



# Modélisation de Systèmes Numériques Intégrés Introduction à VHDL

Alain Vachoux  
Xemics S.A.  
[alain.vachoux@xemics.ch](mailto:alain.vachoux@xemics.ch)

Notes de cours à option 2ème cycle  
Hiver 1998-1999



# Table des matières

<b>1. Introduction</b>	<b>7</b>
1.1. Notion de modèle	9
1.2. Représentation de la conception	11
1.3. Langage de description de matériel	13
1.4. Exemple de description comportementale, structurelle et géométrique	15
1.5. Simulation logique	17
1.6. Synthèse logique	19
<b>2. VHDL</b>	<b>21</b>
2.1. Flot de conception basé sur VHDL	23
2.2. Environnement de travail VHDL	25
2.3. Modèle sémantique de VHDL	27
2.4. Organisation d'un modèle VHDL	29
Unités de conception	29
Entité de conception	29
2.5. Modèles VHDL d'un registre 4 bits	31
Déclaration d'entité	31
Architecture comportementale	31
Architecture structurelle	33
Environnement de test	35
Déclaration de configuration	37
Instanciation directe	39
Spécification de configuration	39
Remarque	39
2.6. Modèles génériques	41
Paramètres génériques	41
Objets d'interface non contraints	43
Instruction generate	45
2.7. Concurrence et modélisation du temps	47
Processus et signaux	47
Initialisation et cycle de simulation	49
Délai delta	51
Délai inertiel	53
Délai transport	53
Transactions multiples	55
Signaux résolus	57
2.8. VHDL pour la synthèse	59
Types (et sous-types) supportés	59
Objets supportés	61
Valeurs initiales	65
Opérateurs	65
Groupement d'opérateurs	67
Partage de ressources	67
Utilisation de processus et de l'instruction wait	71
Instructions séquentielles supportées	73

Instructions concurrentes supportées .....	79
Paramètres génériques .....	79
<b>3. Modélisation de circuits numériques .....</b>	<b>81</b>
3.1. Circuits logiques combinatoires .....	81
Equations logiques .....	81
Multiplexeurs .....	85
Encodeurs .....	87
Décodeurs .....	90
Comparateurs .....	91
Unité arithmétique et logique (ALU) .....	93
3.2. Circuits logiques séquentiels .....	95
Latches .....	95
Flip-flops .....	96
Registres .....	100
Compteurs .....	103
3.3. Circuits avec état haute-impédance .....	105
3.4. Machines à états finis .....	111
Table des états et diagramme d'états .....	111
Structure d'une machine à états finis .....	112
Modélisation avec un processus et registre d'état explicite .....	113
Modélisation avec trois processus et registre d'état explicite .....	116
Modélisation avec registre d'état implicite .....	118
Reset synchrone/asynchrone et comportement sain .....	119
Encodage des états .....	120
Variantes possibles de machines d'états .....	123
3.5. Opérateurs arithmétiques .....	126
Additionneurs/Soustracteurs .....	126
Multiplieurs .....	137
<b>Annexe A: Syntaxe VHDL .....</b>	<b>A-1</b>
A.1. Format de description de la syntaxe .....	A-1
A.2. Eléments de base .....	A-1
A.2.1 Identificateurs et mots réservés .....	A-1
A.2.2 Littéraux caractère .....	A-2
A.2.3 Littéraux chaînes de caractères et chaînes de bits .....	A-3
A.2.4 Littéraux numériques .....	A-3
A.2.5 Agrégats .....	A-4
A.2.6 Commentaires .....	A-4
A.2.7 Types et sous-types .....	A-5
Types numériques .....	A-5
Types énumérés .....	A-6
Types physiques .....	A-6
Types tableaux .....	A-7
Types enregistrements .....	A-8
Types accès .....	A-8
Types fichiers .....	A-9
A.2.8 Objets .....	A-9
Constantes .....	A-9
Variables .....	A-10
Fichiers .....	A-10

Signaux .....	A-11
A.2.9 Attributs .....	A-12
A.2.10 Alias .....	A-13
A.2.11 Expressions et opérateurs .....	A-14
A.3. Unités de conception .....	A-16
A.3.1 Clause de contexte .....	A-16
A.3.2 Déclaration d'entité .....	A-17
A.3.3 Corps d'architecture .....	A-17
A.3.4 Déclaration de paquetage .....	A-18
A.3.5 Corps de paquetage .....	A-18
A.3.6 Déclaration de configuration .....	A-19
A.3.7 Déclarations d'interfaces .....	A-20
A.3.8 Déclaration de composant .....	A-21
A.3.9 Association .....	A-21
A.4. Instructions concurrentes .....	A-21
A.4.1 Processus .....	A-21
A.4.2 Assignation concurrente de signal .....	A-22
Assignation concurrente simple .....	A-22
Assignation concurrente conditionnelle .....	A-23
Assignation concurrente sélective .....	A-24
A.4.3 Instance de composant .....	A-25
A.4.4 Génération d'instructions .....	A-25
A.5. Instructions séquentielles .....	A-26
A.5.1 Assignation de signal .....	A-26
A.5.2 Assignation de variable .....	A-26
A.5.3 Instruction conditionnelle .....	A-26
A.5.4 Instruction sélective .....	A-27
A.5.5 Instructions de boucle .....	A-28
A.5.6 Instruction wait .....	A-29
A.6. Sous-programmes .....	A-30
A.6.1 Procédure .....	A-30
A.6.2 Fonction .....	A-31
A.6.3 Surcharge .....	A-32
A.6.4 Sous-programmes dans un paquetage .....	A-32
<b>Annexe B: Paquetages VHDL standard .....</b>	<b>B-1</b>
B.1. Paquetage STANDARD .....	B-1
B.2. Paquetage TEXTIO .....	B-2
B.3. Paquetage STD_LOGIC_1164 .....	B-4
<b>Annexe C: Autres paquetages VHDL .....</b>	<b>C-1</b>
C.1. Paquetage STD_LOGIC_ARITH .....	C-1
<b>Annexe D: LFSR .....</b>	<b>D-1</b>
<b>Références .....</b>	<b>R-1</b>



# 1. Introduction

L'évolution constante des techniques de fabrication permet d'envisager la conception de systèmes matériels (*hardware systems*) de plus en plus complexes. Un système qui occupait il y a peu une ou plusieurs cartes (circuits imprimés, *printed circuit board* (PCB)) peut désormais tenir sur une seule puce de silicium (circuit intégré, *integrated circuit* (IC)). Le processus de conception doit donc faire face à une quantité croissante d'informations devenant elles-mêmes plus complexes. D'un autre côté, le marché impose un temps de conception (et de fabrication) le plus court possible de manière à réagir rapidement à la demande. Ceci impose dès lors un certain nombre de contraintes sur le cycle de conception:

- Une exploration efficace de l'espace des solutions. Il s'agit de prendre les bonnes décisions le plus tôt possible.
- Une réutilisation optimum de l'expertise acquise. Il s'agit d'éviter de repartir systématiquement de zéro à chaque fois.
- La possibilité de changer facilement de style de conception (*design style*) et/ou de technologie tout en garantissant une nouvelle version optimisée du système. Par exemple, une réalisation sur FPGA (*Field Programmable Gate Array*) peut constituer la version prototype, alors que la version finale sera réalisée sous la forme d'un ASIC (*Application Specific Integrated Circuit*).

L'utilisation d'outils logiciels d'aide à la conception devient ainsi essentielle pour satisfaire les contraintes ci-dessus, principalement grâce à leur faculté de traiter un très grand nombre d'information et d'automatiser certaines tâches de routine ou répétitives. Parmi ceux-ci, les outils EDA (*Electronic Design Automation*) tentent plus particulièrement de prendre en charge des activités de synthèse pour lesquelles le concepteur fournit une description relativement abstraite des spécifications du système (comportement désiré et contraintes à satisfaire) et laisse l'outil lui proposer des solutions sous la forme de structures possibles (architectures, schéma logique, schéma électrique, *layout*). La validation des descriptions initiales et des solutions trouvées se fait alors par simulation ou par preuve formelle.

Le processus de conception passe ainsi par un certain nombre d'étapes, chacune d'elles nécessitant une description de l'état du système sous forme graphique (diagrammes, schémas, etc.) ou textuelle (algorithmes, liste de pièces et connectivité (*netlist*), etc.). Ces descriptions peuvent être fournies par le concepteur ou produites par des outils logiciels. On peut distinguer deux types de descriptions:

- Les descriptions qui ne sont destinées à être lues et comprises que par des outils logiciels. On parle dans ce cas de **formats d'échange** (*interchange format*). On trouve par exemple dans cette catégorie les formats CIF et GDSII pour le layout et EDIF pour le schéma et le layout.
- La seconde famille regroupe les descriptions destinées à être en plus lues et comprises par des concepteurs. On parle alors de **langages de description de matériel** (*hardware description language*, HDL) et le langage VHDL en est un exemple.

## Notion de modèle

- **Support pour:**
  - **Analyse: extraction de propriétés**
  - **Synthèse: dérivation d'une description plus détaillée et éventuellement optimisée**
  
- **Nécessité d'un modèle:**
  - **Manque d'information**
  - **Surplus d'information**
  
- **Mode de fonctionnement d'un système matériel:**
  - **Logique, dirigé par événements**
  - **Analogique, continu**
  - **Mixte**
  
- **Technique de modélisation:**
  - **Discrète**
  - **Continue**
  - **Mixte**



## 1.1. Notion de modèle

La modélisation est une activité essentielle dans le processus de conception d'un système matériel, car elle est à la fois le support pour l'analyse et la synthèse.

L'*analyse* d'un système matériel consiste à extraire un certain nombre de propriétés à partir d'une description du système. L'analyse peut être faite par simulation, c'est-à-dire que la description est soumise à un ensemble de stimuli et le programme de simulation calcule l'état résultant du modèle (p. ex. une simulation logique temporelle). L'analyse peut aussi être faite par examen des propriétés du système sans qu'il soit nécessaire de lui appliquer des stimuli (p. ex. le calcul des délais des chemins d'entrée-sortie ou la vérification qu'une connexion de transistors ne possède pas de court-circuits ou de noeuds non connectés).

La *synthèse* d'un système matériel consiste à transformer une description de départ en une nouvelle description plus détaillée et éventuellement optimisée pour tenir compte de contraintes imposées (p. ex. surface minimum, délais minimum). Par exemple, partir d'un algorithme décrivant la fonction du système et en dériver un circuit logique capable de réaliser cette fonction tout en garantissant un temps de propagation donné.

Un modèle est une description abstraite d'un ensemble de phénomènes physiques: seuls les paramètres nécessaires à la tâche d'analyse ou de synthèse sont pris en compte. La nécessité de travailler avec une description abstraite peut s'expliquer par deux raisons mutuellement exclusives:

- **Manque d'information:** le système est en cours de conception et tous ses paramètres ne sont pas encore connus. Par exemple, l'algorithme d'une fonction à réaliser définit à la fois la séquence des opérations et les opérations elles-mêmes à effectuer, mais pas l'architecture du circuit ou le circuit lui-même qui réalisera la fonction donnée.
- **Surplus d'information:** l'analyse d'un système peut devenir fastidieuse, voire impossible, si la description est trop complexe. Il n'est par exemple pas réaliste d'analyser le comportement d'un microprocesseur si ce dernier est décrit au niveau des portes logiques et des transistors qui le composent et s'il ne s'agit que de vérifier son jeu d'instructions.

Un système matériel est usuellement caractérisé comme ayant un fonctionnement *logique* (*digital*) ou *analogique*. Un fonctionnement logique, ou discret, est défini par un nombre fini d'états possibles et d'actions (événements, *event-driven behaviour*) qui permettent d'atteindre ces états. Les états sont des grandeurs logiques (quantifiées) ou continues seulement définies à des instants particuliers. Un fonctionnement analogique, ou continu, est défini par un ensemble fini d'équations (différentielles, algébriques, linéaires ou non-linéaires) dont la solution est un ensemble de formes d'ondes (*waveforms*) continues fonction d'une variable indépendante (le temps ou la fréquence). Un système matériel peut également exhiber un comportement mixte logique-analogique.

La *technique de modélisation* peut être quant-à-elle discrète ou continue, ou un mélange des deux, selon le genre d'information que l'on désire obtenir par l'analyse et ceci indépendamment du *mode de fonctionnement* logique ou analogique du système modélisé. Par exemple, il est concevable d'analyser un circuit logique en considérant sa description équivalente au niveau du transistor pour en extraire des valeurs précises de délais. Il est aussi concevable de décrire le comportement d'un comparateur analogique comme un modèle à deux états dirigé par un événement qui est le changement de tension à l'entrée. Il est finalement naturel de décrire le comportement d'un convertisseur analogique-logique ou logique-analogique comme une combinaison de fonctionnements dirigés par événements et continus. Le choix de la bonne technique de modélisation est aussi conditionné par les algorithmes mis en oeuvre par le simulateur utilisé.

## Représentation de la conception

Niveaux d'abstraction	Domaines de description		
	Comportement	Structure	Géométrie
<b>Système/ Architecture</b>	Spécification de performances	Connection topologique de processeurs, mémoires, bus	Partition en chips, modules, cartes, sous-systèmes
<b>Algorithmique</b>	Description impérative	Structures de données, décomposition en procédures	????
<b>Transfert de registres (RTL)</b>	Comportement concurrent (déclaratif), transfert de données entre registres	Connection topologique de blocs fonctionnels (ALU, mémoires, multiplexeurs, registres)	Partition et placement de blocs fonctionnels ( <i>floorplan</i> )
<b>Logique, porte</b>	Equations booléennes	Connection topologique de portes	Placement et routage de cellules
<b>Interrupteur</b>	Equations discrètes	Connection topologique de transistors	Placement et routage de transistors
<b>Electrique</b>	Equations différentielles non linéaires	Connection topologique d'éléments électriques (transistor, capacité, résistance, sources, etc.)	Placement et routage d'éléments électriques de base ( <i>layout</i> )

**Table 1:** Niveaux d'abstraction et domaines de description

## 1.2. Représentation de la conception

Le processus d'abstraction consiste à ne garder que les informations nécessaires et d'ignorer *délibérément* les détails inutiles. Il permet ainsi de diminuer la quantité d'information à manipuler. On définit plusieurs *niveaux d'abstraction* (*abstraction levels*), chacun d'eux étant caractérisé par un ensemble de composants de base (primitives) et par la nature des informations qu'il considère:

- Le *niveau électrique* (*electrical level*) est le niveau le plus détaillé. Les composants de base sont les éléments électriques traditionnels (transistors, capacité, résistance, sources, etc.) dont les comportements sont représentés par des équations mathématiques impliquant des fonctions du temps ou de la fréquence.
- Le *niveau interrupteur* (*switch level*) ne considère que des transistors modélisés par des interrupteurs plus ou moins idéaux. L'information est constituée par des paires de valeurs discrètes: un niveau logique (abstraction d'une certaine tension) et une force (abstraction d'une certaine impédance). A partir de ce niveau, seule une description temporelle reste possible. Le temps peut être une valeur continue ou discrète.
- Le *niveau logique* ou *porte* (*gate level*) est basé sur l'algèbre de Boole avec quelques extensions pour introduire des aspects temporels (délais). La correspondance entre équations booléennes et portes logiques est immédiate. L'information est quantifiée sous la forme de valeurs 0 et 1, ou sous forme multi-valuée (0, 1, X, Z, ...).
- Le niveau *transfert de registres* (*register transfer level, RTL*) est une représentation synchrone du système décomposé en une partie de contrôle et une partie opérative travaillant de manière concurrente. Les composants de base sont des modules logiques complexes, tels que ALU, multiplexeur, décodeur, registre, etc. L'information est constituée par des bits et des mots et le temps est réduit à des coups d'horloge.
- Le niveau *algorithmique* (*algorithmic level*) voit le système comme un programme constitué d'une séquence d'opérations. L'information peut être de n'importe quel type et le temps peut être explicite (réduit à des coups d'horloge) ou implicite (les événements sont gérés par la causalité).
- Le niveau *système* (*system level*), ou *architecture*, est le niveau le moins détaillé. Le système à concevoir est vu comme un ensemble de processus communicants, mais représentant des fonctionnalités de très haut niveau comme des processeurs, des mémoires, des canaux de communication. La manière dont les processus communiquent entre eux est plus importante que le comportement des processus eux-mêmes. Le but principal est d'évaluer des réalisations de spécifications pour différentes décompositions fonctionnelles et différentes utilisations des ressources (débit de traitement, espace mémoire, etc.). Les modèles sont dits *non interprétés* (*uninterpreted models*).

A chaque niveau, il est en plus possible de représenter trois aspects, vues ou *domaines de description*, du système:

- La *vue comportementale*, représente ce que le système fait sous la forme d'un comportement entrée-sortie (boîte noire). Toute décomposition hiérarchique est purement fonctionnelle et n'implique pas forcément une structure.
- La *vue structurelle* représente comment le système est logiquement construit sous la forme d'une interconnexion de composants. Cette vue ne prend pas en compte l'aspect géométrique.
- La *vue géométrique (ou physique)* représente comment le système est réellement construit. Elle prend en compte les aspects de taille, de forme et de position des composants.

## Langage de description de matériel

- **Description de tous les aspects d'un système matériel:**
  - **Structure topologique, hiérarchie**
  - **Comportement**
  - **Assemblage, structure physique**
  
- **Description sur plusieurs niveaux d'abstraction**
  
- **Modèle logiciel d'un système matériel**
  - **Modularité, extensibilité, typage des données**
  
- **Indépendant de:**
  - **Technologie (MOS, bipolaire, etc.)**
  - **Méthode de conception (*top-down*, *bottom-up*, etc.)**
  - **Style de conception (*full custom*, cellules standard, FPGA, etc.)**
  - **Outils de conception (simulateurs, synthétiseurs, etc.)**
  
- **Usage:**
  - **Documentation**
  - **Spécification**
  - **Simulation**
  - **Synthèse**

### 1.3. Langage de description de matériel

Un langage de description de matériel (*HDL – Hardware Description Language*) est un outil de description, éventuellement formel, permettant la description du comportement et de la structure d'un système matériel. Un langage de description de matériel idéal a les propriétés suivantes:

- Il supporte la description d'une large gamme de systèmes à la fois logiques (numériques) et analogiques. Pour les systèmes logiques, il supporte les systèmes combinatoires et séquentiels, synchrones et asynchrones. Pour les systèmes analogiques, il supporte non seulement des systèmes électriques, mais aussi mécaniques, thermiques, acoustiques, etc. Ce dernier aspect est très important pour la conception de systèmes car il s'agit de prendre en compte les interfaces avec l'environnement extérieur dans lequel le système conçu sera testé et dans lequel il fonctionnera.
- Il permet la description de l'état de la conception pour toutes les étapes du processus. Le fait d'utiliser un langage unique renforce la cohérence entre différentes représentations d'un même système.
- Il renforce aussi la cohérence des outils logiciels utilisés pour la simulation et la synthèse. Il peut être directement compris comme langage d'entrée pour de tels outils.
- Il est indépendant de toute méthodologie de conception, de toute technologie de fabrication et de tout outil logiciel. Il permet au concepteur d'organiser le processus de conception en fonction des besoins (conception descendante - *top-down design*, conception ascendante - *bottom-up design*, séparation des parties logiques et analogiques, séparation des parties de contrôle et des parties opératives, etc.).
- Il supporte plusieurs niveaux d'abstraction et autorise des descriptions hiérarchiques. Il supporte des descriptions comportementales (fonctionnelles) aussi bien que structurales. Pour chaque niveau d'abstraction, il supporte les primitives correspondantes et leurs caractéristiques et permet d'exprimer les contraintes de conception. Un aspect important est la possibilité de spécifier des caractéristiques temporelles comme le cycle d'horloge, des délais, des temps de montée, de descente, de prépositionnement (*setup time*) et de maintien (*hold time*).
- Il est extensible: il permet au concepteur de définir de nouveaux types d'objets et les nouvelles opérations correspondantes.
- Il est plus qu'un simple format d'échange entre outils logiciels. Il renforce la communication et la cohésion à l'intérieur des équipes de conception et entre les différentes communautés de concepteurs parce qu'il est lisible (format texte) et qu'une description écrite dans un tel langage contient beaucoup d'information sur l'expertise derrière la conception. Il améliore donc grandement les phases de spécification et de documentation.
- Il est standardisé par l'intermédiaire d'organisations reconnues comme l'IEEE, l'ANSI ou l'ISO. Ceci favorise une large acceptation à la fois de la part des fournisseurs d'outils CAO et des différentes communautés de concepteurs.

