitalDelayType      := VitalZeroDelay;
        CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
        CONSTANT XOn                : IN    BOOLEAN             := TRUE;
        CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
        CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
        CONSTANT NegPreemptOn       : IN    BOOLEAN             := FALSE;
CONSTANT IgnoreDefaultDelay : IN    BOOLEAN             := FALSE   );
    PROCEDURE VitalPathDelay01 (
        SIGNAL   OutSignal          : OUT   std_logic;
        VARIABLE GlitchData         : INOUT VitalGlitchDataType;
        CONSTANT OutSignalName      : IN    string;
        CONSTANT OutTemp            : IN    std_logic;
        CONSTANT Paths              : IN    VitalPathArray01Type;
        CONSTANT DefaultDelay       : IN    VitalDelayType01    :=
VitalZeroDelay01;
        CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
        CONSTANT XOn                : IN    BOOLEAN             := TRUE;
        CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
        CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
        CONSTANT NegPreemptOn       : IN    BOOLEAN             := FALSE;
        CONSTANT IgnoreDefaultDelay : IN    BOOLEAN             := FALSE;
        CONSTANT RejectFastPath     : IN    BOOLEAN             := FALSE   );
    PROCEDURE VitalPathDelay01Z (
        SIGNAL   OutSignal          : OUT   std_logic;
        VARIABLE GlitchData         : INOUT VitalGlitchDataType;
        CONSTANT OutSignalName      : IN    string;
        CONSTANT OutTemp            : IN    std_logic;
        CONSTANT Paths              : IN    VitalPathArray01ZType;
        CONSTANT DefaultDelay       : IN    VitalDelayType01Z   :=
VitalZeroDelay01Z;
        CONSTANT Mode               : IN    VitalGlitchKindType := OnEvent;
        CONSTANT XOn                : IN    BOOLEAN             := TRUE;
        CONSTANT MsgOn              : IN    BOOLEAN             := TRUE;
        CONSTANT MsgSeverity        : IN    SEVERITY_LEVEL      := WARNING;
        CONSTANT OutputMap          : IN    VitalOutputMapType  :=
VitalDefaultOutputMap;
        CONSTANT NegPreemptOn       : IN    BOOLEAN             := FALSE;
        CONSTANT IgnoreDefaultDelay : IN    BOOLEAN             := FALSE;
        CONSTANT RejectFastPath     : IN    BOOLEAN             := FALSE
    PROCEDURE VitalWireDelay (
            SIGNAL   OutSig     : OUT   std_ulogic;
            SIGNAL   InSig      : IN    std_ulogic;
            CONSTANT twire      : IN    VitalDelayType);
```

```
PROCEDURE VitalWireDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT twire      : IN    VitalDelayType01);
PROCEDURE VitalWireDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT twire      : IN    VitalDelayType01Z  );
PROCEDURE VitalSignalDelay (
        SIGNAL   OutSig     : OUT   std_ulogic;
        SIGNAL   InSig      : IN    std_ulogic;
        CONSTANT dly        : IN   TIME);
PROCEDURE VitalSetupHoldCheck (
        VARIABLE Violation     : OUT    X01;
        VARIABLE TimingData    : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal    : IN     std_ulogic;
        CONSTANT TestSignalName: IN     STRING := "";
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        SIGNAL   RefSignal     : IN     std_ulogic;
        CONSTANT RefSignalName : IN     STRING := "";
        CONSTANT RefDelay      : IN     TIME := 0 ns;
        CONSTANT SetupHigh     : IN     TIME := 0 ns;
        CONSTANT SetupLow      : IN     TIME := 0 ns;
        CONSTANT HoldHigh      : IN     TIME := 0 ns;
        CONSTANT HoldLow       : IN     TIME := 0 ns;
        CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg     : IN     STRING := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT EnableSetupOnTest : IN BOOLEAN := TRUE;
          CONSTANT EnableSetupOnRef  : IN BOOLEAN := TRUE;
      CONSTANT EnableHoldOnRef   : IN BOOLEAN := TRUE;
          CONSTANT EnableHoldOnTest  : IN BOOLEAN := TRUE );
PROCEDURE VitalSetupHoldCheck (
        VARIABLE Violation     : OUT    X01;
        VARIABLE TimingData    : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal    : IN     std_logic_vector;
        CONSTANT TestSignalName: IN     STRING := "";
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        SIGNAL   RefSignal     : IN     std_ulogic;
        CONSTANT RefSignalName : IN     STRING := "";
        CONSTANT RefDelay      : IN     TIME := 0 ns;
        CONSTANT SetupHigh     : IN     TIME := 0 ns;
        CONSTANT SetupLow      : IN     TIME := 0 ns;
        CONSTANT HoldHigh      : IN     TIME := 0 ns;
        CONSTANT HoldLow       : IN     TIME := 0 ns;
        CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
        CONSTANT RefTransition : IN     VitalEdgeSymbolType;
        CONSTANT HeaderMsg     : IN     STRING := " ";
        CONSTANT XOn           : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT EnableSetupOnTest : IN BOOLEAN := TRUE;
         CONSTANT EnableSetupOnRef  : IN BOOLEAN := TRUE;
         CONSTANT EnableHoldOnRef   : IN BOOLEAN := TRUE;
          CONSTANT EnableHoldOnTest  : IN BOOLEAN := TRUE    );
PROCEDURE VitalRecoveryRemovalCheck (
        VARIABLE Violation     : OUT    X01;
        VARIABLE TimingData    : INOUT  VitalTimingDataType;
        SIGNAL   TestSignal    : IN     std_ulogic;
        CONSTANT TestSignalName: IN     STRING := "";
        CONSTANT TestDelay     : IN     TIME := 0 ns;
        SIGNAL   RefSignal     : IN     std_ulogic;
        CONSTANT RefSignalName : IN     STRING := "";
        CONSTANT RefDelay      : IN     TIME := 0 ns;
        CONSTANT Recovery      : IN     TIME := 0 ns;
```

```
            CONSTANT Removal       : IN     TIME := 0 ns;
            CONSTANT ActiveLow     : IN     BOOLEAN := TRUE;
            CONSTANT CheckEnabled  : IN     BOOLEAN := TRUE;
            CONSTANT RefTransition : IN     VitalEdgeSymbolType;
            CONSTANT HeaderMsg     : IN     STRING := " ";
            CONSTANT XOn           : IN     BOOLEAN := TRUE;
            CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
            CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING;
          CONSTANT EnableRecOnTest : IN   BOOLEAN := TRUE;
              CONSTANT EnableRecOnRef  : IN   BOOLEAN := TRUE;
              CONSTANT EnableRemOnRef  : IN   BOOLEAN := TRUE;
              CONSTANT EnableRemOnTest : IN   BOOLEAN := TRUE);
    PROCEDURE VitalPeriodPulseCheck  (
            VARIABLE Violation      : OUT    X01;
            VARIABLE PeriodData     : INOUT  VitalPeriodDataType;
            SIGNAL   TestSignal     : IN     std_ulogic;
            CONSTANT TestSignalName : IN     STRING := "";
            CONSTANT TestDelay      : IN     TIME := 0 ns;
            CONSTANT Period         : IN     TIME := 0 ns;
            CONSTANT PulseWidthHigh : IN     TIME := 0 ns;
            CONSTANT PulseWidthLow  : IN     TIME := 0 ns;
            CONSTANT CheckEnabled   : IN     BOOLEAN := TRUE;
            CONSTANT HeaderMsg      : IN     STRING := " ";
            CONSTANT XOn           : IN     BOOLEAN := TRUE;
            CONSTANT MsgOn         : IN     BOOLEAN := TRUE;
            CONSTANT MsgSeverity   : IN     SEVERITY_LEVEL := WARNING);
    PROCEDURE VitalInPhaseSkewCheck (
        VARIABLE Violation         : OUT    X01;
        VARIABLE SkewData          : INOUT  VitalSkewDataType;
        SIGNAL   Signal1           : IN     std_ulogic;
        CONSTANT Signal1Name       : IN     STRING := "";
        CONSTANT Signal1Delay      : IN     TIME := 0 ns;
        SIGNAL   Signal2           : IN     std_ulogic;
        CONSTANT Signal2Name       : IN     STRING := "";
        CONSTANT Signal2Delay      : IN     TIME := 0 ns;
        CONSTANT SkewS1S2RiseRise  : IN     TIME := TIME'HIGH;
        CONSTANT SkewS2S1RiseRise  : IN     TIME := TIME'HIGH;
        CONSTANT SkewS1S2FallFall  : IN     TIME := TIME'HIGH;
        CONSTANT SkewS2S1FallFall  : IN     TIME := TIME'HIGH;
        CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
        CONSTANT XOn               : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT HeaderMsg         : IN     STRING  := "";
        SIGNAL   Trigger           : INOUT  std_ulogic );
    PROCEDURE VitalOutPhaseSkewCheck (
        VARIABLE Violation         : OUT    X01;
        VARIABLE SkewData          : INOUT  VitalSkewDataType;
        SIGNAL   Signal1           : IN     std_ulogic;
        CONSTANT Signal1Name       : IN     STRING := "";
        CONSTANT Signal1Delay      : IN     TIME := 0 ns;
        SIGNAL   Signal2           : IN     std_ulogic;
        CONSTANT Signal2Name       : IN     STRING := "";
        CONSTANT Signal2Delay      : IN     TIME := 0 ns;
        CONSTANT SkewS1S2RiseFall  : IN     TIME := TIME'HIGH;
        CONSTANT SkewS2S1RiseFall  : IN     TIME := TIME'HIGH;
        CONSTANT SkewS1S2FallRise  : IN     TIME := TIME'HIGH;
        CONSTANT SkewS2S1FallRise  : IN     TIME := TIME'HIGH;
        CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
        CONSTANT XOn               : IN     BOOLEAN := TRUE;
        CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
        CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
        CONSTANT HeaderMsg         : IN     STRING  := "";
        SIGNAL   Trigger           : INOUT  std_ulogic);
END VITAL_Timing;
```

12.3.3 VITAL_Primitives