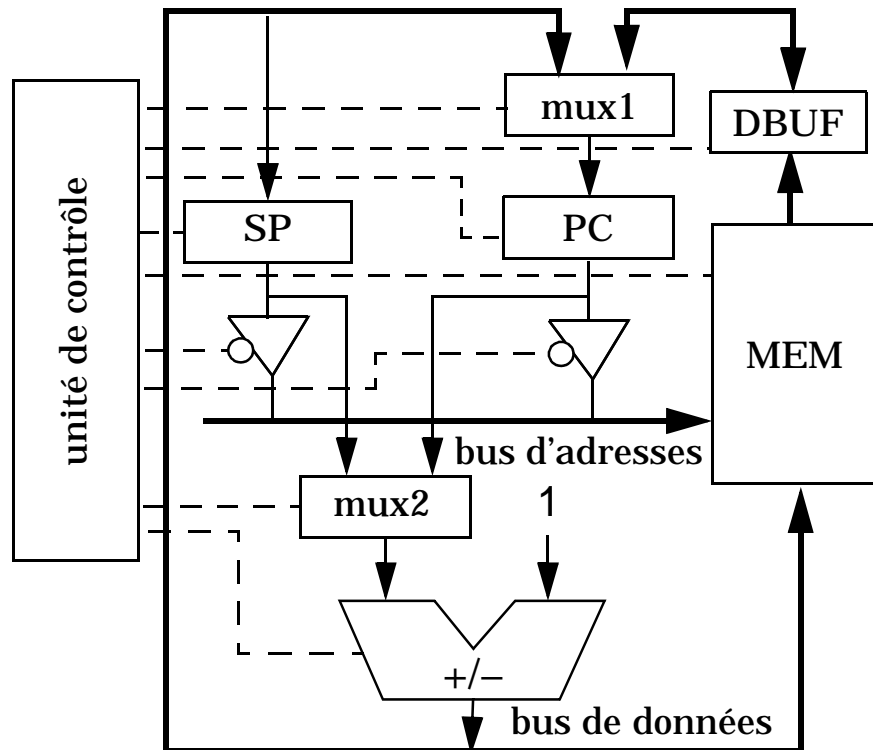
## Exemple de description comportementale, structurelle et géométrique

```

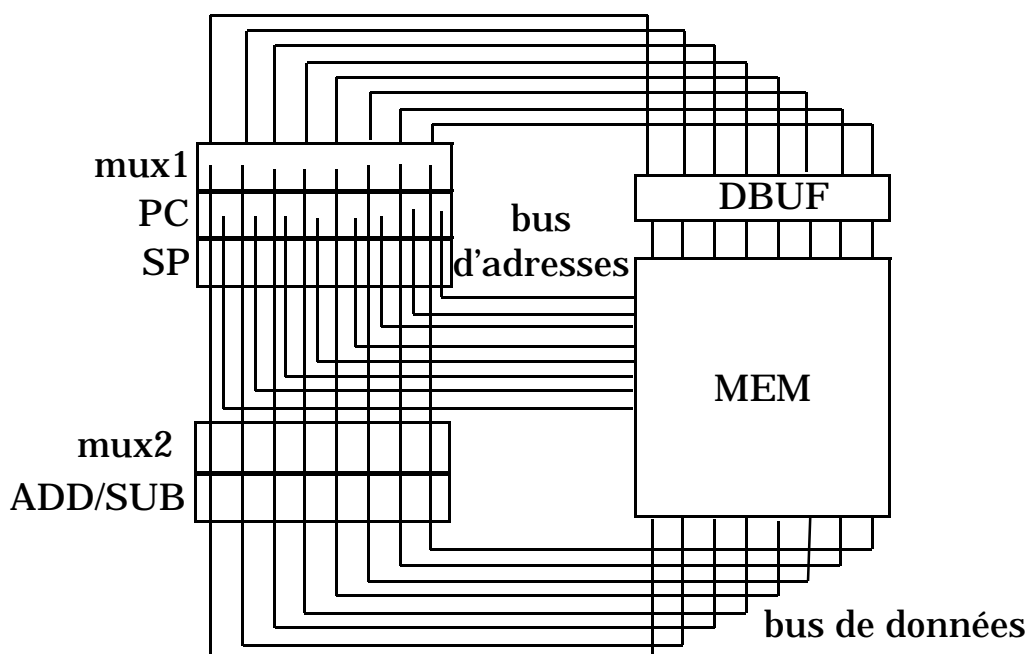
if IR(3) = '0' then
  PC := PC + 1;
else
  DBUF := MEM(PC);
  MEM(SP) := PC + 1;
  SP := SP + 1;
  PC := DBUF;
end if;

```

**Code 1.** Description comportementale (extrait).



**Figure 1.** Description structurelle.



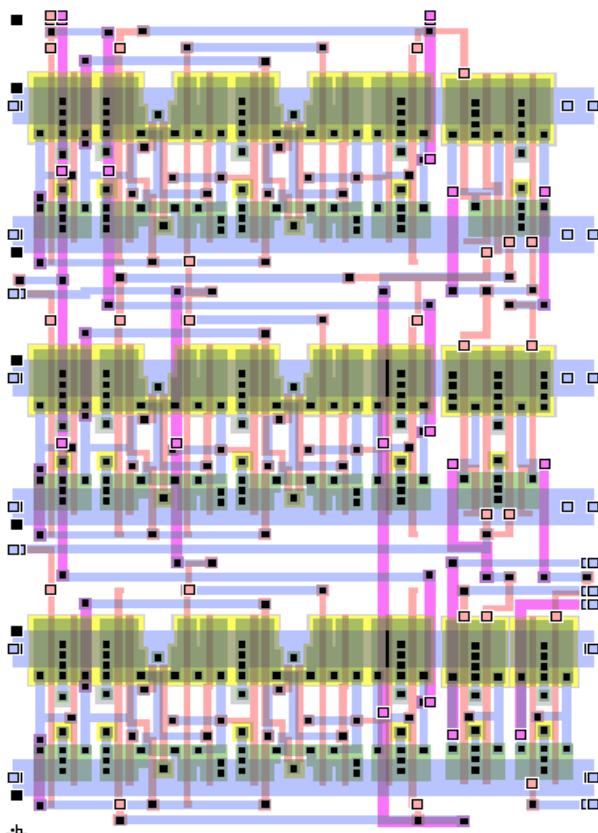
**Figure 2.** Description géométrique.

## 1.4. Exemple de description comportementale, structurelle et géométrique

Le [Code 1](#) donne un extrait du comportement d'une instruction d'appel conditionnel de sous-programme d'un processeur hypothétique en langage VHDL. Selon la valeur du bit 3 du registre d'instruction IR, soit le compteur de programme PC est incrémenté, soit une adresse de sous-programme est écrite dans le compteur de programme, le contexte est sauvé en mémoire en vue de l'exécution du sous-programme. Une description plus complète du comportement du processeur devrait inclure toutes les instructions et les mode d'adressages. Les opérateurs "+" et "-" sont purement abstraits à ce niveau et ne correspondent pas (encore) à des blocs fonctionnels. Il s'agira certainement d'optimiser l'usage des ressources en partageant le plus possible les blocs pour plusieurs opérations.

La [Figure 1](#) illustre une description structurelle au niveau RTL correspondant au comportement du [Code 1](#). Cette description fait apparaître des blocs fonctionnels (unité arithmétique, registres, mémoire, etc.) ainsi que les bus nécessaires aux interconnexions. La partie contrôle, responsable du séquençement des opérations, est aussi clairement identifiée. Chaque bloc fonctionnel est décrit de manière comportementale. Par exemple, la partie contrôle comme une machine à états finis, la mémoire comme un tableau à deux dimensions. Plusieurs descriptions structurelles différentes au niveau RTL sont possibles (architecture pipeline, p. ex.).

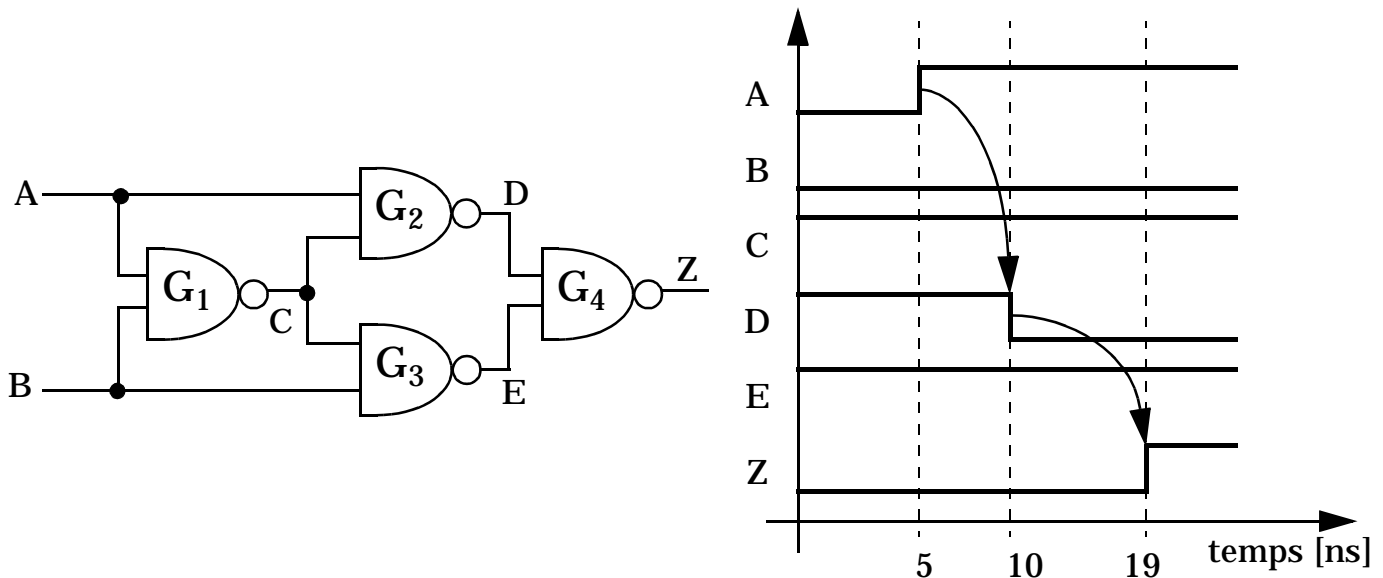
La [Figure 2](#) illustre une description géométrique donnant le plan d'implantation, ou *floorplan*, des blocs fonctionnels. Les dimensions et les positions relatives des blocs ont maintenant une importance. Dans cet exemple, la mémoire MEM et le registre DBUF sont placés à droite du plan, alors que les autres blocs sont groupés à gauche dans une structure dite de chemin de données (*datapath*) pour laquelle les opérateurs sont découpés en tranches de bits (*bit slices*). Les bus prennent leur forme et leur longueur finales. Il s'agit de minimiser la surface occupée et/ou les délais associés aux interconnexions.



**Figure 3.** Exemple de layout en style cellules standard

La [Figure 3](#) donne un autre exemple de description géométrique d'une partie de l'un des blocs de la [Figure 2](#) (le bloc de contrôle p. ex.). Le style utilisé est celui des cellules standard. Les portes logiques sont placées et aboutées sur des rangées séparées par des canaux réservés aux interconnexions. L'implantation sur silicium de chaque transistor est décrite de manière abstraite par des polygones dont les couleurs font référence à des niveaux de fabrication (métal, polysilicium, diffusion, etc.).

## Simulation logique



**Figure 4.** Exemple de circuit logique réalisant la fonction  $A \downarrow B$ .

```

Tn := 0; -- temps de simulation
while Tn < Tstop {
  Tn := temps du prochain événement;
  foreach entrée primaire Ek à Tn {
    if Ek est active {
      forall portes Gi in Fanout(Ek) { Schedule(Gi, Tn) }}
  foreach porte G active à Tn {
    Prendre les valeurs des entrées de G;
    Calculer la nouvelle sortie de G;
    if sortie de G a changé {
      Calculer le délai de changement Dt;
      forall portes Gi in Fanout(G) { Schedule(Gi, Tn + Dt) }}
  }
}

```

**Code 2.** Pseudo-code simplifié de simulation logique dirigée par les événements.



## 1.5. Simulation logique

La simulation logique a pour but la vérification de circuits intégrés de grande taille LSI et VLSI (50'000 à 1'000'000 de transistors). Il s'agit de circuits fonctionnant en mode logique qu'il n'est plus possible de vérifier au niveau électrique vu l'énorme quantité d'information qu'il faudrait stocker et gérer au niveau du transistor. Il est ainsi plus naturel et plus efficace de considérer une description du circuit à un niveau d'abstraction plus élevé, le niveau logique, pour lequel les primitives sont des portes logiques (ET, OU, NON, etc.) et les signaux représentés par des formes d'ondes logiques ne pouvant prendre qu'un nombre limité d'états. Les modèles des primitives sont simples et le circuit comporte moins d'éléments (chaque primitive regroupe en fait plusieurs transistors). De plus, l'effort nécessaire pour évaluer l'état du circuit à un moment donné est faible car l'arithmétique est basée sur des opérateurs booléens. Il est ainsi possible d'effectuer des simulations 10 à 100 fois plus rapides que les formes de simulation électrique les plus efficaces. Evidemment, la rapidité de simulation est au prix d'un appauvrissement dans le détail de l'information obtenue. Les états logiques ne représentent que partiellement les signaux électriques tension et courant et les délais doivent être modélisés séparément si l'on ne désire pas seulement effectuer une vérification fonctionnelle.

La [Figure 4](#) illustre un circuit logique dont la sortie vaut '1' ou la valeur VRAIE si les entrées A et B sont différentes et '0' ou la valeur FAUSSE si les entrées sont les mêmes. Un chronogramme illustrant l'évolution des signaux logiques du circuit est également donné en supposant une variation de valeur sur l'entrée A.

Le [Code 2](#) donne le pseudo-code simplifié de l'algorithme de simulation logique dirigée par les événements. Le temps est ici une variable entière multiple d'une unité de base appelée *temps de résolution minimum* (*Minimum Resolvable Time* - MRT). Toute valeur de délai non représentable entièrement dans l'échelle utilisée est tronquée (p. ex. un délai de 1,5 ns sera tronqué à 1 ns si la base de temps est la ns). L'usage d'une base de temps plus fine à par contre pour conséquence de limiter la valeur maximum du temps simulé car le codage du temps est fait avec un nombre limité de bits (32 ou 64 bits). La [Table 2](#) donne la dynamique approximative des valeurs de temps pour différentes bases de temps.

Base de temps	Résolution (s)	64 bits en compl. à 2	32 bits en compl. à 2
ms	1e-3	300 millions d'années	25 jours
us	1e-6	300'000 ans	36 minutes
ns	1e-9	300 ans	2 secondes
ps	1e-12	3 ans	2 millisecondes
fs	1e-15	3 heures	2 microsecondes

**Table 2:** Dynamique approximative des valeurs de temps pour différentes bases de temps.

Un événement consiste en un changement de valeur d'un signal aux entrées primaires ou aux noeuds internes du circuit. L'état du circuit n'est réévalué qu'aux instants auxquels un événement a lieu. La fonction Fanout(G) retourne l'ensemble des portes dont l'une des entrées au moins est connectée à la sortie de la porte G (p. ex.: les portes G2 et G3 font partie de l'ensemble Fanout(G1)). La procédure Schedule(G, Tn) marque la porte G comme devant être réévaluée au temps Tn.

Un aspect important de la simulation logique est le nombre d'états considérés. Le système à deux états binaires n'est pas suffisant pour modéliser et observer des états indéfinis, inconnus, ou à haute impédance. Il peut aussi être utile de modéliser des comportements résistifs au niveau logique au moyen d'états dits faibles (un '0' ou un '1' faible p. ex.).

## Synthèse logique

```

entity exsynt is
  port (A, B, C, E, F, G: in bit; Z: out bit);
end;

architecture comb of exsynt is
  signal X1, X2, X3: bit;
begin
  Z <= (A and B) or X1;
  X1 <= X3 xor G;
  X2 <= E or (not F);
  X3 <= C and X2;
end;

```

Code 3. Exemple de comportement combinatoire en VHDL.

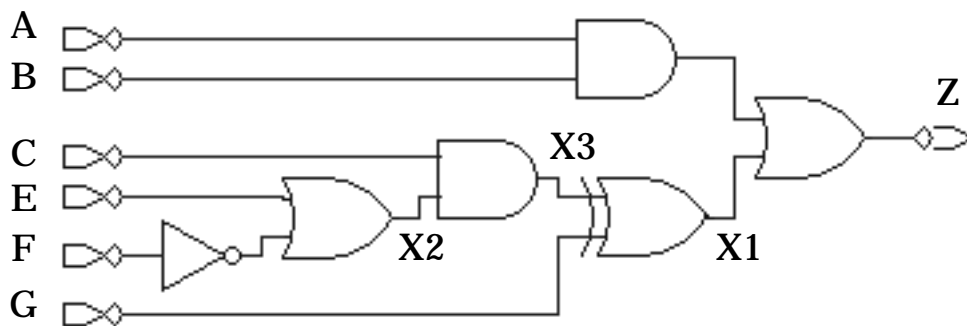


Figure 5. Circuit obtenu par traduction directe du Code 3.