```
LIBRARY IEEE;USE IEEE.Std_Logic_1164.ALL;
LIBRARY vital2000;USE vital2000.VITAL_Timing.ALL;
PACKAGE VITAL_Primitives IS
    SUBTYPE VitalTruthSymbolType IS VitalTableSymbolType RANGE 'X' TO 'Z';
    SUBTYPE VitalStateSymbolType IS VitalTableSymbolType RANGE '/' TO 'S';
    TYPE VitalTruthTableType IS ARRAY ( NATURAL RANGE <>, NATURAL RANGE <> )
         OF VitalTruthSymbolType;
    TYPE VitalStateTableType IS ARRAY ( NATURAL RANGE <>, NATURAL RANGE <> )
         OF VitalStateSymbolType;
    CONSTANT VitalDefDelay01  : VitalDelayType01;
    CONSTANT VitalDefDelay01Z : VitalDelayType01Z
    FUNCTION VitalAND    (
            CONSTANT      Data : IN std_logic_vector;
            CONSTANT ResultMap : IN VitalResultMapType := VitalDefaultResultMap
          ) RETURN std_ulogic;
    FUNCTION VitalOR     (
            CONSTANT      Data : IN std_logic_vector;
            CONSTANT ResultMap : IN VitalResultMapType := VitalDefaultResultMap
          ) RETURN std_ulogic;
    FUNCTION VitalXOR    (
            CONSTANT      Data : IN std_logic_vector;
            CONSTANT ResultMap : IN VitalResultMapType := VitalDefaultResultMap
          ) RETURN std_ulogic;
    FUNCTION VitalNAND   (
            CONSTANT      Data : IN std_logic_vector;
            CONSTANT ResultMap : IN VitalResultMapType := VitalDefaultResultMap
          ) RETURN std_ulogic;
    FUNCTION VitalNOR    (
            CONSTANT      Data : IN std_logic_vector;
            CONSTANT ResultMap : IN VitalResultMapType := VitalDefaultResultMap
          ) RETURN std_ulogic;
    FUNCTION VitalXNOR   (
            CONSTANT      Data : IN std_logic_vector;
            CONSTANT ResultMap : IN VitalResultMapType := VitalDefaultResultMap
          ) RETURN std_ulogic;
    PROCEDURE VitalAND   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType:= VitalDefaultResultMap );
    PROCEDURE VitalOR    (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType:= VitalDefaultResultMap );
    PROCEDURE VitalXOR   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType:=VitalDefaultResultMap );
    PROCEDURE VitalNAND  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType:= VitalDefaultResultMap );
    PROCEDURE VitalNOR   (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
            CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT  ResultMap :  IN VitalResultMapType := VitalDefaultResultMap
);
    PROCEDURE VitalXNOR  (
            SIGNAL            q : OUT std_ulogic;
            SIGNAL         Data :  IN std_logic_vector;
```

```
        CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
FUNCTION VitalAND2   (
        CONSTANT       a, b :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalOR2    (
        CONSTANT       a, b :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalXOR2   (
        CONSTANT       a, b :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalNAND2  (
        CONSTANT       a, b :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalNOR2   (
        CONSTANT       a, b :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalXNOR2  (
        CONSTANT       a, b :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalAND3   (
        CONSTANT    a, b, c :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalOR3    (
        CONSTANT    a, b, c :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalXOR3   (
        CONSTANT    a, b, c :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalNAND3  (
        CONSTANT    a, b, c :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalNOR3   (
        CONSTANT    a, b, c :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalXNOR3  (
        CONSTANT    a, b, c :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalAND4   (
        CONSTANT a, b, c, d :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
```

```
FUNCTION VitalOR4    (
        CONSTANT a, b, c, d :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalXOR4   (
        CONSTANT a, b, c, d :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalNAND4  (
        CONSTANT a, b, c, d :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalNOR4   (
        CONSTANT a, b, c, d :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalXNOR4  (
        CONSTANT a, b, c, d :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap
      ) RETURN std_ulogic;
PROCEDURE VitalAND2  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL          a, b :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalOR2   (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL          a, b :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalXOR2  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL          a, b :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalNAND2 (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL          a, b :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalNOR2  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL          a, b :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalXNOR2 (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL          a, b :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalAND3  (
```

```
        SIGNAL            q : OUT std_ulogic;
        SIGNAL      a, b, c :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalOR3   (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL      a, b, c :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalXOR3  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL      a, b, c :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalNAND3 (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL      a, b, c :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalNOR3  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL      a, b, c :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalXNOR3 (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL      a, b, c :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalAND4  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalOR4   (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalXOR4  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic;
```

```
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalNAND4 (
        SIGNAL           q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalNOR4  (
        SIGNAL           q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalXNOR4 (
        SIGNAL           q : OUT std_ulogic;
        SIGNAL   a, b, c, d :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_b_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_c_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT    tpd_d_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
FUNCTION VitalBUF    (
        CONSTANT         Data :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalBUFIF0 (
        CONSTANT Data, Enable :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap
      ) RETURN std_ulogic;
FUNCTION VitalBUFIF1 (
        CONSTANT Data, Enable :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap
      ) RETURN std_ulogic;
FUNCTION VitalIDENT  (
        CONSTANT         Data :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap
      ) RETURN std_ulogic;
PROCEDURE VitalBUF   (
        SIGNAL           q : OUT std_ulogic;
        SIGNAL           a :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                  := VitalDefaultResultMap );
PROCEDURE VitalBUFIF0 (
        SIGNAL             q : OUT std_ulogic;
        SIGNAL          Data :  IN std_ulogic;
        SIGNAL        Enable :  IN std_ulogic;
        CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                  := VitalDefaultResultZMap);
PROCEDURE VitalBUFIF1 (
```

```
        SIGNAL              q : OUT std_ulogic;
        SIGNAL           Data :  IN std_ulogic;
        SIGNAL         Enable :  IN std_ulogic;
        CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap);
PROCEDURE VitalIDENT (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL            a :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
        CONSTANT  ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap );
FUNCTION VitalINV    (
        CONSTANT         Data :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalINVIF0 (
        CONSTANT Data, Enable :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap
      ) RETURN std_ulogic;
FUNCTION VitalINVIF1 (
        CONSTANT Data, Enable :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap
      ) RETURN std_ulogic;
PROCEDURE VitalINV   (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL            a :  IN std_ulogic;
        CONSTANT    tpd_a_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap );
PROCEDURE VitalINVIF0 (
        SIGNAL              q : OUT std_ulogic;
        SIGNAL           Data :  IN std_ulogic;
        SIGNAL         Enable :  IN std_ulogic;
        CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap);
PROCEDURE VitalINVIF1 (
        SIGNAL              q : OUT std_ulogic;
        SIGNAL           Data :  IN std_ulogic;
        SIGNAL         Enable :  IN std_ulogic;
        CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_enable_q :  IN VitalDelayType01Z   := VitalDefDelay01Z;
        CONSTANT    ResultMap :  IN VitalResultZMapType
                                    := VitalDefaultResultZMap);
FUNCTION VitalMUX   (
        CONSTANT       Data :  IN std_logic_vector;
        CONSTANT    dSelect :  IN std_logic_vector;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalMUX2  (
        CONSTANT Data1, Data0 :  IN std_ulogic;
        CONSTANT      dSelect :  IN std_ulogic;
        CONSTANT    ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_ulogic;
FUNCTION VitalMUX4  (
        CONSTANT       Data :  IN std_logic_vector4;
        CONSTANT    dSelect :  IN std_logic_vector2;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_ulogic;
```