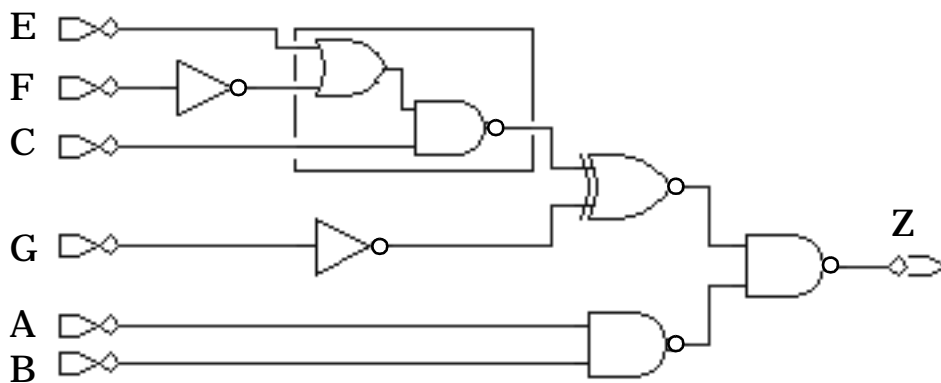


Figure 6. Circuit avec chemin critique optimisé.

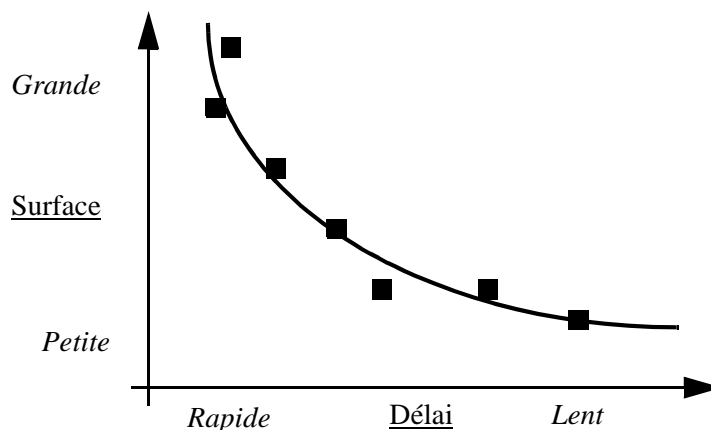
## 1.6. Synthèse logique

La synthèse logique détermine une architecture au niveau logique en identifiant les portes logiques nécessaires à la réalisation des blocs définis au niveau RTL et en déterminant une interconnexion optimale de ces portes. Deux tâches principales sont effectuées:

- Une phase d'**optimisation logique** (*logic optimization* ou *logic minimization*) cherche à reformuler les équations logiques (pour un circuit combinatoire) ou à minimiser le nombre d'états (pour un circuit séquentiel) représentant la fonction d'un bloc RTL.
- Une phase d'**allocation de cellules** (*library binding*) détermine la meilleure structure à base de cellules standard dans une technologie donnée ou la meilleure programmation d'un circuit FPGA.

Le [Code 3](#) donne un exemple de description VHDL d'un comportement combinatoire simple. La synthèse logique va tout d'abord compiler le modèle pour en dériver une représentation interne sous la forme d'un graphe. La [Figure 5](#) illustre le circuit logique obtenu par traduction directe du comportement, sans optimisation. La première phase d'optimisation logique n'a pas encore eu lieu et les portes dessinées ne correspondent pas encore à des cellules d'une bibliothèque réelle, mais à des cellules génériques. La [Figure 6](#) donne le résultat de l'optimisation logique et de l'allocation des cellules de la bibliothèque de cellules standard réelle. Dans ce cas l'optimisation a eu lieu sur le chemin critique F-X2-X3-X1-Z qui valait 1.25 ns initialement et 0.89 ns après optimisation (pour une technologie 0.6 micron CMOS). Le prix à payer est une augmentation de surface estimée de 49 microns carrés à 62 microns carrés, soit environ 27%. On parle bien d'estimation de surface ici car la place effective prise par les interconnexions n'est pas encore connue. Un modèle de calcul se basant sur le nombre de fils associés aux noeuds (équivalent à la fonction Fanout(G) discutée pour la simulation logique) a été utilisé.

La [Figure 7](#) donne la courbe typique surface-délai que l'on suit lorsque l'on explore l'espace des solutions avec la synthèse logique. Il n'est donc pas possible de minimiser à la fois la surface du circuit et son chemin critique. Une architecture parallèle (pipeline p. ex.) sera ainsi plus rapide qu'une architecture série, mais elle occupera plus de place à cause de la duplication nécessaire des ressources.



**Figure 7.** Courbe surface-délai typique pour la synthèse logique.



## 2. VHDL

Le langage VHDL<sup>1</sup> permet la description de tous les aspects d'un système matériel (*hardware system*): son comportement, sa structure et ses caractéristiques temporelles. Par système matériel, on entend un système électronique arbitrairement complexe réalisé sous la forme d'un circuit intégré ou d'un ensemble de cartes. Le comportement définit la ou les fonctions que le système remplit (p. ex. le comportement d'un microprocesseur comporte, entre autres, des fonctions arithmétiques et logiques). La structure définit l'organisation du système en une hiérarchie de composants (p. ex. un microprocesseur est constitué d'une unité de contrôle et d'une unité opérative; cette dernière est elle-même, entre autres, constituée d'un composant réalisant les opérations arithmétiques entières et d'un composant réalisant les opérations arithmétiques en virgule flottante). Les caractéristiques temporelles définissent des contraintes sur le comportement du système (p. ex. les signaux d'un bus de données doivent être stables depuis un temps minimum donné par rapport à un flanc d'horloge pour qu'une opération d'écriture dans la mémoire soit valable).

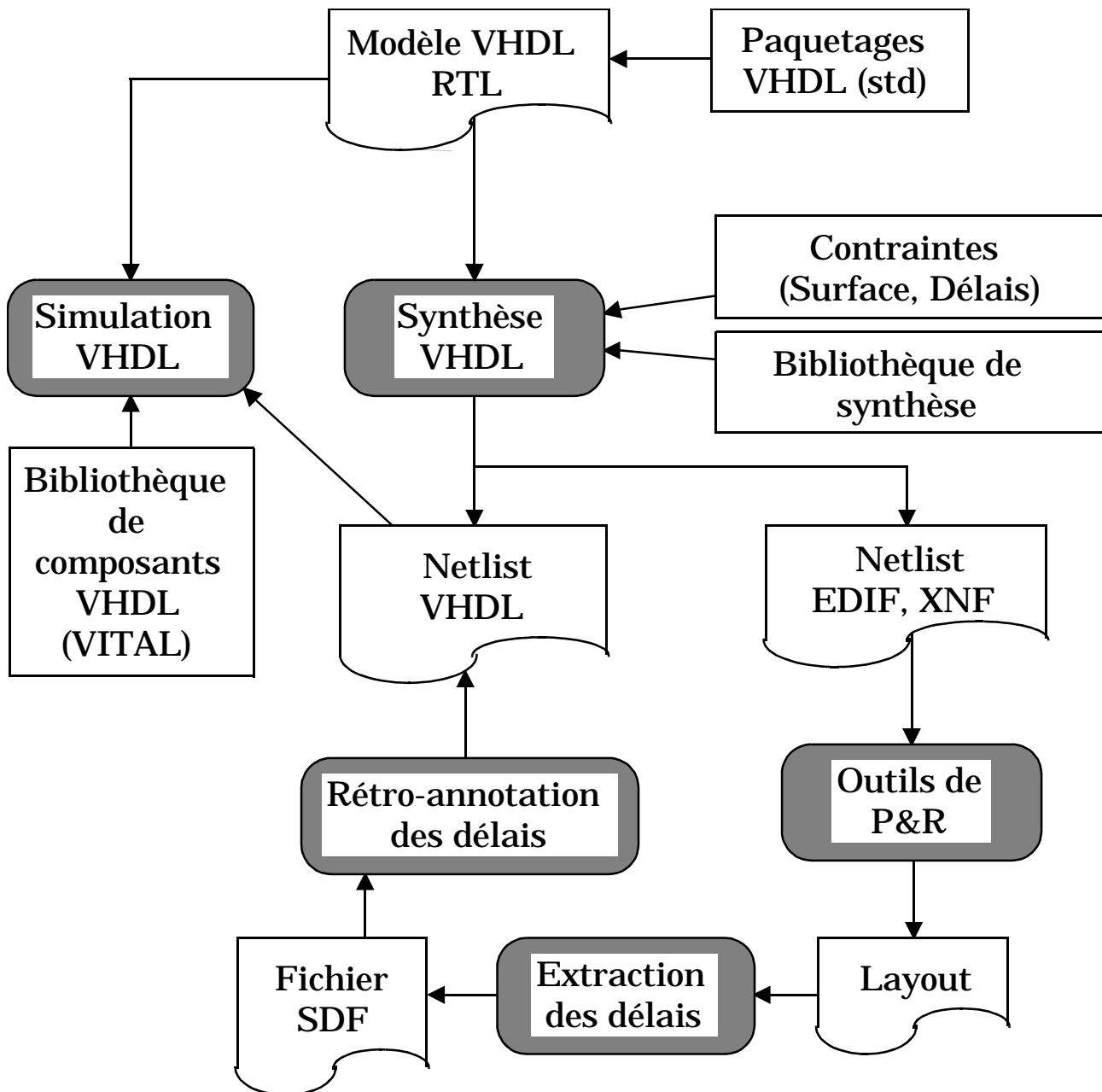
La description d'un système matériel en VHDL est en plus simulable. Il est possible de lui appliquer des stimuli (également décrits en VHDL) et d'observer l'évolution des signaux du modèle dans le temps. La sémantique d'un modèle VHDL est basée sur un modèle de simulation discrète dirigée par les événements (*event-driven discrete simulation*). Le modèle VHDL ne peut prendre un nouvel état en simulation que lorsqu'un stimuli change de valeur et que ce changement est propagé dans le modèle. Le langage VHDL définit des règles précises pour l'évaluation de l'état d'un modèle en présence d'un changement de stimuli. Ces règles garantissent que l'évaluation abouti au même résultat quel que soit l'outil de simulation utilisé.

Le langage VHDL est aussi très utilisé pour la synthèse, par exemple pour dériver automatiquement un circuit à base de portes logique optimisé à partir d'une description au niveau RTL (*Register-Transfer Level*) ou algorithmique. Cette application très importante du langage sort toutefois du cadre de sa définition standard et comporte des limitations dont les plus importantes seront présentées dans ce chapitre.

Finalement, le langage VHDL est un standard IEEE<sup>2</sup> depuis 1987 sous la dénomination IEEE Std. 1076-1987 (VHDL-87). Il est sujet à une nouvelle version tous les cinq ans. La dernière version est celle de 1993 (IEEE Std. 1076-1993, VHDL-93) [LRM93][Berg93]. Elle corrige certaines incohérences de la version initiale et ajoute de nouvelles fonctionnalités.

- 
1. VHDL est l'acronyme de *VHSIC Hardware Description Language*, où VHSIC signifie *Very High-Speed Integrated Circuit*.
  2. L'IEEE (*Institute of Electrical and Electronics Engineers*) est un organisme mondial qui définit entre autres des normes pour la conception et l'usage de systèmes électriques et électroniques.

## Flot de conception basé sur VHDL



## 2.1. Flot de conception basé sur VHDL

Le flot de conception actuel basé sur VHDL part d'une description du système à réaliser au niveau RTL. Les fonctions complexes sont décrites de manière comportementale. Par exemple, un contrôleur (ou séquenceur) est décrit comme une machine à états finis (*finite state machine*), une partie opérative comme une unité arithmétique et logique (ALU) est décrite comme un flot de données régit par des équations booléennes et contrôlé par un signal d'horloge.

Un tel modèle RTL utilise normalement un ensemble de déclarations groupées dans des paquetages (*packages*) VHDL. Ces déclarations peuvent définir les types d'information manipulés par le modèle. Par exemple, le paquetage standard IEEE STD\_LOGIC\_1164 définit un système de valeurs logiques à 9 états [STD1164].

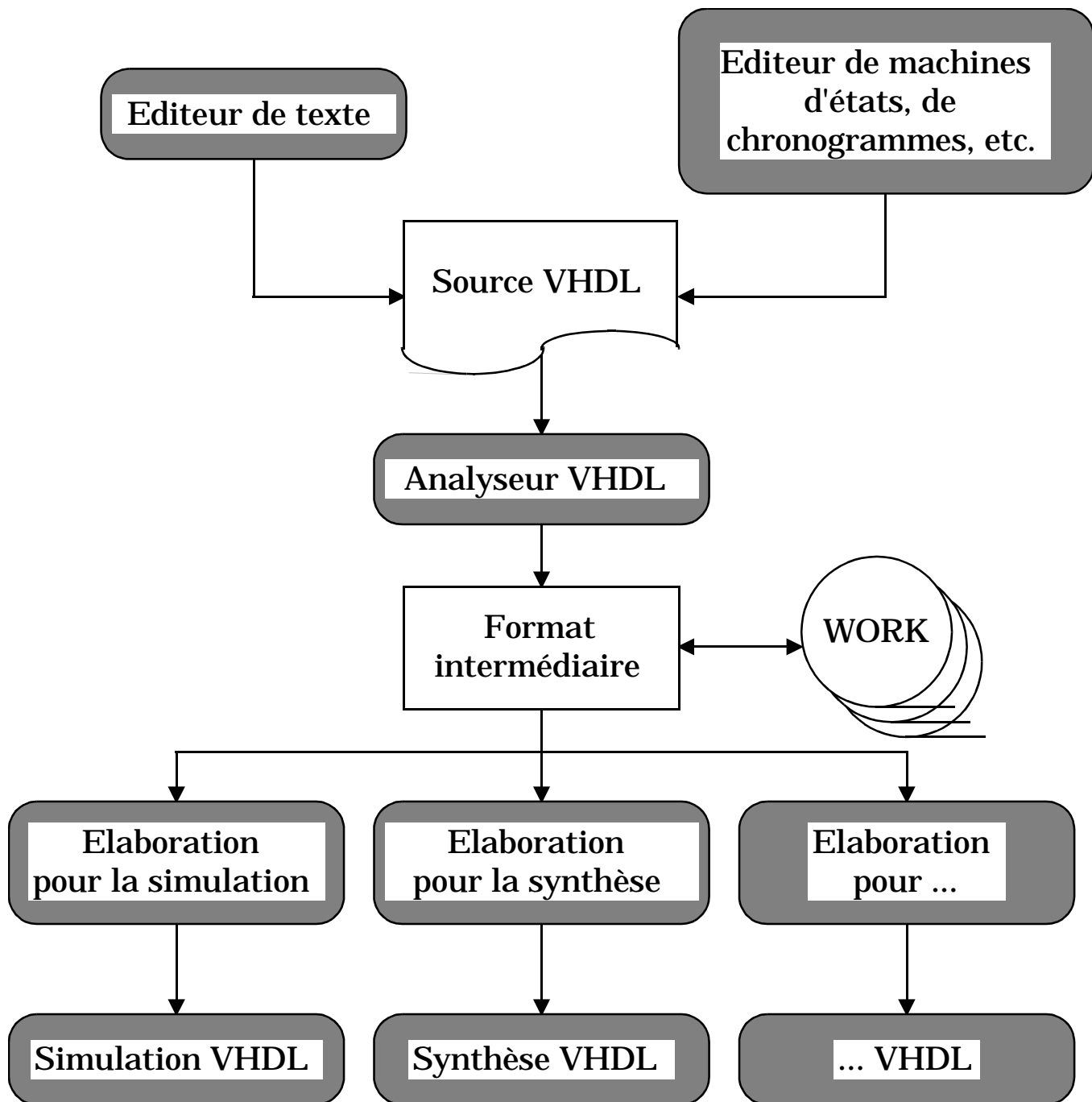
Le modèle RTL peut être validé par simulation logique. Un environnement de test (*testbench*) peut être écrit en VHDL. Il déclare une instance du composant à tester, le modèle RTL, et un ensemble de stimuli, ou vecteurs de test. Les fonctions du système peuvent être ainsi vérifiées avant de disposer d'une réalisation détaillée de celles-ci.

Le modèle RTL peut être ensuite synthétisé. Un outil de synthèse logique (*logic synthesis*) est capable de transformer une description comportementale en un circuit optimisé à base de portes logiques (*gate-level netlist*). L'optimisation est gouvernée par un ensemble de contraintes fournies par l'utilisateur sur la surface et/ou les délais que doit satisfaire le circuit. La définition de ces contraintes est faite dans un langage propre à l'outil de synthèse. La synthèse se base aussi sur une bibliothèque de synthèse contenant les descriptions de toutes les cellules (portes) disponibles dans la technologie utilisée (p. ex. 0.6 micron CMOS). Les informations essentielles sont, pour chaque cellule: sa fonction (p. ex. NAND, flip-flop), sa surface et les délais associés à ses chemins d'entrées-sorties. Là aussi le format de la bibliothèque dépend de l'outil de synthèse.

Le résultat de la synthèse logique est un circuit de portes logiques dont on peut en tirer une nouvelle description VHDL. Les modèles VHDL des portes logiques sont stockés dans une bibliothèque mise à disposition par le fournisseur technologique (fondeur ou fabricant FPGA). Ces modèles de portes sont usuellement définis selon la norme VITAL (*VHDL Initiative Towards VHDL Libraries*) [STD1076.4] pour permettre la rétroannotation des délais dus aux interconnexions. La simulation après synthèse logique ne prend cependant en compte que les délais des portes.

La réalisation du circuit sous forme de layout est faite par un outil de placement et de routage qui nécessite une description d'entrée dans un format différent de VHDL. Les formats EDIF (pour les cellules standard) ou XNF (pour les FPGA de type Xilinx) sont alors utilisés. Une fois le layout obtenu il est possible d'en extraire les valeurs des éléments parasites (résistances, capacités) associés aux interconnexions et de calculer les délais correspondants. Ces délais sont stockés dans un fichier au format SDF (*Standard Delay Format*) [SDF], puis réinjectés dans la description VHDL obtenue après synthèse logique (rétroannotation, *backannotation*). Cette fois-ci la simulation prend en compte les délais des portes logiques et des interconnexions. Les résultats sont suffisamment précis pour que les fondeurs les acceptent comme référence (*sign-off simulation*) sans qu'il soit nécessaire de recourir à des simulations au niveau électrique (de type SPICE) coûteuses.

## Environnement de travail VHDL





## 2.2. Environnement de travail VHDL

L'*interface graphique* peut se réduire à un simple éditeur de texte. Les outils CAO du marché utilisent en plus leur éditeur de schémas pour générer automatiquement le squelette d'un modèle VHDL, c'est-à-dire au moins la déclaration d'entité avec ses ports et un corps d'architecture minimum. Des outils plus avancés permettent de décrire le comportement du système à modéliser sous la forme de machines d'états, de chronogrammes ou de tables de vérité.

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

Chaque concepteur possède une *bibliothèque de travail* (*working library*) de nom logique WORK (le nom est standard) dans laquelle sont placés tous les modèles compilés. Le lien du nom logique avec l'emplacement physique de la bibliothèque dépend de l'outil de simulation ou de synthèse utilisé.

Il est aussi possible de faire référence, en mode de lecture seule, à d'autres bibliothèques, des *bibliothèques de ressources*, contenant d'autres modèles ou des utilitaires. Plusieurs bibliothèques peuvent être actives simultanément. Chaque bibliothèque contient une collection de modèles mémorisés dans un format intermédiaire. Elle contient également un certain nombre de relations et d'attributs liant, si nécessaire, les différents modèles entre eux.

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

Un *outil de synthèse* est capable de générer une netlist de portes logiques (synthèse logique), et même maintenant une architecture RTL (synthèse de haut niveau), à partir d'une description VHDL comportementale. Le résultat de la synthèse peut lui aussi être décrit en VHDL pour simulation. Un tel outil n'accepte en général qu'un sous-ensemble du langage [Airi94] [Ott94] [Bhas96] [STD1076.3].

La phase d'*élaboration* est propre à chaque application (simulation, synthèse, etc.). Elle consiste en une construction des structures de données et permet la détection d'erreurs globales, qui concernent l'ensemble des unités de la description. Cette phase est normalement cachée pour la simulation, alors qu'elle est explicite pour la synthèse. En effet, dans ce dernier cas elle permet de dériver une structure générique, indépendante de la technologie cible, sur laquelle il est possible de définir des contraintes qui gouverneront la phase d'optimisation.

### Modèle sémantique de VHDL

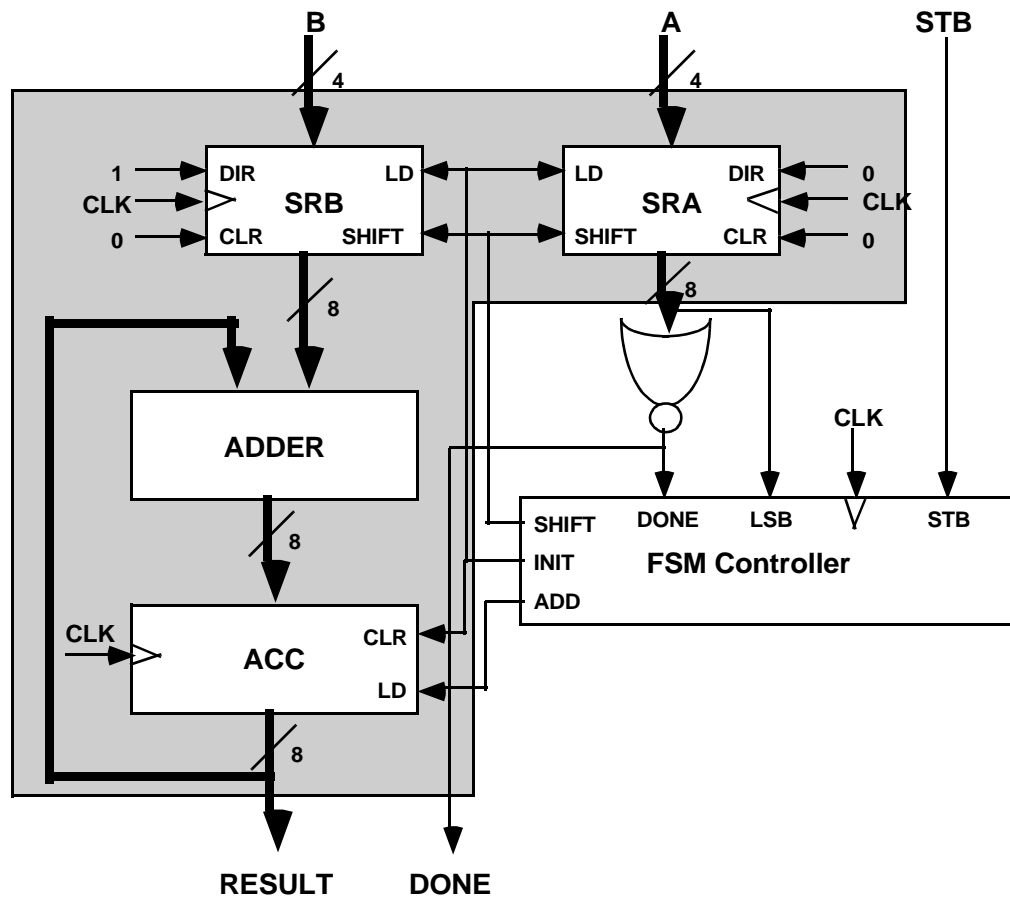


Figure 8. Vue RTL d'un multiplieur à additions et décalages.

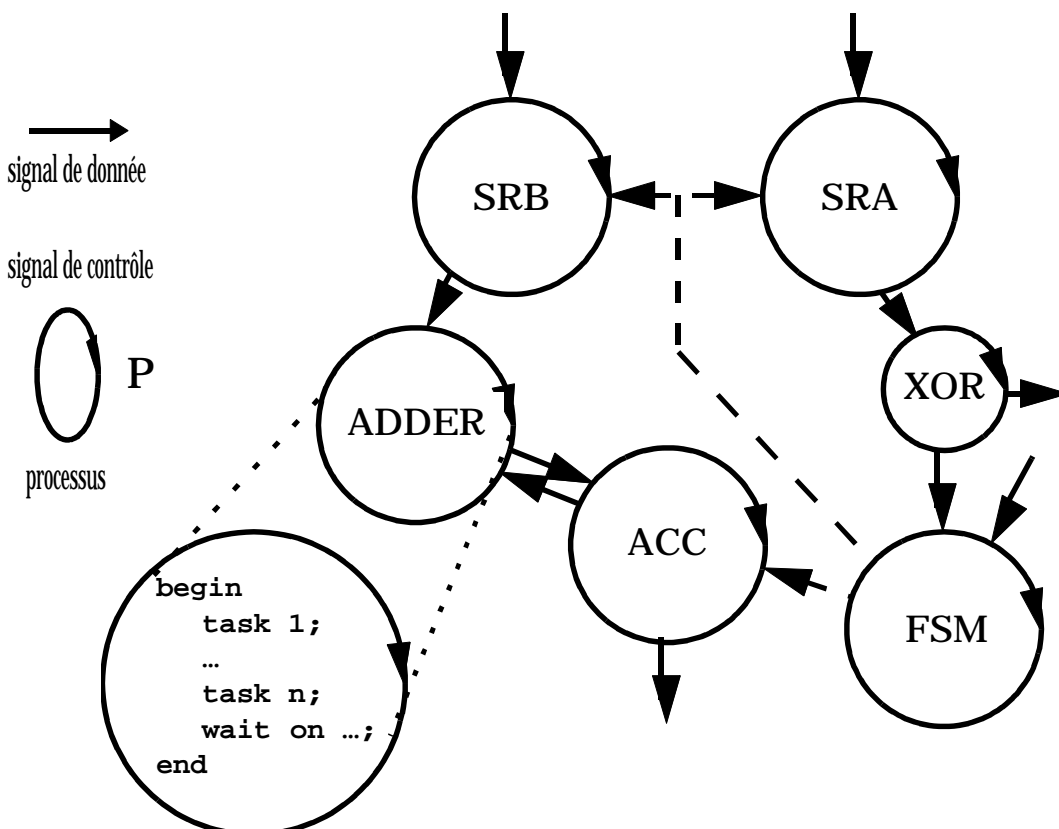


Figure 9. Vue "processus" du multiplieur.

### 2.3. Modèle sémantique de VHDL

Tout modèle VHDL peut se ramener à un ensemble de *processus concurrents* dont l'exécution peut être activée de manière individuelle (asynchrone) en fonction de l'activité de *signaux*. Chaque processus est sensible sur un certain nombre de signaux dont les événements (les changements de valeurs) activent l'exécution des instructions du processus. Ces instructions sont exécutées de manière séquentielle dans l'ordre de leur apparition. Les signaux représentent ainsi le moyen de communiquer des informations de manière globale dans le modèle VHDL<sup>1</sup>.

Un *cycle de simulation canonique* fait partie de la définition du langage VHDL. Il définit la manière de mettre à jour les signaux et à quel moment les processus doivent être activés. Il faut noter que l'ordre d'exécution des processus activés peut être quelconque et dépend de l'outil de simulation. Il est toutefois garanti que les signaux possèdent les mêmes valeurs à la fin d'un cycle de simulation quelque soit l'ordre d'exécution des processus.

Le modèle VHDL d'un système complexe est normalement décrit de manière hiérarchique comme une structure de composants. Chaque composant peut encapsuler du comportement sous forme de processus concurrents.

A titre d'exemple, considérons le modèle au niveau RTL d'un multiplieur de deux mots de 4 bits A et B réalisant l'algorithme classique basé sur des additions et des décalages successifs (Figure 8). La description est structurelle et définit les composants de la partie contrôle (le contrôleur FSM) et de la partie opérative (les registres SRA et SRB, l'additionneur ADDER et l'accumulateur ACC). On suppose que le comportement de chaque composant est décrit séparément comme une série d'instructions à exécuter en séquence.

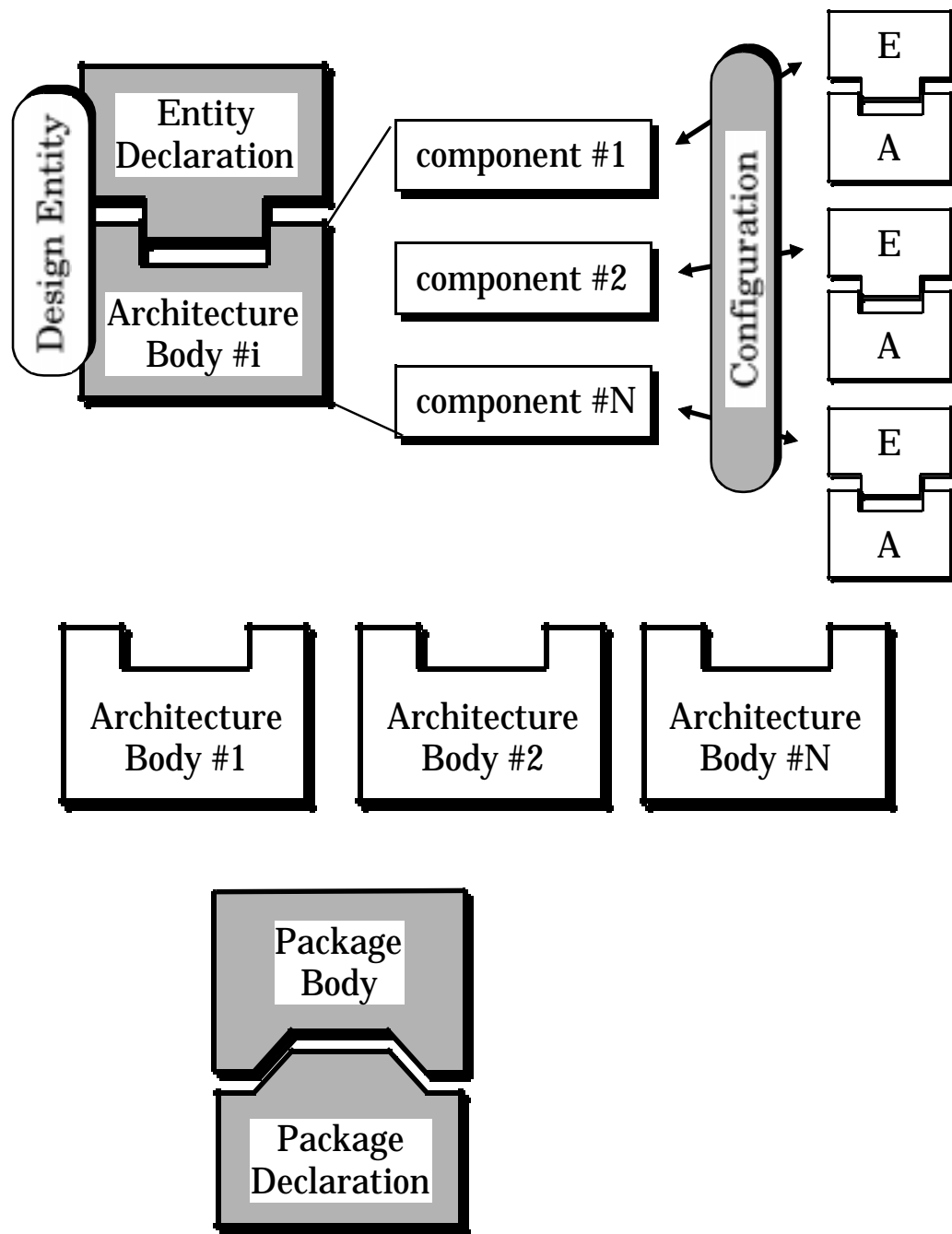
Le modèle VHDL du multiplieur peut être vu comme un ensemble de processus concurrents, chaque processus correspondant à un composant (Figure 9). Les processus sont liés entre eux par un réseau de signaux qui représentent les données et les signaux de contrôle. L'activité de ces signaux conditionne l'exécution des processus. Dès qu'un processus est activé il exécute ses instructions jusqu'à un point d'arrêt. Le point d'arrêt est une instruction VHDL particulière qui dit essentiellement d'attendre jusqu'à ce qu'il y ait un nouvel événement sur les signaux sensibles du processus. L'exécution d'un processus est *cyclique*: la séquence d'instructions recommence au début une fois la dernière instruction exécutée.

Par exemple, un flanc montant du signal d'horloge CLK et un signal SHIFT actif - égal à '1' - va simultanément<sup>2</sup> effectuer un décalage à gauche du registre SRB et à droite du registre SRA. Une fois les opérations effectuées chaque processus se met en état de veille jusqu'à ce qu'un nouvel événement significatif le réactive à nouveau.

Toutes les données nécessaires à l'exécution d'un processus (p. ex. des variables locales) sont ainsi créées avant de démarrer la simulation proprement dite et ne sont détruites qu'à la fin de la simulation.

- 
1. VHDL-93 supporte aussi le concept de variables globales (*shared variables*), mais leur usage est plus délicat. L'accès simultané à la même variable globale à partir de plusieurs processus n'est pas déterministe.
  2. L'effet sera effectivement simultané pour l'utilisateur, même si les deux processus SRA et SRB sont activés l'un après l'autre dans un ordre quelconque.

## Organisation d'un modèle VHDL



**Figure 10.** Unités de conception VHDL (en gris).

## 2.4. Organisation d'un modèle VHDL

### Unités de conception

L'**unité de conception** (*design unit*) est le plus petit module compilable séparément. VHDL offre cinq types d'unités de conception (Figure 10):

- la déclaration d'entité (*entity declaration*);
- le corps d'architecture (*architecture body*), ou plus simplement architecture;
- la déclaration de configuration (*configuration declaration*);
- la déclaration de paquetage (*package declaration*);
- le corps de paquetage (*package body*).

Les trois premières unités de conception (déclaration d'entité, architecture et déclaration de configuration) permettent la description de l'aspect matériel d'un système, alors que les deux dernières (déclaration et corps de paquetage) permettent de grouper des informations pouvant être réutilisées pour la description de plusieurs systèmes différents.

Les trois unités de conception: déclaration d'entité, déclaration de configuration et déclaration de paquetage, sont qualifiées de **primaires** (*primary unit*), car elles décrivent une vue externe (le "quoi" de la boîte noire). Les unités primaires déclarent ce qui est accessible aux autres parties du modèle. Elles permettent le partage et la réutilisation de ressources (modèles et algorithmes). Elles ne contiennent que le minimum d'information nécessaire pour l'utilisation de ces ressources.

Les deux autres unités de conception: architecture et corps de paquetage, sont qualifiées de **secondaires** (*secondary units*), car elles décrivent une vue interne particulière (une réalisation ou le "comment" de la boîte noire). Les unités secondaires regroupent des ressources seulement visibles par leur unité primaire correspondante (déclaration d'entité pour l'architecture et déclaration de paquetage pour le corps de paquetage). Elles contiennent des détails d'implémentation que l'on masque pour a) éviter de surcharger l'utilisateur avec des détails inutiles et/ou b) éviter des modifications malencontreuses de modèles et d'algorithmes de base. Plusieurs unités secondaires peuvent être définies pour une même unité primaire.

### Entité de conception

L'**entité de conception** (*design entity*) est l'abstraction de base en VHDL. Elle représente une portion d'un système matériel possédant une interface entrée-sortie et une fonction bien définies. Une entité de conception est constituée d'une déclaration d'entité et d'un corps d'architecture correspondant.

Une entité de conception peut représenter un système matériel à plusieurs niveaux de complexité: un système entier, un sous-système, une carte, un circuit intégré, une cellule complexe (p.ex. ALU, mémoire, etc.), une porte logique.

Une **configuration** (*configuration*) permet de définir comment plusieurs entités de conception sont assemblées pour constituer le système complet. Une entité de conception peut être décrite comme une hiérarchie de **composants** (*component*) interconnectés. Chaque composant peut être lié à une entité de conception de plus bas niveau qui définit la structure ou le comportement de ce composant. Une telle décomposition d'une entité de conception et les liens de ses composants éventuels à d'autres unités de conception forment une hiérarchie représentant le système complet.

## Modèle comportemental d'un registre 4 bits

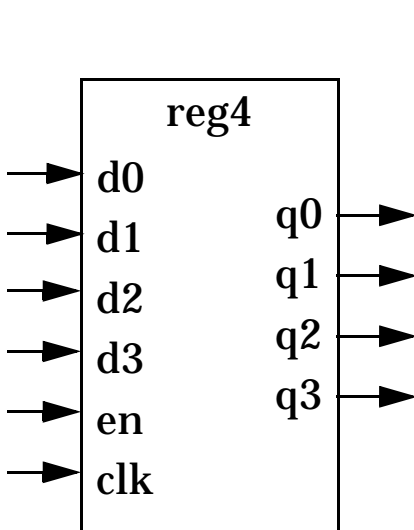


Figure 11. Vue externe du registre 4 bits.

```
entity reg4 is
    port (d0, d1, d2, d3,
          en, clk: in bit;
          q0, q1, q2, q3: out bit);
end entity reg4;
```

Code 4. Déclaration d'entité du registre 4 bits. Fichier: reg4\_ent.vhd.

```
architecture bhv of reg4 is
begin
    process is
        variable d0_reg, d1_reg, d2_reg, d3_reg: bit;
    begin
        if en = '1' and clk = '1' then -- mémorise les entrées
            d0_reg := d0;
            d1_reg := d1;
            d2_reg := d2;
            d3_reg := d3;
        end if;
        -- modifie les signaux de sortie
        q0 <= d0_reg after 5 ns;
        q1 <= d1_reg after 5 ns;
        q2 <= d2_reg after 5 ns;
        q3 <= d3_reg after 5 ns;
        wait on d0, d1, d2, d3, en, clk;
        -- attends le prochain événement sur l'un des signaux
    end process;
end architecture bhv;
```

Code 5. Modèle comportemental du registre 4 bits. Fichier: reg4\_bhv.vhd.

## 2.5. Modèles VHDL d'un registre 4 bits

On donne ici plusieurs versions possibles d'un modèle de registre 4 bits en VHDL pour illustrer l'usage des différentes unités de conception et des styles de description disponibles en VHDL. Se référer à l'[Annexe A](#) pour plus de détails sur la syntaxe du langage. Les mots-clés réservés du langage sont indiqués en caractères gras.