```
FUNCTION VitalMUX8  (
        CONSTANT       Data :  IN std_logic_vector8;
        CONSTANT    dSelect :  IN std_logic_vector3;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_ulogic;
PROCEDURE VitalMUX   (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL         Data :  IN std_logic_vector;
        SIGNAL         dSel :  IN std_logic_vector;
        CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
        CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap );
PROCEDURE VitalMUX2  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL       d1, d0 :  IN std_ulogic;
        SIGNAL         dSel :  IN std_ulogic;
        CONSTANT   tpd_d1_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT   tpd_d0_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT tpd_dsel_q :  IN VitalDelayType01    := VitalDefDelay01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap );
PROCEDURE VitalMUX4  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL         Data :  IN std_logic_vector4;
        SIGNAL         dSel :  IN std_logic_vector2;
        CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
        CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap );
PROCEDURE VitalMUX8  (
        SIGNAL            q : OUT std_ulogic;
        SIGNAL         Data :  IN std_logic_vector8;
        SIGNAL         dSel :  IN std_logic_vector3;
        CONSTANT tpd_data_q :  IN VitalDelayArrayType01;
        CONSTANT tpd_dsel_q :  IN VitalDelayArrayType01;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap );
FUNCTION VitalDECODER   (
        CONSTANT       Data :  IN std_logic_vector;
        CONSTANT     Enable :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_logic_vector;
FUNCTION VitalDECODER2  (
        CONSTANT       Data :  IN std_ulogic;
        CONSTANT     Enable :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_logic_vector2;
FUNCTION VitalDECODER4  (
        CONSTANT       Data :  IN std_logic_vector2;
        CONSTANT     Enable :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_logic_vector4;
FUNCTION VitalDECODER8  (
        CONSTANT       Data :  IN std_logic_vector3;
        CONSTANT     Enable :  IN std_ulogic;
        CONSTANT  ResultMap :  IN VitalResultMapType
                                    := VitalDefaultResultMap
      ) RETURN std_logic_vector8;
PROCEDURE VitalDECODER   (
        SIGNAL             q : OUT std_logic_vector;
        SIGNAL          Data :  IN std_logic_vector;
        SIGNAL        Enable :  IN std_ulogic;
        CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
```

```
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap );
    PROCEDURE VitalDECODER2  (
            SIGNAL              q : OUT std_logic_vector2;
            SIGNAL           Data :  IN std_ulogic;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap );
    PROCEDURE VitalDECODER4  (
            SIGNAL              q : OUT std_logic_vector4;
            SIGNAL           Data :  IN std_logic_vector2;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap );
    PROCEDURE VitalDECODER8  (
            SIGNAL              q : OUT std_logic_vector8;
            SIGNAL           Data :  IN std_logic_vector3;
            SIGNAL         Enable :  IN std_ulogic;
            CONSTANT   tpd_data_q :  IN VitalDelayArrayType01;
            CONSTANT tpd_enable_q :  IN VitalDelayType01    := VitalDefDelay01;
            CONSTANT    ResultMap :  IN VitalResultMapType
                                        := VitalDefaultResultMap );
    FUNCTION VitalTruthTable  (
            CONSTANT TruthTable   : IN VitalTruthTableType;
            CONSTANT DataIn       : IN std_logic_vector
          ) RETURN std_logic_vector;
    FUNCTION VitalTruthTable  (
            CONSTANT TruthTable   : IN VitalTruthTableType;
            CONSTANT DataIn       : IN std_logic_vector
          ) RETURN std_logic;
    PROCEDURE VitalTruthTable (
            SIGNAL   Result       : OUT std_logic_vector;
            CONSTANT TruthTable   : IN VitalTruthTableType;
            SIGNAL   DataIn       : IN std_logic_vector  );
    PROCEDURE VitalTruthTable (
            SIGNAL   Result       : OUT std_logic;
            CONSTANT TruthTable   : IN VitalTruthTableType;
            SIGNAL   DataIn       : IN std_logic_vector);
   PROCEDURE VitalStateTable (
            VARIABLE Result         : INOUT std_logic_vector;
            VARIABLE PreviousDataIn : INOUT std_logic_vector;
            CONSTANT StateTable     : IN VitalStateTableType;
            CONSTANT DataIn         : IN  std_logic_vector;
            CONSTANT NumStates      : IN NATURAL  );
   PROCEDURE VitalStateTable (
            VARIABLE Result         : INOUT std_logic;
            VARIABLE PreviousDataIn : INOUT std_logic_vector;
            CONSTANT StateTable     : IN VitalStateTableType;
            CONSTANT DataIn         : IN  std_logic_vector  );
   PROCEDURE VitalStateTable (
            SIGNAL   Result     : INOUT std_logic_vector;
            CONSTANT StateTable : IN VitalStateTableType;
            SIGNAL   DataIn     : IN std_logic_vector;
            CONSTANT NumStates  : IN NATURAL );
   PROCEDURE VitalStateTable (
            SIGNAL   Result     : INOUT std_logic;
            CONSTANT StateTable : IN VitalStateTableType;
            SIGNAL   DataIn     : IN std_logic_vector);
    PROCEDURE VitalResolve (
            SIGNAL              q : OUT std_ulogic;
            SIGNAL           Data :  IN std_logic_vector); --IR236 4/2/98
END VITAL_Primitives;
```

12.3.4 VITAL_MEMORY

```
LIBRARY IEEE;USE IEEE.STD_LOGIC_1164.ALL;
LIBRARY vital2000 USE vital2000.Vital_Timing.ALL;USE
vital2000.Vital_Primitives.ALL;
LIBRARY STD; USE     STD.TEXTIO.ALL;
PACKAGE Vital_Memory IS
TYPE VitalMemoryArcType       IS (ParallelArc, CrossArc, SubwordArc);
TYPE OutputRetainBehaviorType IS (BitCorrupt, WordCorrupt);
TYPE VitalMemoryMsgFormatType IS (Vector, Scalar, VectorEnum);
TYPE X01ArrayT IS ARRAY (NATURAL RANGE <> ) OF X01;
TYPE X01ArrayPT IS ACCESS X01ArrayT;
TYPE VitalMemoryViolationType IS ACCESS X01ArrayT;
CONSTANT DefaultNumBitsPerSubword : INTEGER := -1;
ATTRIBUTE VITAL_Level1_Memory : BOOLEAN;
TYPE VitalMemoryScheduleDataType IS RECORD
  OutputData        : std_ulogic;
  NumBitsPerSubWord : INTEGER;
  ScheduleTime      : TIME;
  ScheduleValue     : std_ulogic;
  LastOutputValue   : std_ulogic;
  PropDelay         : TIME;
  OutputRetainDelay : TIME;
  InputAge          : TIME;
END RECORD;
TYPE VitalMemoryTimingDataType IS RECORD
  NotFirstFlag : BOOLEAN;
  RefLast      : X01;
  RefTime      : TIME;
  HoldEn       : BOOLEAN;
  TestLast     : std_ulogic;
  TestTime     : TIME;
  SetupEn      : BOOLEAN;
  TestLastA    : VitalLogicArrayPT;
  TestTimeA    : VitalTimeArrayPT;
  RefLastA     : X01ArrayPT;
  RefTimeA     : VitalTimeArrayPT;
  HoldEnA      : VitalBoolArrayPT;
  SetupEnA     : VitalBoolArrayPT;
END RECORD;
TYPE VitalPeriodDataArrayType IS ARRAY (NATURAL RANGE <>) OF
  VitalPeriodDataType;
TYPE VitalMemoryScheduleDataVectorType IS ARRAY (NATURAL RANGE <> ) OF
  VitalMemoryScheduleDataType;
TYPE VitalPortStateType IS (UNDEF, READ, WRITE, CORRUPT, HIGHZ);
TYPE VitalPortFlagType IS RECORD
  MemoryCurrent   : VitalPortStateType;
  MemoryPrevious  : VitalPortStateType;
  DataCurrent     : VitalPortStateType;
  DataPrevious    : VitalPortStateType;
  OutputDisable   : BOOLEAN;
END RECORD;
CONSTANT VitalDefaultPortFlag : VitalPortFlagType := (
  MemoryCurrent   => READ,
  MemoryPrevious  => UNDEF,
  DataCurrent     => READ,
  DataPrevious    => UNDEF,
  OutputDisable   => FALSE);
TYPE VitalPortFlagVectorType IS
  ARRAY (NATURAL RANGE <>) OF VitalPortFlagType;
PROCEDURE VitalMemoryInitPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  VARIABLE OutputDataArray   : IN STD_LOGIC_VECTOR;
  CONSTANT NumBitsPerSubWord : IN INTEGER := DefaultNumBitsPerSubword);
PROCEDURE VitalMemoryInitPathDelay (
```