### Déclaration d'entité

La [Figure 11](#) illustre la vue externe du registre 4 bits. Le registre a pour nom REG4 et possède 6 entrées (les données D0, D1, D2, D3 et le signal de contrôle EN et l'horloge CLK) et 4 sorties (les données Q0, Q1, Q2, et Q3). Le [Code 4](#) définit la déclaration d'entité VHDL du registre. On peut distinguer les aspects principaux suivants:

- Les *ports* (*ports*) définissent les canaux de communication entre le modèle et le monde extérieur. Plus précisément ils définissent les types de signaux transitant par ces canaux ainsi que les directions de transition. Dans l'exemple, tous les signaux sont du type prédéfini<sup>1</sup> `bit`, c'est-à-dire pouvant prendre la valeur '0' ou '1', et les signaux d'entrées (resp. de sortie) sont définis par le *mode in* (resp. par le mode *out*).
- La déclaration d'entité peut être stockée dans un fichier dont le nom suggéré est `reg4_ent.vhd`. Elle peut être compilée et le résultat de la compilation sera placé dans la bibliothèque de travail WORK.

### Architecture comportementale

Le [Code 5](#) donne le corps d'architecture définissant un modèle comportemental du registre. On peut distinguer les aspects principaux suivants:

- L'architecture a pour nom BHV et fait référence à la déclaration d'entité de nom REG4. Tous les objets déclarés dans l'entité (ici les signaux d'interface) sont visibles dans l'architecture sans déclaration supplémentaire.
- Le comportement du registre est défini à l'aide d'un processus unique délimité par les mots-clés **process** et **end process**. Le processus définit une séquence d'instructions qui:
  - 1) mémorisent conditionnellement l'état des entrées du registre lorsque le signal EN et l'horloge CLK sont actifs (ici égaux à la valeur '1'),
  - 2) modifient les valeurs des signaux de sortie du registre avec un délai de 5 ns (ce délai modélise le fait que le registre réel ne réagira pas de manière instantanée),
  - 3) mettent le processus en état de veille jusqu'à un prochain événement *sur l'un quelconque* des signaux d'entrée (instruction **wait**).
 Un événement sur l'un des signaux d'entrée du registre réactivera le processus qui exécutera ses instructions en repartant de la première instruction après le mot-clé **begin**.
- Le processus déclare quatre variables locales de type `bit` qui servent à mémoriser l'état interne du registre. Ces variables sont initialisées à la valeur '0' au début de la simulation et conservent leurs valeurs d'une activation du processus à l'autre.
- Le corps d'architecture peut être stocké dans un fichier propre dont le nom suggéré est `reg4_bhv.vhd`. Il peut être compilé et le résultat de la compilation sera placé dans la bibliothèque de travail WORK.

---

1. Un certain nombre de déclarations sont prédéfinies en VHDL et stockées dans le paquetage nommé STANDARD de la bibliothèque nommée STD.

## Modèle structurel d'un registre 4 bits

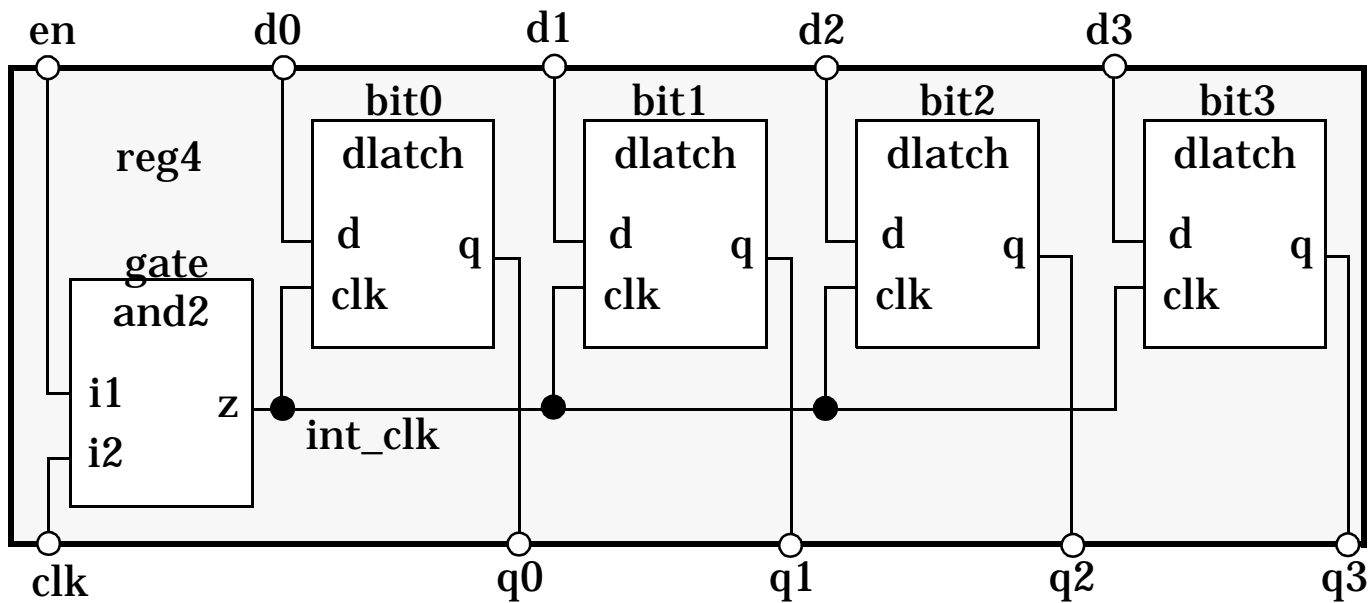


Figure 12. Vue structurelle du registre 4 bits.

```

architecture str of reg4 is

    -- déclaration des composants utilisés
    component and2 is
        port (i1, i2: in bit; z: out bit);
    end component and2;

    component dlatch is
        port (d, clk: in bit; q: out bit);
    end component dlatch;

    signal int_clk: bit; -- signal interne
begin
    -- instances de latches
    bit0: dlatch port map (d => d0, clk => int_clk, q => q0);
    bit1: dlatch port map (d => d1, clk => int_clk, q => q1);
    bit2: dlatch port map (d => d2, clk => int_clk, q => q2);
    bit3: dlatch port map (d => d3, clk => int_clk, q => q3);

    -- génération de l'horloge interne
    gate: and2 port map (en, clk, int_clk);
end architecture str;

```

Code 6. Modèle structurel du registre 4 bits. Fichier: reg4\_str.vhd.



## Architecture structurelle

La [Figure 12](#) représente une vue structurelle du registre 4 bits composée de composants de types latch et porte AND interconnectés. Le [Code 6](#) donne l'architecture VHDL correspondante. On peut distinguer les aspects principaux suivants:

- L'architecture nommée STR fait référence à la même déclaration d'entité que le modèle comportemental précédant, ce qui est normal vu que la vue externe du modèle reste la même.
- L'architecture reflète fidèlement la structure de la [Figure 12](#). Elle déclare dans un premier temps les composants nécessaires LATCH et AND2 dont un certain nombre d'instances seront "placées" dans le modèle.
- Chaque instance de composant définit les associations entre ses *ports formels* (*formal ports*), c'est-à-dire les ports définis dans sa déclaration, et ses *ports effectifs* (*actual ports*), c'est-à-dire les signaux d'interface ou le signal local du registre. Dans l'exemple, les instances de latches utilisent une *association par nom* (*named association*), alors que l'instance de la porte AND2 utilise une *association par position* (*positional association*).
- L'*étiquette* (*label*), telle que BIT0, est obligatoire pour nommer chaque instance de manière unique.
- L'ordre de déclaration des instances de composants peut être quelconque. Chaque instance peut être vue comme un processus activable par un événement sur l'un des signaux effectifs associé à un port formel de mode **in**.
- Les modèles des composants utilisés ne sont pas définis à ce niveau et qu'une étape de *configuration* sera encore nécessaire pour rendre le modèle structurel du registre simulable. Le rôle de la configuration est d'associer une entité de conception (paire entité/architecture) à chaque instance de composant. C'est pourquoi les noms des composants déclarés et de leurs ports formels peuvent être quelconques.
- Le corps d'architecture peut être stocké dans un fichier propre dont le nom suggéré est `reg4_str.vhd`. Il peut être compilé et le résultat de la compilation sera placé dans la bibliothèque de travail WORK.

## Environnement de test

```
entity reg4_tb is
end entity reg4_tb;

architecture test of reg4_tb is

    component c_reg4 is
        port (p_d0, p_d1, p_d2, p_d3, p_en, p_clk: in bit;
              p_q0, p_q1, p_q2, p_q3: out bit);
    end component c_reg4;

    signal s_d0, s_d1, s_d2, s_d3, s_en, s_clk,
           s_q0, s_q1, s_q2, s_q3: bit;
begin
    -- composant à tester
    UUT: c_reg4 port map (s_d0, s_d1, s_d2, s_d3, s_en, s_clk,
                          s_q0, s_q1, s_q2, s_q3);

    -- stimulis
    s_clk <= not s_clk after 20 ns; -- période de 40 ns
    process
    begin
        s_d0 <= '1'; s_d1 <= '1'; s_d2 <= '1'; s_d3 <= '1';
        s_en <= '0';
        wait for 40 ns;
        s_en <= '1';
        wait for 40 ns;
        s_d0 <= '0'; s_d1 <= '0'; s_d2 <= '0'; s_d3 <= '0';
        wait for 40 ns;
        s_en <= '1';
        wait for 40 ns;
        ...
        wait;
    end process;
end architecture test;
```

**Code 7.** Modèle VHDL de l'environnement de test. Fichier: reg4\_tb.vhd.

## Environnement de test

L'environnement de test (*testbench*) d'un modèle VHDL peut être lui-même décrit comme un modèle VHDL. La Figure 13 illustre la structure d'un tel environnement.

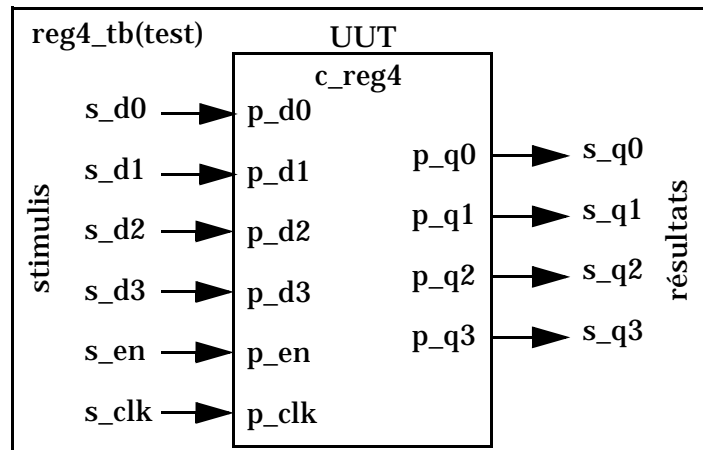


Figure 13. Environnement de test du registre 4 bits.

Le modèle à tester est instancié comme un composant d'un modèle qui ne possède pas d'interactions avec le monde extérieur. Le reste du modèle définit les stimuli à appliquer au composant testé et un traitement éventuel des résultats obtenus aux sorties du composant. Un environnement de test VHDL est usuellement un mélange de structure et de comportement.

Le Code 7 donne le modèle VHDL de l'environnement de test du modèle du registre 4 bits. Les points suivants sont à considérer :

- La déclaration d'entité se limite à sa plus simple expression.
- Le composant à tester est déclaré puis instancié. Pour l'instant rien ne lie l'instance de composant avec une entité de conception particulière. Il s'agit encore de définir une *configuration* pour rendre l'environnement de test simulable.
- Le comportement du signal d'horloge CLK est défini au moyen d'une assignation concurrente de signal. Le signal CLK a une valeur initiale par défaut égale à '0', qui est la première valeur du type énuméré prédéfini `bit`. Le signal aura une période de 40 ns.
- Les autres stimuli sont définis dans un processus. Plusieurs séquences de stimuli délimitées par des instructions `wait` sont définies. Lorsque tous les stimuli ont été appliqués, la dernière instruction `wait` sans paramètre stoppe le processus indéfiniment.
- Le modèle ne contient pas d'instructions particulières pour le traitement des résultats, c'est-à-dire des signaux Q0, Q1, Q2 et Q3. On se contente dans ce cas de visualiser les formes d'ondes dans l'environnement de simulation.
- Un environnement de test plus élaboré pourrait contenir des composants supplémentaires pour la génération des stimuli et le traitement des signaux de sortie.
- Comme la déclaration d'entité est très simple, il est recommandé de maintenir la déclaration d'entité et le corps d'architecture dans le même fichier dont le nom suggéré est `reg4_tb.vhd`. La compilation de ce fichier va créer *deux* unités de conception dans la bibliothèque de travail WORK.

## Déclaration de configuration

```

architecture test of reg4_tb is
  component c_reg4 is
    port (p_d0, p_d1, p_d2, p_d3, p_en, p_clk: in bit;
          p_q0, p_q1, p_q2, p_q3: out bit);
  end component c_reg4;
  ...
begin
  ...
end architecture test;


```

---

```

configuration reg4_cfg_tb_bhv of reg4_tb is
  for test
    for UUT: c_reg4 use entity WORK.reg4(bhv)
      port map (d0 => p_d0, d1 => p_d1, d2 => p_d2, d3 => p_d3,
                en => p_en, clk => p_clk,
                q0 => p_q0, q1 => p_q1, q2 => p_q2, q3 => p_q3);
    end for; -- UUT
  end for; -- test
end configuration reg4_cfg_tb_bhv;

```



**Code 8.** Déclaration de configuration pour le test du modèle comportemental du registre 4 bits.  
Fichier: reg4\_cfg\_tb\_bhv.vhd.

```

library GATES;
configuration reg4_cfg_tb_str of reg4_tb is
  for test
    for UUT: c_reg4 use entity WORK.reg4(str)
      port map (p_d0, p_d1, p_d2, p_d3, p_en, p_clk,
                p_q0, p_q1, p_q2, p_q3);
      for str
        for all: dlatch use entity GATES.dlatch(bhv); end for;
        for all: and2 use entity GATES.and2(bhv)
          port map (a => i1, b => i2, z => z);
        end for; -- and2
      end for; -- str
    end for; -- UUT
  end for; -- test
end configuration reg4_cfg_tb_str;

```

**Code 9.** Déclaration de configuration pour le test du modèle structurel du registre 4 bits.  
Fichier: reg4\_cfg\_tb\_str.vhd.

## Déclaration de configuration

La déclaration de configuration définit la vue de plus haut niveau d'un modèle. Elle définit les associations globales entre les instances de composants d'une architecture et les entités de conception disponibles dans une bibliothèque (WORK ou autre).

Le [Code 8](#) donne la déclaration de configuration de l'environnement de test du registre 4 bits utilisant le modèle comportemental du registre ([Code 5](#)).

- On suppose que l'entité de conception `reg4(bhv)` a déjà été compilée sans erreur dans la bibliothèque WORK.
- L'association **port map** est nécessaire parce que les noms donnés aux ports de la déclaration de composant dans l'environnement de test ne sont pas identiques à ceux de la déclaration d'entité du registre. Cette association pourrait être omise si les noms étaient identiques.
- La déclaration de configuration est stockée dans un fichier dont le nom suggéré est `reg4_cfg_tb_bhv.vhd`.

Le test de l'architecture structurelle du registre requiert la définition d'une autre configuration sans qu'il soit nécessaire de modifier le modèle de l'environnement de test. Le [Code 9](#) donne la nouvelle déclaration de configuration.

- On suppose que les entités de conception des portes DLATCH ([Code 10](#)) et AND2 ([Code 11](#)) ont été préalablement compilées sans erreur dans la bibliothèque GATES.
- La déclaration de configuration comporte dans ce cas un niveau hiérarchique supplémentaire pour associer les instances de portes logiques.
- Comme les noms des ports de la déclaration d'entité DLATCH sont identiques à ceux de la déclaration du composant du même nom dans l'architecture STR, la partie **port map** n'est pas nécessaire.
- La déclaration de configuration est stockée dans un fichier dont le nom suggéré est `reg4_cfg_tb_str.vhd`.

Les déclarations de configurations dans cet exemple représentent les descriptions de plus haut niveau du modèle. Ce sont les unités de conception à charger dans un simulateur. Toutes les autres unités de conception qui en dépendent seront chargées dynamiquement avant la phase d'élaboration.

## Déclaration de configuration (suite)

```

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

architecture bhv of dlatch is
begin
  process is
  begin
    if clk = '1' then
      q <= d after 2 ns;
    end if;
    wait on clk, d;
  end process;
end architecture bhv;

```

**Code 10.** Modèle comportemental d'un composant latch. Fichier: dlatch\_bhv.vhd.

```

entity and2 is
  port (a, b: in bit;
        z      : out bit);
end entity and2;

architecture bhv of and2 is
begin
  process is
  begin
    z <= a and b after 2 ns;
    wait on a, b;
  end process;
end architecture bhv;

```

**Code 11.** Modèle comportemental d'un composant and2. Fichier: and2\_bhv.vhd.

```

library GATES;
architecture str2 of reg4 is
  signal int_clk: bit; -- signal interne
begin
  -- instantiations directes des latches
  bit0: entity GATES.dlatch(bhv)
    port map (d => d0, clk => int_clk, q => q0);
  bit1: entity GATES.dlatch(bhv)
    port map (d => d1, clk => int_clk, q => q1);
  bit2: entity GATES.dlatch(bhv)
    port map (d => d2, clk => int_clk, q => q2);
  bit3: entity GATES.dlatch(bhv)
    port map (d => d3, clk => int_clk, q => q3);
  -- instantiation directe de la porte and2
  gate: entity GATES.and2(bhv)
    port map (en, clk, int_clk);
end architecture str2;

```

**Code 12.** Instanciation directe.

## Instanciation directe

L'*instanciation directe* (*direct instantiation*) a été introduite en VHDL-93 pour offrir une version simplifiée du mécanisme de configuration. Il n'est plus nécessaire de déclarer de composants et de configuration. Chaque instance est directement associée à une entité de conception.

Le [Code 12](#) donne une nouvelle version de l'architecture structurelle du registre 4 bits utilisant l'instanciation directe. Chaque instance doit être associée indépendamment à une entité de conception. Il n'est pas possible d'utiliser de spécification **for all** dans ce cas.

## Spécification de configuration

La *spécification de configuration* (*configuration specification*) est une autre forme de configuration qui ne définit pas une nouvelle unité de conception. Elle revient à définir l'association d'une instance de composant avec son entité de conception correspondante directement dans le corps d'architecture, en nécessitant toutefois toujours une déclaration de composant préalable. Le [Code 13](#) donne des exemples de spécifications de configurations pour l'architecture structurelle du registre 4 bits. Dans ce cas, la déclaration de configuration de l'environnement de test du [Code 9](#) s'en trouve simplifiée.

```

library GATES;
architecture str2 of reg4 is

    component and2 is
        port (i1, i2: in bit; z: out bit);
    end component and2;
    for all: and2 use entity GATES.and2(bhv) port map (i1, i2, z);

    component dlatch is
        port (d, clk: in bit; q: out bit);
    end component dlatch;
    for all: dlatch use entity GATES.dlatch(bhv);
    ...
begin
    ...
end architecture str2;

```

**Code 13.** Utilisation de spécifications de configurations pour l'architecture structurelle du registre 4 bits; le reste du modèle est identique à celui du [Code 6](#).