```
  VARIABLE ScheduleData      : INOUT VitalMemoryScheduleDataType;
  VARIABLE OutputData        : IN STD_ULOGIC);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_ULOGIC;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTime      : INOUT Time;
  CONSTANT PathDelay            : IN VitalDelayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray    : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray    : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_ULOGIC;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTime      : INOUT Time;
  CONSTANT PathDelay            : IN VitalDelayType01;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
```

```
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray  : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_ULOGIC;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTime      : INOUT Time;
  CONSTANT PathDelay            : IN VitalDelayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01Z;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01Z;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt);
PROCEDURE VitalMemoryAddPathDelay (
```

```
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01Z;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_ULOGIC;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTime      : INOUT Time;
  CONSTANT PathDelay            : IN VitalDelayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition     : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_ULOGIC;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTime   : INOUT Time;
  CONSTANT PathDelayArray    : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType           : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray: IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag  : IN BOOLEAN := FALSE);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleData         : INOUT VitalMemoryScheduleDataType;
  SIGNAL   InputSignal          : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName     : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathCondition        : IN BOOLEAN := TRUE;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt);
PROCEDURE VitalMemoryAddPathDelay (
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType;
```

```
  SIGNAL   InputSignal       : IN STD_LOGIC_VECTOR;
  CONSTANT OutputSignalName  : IN STRING := "";
  VARIABLE InputChangeTimeArray : INOUT VitalTimeArrayT;
  CONSTANT PathDelayArray       : IN VitalDelayArrayType01ZX;
  CONSTANT ArcType              : IN VitalMemoryArcType := CrossArc;
  CONSTANT PathConditionArray   : IN VitalBoolArrayT;
  CONSTANT OutputRetainFlag     : IN BOOLEAN := FALSE;
  CONSTANT OutputRetainBehavior : IN OutputRetainBehaviorType := BitCorrupt);
PROCEDURE VitalMemorySchedulePathDelay (
  SIGNAL   OutSignal         : OUT std_logic_vector;
  CONSTANT OutputSignalName  : IN STRING := "";
  CONSTANT PortFlag          : IN VitalPortFlagType := VitalDefaultPortFlag;
  CONSTANT OutputMap : IN VitalOutputMapType := VitalDefaultOutputMap;
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType);
PROCEDURE VitalMemorySchedulePathDelay (
  SIGNAL   OutSignal         : OUT std_logic_vector;
  CONSTANT OutputSignalName  : IN STRING := "";
  CONSTANT PortFlag          : IN VitalPortFlagVectorType;
  CONSTANT OutputMap : IN VitalOutputMapType := VitalDefaultOutputMap;
  VARIABLE ScheduleDataArray : INOUT VitalMemoryScheduleDataVectorType);
PROCEDURE VitalMemorySchedulePathDelay (
  SIGNAL   OutSignal         : OUT std_ulogic;
  CONSTANT OutputSignalName  : IN STRING := "";
  CONSTANT PortFlag          : IN VitalPortFlagType := VitalDefaultPortFlag;
  CONSTANT OutputMap  : IN VitalOutputMapType := VitalDefaultOutputMap;
  VARIABLE ScheduleData      : INOUT VitalMemoryScheduleDataType);
FUNCTION VitalMemoryTimingDataInit RETURN VitalMemoryTimingDataType;
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_ulogic;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     TIME := 0 ns;
  SIGNAL   RefSignal         : IN     std_ulogic;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     TIME := 0 ns;
  CONSTANT SetupHigh         : IN     VitalDelayType;
  CONSTANT SetupLow          : IN     VitalDelayType;
  CONSTANT HoldHigh          : IN     VitalDelayType;
  CONSTANT HoldLow           : IN     VitalDelayType;
  CONSTANT CheckEnabled      : IN     VitalBoolArrayT;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE);
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_ulogic;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     TIME := 0 ns;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
```

```
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE);
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_ulogic;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     TIME := 0 ns;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     VitalBoolArrayT;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE);
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_logic_vector;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     VitalDelayArrayType;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE  );
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01ArrayT;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_logic_vector;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     VitalDelayArrayType;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
```

```
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     VitalBoolArrayT;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE  );
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_ulogic;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     TIME := 0 ns;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE  );
PROCEDURE VitalMemorySetupHoldCheck (
  VARIABLE Violation         : OUT    X01;
  VARIABLE TimingData        : INOUT  VitalMemoryTimingDataType;
  SIGNAL   TestSignal        : IN     std_logic_vector;
  CONSTANT TestSignalName    : IN     STRING := "";
  CONSTANT TestDelay         : IN     VitalDelayArrayType;
  SIGNAL   RefSignal         : IN     std_logic_vector;
  CONSTANT RefSignalName     : IN     STRING := "";
  CONSTANT RefDelay          : IN     VitalDelayArrayType;
  CONSTANT SetupHigh         : IN     VitalDelayArrayType;
  CONSTANT SetupLow          : IN     VitalDelayArrayType;
  CONSTANT HoldHigh          : IN     VitalDelayArrayType;
  CONSTANT HoldLow           : IN     VitalDelayArrayType;
  CONSTANT CheckEnabled      : IN     BOOLEAN := TRUE;
  CONSTANT RefTransition     : IN     VitalEdgeSymbolType;
  CONSTANT HeaderMsg         : IN     STRING := " ";
  CONSTANT XOn               : IN     BOOLEAN := TRUE;
  CONSTANT MsgOn             : IN     BOOLEAN := TRUE;
  CONSTANT MsgSeverity       : IN     SEVERITY_LEVEL := WARNING;
  CONSTANT ArcType           : IN     VitalMemoryArcType := CrossArc;
  CONSTANT NumBitsPerSubWord : IN     INTEGER := 1;
  CONSTANT MsgFormat         : IN     VitalMemoryMsgFormatType;
  CONSTANT EnableSetupOnTest : IN     BOOLEAN := TRUE;
  CONSTANT EnableSetupOnRef  : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnRef   : IN     BOOLEAN := TRUE;
  CONSTANT EnableHoldOnTest  : IN     BOOLEAN := TRUE
PROCEDURE  VitalMemoryPeriodPulseCheck  (
  VARIABLE Violation      : OUT   X01ArrayT;
```

```
  VARIABLE PeriodData     : INOUT VitalPeriodDataArrayType;
  SIGNAL   TestSignal     : IN    std_logic_vector;
  CONSTANT TestSignalName : IN    STRING := "";
  CONSTANT TestDelay      : IN    VitalDelayArrayType;
  CONSTANT Period         : IN    VitalDelayArrayType;
  CONSTANT PulseWidthHigh : IN    VitalDelayArrayType;
  CONSTANT PulseWidthLow  : IN    VitalDelayArrayType;
  CONSTANT CheckEnabled   : IN    BOOLEAN := TRUE;
  CONSTANT HeaderMsg      : IN    STRING := " ";
  CONSTANT XOn            : IN    BOOLEAN := TRUE;
  CONSTANT MsgOn          : IN    BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN    SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat      : IN    VitalMemoryMsgFormatType);
PROCEDURE  VitalMemoryPeriodPulseCheck  (
  VARIABLE Violation      : OUT   X01;
  VARIABLE PeriodData     : INOUT VitalPeriodDataArrayType;
  SIGNAL   TestSignal     : IN    std_logic_vector;
  CONSTANT TestSignalName : IN    STRING := "";
  CONSTANT TestDelay      : IN    VitalDelayArrayType;
  CONSTANT Period         : IN    VitalDelayArrayType;
  CONSTANT PulseWidthHigh : IN    VitalDelayArrayType;
  CONSTANT PulseWidthLow  : IN    VitalDelayArrayType;
  CONSTANT CheckEnabled   : IN    BOOLEAN := TRUE;
  CONSTANT HeaderMsg      : IN    STRING := " ";
  CONSTANT XOn            : IN    BOOLEAN := TRUE;
  CONSTANT MsgOn          : IN    BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN    SEVERITY_LEVEL := WARNING;
  CONSTANT MsgFormat      : IN    VitalMemoryMsgFormatType);
TYPE MemoryWordType        IS ARRAY (NATURAL RANGE <>) OF UX01;
TYPE MemoryWordPtr         IS ACCESS MemoryWordType;
TYPE MemoryArrayType       IS ARRAY (NATURAL RANGE <>) OF MemoryWordPtr;
TYPE MemoryArrayPtrType    IS ACCESS MemoryArrayType;
TYPE VitalMemoryArrayRecType IS RECORD
NoOfWords           : POSITIVE;
NoOfBitsPerWord     : POSITIVE;
NoOfBitsPerSubWord  : POSITIVE;
NoOfBitsPerEnable   : POSITIVE;
MemoryArrayPtr      : MemoryArrayPtrType;
END RECORD;
TYPE VitalMemoryDataType    IS ACCESS VitalMemoryArrayRecType;
TYPE VitalTimingDataVectorType IS
ARRAY (NATURAL RANGE <>) OF VitalTimingDataType;
TYPE VitalMemoryViolFlagSizeType IS ARRAY (NATURAL RANGE <>) OF INTEGER;
TYPE VitalMemorySymbolType IS (
  '/',      -- 0 -> 1
  '\',      -- 1 -> 0
  'P',      -- Union of '/' and '^' (any edge to 1)
  'N',      -- Union of '\' and 'v' (any edge to 0)
  'r',      -- 0 -> X
  'f',      -- 1 -> X
  'p',      -- Union of '/' and 'r' (any edge from 0)
  'n',      -- Union of '\' and 'f' (any edge from 1)
  'R',      -- Union of '^' and 'p' (any possible rising edge)
  'F',      -- Union of 'v' and 'n' (any possible falling edge)
  '^',      -- X -> 1
  'v',      -- X -> 0
  'E',      -- Union of 'v' and '^' (any edge from X)
  'A',      -- Union of 'r' and '^' (rising edge to or from 'X')

  'D',      -- Union of 'f' and 'v' (falling edge to or from 'X')

  '*',      -- Union of 'R' and 'F' (any edge)
  'X',      -- Unknown level
  '0',      -- low level
  '1',      -- high level
  '-',      -- don't care
  'B',      -- 0 or 1
  'Z',      -- High Impedance
```

```
  'S',      -- steady value

  'g',      -- Good address (no transition)
  'u',      -- Unknown address (no transition)
  'i',      -- Invalid address (no transition)
  'G',      -- Good address (with transition)
  'U',      -- Unknown address (with transition)
  'I',      -- Invalid address (with transition)
  'w',      -- Write data to memory
  's',      -- Retain previous memory contents
  'c',      -- Corrupt entire memory with 'X'
  'l',      -- Corrupt a word in memory with 'X'
  'd',      -- Corrupt a single bit in memory with 'X'
  'e',      -- Corrupt a word with 'X' based on data in
  'C',      -- Corrupt a sub-word entire memory with 'X'
  'L',      -- Corrupt a sub-word in memory with 'X'
  'M',      -- Implicit read data from memory
  'm',      -- Read data from memory
  't'       -- Immediate assign/transfer data in
);
TYPE VitalMemoryTableType IS ARRAY ( NATURAL RANGE <>, NATURAL RANGE <> )
  OF VitalMemorySymbolType;

TYPE VitalMemoryViolationSymbolType IS ('X', '0',   '-'   );
TYPE VitalMemoryViolationTableType IS
  ARRAY ( NATURAL RANGE <>, NATURAL RANGE <> )
  OF VitalMemoryViolationSymbolType;
TYPE VitalPortType IS (UNDEF,  READ, WRITE,  RDNWR);
TYPE VitalCrossPortModeType IS (  CpRead, WriteContention, WrContOnly,
ReadWriteContention, CpReadAndWriteContention, CpReadAndReadContention);
SUBTYPE VitalAddressValueType IS INTEGER;
TYPE VitalAddressValueVectorType IS
  ARRAY (NATURAL RANGE <>) OF VitalAddressValueType;
FUNCTION VitalDeclareMemory (
  CONSTANT NoOfWords          : IN POSITIVE;
  CONSTANT NoOfBitsPerWord    : IN POSITIVE;
  CONSTANT NoOfBitsPerSubWord : IN POSITIVE;
  CONSTANT MemoryLoadFile     : IN string := "";
  CONSTANT BinaryLoadFile     : IN BOOLEAN := FALSE
) RETURN VitalMemoryDataType;
FUNCTION VitalDeclareMemory (
  CONSTANT NoOfWords          : IN POSITIVE;
  CONSTANT NoOfBitsPerWord    : IN POSITIVE;
  CONSTANT MemoryLoadFile     : IN string := "";
  CONSTANT BinaryLoadFile     : IN BOOLEAN := FALSE) RETURN VitalMemoryDataType;
PROCEDURE VitalMemoryTable (
  VARIABLE DataOutBus     : INOUT std_logic_vector;
  VARIABLE MemoryData     : INOUT VitalMemoryDataType;
  VARIABLE PrevControls   : INOUT std_logic_vector;
  VARIABLE PrevDataInBus  : INOUT std_logic_vector;
  VARIABLE PrevAddressBus : INOUT std_logic_vector;
  VARIABLE PortFlag       : INOUT VitalPortFlagVectorType;
  CONSTANT Controls       : IN std_logic_vector;
  CONSTANT DataInBus      : IN std_logic_vector;
  CONSTANT AddressBus     : IN std_logic_vector;
  VARIABLE AddressValue   : INOUT VitalAddressValueType;
  CONSTANT MemoryTable    : IN VitalMemoryTableType;
  CONSTANT PortType       : IN VitalPortType := UNDEF;
  CONSTANT PortName       : IN STRING := "";
  CONSTANT HeaderMsg      : IN STRING := "";
  CONSTANT MsgOn          : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING);
PROCEDURE VitalMemoryTable (
  VARIABLE DataOutBus     : INOUT std_logic_vector;
  VARIABLE MemoryData     : INOUT VitalMemoryDataType;
  VARIABLE PrevControls   : INOUT std_logic_vector;
  VARIABLE PrevEnableBus  : INOUT std_logic_vector;
  VARIABLE PrevDataInBus  : INOUT std_logic_vector;
```

```
  VARIABLE PrevAddressBus : INOUT std_logic_vector;
  VARIABLE PortFlagArray  : INOUT VitalPortFlagVectorType;
  CONSTANT Controls       : IN std_logic_vector;
  CONSTANT EnableBus      : IN std_logic_vector;
  CONSTANT DataInBus      : IN std_logic_vector;
  CONSTANT AddressBus     : IN std_logic_vector;
  VARIABLE AddressValue   : INOUT VitalAddressValueType;
  CONSTANT MemoryTable    : IN VitalMemoryTableType;
  CONSTANT PortType       : IN VitalPortType := UNDEF;
  CONSTANT PortName       : IN STRING := "";
  CONSTANT HeaderMsg      : IN STRING := "";
  CONSTANT MsgOn          : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity    : IN SEVERITY_LEVEL := WARNING);
PROCEDURE VitalMemoryCrossPorts (
  VARIABLE DataOutBus             : INOUT std_logic_vector;
  VARIABLE MemoryData             : INOUT VitalMemoryDataType;
  VARIABLE SamePortFlag           : INOUT VitalPortFlagVectorType;
  CONSTANT SamePortAddressValue   : IN VitalAddressValueType;
  CONSTANT CrossPortFlagArray     : IN VitalPortFlagVectorType;
  CONSTANT CrossPortAddressArray  : IN VitalAddressValueVectorType;
  CONSTANT CrossPortMode          : IN VitalCrossPortModeType
                                     := CpReadAndWriteContention;
  CONSTANT PortName               : IN STRING := "";
  CONSTANT HeaderMsg              : IN STRING := "";
  CONSTANT MsgOn                  : IN BOOLEAN := TRUE) ;
PROCEDURE VitalMemoryCrossPorts (
  VARIABLE MemoryData             : INOUT VitalMemoryDataType;
  CONSTANT CrossPortFlagArray     : IN VitalPortFlagVectorType;
  CONSTANT CrossPortAddressArray  : IN VitalAddressValueVectorType;
  CONSTANT HeaderMsg              : IN STRING := "";
  CONSTANT MsgOn                  : IN BOOLEAN := TRUE) ;
PROCEDURE VitalMemoryViolation (
  VARIABLE DataOutBus           : INOUT std_logic_vector;
  VARIABLE MemoryData           : INOUT VitalMemoryDataType;
  VARIABLE PortFlag             : INOUT VitalPortFlagVectorType;
  CONSTANT DataInBus            : IN std_logic_vector;
  CONSTANT AddressValue         : IN VitalAddressValueType;
  CONSTANT ViolationFlags       : IN std_logic_vector;
  CONSTANT ViolationFlagsArray  : IN X01ArrayT;
  CONSTANT ViolationSizesArray  : IN VitalMemoryViolFlagSizeType;
  CONSTANT ViolationTable       : IN VitalMemoryTableType;
  CONSTANT PortType             : IN VitalPortType;
  CONSTANT PortName             : IN STRING := "";
  CONSTANT HeaderMsg            : IN STRING := "";
  CONSTANT MsgOn                : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity          : IN SEVERITY_LEVEL := WARNING) ;
PROCEDURE VitalMemoryViolation (
  VARIABLE DataOutBus           : INOUT std_logic_vector;
  VARIABLE MemoryData           : INOUT VitalMemoryDataType;
  VARIABLE PortFlag             : INOUT VitalPortFlagVectorType;
  CONSTANT DataInBus            : IN std_logic_vector;
  CONSTANT AddressValue         : IN VitalAddressValueType;
  CONSTANT ViolationFlags       : IN std_logic_vector;
  CONSTANT ViolationTable       : IN VitalMemoryTableType;
  CONSTANT PortType             : IN VitalPortType;
  CONSTANT PortName             : IN STRING := "";
  CONSTANT HeaderMsg            : IN STRING := "";
  CONSTANT MsgOn                : IN BOOLEAN := TRUE;
  CONSTANT MsgSeverity          : IN SEVERITY_LEVEL := WARNING
) ;
END Vital_Memory;
```

12.4 Analogique

12.4.1 [AMS]IEEE.FUNDAMENTAL_CONSTANTS