## Remarque

Il existe une forte ressemblance entre une déclaration de composant et une déclaration d'entité. Une différence essentielle est que la déclaration d'entité définit ce que le modèle d'un composant offre au monde extérieur, alors que la déclaration de composant définit ce dont le modèle qui utilise des instances de ce composant a besoin. D'une manière imagée on peut dire que la déclaration de configuration permet de relier l'offre à la demande lorsque ces dernières n'utilisent pas les mêmes conventions de nommage des paramètres ou des ports.

## Modèles génériques

- Paramètres (constantes) génériques
- Objets d'interface non contraints
- Instruction concurrente `generate`

```

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

architecture bhv of dff is
begin
  behavior: q <= d after Tpd_clk_q when clk = '1' and clk'event;
  check_setup: process is
  begin
    wait until clk = '1';
    assert d'last_event >= Tsu_d_clk report "Setup violation";
  end process check_setup;
  check_hold: process is
  begin
    wait until clk'delayed(Th_d_clk) = '1';
    assert d'delayed'last_event >= Th_d_clk
      report "Hold violation";
  end process check_hold;
end architecture bhv;

FF1: entity WORK.dff(bhv)
  generic map (Tpd_clk_q => 4 ns,
              Tsu_d_clk => 3 ns, Th_d_clk => 1 ns)
  port map (clk => master_clk, d => data, q => data_reg);

```

**Code 14.** Exemple d'utilisation de paramètres génériques spécifiant les caractéristiques temporelles d'un modèle de flip-flop D et exemple d'instance de composant avec spécification des valeurs effectives pour les paramètres.



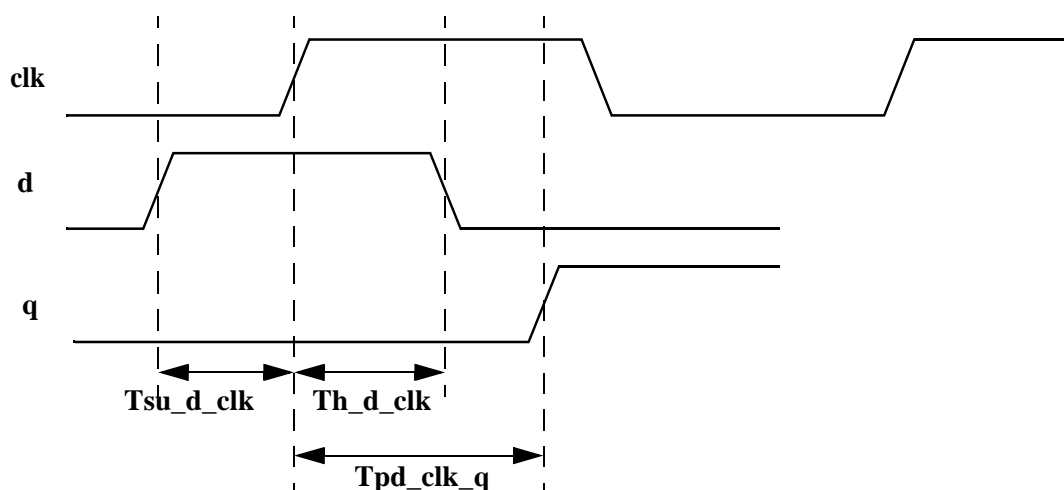
## 2.6. Modèles génériques

VHDL permet le développement de modèles dont le comportement ou la structure peuvent être paramétrés au moyen de trois mécanismes complémentaires:

- Les paramètres génériques.
- Les objets d'interface non contraints.
- L'instruction concurrente **generate**.

### Paramètres génériques

Le [Code 14](#) donne l'exemple d'un modèle de flip-flop D pour lequel les caractéristiques temporelles (temps de propagation  $TPD\_CLK\_Q$ , temps de prépositionnement (*setup time*)  $TSU\_D\_CLK$  et temps de maintien (*hold time*)  $TH\_D\_CLK$ ) sont définis comme des paramètres génériques.



**Figure 14.** Paramètres temporels du flip-flop D.

Le corps d'architecture est composé de trois processus concurrents. Le premier, nommé `behavior`, définit le comportement du composant en tenant compte du temps de propagation. Le deuxième, nommé `check_setup`, vérifie le temps de prépositionnement. Le troisième, nommé `check_hold`, vérifie le temps de maintien. Noter l'usage des attributs prédéfinis sur les signaux `'last_event` et `'delayed`.

Les valeurs effectives des paramètres seront spécifiées pour chaque instance du composant au moyen d'une spécification **generic map**.

## Modèles génériques (suite)

```

entity reg is
  port (clk, rst: in bit; d: in bit_vector; q: out bit_vector);
begin
  assert d'length = q'length;
end entity reg;

architecture bhv of reg is
begin
  process (clk, rst) is
    constant zero: bit_vector(q'length) := (others => '0');
  begin
    if rst = '1' then
      q <= zero;
    elsif clk'event and clk = '1' then
      q <= d; -- erreur si d'length /= q'length!
    end if;
  end process;
end architecture bhv;

signal data, data_reg: bit_vector(15 downto 0);

regA: entity WORK.reg(bhv) port map (data, data_reg);

```

**Code 15.** Exemple de modèle de registre de taille quelconque utilisant des signaux d'interface non contraints.

```

entity reg is
  generic (width: positive := 4);
  port (clk, rst: in bit;
        d: in bit_vector(width-1 downto 0);
        q: out bit_vector(width-1 downto 0));
end entity reg;

constant databus_size: positive := 16;
signal data, data_reg: bit_vector(databus_size-1 downto 0);

regA: entity WORK.reg(bhv)
  generic map (width => databus_size)
  port map (data, data_reg);

```

**Code 16.** Modèle plus robuste quant aux tailles des signaux.

## Objets d'interface non contraints

Le [Code 15](#) illustre l'exemple du modèle d'un registre de taille quelconque dont les signaux d'interfaces sont déclarés comme des tableaux non contraints. La déclaration d'entité possède une partie d'instruction qui vérifie que les tailles des signaux D et Q sont les mêmes, sinon il y aura une erreur au moment où une instance du composant tentera d'assigner la valeur du signal D au signal Q. L'avantage de mettre ce test dans la déclaration d'entité est de le rendre effectif quelle que soit l'architecture utilisée.

Le corps d'architecture définit un reset asynchrone. Dès que le signal RST est actif, le registre est mis à zéro. La taille de la constante ZERO est paramétrée en fonction de la taille du signal Q. Noter l'usage de l'agrégat avec le mot clé **others** pour initialiser la constante sans devoir connaître sa taille exacte.

Une instance du composant REG doit nécessairement spécifier les tailles des ports effectifs. Les déclarations des signaux associés aux ports fixent automatiquement ces tailles. Si les tailles ne sont pas les mêmes dans ce cas, une erreur sera reportée par le modèle.

Le [Code 16](#) donne une version plus robuste de la déclaration d'entité. La définition du paramètre générique WIDTH permet d'assurer automatiquement la contrainte sur les tailles des signaux D et Q. Les déclarations de signaux d'interface spécifient des tailles identiques. Le corps d'architecture peut rester identique à celui du [Code 15](#). Par contre, chaque instance de composant REG qui définit un registre de taille différente de 4 (taille par défaut) doit posséder une déclaration **generic map** pour associer la valeur effective au paramètre WIDTH.

## Modèles génériques (suite)

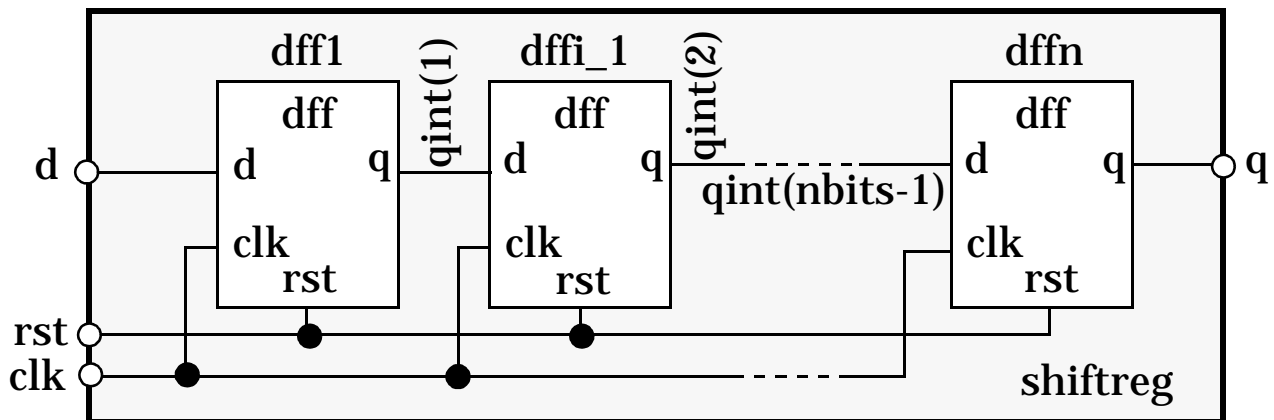


Figure 15. Registre à décalage générique.

```

entity shiftreg is
    generic (nbites: positive := 8);
    port (clk, rst, d: in bit; q: out bit);
end entity shiftreg;

architecture str of shiftreg is

    component dff is
        port (clk, rst, d: in bit; q: out bit);
    end component dff;

    signal qint: bit_vector(1 to nbites-1);
begin
    cell_array: for i in 1 to nbites generate

        first_cell: if i = 1 generate -- première cellule
            dff1: dff port map (clk, rst, d, qint(1));
        end generate first_cell;

        -- cellules internes
        int_cell: if i > 1 and i < nbites generate
            dffi: dff port map (clk, rst, qint(i-1), qint(i));
        end generate int_cell;

        last_cell: if i = nbites generate -- dernière cellule
            dffn: dff port map (clk, rst, qint(nbites-1), q);
        end generate last_cell;

    end generate cell_array;
end architecture str;

```

Code 17. Modèle générique de registre à décalage utilisant les deux formes de l'instruction `generate`.

## Instruction generate

L'instruction concurrente **generate** permet de dupliquer des instructions concurrentes de manière itérative ou conditionnelle. La [Figure 15](#) illustre la structure d'un registre à décalage générique de N bits. Le [Code 17](#) donne le modèle VHDL générique correspondant.

- La déclaration d'entité déclare le paramètre générique NBITS dont la valeur par défaut est 8.
- Le corps d'architecture déclare le composant DFF à dupliquer et un signal interne QINT qui permettra de connecter les instances internes entre elles.
- Une boucle de génération itérative globale (étiquette `cell_array`) va générer le nombre d'instances nécessaire. La variable d'indice `i` n'a pas besoin d'être déclarée.
- Les générations de la première, des instances internes et de la dernière instance requièrent des traitements spécifiques, des générations conditionnelles, pour tenir compte des associations de ports différentes.
- Les étiquettes des instructions **generate** sont obligatoires.

**NOTE:** VHDL-93 permet les déclarations locales à une instruction generate. Pour l'exemple du [Code 17](#) cela permettrait de remplacer la déclaration du signal QINT de type tableau au niveau de l'architecture par la déclaration d'un signal QINT de type `bit` et ainsi de modifier les associations de ports.

L'exemple ne duplique que des instances de composants, mais l'instruction **generate** peut générer n'importe quelle instruction concurrente.

La duplication des instructions concurrentes a lieu durant la phase d'élaboration avant de commencer la simulation proprement dite du modèle.

## Processus et signaux

```

signal S: bit;

process
begin
  wait for 5 ns;
  S <= '0', '1' after 10 ns, '0' after 18 ns, '1' after 25 ns;
  wait;
end process;

```

Code 18. Assignation d'une forme d'onde à un signal.

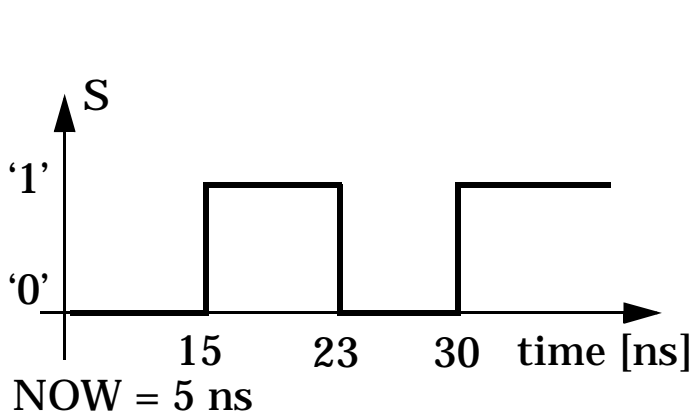


Figure 16. Forme d'onde résultant de l'assignation du Code 18 en supposant que l'assignation a lieu au temps NOW = 5 ns.

		transactions		
valeur courante				
S:	'0'	15	23	30
	'1'			

Figure 17. Pilote du signal S.

- **Un signal ne prend jamais sa nouvelle valeur immédiatement**
- **Types de délais:**
  - **Délai delta (assignation à délai nul)**
  - **Délai inertiel**
  - **Délai de transport**

## 2.7. Concurrency et modélisation du temps

### Processus et signaux

Les signaux sont les objets nécessaires à la communication entre processus et entre entités de conception. Ils représentent une abstraction de formes d'ondes temporelles dont les valeurs sont d'un type donné.

Un signal possède une structure de donnée complexe appelée un *pilote* (*driver*) qui stocke la valeur courante du signal et éventuellement ses valeurs futures. Le [Code 18](#) illustre l'assignation d'une forme d'onde à un signal S. La forme d'onde spécifie un certain nombre de *transactions* ordonnées dans le temps qui constituent la *forme d'onde projetée* (*projected waveform*). Tous les délais sont relatifs au temps auquel l'instruction est exécutée.

La [Figure 16](#) montre la forme d'onde résultante de l'assignation en supposant que l'assignation a lieu au temps  $NOW = 5$  ns.

La [Figure 17](#) montre la structure du pilote du signal S. Le pilote définit une *source* pour le signal. Un processus contenant plusieurs instructions d'assignation de signal définit des pilotes (ou des sources) distincts pour chaque signal assigné. Un signal normal ne peut pas posséder plus d'une source. L'usage de signaux multi-sources est cependant nécessaire pour modéliser des bus ou des fonctions ET et OU câblées. Ceci est possible en VHDL grâce à un type particulier de signal appelé *signal résolu* (*resolved signal*).

Un signal ne prend jamais une nouvelle valeur immédiatement, même si l'instruction d'assignation ne spécifie pas de clause de délai (clause **after**). L'assignation définit une transaction qui sera exécutée une fois que le processus est suspendu par une instruction **wait**. L'exécution d'une transaction induisant un changement de valeur du signal définit un *événement* (*event*) sur le signal. Un événement est toujours le résultat d'une transaction. Une transaction ne cause pas forcément un événement.

VHDL supporte trois types de délais:

- Le *délai delta* (*delta delay*) modélise une assignation de signal à délai nul. Il est essentiel pour garantir des résultats de simulation corrects quel que soit l'ordre d'exécution des processus.
- Le *délai inertiel* (*inertial delay*) modélise un temps de réaction non nul. Tout événement se passant plus rapidement que le délai inertiel est filtré et est donc annulé. C'est le mode par défaut lorsqu'une clause de délai (clause **after**) est spécifiée.
- Le *délai de transport* (*transport delay*) modélise une réponse fréquentielle infinie pour laquelle toute impulsion, quelle que soit sa durée, est transmise.

Dans le cas où l'instruction d'assignation de signal contient plusieurs éléments, comme dans le cas du [Code 18](#), le mécanisme de délai s'applique seulement sur le premier élément. Les autres éléments de la forme d'onde projetée sont traités avec un délai de transport.

## Initialisation et cycle de simulation

- **Initialisation**

- I1) Assigner les valeurs par défaut ou les valeurs initiales à toutes les variables et les signaux
- I2)  $T := 0$  ns
- I3) Exécuter tous les processus jusqu'à leur première instruction `wait`

- **Cycle de simulation**

- S1)  $T :=$  temps de la (des) prochaine(s) transaction(s)
- S2) Mise à jour des signaux (si événement)
- S3) Exécuter tous les processus sensibles aux signaux  
mis à jour jusqu'à leur prochaine instruction `wait`
- S4) Si nouvelles transactions au temps courant:  
retour en S2 (cycle delta)  
Sinon, si  $T = \text{Time'High}$  ou plus de transactions:  
STOP  
Sinon: retour en S1



## Initialisation et cycle de simulation

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

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

**NOTE:** La valeur initiale par défaut d'un objet de type T est définie comme étant la valeur correspondant à T'left.

Exemples de valeurs initiales par défaut:

```
signal S1: bit;                -- S1 = '0'  
signal S2: integer;           -- S2 = -32.... (32 bits)  
signal S3: bit_vector(0 to 7); -- S3 = "00000000"
```

Le cycle de simulation proprement dit commence par déterminer le temps de la (ou des) transactions la (les) plus proche(s). Toutes les transactions prévues à ce temps sont alors appliquées, ce qui se traduit par une mise à jour de la valeur courante de signaux. Une transaction aboutissant à un changement de valeur d'un signal définit un événement sur ce signal. Tous les processus sensibles aux signaux qui ont subi un événement sont ensuite réactivés et leurs instructions exécutées jusqu'à la prochaine instruction **wait**. Cette étape génère normalement de nouvelles transactions sur des signaux, soit au temps de simulation courant (délai delta), soit à un temps futur. Si ce n'est pas le cas, ou si l'on a atteint la valeur maximum du temps représentable pour la base de temps utilisée (voir la Table 2), la simulation stoppe.

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

## Délai delta

```

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

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

```

Code 19. Modèle comportemental à délais nuls du circuit de la [Figure 4](#).

- **Etat initial:** A = '0', B = '0', C,D,E = '1' et Z = '0'
- **A = '1' @ 10 ns**
- **P1:** C <= '1' nand '0' => transaction ('1' @<sup>1</sup>1) dans P1\_C  
**P2:** D <= '1' nand '1' => transaction ('0' @<sup>1</sup>1) dans P2\_D
- **Délai  $\partial 1$**   
 Événement sur P2\_D => D = '0'
- **P4:** Z <= '0' nand '1', => ('1' @<sup>1</sup>2) dans P4\_Z
- **Délai  $\partial 2$**   
 Événement sur P4\_Z => Z = '1'
- **Pas d'autre processus sensible sur Z => état stable atteint**
- **Prochain cycle de simulation à T > 10 ns sur un événement sur A ou B**

## Délai delta

Pour illustrer le fonctionnement du délai delta (*delta cycle*), considérons à nouveau le circuit logique de la [Figure 4](#) réalisant une fonction OU exclusif. Le [Code 19](#) donne le modèle VHDL correspondant. Ce modèle définit quatre processus concurrents P1, P2, P3 et P4, chacun d'eux assignant une valeur à un seul signal. On aura ainsi après élaboration six pilotes créés que l'on note P\_A, P\_B, P1\_C, P2\_D, P3\_E et P4\_Z.