```
library IEEE; use IEEE.MATH_REAL.all;
package FUNDAMENTAL_CONSTANTS is
-- some declarations
  attribute SYMBOL : STRING;
  attribute UNIT   : STRING;
-- PHYSICAL CONSTANTS
   -- electron charge <COULOMB>
   -- NIST value: 1.602 176 462e-19 coulomb
   -- uncertainty: 0.000 000 063e-19
   constant PHYS_Q : REAL :=  1.602_176_462e-19;
   -- permittivity of vacuum <FARADS/METER>
   -- NIST value: 8.854 187 817e-12 farads/meter
   -- uncertainty: exact
   constant PHYS_EPS0 : REAL := 8.854_187_817e-12;
   -- permeability of vacuum <HENRIES/METER>
   -- NIST value: 4e-7*pi henries/meter
   -- uncertainty: exact
   constant PHYS_MU0 : REAL := 4.0e-7 * MATH_PI;
   -- Boltzmann's constant <JOULES/KELVIN>
   -- NIST value: 1.380 6503e-23 joules/kelvin
   -- uncertainty: 0.000 0024e-23:
   constant PHYS_K : REAL :=  1.380_6503e-23;
   -- Acceleration due to gravity <METERS/SECOND_SQUARED>
   -- NIST value: 9.806 65 meters/square second
   -- uncertainty: exact
   constant PHYS_GRAVITY : REAL := 9.806_65;
   -- Conversion between Kelvin and degree Celsius
   -- NIST value: 273.15
   -- uncertainty: exact
   constant PHYS_CTOK : REAL := 273.15;
   -- object declarations
   -- common scaling factors
   constant  YOCTO : REAL := 1.0e-24;
   constant  ZEPTO : REAL := 1.0e-21;
   constant  ATTO  : REAL := 1.0e-18;
   constant  FEMTO : REAL := 1.0e-15;
   constant  PICO  : REAL := 1.0e-12;
   constant  NANO  : REAL := 1.0e-9;
   constant  MICRO : REAL := 1.0e-6;
   constant  MILLI : REAL := 1.0e-3;
   constant  CENTI : REAL := 1.0e-2;
   constant  DECI  : REAL := 1.0e-1;
   constant  DEKA  : REAL := 1.0e+1;
   constant  HECTO : REAL := 1.0e+2;
   constant  KILO  : REAL := 1.0e+3;
   constant  MEGA  : REAL := 1.0e+6;
   constant  GIGA  : REAL := 1.0e+9;
   constant  TERA  : REAL := 1.0e+12;
   constant  PETA  : REAL := 1.0e+15;
   constant  EXA   : REAL := 1.0e+18;
   constant  ZETTA : REAL := 1.0e+21;
   constant  YOTTA : REAL := 1.0e+24;
   alias DECA is DEKA;
end package FUNDAMENTAL_CONSTANTS;
```

12.4.2 [AMS] DISCIPLINES.ELECTRICAL_SYSTEMS

```
library IEEE;
    use IEEE.FUNDAMENTAL_CONSTANTS.all;
package ELECTRICAL_SYSTEMS is
   subtype VOLTAGE     is REAL tolerance "DEFAULT_VOLTAGE";
   subtype CURRENT     is REAL tolerance "DEFAULT_CURRENT";
   subtype CHARGE      is REAL tolerance "DEFAULT_CHARGE";
   subtype RESISTANCE  is REAL tolerance "DEFAULT_RESISTANCE";
   subtype CAPACITANCE is REAL tolerance "DEFAULT_CAPACITANCE";
   subtype MMF         is REAL tolerance "DEFAULT_MMF";
   subtype FLUX        is REAL tolerance "DEFAULT_FLUX";
   subtype INDUCTANCE  is REAL tolerance "DEFAULT_INDUCTANCE";
   -- attribute declarations
   -- Use of UNIT to designate units
   attribute UNIT of VOLTAGE       : subtype is "Volt";
   attribute UNIT of CURRENT       : subtype is "Ampere";
   attribute UNIT of CHARGE        : subtype is "Coulomb";
   attribute UNIT of RESISTANCE    : subtype is "Ohm";
   attribute UNIT of CAPACITANCE   : subtype is "Farad";

   attribute UNIT of MMF           : subtype is "Ampere-turns";
   attribute UNIT of FLUX          : subtype is "Weber";
   attribute UNIT of INDUCTANCE    : subtype is "Henry";
   -- Use of SYMBOL to designate abbreviation of units
   attribute SYMBOL of VOLTAGE     : subtype is "V";
   attribute SYMBOL of CURRENT     : subtype is "A";
   attribute SYMBOL of CHARGE      : subtype is "C";
   attribute SYMBOL of RESISTANCE  : subtype is "Ohm";
   attribute SYMBOL of CAPACITANCE : subtype is "F";
   attribute SYMBOL of MMF         : subtype is "A-t";
   attribute SYMBOL of FLUX        : subtype is "W";
   attribute SYMBOL of INDUCTANCE  : subtype is "H";
   -- nature declarations
   nature ELECTRICAL is VOLTAGE across
                        CURRENT  through
                        ELECTRICAL_REF reference;
   nature ELECTRICAL_VECTOR is array (NATURAL range <>) of ELECTRICAL;
   -- vector subtypes for array types
   nature MAGNETIC is MMF  across
                      FLUX  through
                      MAGNETIC_REF reference;
   nature MAGNETIC_VECTOR is array (NATURAL range <>) of MAGNETIC;
   -- vector subtype declarations
   subtype VOLTAGE_VECTOR    is ELECTRICAL_VECTOR'across;
   subtype CURRENT_VECTOR    is ELECTRICAL_VECTOR'through;
   subtype CHARGE_VECTOR     is REAL_VECTOR tolerance "DEFAULT_CHARGE";
   subtype RESISTANCE_VECTOR is REAL_VECTOR tolerance "DEFAULT_RESISTANCE";
   subtype MMF_VECTOR        is MAGNETIC_VECTOR'across;
   subtype FLUX_VECTOR       is MAGNETIC_VECTOR'through;
   subtype INDUCTANCE_VECTOR is REAL_VECTOR tolerance "DEFAULT_INDUCTANCE";
   -- attributes of vector subtypes
   -- Use of UNIT to designate units
   attribute UNIT of VOLTAGE_VECTOR       : subtype is "Volt";
   attribute UNIT of CURRENT_VECTOR       : subtype is "Ampere";
   attribute UNIT of CHARGE_VECTOR        : subtype is "Coulomb";
   attribute UNIT of RESISTANCE_VECTOR    : subtype is "Ohm";
   attribute UNIT of CAPACITANCE_VECTOR   : subtype is "Farad";
```

```
   attribute UNIT of MMF_VECTOR           : subtype is "Ampere-turns";
   attribute UNIT of FLUX_VECTOR          : subtype is "Weber";
   attribute UNIT of INDUCTANCE_VECTOR    : subtype is "Henry";
   -- Use of SYMBOL to designate abbreviation of units
   attribute SYMBOL of VOLTAGE_VECTOR     : subtype is "V";
   attribute SYMBOL of CURRENT_VECTOR     : subtype is "A";
   attribute SYMBOL of CHARGE_VECTOR      : subtype is "C";
   attribute SYMBOL of RESISTANCE_VECTOR  : subtype is "Ohm";
   attribute SYMBOL of CAPACITANCE_VECTOR : subtype is "F";
   attribute SYMBOL of MMF_VECTOR         : subtype is "A-t";
   attribute SYMBOL of FLUX_VECTOR        : subtype is "W";
   attribute SYMBOL of INDUCTANCE_VECTOR  : subtype is "H";
   alias GROUND is ELECTRICAL_REF;
end package ELECTRICAL_SYSTEMS;
```

13 Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

14 Tables des figures

15 Bibliographie

- [AIR] VHDL: Langage, Modélisation, Synthèse
  Roland Airiau, Jean-Michel Bergé, Vincent Olive, Jacques Rouillard,
  Presses Polytechniques Universitaires Romandes
  ISBN 2-88074-361-3

- [ASH1] The VHDL Cookbook (anglais), une bonne et brève introduction à VHDL'87
  Peter Ashenden
  http://tams-www.informatik.uni-hamburg.de/vhdl/ ……
  …… doc/cookbook/VHDL-Cookbook.pdf

- [ASH2] VHDL-2008, Just the new stuff
  Peter Ashenden
  Morgan Kauffman
  ISBN 978-0-12-374249-0

- [BER] VHDL Designer's Reference (anglais)
  Jean-Michel Bergé, Alain Fonkoua, Serge Maginot, Jacques Rouillard
  Kluwer Academic Publishers
  ISBN 0-7923-1756-4

- [CHR] Analog & Mixed-Signal extensions to VHDL (anglais)
  Ernst Christen, Kenneth Bakalar, in CIEM 5 Current Issues in Electronic Modeling
  "Analog & Mixed-Signal Hardware Description Languages",
  edited by *Alain Vachoux, Jean-Michel Bergé, Oz Levia, Jacques Rouillard*
  Kluwer Academic Publishers
  ISBN 0-7923-9875-0

- [COE] The VHDL Handbook
  David R. Coelho
  Kluwer Academic Publishers
  ISBN 0-7923-9031-8

- [HAM] Les archives VHDL de l'université de Hambourg
  http://tams-www.informatik.uni-hamburg.de/research/vlsi/vhdl/

- [HER] VHDL-AMS Applications et enjeux industriels
  Yannick Hervé, Dunod
  ISBN 9-782100-0058884

- [IEEE] Publications IEEE : Standards (anglais)
  - [IE1] 1076-2002 IEEE Standard **VHDL** Language Reference Manual
    Publication IEEE
    E-ISBN: 0-7381-3248-9
    ISBN: 0-7381-3247-0
  - [IE2] 1076.1-2007 IEEE Standard **VHDL Analog** and Mixed-Signal Extensions
    Publication IEEE
    E-ISBN: 0-7381-5628-0
    ISBN: 0-7381-5627-2
  - [IE3] 1076.4-2000 IEEE standard for **VITAL** ASIC modeling specification
    Publication IEEE
    E-ISBN: 0-7381-2692-4
    ISBN: 0-7381-2691-0
  - [IE4] 1497 IEEE Standard Delay Format (**SDF**) for the electronic design process
    Publication IEEE
    E-ISBN: 0-7381-3075-3
    ISBN: 0-7381-3074-5
  - [IE5] 1029.1-1998 IEEE Standard for Vhdl Waveform and Vector Exchange to Support Design and Test Verification (**WAVES**) Language Reference Manual
    Publication IEEE
    ISBN: 0-7381-1445-6
    E-ISBN: 0-7381-1446-4

- [MUN] ASIC and FPGA verification: a guide to component modeling (anglais)
  Une excellente introduction à VITAL
  Richard Munden, Morgan Kaufmann Publishers
  ISBN: 0-12-510581-9
  Version pdf accessible: http://www.muco17.com/fpga/FPGA.pdf

- [ROU1] Lire & Comprendre VHDL
  Jacques Rouillard, Lulu.com
  ISBN 978-1-4092-2787-8
  Version pdf accessible : http://www.rouillard.org/lire-vhdl-et-ams.pdf

- [ROU2] Aide Mémoire VHDL, une version pdf de celui qui est à la fin de ce manuel.
  Jacques Rouillard
  http://www.rouillard.org/memovhdl.pdf

- [TUT] L'excellent tutorial sur VHDL-AMS sur le site de référence [HAM] (anglais):
  Ernst Christen, Kenneth Bakalar, Allen M. Dewey, Eduard Moser
  http://tams-www.informatik.uni-hamburg.de/vhdl/doc/P1076.1/tutdac99.pdf

- [VAC] VHDL(-AMS) Essentiel
  Alain Vachoux
  http://lsmwww.epfl.ch/Education/former/ ….
  ….. 2002-2003/modelmix03/Documents/VHDLAMS_instant.pdf

Aide-Mémoire

Cet aide-mémoire peut être découpé ou photocopié, et agrafé. Une version électronique pdf se trouve sur http://rouillard.org/memovhdl.pdf

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`

```
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'
```

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;`

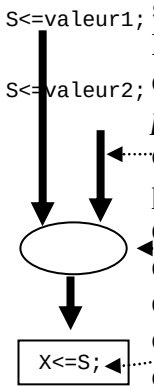
[AMS] Natures: *voir page 6.*

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 »[14] et c'est cette résolution qui devient la valeur observable[15]. 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.*

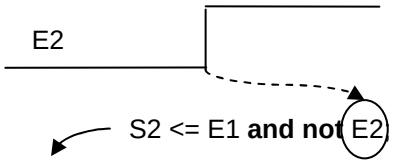
[14] C'est là que les signaux forts l'emportent sur les faibles ou que les conflits sont détectés. Voir le paquetage STD_LOGIC_1164, section **paquetages standards.**

[15] 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.*

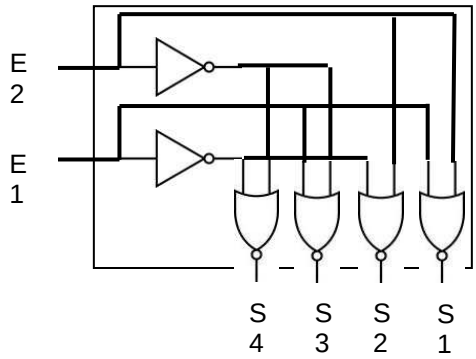
```
library IEEE ;use IEEE.std_logic_1164.all ;
entity decodeur is
      port (E1,E2 : in std_logic ; S1,S2,S3,S4 : out std_logic ) ;
end decodeur;

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 :



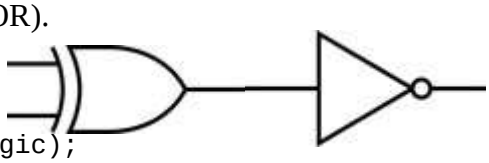

```
library lib4212;use lib4212.delaypkg.all;
architecture structure of decodeur is
  component decodeur
      port (E1,E2 : in std_logic ; S1,S2,S3,S4 : out
std_logic ) ;
  end component;
  signal Ss1,Sz2,Ss3,Ss4: std_logic;
  for dec: decodeur use entity lib4212.deco2to4(bla666);
begin
dec :decodeur 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).



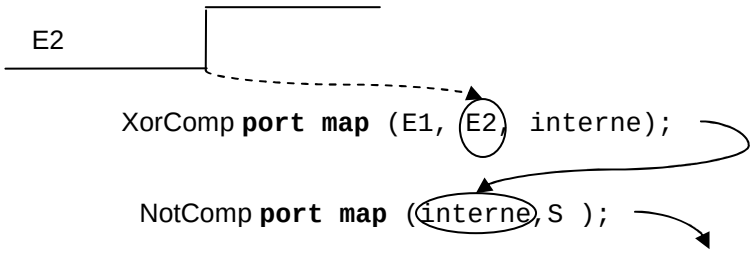
```
library IEEE ;use IEEE.std_logic_1164.all ;
entity XnorGate is
 port (E1,E2 : in std_logic ; S :out std_logic);
end XnorGate;
```

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;[16]
 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.

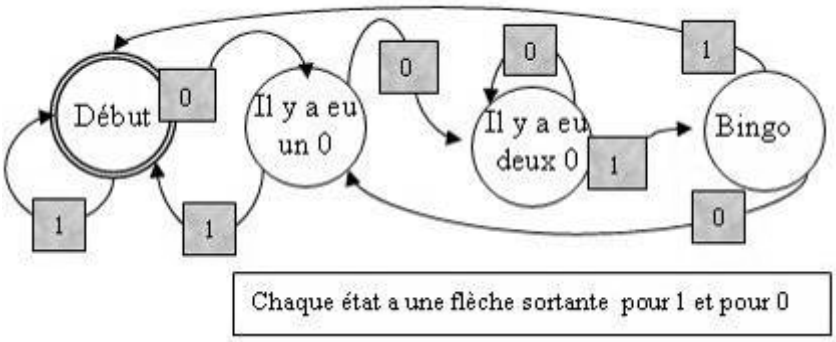
[16] La configuration peut être simple comme ici (simplement renvoyer à une entité existante) ou encore spécifier de quelle architecture on parle s'il y en a plusieurs, et enfin dire explicitement quel port du composant correspond à quel port de l'entité. Une forme plus complète (hors généricité) est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