À  $T = 0$  ns les signaux ont les valeurs suivantes<sup>1</sup>:

A	B	C	D	E	Z
'0'	'0'	'1'	'1'	'1'	'0'

Supposons que le signal A prenne la valeur '1' à  $T = 10$  ns. Une transaction ('1' @10 ns) a ainsi été prévue dans le pilote du signal A. Le cycle de simulation effectuée alors les actions suivantes à  $T = 10$  ns:

- Le signal A est mis à jour car la transaction produit un changement d'état (un événement).
- Processus sensibles à l'événement sur A: P1 et P2.
- Délai delta  $\delta 0$   
 Exécution du processus P1:  $C \leq '1' \text{ nand } '0'$ , d'où une transaction ('1' @ $\delta 1$ ) dans P1\_C.  
 Exécution du processus P2:  $D \leq '1' \text{ nand } '1'$  (valeur *courante* de C), d'où une transaction ('0' @ $\delta 1$ ) dans P2\_D.
- Nouveau cycle de simulation à  $T = 10$  ns (délai delta  $\delta 1$ )
- Mise à jour des signaux  
 La transaction ('1' @ $\delta 1$ ) dans P1\_C n'abouti pas à un événement.  
 La transaction ('0' @ $\delta 1$ ) dans P2\_D abouti à un événement.  $D = '0'$ .
- Processus sensible à l'événement sur P2\_D: P4.
- Délai delta  $\delta 1$   
 Exécution du processus P4:  $Z \leq '0' \text{ nand } '1'$ , d'où une transaction ('1' @ $\delta 2$ ) dans P4\_Z.
- Nouveau cycle de simulation à  $T = 10$  ns (délai delta  $\delta 2$ )
- Mise à jour des signaux  
 La transaction ('1' @ $\delta 2$ ) dans P4\_Z abouti à un événement.  $Z = '1'$ .
- Aucun processus sensible à un événement sur Z. L'état du modèle est stable. Le cycle de simulation ne redémarrera qu'à l'occasion d'un événement sur l'un des signaux d'entrée A ou B et ceci pour un temps  $T > 10$  ns.

---

1. Ces valeurs sont aussi obtenues après plusieurs cycles delta. On peut s'en convaincre à titre d'exercice.

## Délais inertiel et transport

```

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

```

Code 20. Exemple d'utilisation du délai inertiel.

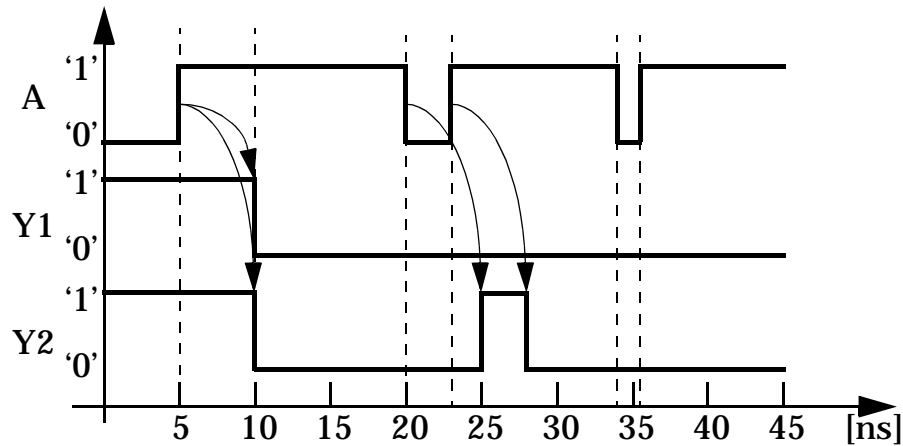


Figure 18. Résultats des assignations du Code 20.

```

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

```

Code 21. Exemple d'utilisation du délai transport.

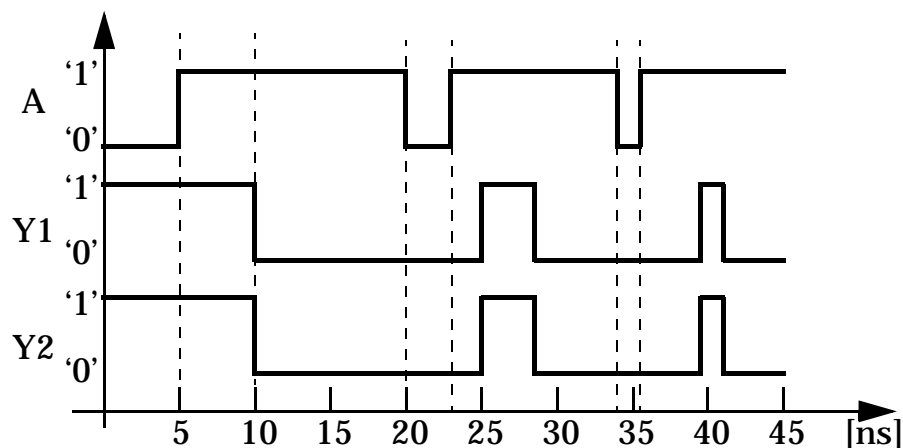


Figure 19. Résultats des assignations du Code 21.

## Délai inertiel

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

Le délai inertiel est le mécanisme par défaut en VHDL pour une assignation de signal du type:

```
S <= valeur after délai;
```

**VHDL-93:** VHDL-93 permet de spécifier le mot-clé **inertial**:

```
S <= inertial valeur after délai;
```

VHDL-93 permet aussi de spécifier une valeur de rejection plus petite que le délai inertiel:

```
S <= reject valeur-rejection inertial valeur after délai;
```

La durée de la valeur de rejection doit être positive et plus petite ou égale au délai inertiel.

Le [Code 20](#) donne un extrait de modèle utilisant un délai inertiel pour définir le comportement d'un inverseur. La [Figure 18](#) illustre les résultats de l'assignation des signaux Y1 et Y2. Le changement de valeur du signal A à 5 ns est propagé sur Y1 et Y2 avec un délai de 5 ns. L'impulsion sur A dès 20 ns et d'une durée de 3 ns n'est pas propagée sur Y1 car sa durée est plus petite que le délai inertiel de 5 ns. Elle est par contre propagée sur Y2 car l'assignation spécifie un délai de rejection de 2 ns. La deuxième impulsion sur A dès 34 ns et d'une durée de 1,5 ns n'est par contre propagée ni sur Y1 ni sur Y2.

## Délai transport

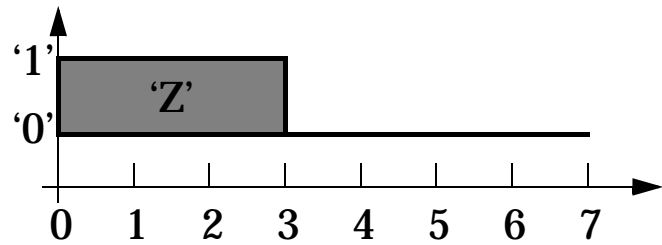
Le délai transport modélise un comportement idéal pour lequel toute impulsion, quelle que soit sa durée, est transmise.

Le [Code 21](#) donne un extrait de modèle utilisant un délai transport pour définir le comportement d'un inverseur. Il montre aussi une formulation équivalente utilisant un délai inertiel et une durée de rejection nulle. La [Figure 19](#) illustre les résultats de l'assignation des signaux Y1 et Y2. Les transactions impliquées par des changements de valeurs sur le signal A dans un délai plus petit que 5 ns sont simplement ajoutées au pilote du signal assigné.

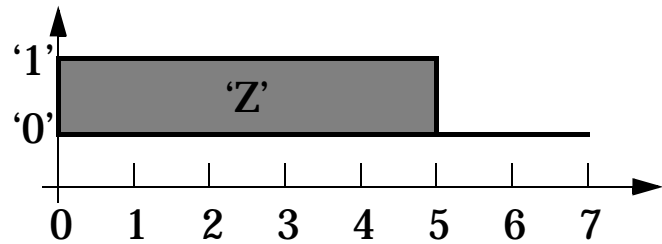
## Transactions multiples

```
type logic3 is ('Z', '0', '1');
signal X: logic3;
```

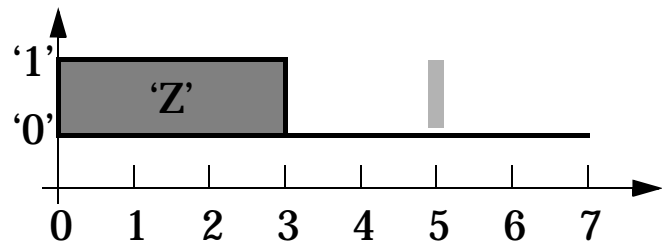
```
Cas1: process begin
  X <= '1' after 5 ns;
  X <= '0' after 3 ns;
  wait;
end process Cas1;
```



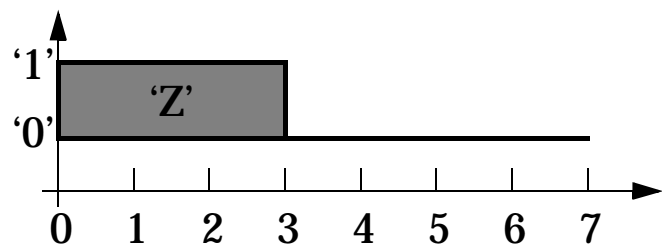
```
Cas2: process begin
  X <= '1' after 3 ns;
  X <= '0' after 5 ns;
  wait;
end process Cas2;
```



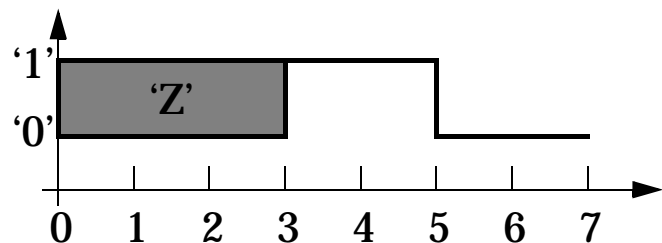
```
Cas3: process begin
  X <= '0' after 3 ns;
  X <= '0' after 5 ns;
  wait;
end process Cas3;
```



```
Cas4: process begin
  X <= transport '1' after 5 ns;
  X <= transport '0' after 3 ns;
  wait;
end process Cas4;
```



```
Cas5: process begin
  X <= transport '1' after 3 ns;
  X <= transport '0' after 5 ns;
  wait;
end process Cas5;
```



## Transactions multiples

Des transactions multiples sur un même signal peuvent exister si plusieurs assignations sont définies dans le corps d'un processus. L'ordre d'assignation est dans ce cas important et selon le type de délai on obtient des contenus de pilotes différents et éventuellement des formes d'ondes différentes. On distingue cinq cas:

- Cas 1: Délai inertiel  
La première transaction ('1', 5 ns) est supprimée car la seconde transaction ('0', 3 ns) est prévue avant.
- Cas 2: Délai inertiel  
La première transaction ('1', 3 ns) est supprimée car la seconde transaction ('0', 5 ns) est prévue plus tard et avec une valeur différente.
- Cas 3: Délai inertiel  
Les deux transactions ('0', 3 ns) et ('0', 5 ns) sont conservées. La seconde transaction ne crée pas d'événement.
- Cas 4: Délai transport  
La première transaction ('1', 5 ns) est supprimée car la seconde transaction ('0', 3 ns) est prévue avant.
- Cas 5: Délai transport  
La seconde transaction ('0', 5 ns) est ajoutée dans le pilote car elle est prévue après la première transaction.

La Table 3 résume les différents cas de transactions multiples possibles.

délai	Inertiel	Transport
Nouvelle transaction AVANT une transaction existante	La transaction existante est supprimée du pilote et remplacée par la nouvelle transaction	La transaction existante est supprimée du pilote et remplacée par la nouvelle transaction
Nouvelle transaction APRES une transaction existante	La transaction existante est supprimée du pilote si les valeurs sont différentes, autrement elles sont les deux conservées	La nouvelle transaction est ajoutée dans le pilote après la transaction existante

**Table 3:** Résultats de transactions multiples sur un même signal en fonction du mode de délai.

## Signaux résolus

```

signal S: logic4;
...
P1: process begin
    wait for 10 ns;
    S <= '1';
    wait for 20 ns;
    S <= '0';
end process P1;
P2: process begin
    wait for 20 ns;
    S <= '0';
    wait for 20 ns;
    S <= '1';
end process P2;

```

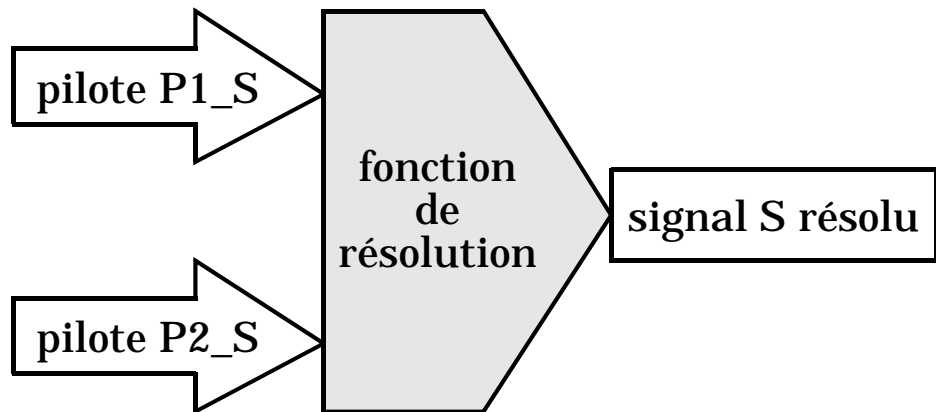


Figure 20. Signal multi-source et fonction de résolution.

```

package logic4_pkg is
    type ulogic4 is ('X', '0', '1', 'Z'); -- type non résolu
    type ulogic4_vector is array (natural range <>) of ulogic4;

    function resolve (sources: ulogic4_vector) return ulogic4;

    subtype logic4 is resolve ulogic4;
    type logic4_vector is array (natural range <>) of logic4;
end package logic4_pkg;

package body logic4_pkg is
    type table is array (ulogic4, ulogic4) of ulogic4;
    constant resolution_table: table :=
        -- 'X' '0' '1' 'Z'
        (( 'X', 'X', 'X', 'X' ), -- 'X'
         ( 'X', '0', 'X', '0' ), -- '0'
         ( 'X', 'X', '1', '1' ), -- '1'
         ( 'X', '0', '1', 'Z' )); -- 'Z'

    function resolve (sources: ulogic4_vector) return ulogic4 is
        variable result: ulogic4 := 'Z';
    begin
        for i in sources'range loop
            result := resolution_table(result, sources(i));
        end loop;
        return result;
    end function resolve;

end package body logic4_pkg;

```

Code 22. Exemple de déclarations pour un type résolu à 4 états.



## Signaux résolus

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

La [Figure 20](#) illustre le cas d'un signal S assigné de deux processus P1 et P2. Pour chaque processus est défini un pilote pour le signal: P1\_S et P2\_S. Une *fonction de résolution* (*resolution function*) est alors utilisée pour calculer la valeur finale du signal S. Le signal S est déclaré comme du type `logic4` qui est un type, ou plutôt un sous-type, résolu.

Le [Code 22](#) donne les déclarations nécessaires à la définition du sous-type résolu `logic4`. Ces déclarations sont groupées dans un paquetage dont le contenu devra être rendu visible par une clause de contexte appropriée, p. ex.:

```
use WORK.logic4_pkg.all;
```

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

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

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

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

Le type non résolu `ulogic4` et le sous-type résolu `logic4` sont compatibles, c'est-à-dire qu'il est légal d'assigner des signaux de ces deux types entre eux:

```
signal S1: ulogic4;
signal S2: logic4;
S2 <= S1; -- légal
```

Par contre, les types tableau `ulogic4_vector` et `logic4_vector` sont deux types différents et ne sont pas compatibles. Il s'agit d'utiliser une conversion de type si l'on veut assigner un type de signal à l'autre:

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

Le paquetage standard IEEE `STD_LOGIC_1164` définit un type logique à 9 états en version non résolue et résolue. Il est recommandé d'utiliser ces types pour favoriser l'échange de modèles de sources différentes. L'[Annexe B](#) décrit le contenu de ce paquetage.

## VHDL pour la synthèse

- **Toutes les instructions VHDL n'ont pas forcément un sens pour la synthèse**
- **Les outils de synthèse du marché reconnaissent des “motifs” particuliers, mais pas forcément les mêmes!**
- **Types supportés:**

- **Types énumérés:** bit, character, boolean, std\_logic

```
type State is (Idle, Init, Shift, Add, Check);
-- encodage:  000  001  010  011  100  (défaut)
```

- **Types entiers:** integer, natural, positive

```
subtype my_byte is integer range -128 to 127; -- 8 bits
```

- **Tableaux (1 dimension):** bit\_vector, std\_logic\_vector  
“Paquets de bits”, pas de MSB/LSB

- **Tableaux (2 dimensions):**

```
subtype word is bit_vector(31 downto 0);
type RAM is array (1023 downto 0) of word;
```

- **Enregistrements: pas de signification matérielle particulière**
- **Types STD\_LOGIC\_1164:**

```
std_ulogic      et      std_logic
std_ulogic_vector et    std_logic_vector
```

### Valeurs logiques:

'0', 'L'	niveau logique bas
'1', 'H'	niveau logique haut
'U', 'X', 'W', '-'	valeurs métalogiques
'Z'	valeur haute impédance

## 2.8. VHDL pour la synthèse

Le langage VHDL n'a pas été conçu dès le départ pour supporter une sémantique de synthèse. Il est toutefois possible de définir un sous-ensemble du langage qui se traduit naturellement en éléments matériels. Le sous-ensemble décrit ici est valable pour la synthèse logique. Il reste encore variable selon les outils de synthèse du marché. Un standard IEEE (IEEE 1076.3) définissant des paquetages standard pour la synthèse est disponible depuis peu [STD1076.3] et un deuxième standard IEEE (IEEE 1076.6) définissant un sous-ensemble synthétisable du langage vient de voir le jour [STD1076.6] et devrait s'imposer pour tous les outils du marché.

Références: [Airi94] [Ott94] [Bhas96] [Pick96] [Rush96] [Rush98].

### *Types (et sous-types) supportés*

**Types énumérés.** Par exemple: `bit`, `boolean`, `std_logic`. Il n'existe pas d'encodage standard pour ces types, mais l'usage est d'utiliser un vecteur de bits de longueur minimum pour représenter tous les cas possibles en commençant par l'indice zéro. D'autres schémas d'encodage (p.ex. one-hot, Gray, Johnson) sont possibles, mais leur spécification dépend de l'outil de synthèse.

**Types entiers.** Par exemple: `integer`, `natural`, `positive`. Les entiers négatifs sont usuellement encodés comme des valeurs binaires en complément à deux.

**NOTE:** Il est hautement recommandé de définir la dynamique réelle de l'entier sous peine de synthétiser des mots de 32 bits.

Il est aussi recommandé de définir des sous-types plutôt que des types car les objets déclarés avec sous-types d'un même type de base restent compatibles.

**Tableaux monodimensionnels.** Par exemple: `bit_vector`, `std_logic_vector`. Le type de l'indice doit être entier. Il n'y a pas de significations MSB et LSB (*Most Significant Bit*, *Less Significant Bit*) associées au tableau. Les types non contraints (dont les indices sont des paramètres génériques) sont supportés.

**NOTE:** Les tableaux bidimensionnels sont supportés s'ils sont déclarés comme des tableaux de tableaux monodimensionnels.