```
for ref-des-composants: nom-du-composant
                        use entity ressources_lib.entité(architecture)
                        port map (port-du-composant => port-de-l'entité,…)[16]
```

`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 »:

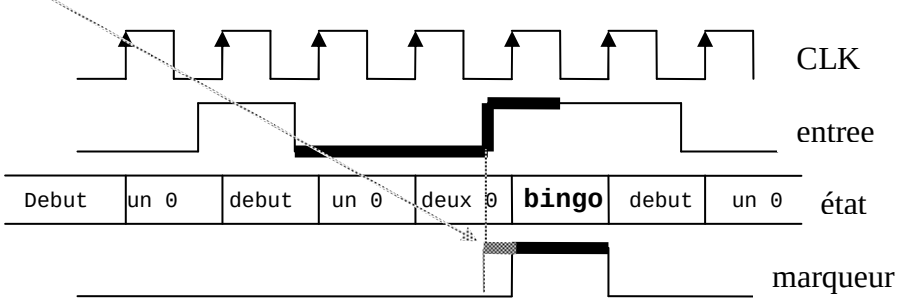
```
type type_etat is (debut,
il_y_a_eu_un_0,
il_y_a_eu_deux_0, bingo) ;
signal etat : type_etat :=
debut ;
```



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 ;
```

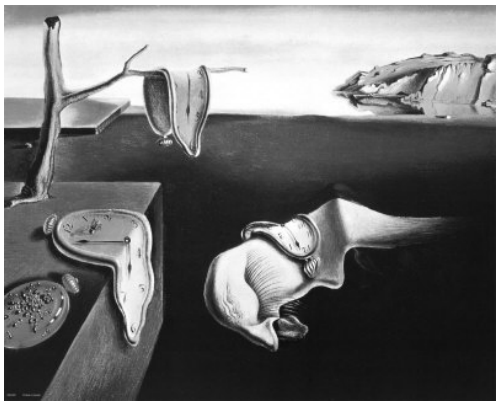


Pour la Simulation:
À chaque front d'horloge, l'entrée est lue par la branche correspondante du case; le marqueur est mis à 1 ou 0, et l'état suivant est calculé.

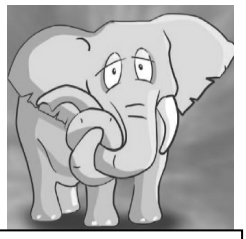
Pour la Synthèse : ici une synthèse « à la main » et en VHDL, sans aucune optimisation. L'état va être codé (ici sur 2 bits, le codage est optimisable). 00=début, 01= etc.

```
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 165. 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 :

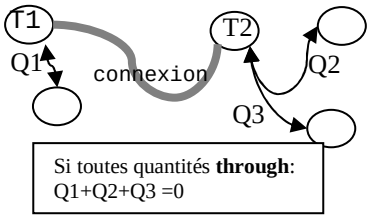
- Beaucoup de constructions sont par essence non synthétisables : fichiers, accès, procédures récursives…La première passe du synthétiseur sert surtout à corriger le VHDL pour le rendre synthétisable.
- Il ne faut pas chercher à synthétiser des blocs réguliers (RAM, etc.) ; il faut appeler des blocs 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.*

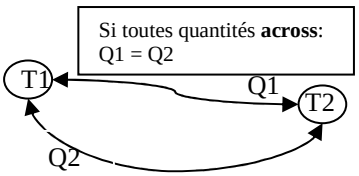
```
Principaux attributs usuels:
S'RAMP S'SLEW rendent des rampes définies par temps ou par pente.
Q'DOT : derivée / temps
Q'INTEG: intégrale depuis 0
Q'DELAYED(T): Q décalé de T
T'REFERENCE : quantité across entre T et sa référence
T'CONTRIBUTION: quantité through somme des quantité through incidentes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

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

```
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 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];
```

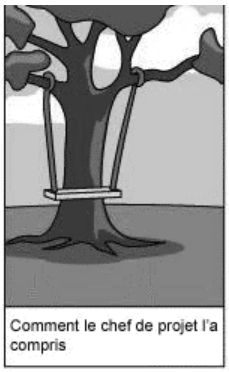
break on instruction concurrente avec équivalence séquentielle, indispensable pour dire au noyau qu'une discontinuité nécessite un recalcul des conditions "initiales".

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

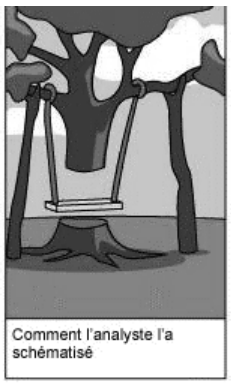
Condition de solvabilité: La somme des quantités libres, des quantités de mode **out** et des quantités **through** d'une unité doit être égale au nombre d'équations (ou équations équivalentes). Ceci est une condition nécessaire, mais ne garantit évidemment pas la convergence.

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`.

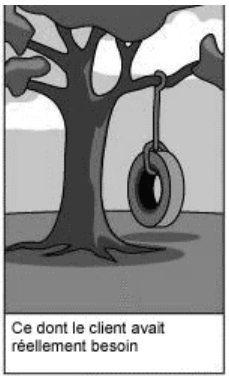


- **Clause use** : on peut « factoriser » un préfixe trop encombrant à écrire, sous réserve que cela ne crée pas d'ambiguïtés. La clause « **`use`** `prefixe.`**`all`** `;` » peut être mise n'importe où dans une zone déclarative. En dessous, les objets déjà visibles (et seulement ceux-là) par la notation `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 165. 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

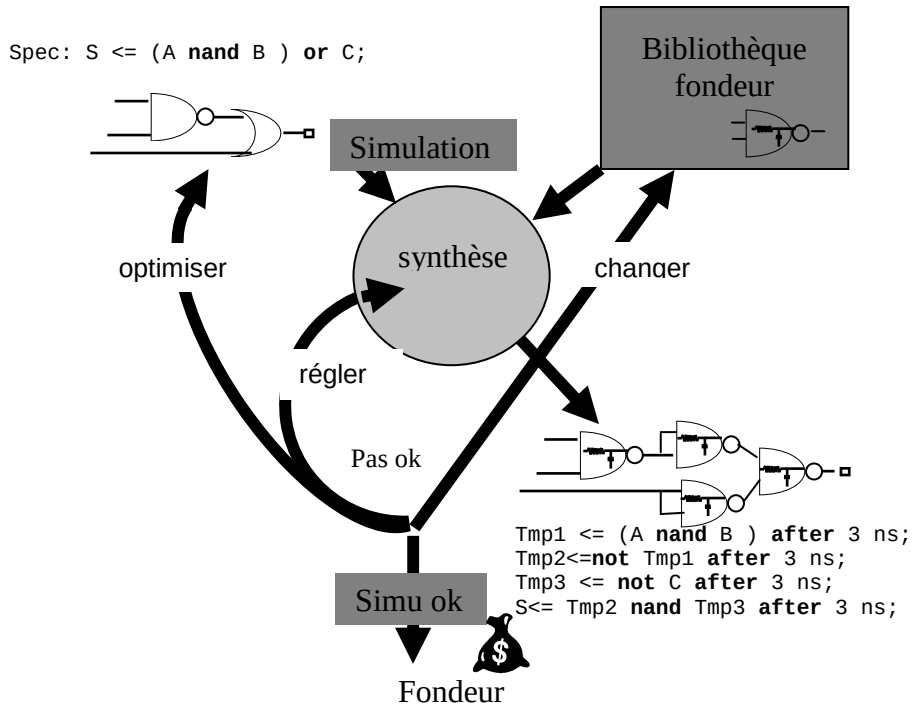
- **IEEE.STD_LOGIC_ARITH :** permet de faire de l'arithmétique sur `std_logic_vector` Pour cela il fournit deux types `signed` et `unsigned` qui ont la même définition que `std_logic_vector` : « `array (natural range <>) of std_logic` ». Suivant la définition des compatibilités de types en VHDL, tout `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 bien configuré
  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".

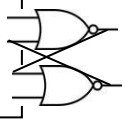
Exemples de code en vrac : (portions significatives du code seulement.)

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_ent
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)