**Types enregistrements.** Ils permettent de grouper des objets sans avoir de signification matérielle spécifique. Il est par contre évident que les types des champs doivent être supportés pour la synthèse.

**Types du paquetage standard STD\_LOGIC\_1164.** Seul un sous-ensemble des 9 états logiques ont une signification pour la synthèse.

Les valeurs logiques '0' et 'L' (respectivement '1' et 'H') sont interprétées comme représentant le niveau logique bas (respectivement haut). Chacun des deux niveaux représente à son tour une gamme de tensions distinctes dans le circuit synthétisé (par exemple, 0,1-0.5V pour le niveau bas et 4,5-5V pour le niveau haut).

Les valeurs 'U', 'X', 'W' et '-' sont appelées *valeurs métalogiques*. Elles définissent le comportement du modèle en simulation plutôt que celui du circuit synthétisé. Elles ne devraient pas être utilisées pour la synthèse. La valeur '-' est aussi appelé *valeur indifférente* (*don't care value*). Cette valeur est utile en simulation. La prise en compte de l'état indifférent en synthèse doit utiliser d'autres techniques qui seront vues plus loin.

La valeur 'Z' est appelée *valeur haute impédance*. Elle représente la valeur d'une source d'un signal lorsque cette source ne contribue pas à la valeur effective (résolue) du signal. L'usage de cette valeur permet d'inférer des buffers à trois états (*three-state buffers*).

## VHDL pour la synthèse

- Types supportés (suite):
  - Types tableau représentant un entier non signé (unsigned) ou signé (signed)

```
-- pour utiliser ces types:
library IEEE;
use IEEE.std_logic_1164.all; -- optionnel
use IEEE.std_logic_arith.all;

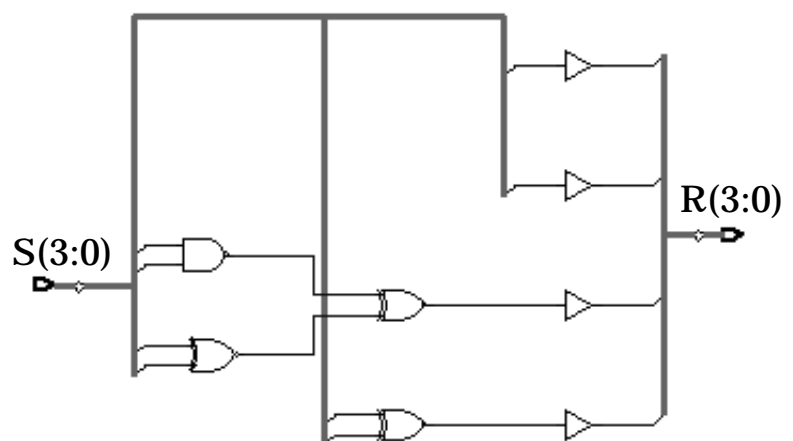
type unsigned is array (natural range <>) of std_logic;
type signed   is array (natural range <>) of std_logic;
```

- Objets supportés
  - Constantes
    - Usage typique: table de vérité, ROM

```
package cst_pkg is
  subtype int16 is integer range 0 to 15;
end;

use WORK.cst_pkg.all;
entity cst is
  port (S: in int16; R:out int16);
end cst;

architecture a of cst is
  constant K: int16 := 5;
begin
  R <= S * K;
end a;
```



**Figure 21.** Exemple d'usage de constante.  
Le circuit inféré ne contient pas de multiplieur.

**Types unsigned et signed.** Ces types représentent respectivement un entier non signé et un entier signé sous la forme d'un tableau unidimensionnel. L'Annexe C donne le contenu de la déclaration du paquetage STD\_LOGIC\_ARITH qui définit ces types ainsi que les opérateurs arithmétiques et relationnels associés. Le paquetage définit aussi des fonctions de conversion entre types.

**NOTE:** L'avantage d'utiliser ces types plutôt que le type `integer` est de pouvoir accéder explicitement à chaque bit de la représentation de l'entier.

**NOTE:** Le paquetage STD\_LOGIC\_ARITH considéré ici est propre à l'environnement Synopsys. Son contenu peut être légèrement différent d'un outil à l'autre. Un paquetage standard IEEE définissant ses propres types `unsigned` et `signed` est disponible depuis peu [STD1076.3].

### Objets supportés

**Constantes.** Une constante n'infère pas de matériel en synthèse et son usage est particulièrement recommandé pour obtenir un circuit optimisé.

La Figure 21 donne un exemple d'utilisation d'une constante. L'opérateur multiplication du modèle ne conduit pas à l'utilisation d'un multiplieur car la valeur de la constante est propagée à l'élaboration dans l'instruction d'assignation du signal R.

Un usage typique de constantes est la déclaration d'une table de vérité ou de contenu d'une mémoire ROM. Le Code 23 donne un exemple d'utilisation d'une table de vérité.

```

package rom_pkg is
    subtype t_word is bit_vector(1 to 2);
    subtype t_address is natural range 0 to 7;
    type t_rom is array (t_address) of t_word;
end rom_pkg;

use WORK.rom_pkg.all;
entity rom is
    port (A: in t_address; S: out t_word);
end;

architecture a of rom is
    constant ROM: t_rom := (
        0          => "00",
        1 | 2      => "10",
        3 | 5 | 6  => "01",
        4          => "10",
        7          => "11");
begin
    S <= ROM(A);
end a;

```

**Code 23.** Exemple d'utilisation d'une constante pour définir la table de vérité d'un additionneur 1 bit complet.

## VHDL pour la synthèse

- Objets supportés (suite)
  - Variables (locales à un processus ou à un sous-programme)

```

entity var is
  port (A, B: in bit; R: out bit);
end;
architecture a of var is
begin
  process
  begin
    variable V: bit;
  begin
    V := A xor B;
    R <= V;
  end process;
end;

```



Figure 22. Exemple d'utilisation d'une variable.

- Signaux (visibilité globale)

```

architecture a of rom is
  constant ROM: t_rom := (
    0      => "00",
    1 | 2  => "10",
    3 | 5 | 6 => "01",
    4      => "10",
    7      => "11");
begin
  S <= ROM(A);
end a;

```

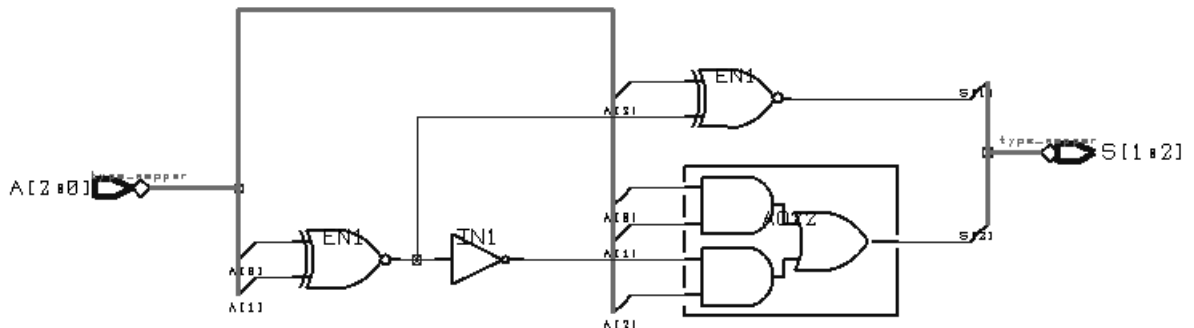


Figure 23. Exemple d'utilisation d'un signal.

**Variables et signaux.** Les variables et les signaux ont des sémantiques différentes en VHDL qui sont conservées pour la synthèse. Un objet variable ou signal peut néanmoins inférer une connexion (*wire*), un registre ou rien du tout (l'objet est éliminé au cours de l'optimisation).

Une variable ne peut être déclarée et utilisée que dans un processus ou un sous-programme. La Figure 22 illustre l'usage d'une variable V qui est éliminée lors de l'optimisation. Le circuit se réduit à une simple porte XOR.

Un signal a une visibilité plus globale. La Figure 23 illustre l'usage d'un signal dont le modèle complet est donné dans le Code 23, L'assignation concurrente au signal S infère un circuit combinatoire produisant un mot de 2 bits en fonction de l'adresse A et du contenu de la "ROM".

Pour illustrer la différence entre une variable et un signal, considérons l'exemple d'un simple registre à décalage (Code 24). L'architecture GOOD utilise des signaux et infère correctement un registre à décalage. L'architecture BAD utilise une variable et ne décrit plus le comportement voulu. L'architecture GOOD2 utilise une variable mais synthétise correctement le registre.

```

entity shiftreg is
  port (CLK, DIN: in bit; DOUT: out bit);
end shiftreg;

architecture good of shiftreg is
  signal SINT: bit;
begin
  process
  begin
    wait until CLK = '1';
    SINT <= DIN;
    DOUT <= SINT;
  end process;
end good;

architecture good2 of shiftreg is
begin
  process
  variable VINT: bit;
  begin
    wait until CLK = '1';
    DOUT <= VINT;
    VINT := DIN;
  end process;
end good2;

architecture bad of shiftreg is
begin
  process
  variable VINT: bit;
  begin
    wait until CLK = '1';
    VINT := DIN;
    DOUT <= VINT;
  end process;
end bad;

```

Code 24. Synthèse d'un registre à décalage.



## VHDL pour la synthèse

- Valeurs initiales pas supportées en synthèse  
=> SET/RESET explicite pour variables et signaux

```

entity E is
  port ( ...; RESET: in bit; ... );
end E;

architecture Sync of E is
  signal S: bit_vector(15 downto 0);
begin
  process
  begin
    wait until CLK = '1';
    if RESET = '1' then -- reset synchrone
      S <= (others => '0');
      -- + autres initialisations
    else
      -- comportement normal...
    end if;
  end process;
end Sync;

```

**Code 25.** Reset synchrone.

```

architecture Async of E is
  signal S: bit_vector(15 downto 0);
begin
  process (CLK, RESET)
  begin
    if RESET = '1' then -- reset asynchrone
      S <= (others => '0');
      -- + autres initialisations
    elsif CLK = '1' and CLK'event then
      -- comportement normal synchrone...
    end if;
  end process;
end Async;

```

**Code 26.** Reset asynchrone.



### Valeurs initiales

Tout objet VHDL possède une valeur initiale, soit par défaut car héritée de la définition de type ou de sous-type auquel l'objet appartient, soit explicite au moyen d'une expression spécifiée au moment de la déclaration. Aucune de ces deux méthodes n'est supportée pour la synthèse. Il s'agit ainsi de prévoir dans le modèle VHDL à synthétiser un comportement de type set/reset explicite.

Un comportement de *reset synchrone* peut être décrit comme mentionné dans l'architecture Sync du [Code 25](#). Le signal RESET est testé lors d'un flanc montant du signal CLK. E est une déclaration d'entité quelconque qui déclare le signal d'interface RESET.

Un comportement de *reset asynchrone* peut être décrit comme mentionné dans l'architecture Async du [Code 26](#). Le signal RESET est testé indépendamment du signal CLK.

### Opérateurs

VHDL possède un certain nombre d'opérateurs pour les types énumérés, numériques et logiques prédéfinis qui sont tous supportés pour la synthèse, avec quelques restrictions parfois. La [Figure 24](#) récapitule ces opérateurs par ordre de priorité croissante.

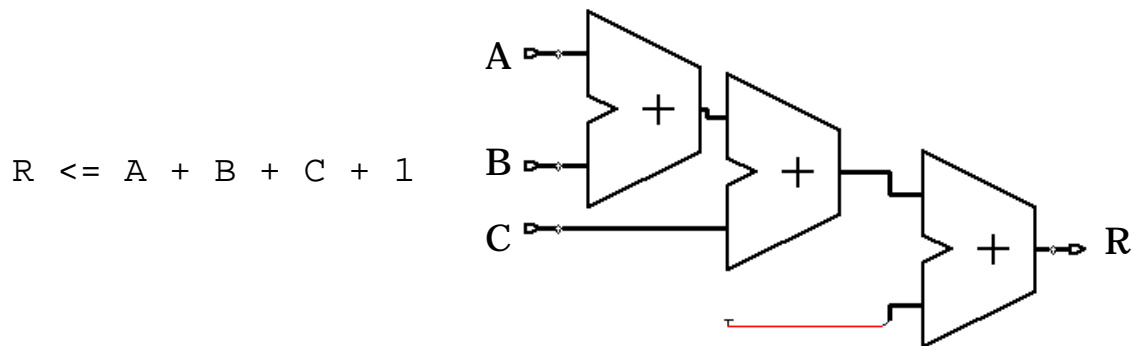
Type	Opérateurs
Logique	<b>or, and, nor, nand, xor, not, xnor</b>
Relationnel <sup>a</sup>	<b>=, /=, &lt;<sup>b</sup>, &gt;Annexe b, &gt;=Annexe b, &lt;=Annexe b</b>
Décalage <sup>c</sup>	<b>sll, srl, sla, sra, rol, ror</b>
Addition	<b>+Annexe b, -Annexe b, &amp;<sup>d</sup></b>
Unaires	<b>+, -</b>
Multiplication	<b>*Annexe b<sup>e, f</sup> /<sup>g, h</sup>, modAnnexe g, remAnnexe g</b>
Autres	<b>**<sup>i</sup> abs, not</b>

- Résultat de type Boolean
- Peut être partagé avec un autre opérateur de même type et de même niveau de priorité
- Introduits depuis VHDL-93; chaque outil de synthèse peut en proposer une version propriétaire
- L'opérateur & (concaténation) peut être utilisé pour émuler les opérateurs de décalage
- Infère en général un circuit combinatoire; le mécanisme dépend de l'outil de synthèse (p. ex. DesignWare de Synopsys)
- Dans le cas où l'opérande de droite est un multiple de 2, le circuit inféré est un ensemble de buffers réalisant un simple décalage à gauche des bits de l'opérande de gauche
- L'opérande de droite doit être positif et égal à une puissance de 2
- Dans le cas où l'opérande de droite est un multiple de 2, le circuit inféré est un ensemble de buffers réalisant un simple décalage à droite des bits de l'opérande de gauche
- L'opérande élevé à la puissance doit être une constante égale à 2

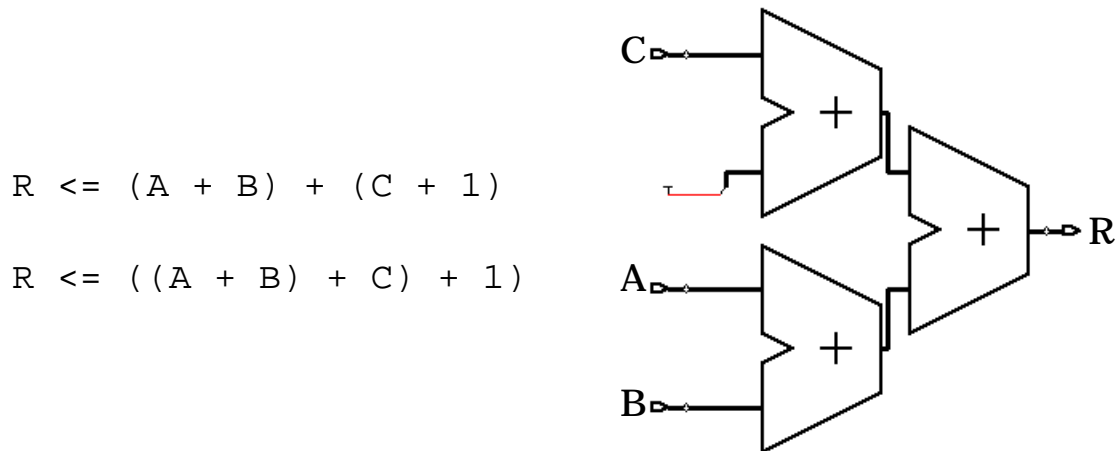
**Figure 24.** Support pour la synthèse des opérateurs VHDL prédéfinis (par ordre de priorité croissante)

## VHDL pour la synthèse

- Groupement d'opérateurs

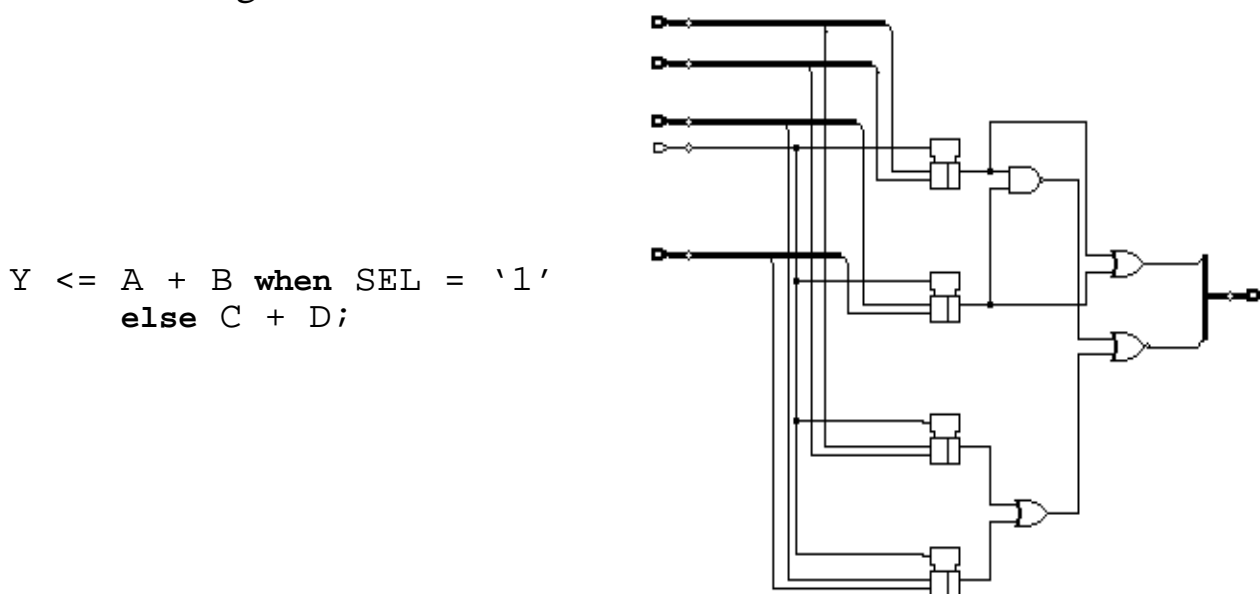


**Figure 25.** Structure inferée sans groupement d'opérateurs.



**Figure 26.** Structure inferée avec groupement d'opérateurs.

- Partage de ressources



**Figure 27.** Un seul additionneur inferé avec entrées multiplexées.

### Groupement d'opérateurs

Le groupement des opérateurs arithmétiques de même priorité au moyen de parenthèses permet d'inférer des circuits remplissant la même fonction, mais caractérisés par des performances différentes.

La [Figure 25](#) illustre la structure inférée par la synthèse d'une expression sans groupement particulier d'opérateurs. Dans ce cas les priorités par défaut sont appliquées.

**NOTE:** La structure illustrée ici est obtenue après la phase d'élaboration, avant l'assignation de cellules standard d'une technologie donnée.

La [Figure 26](#) illustre la structure inférée par une expression parenthésée. La structure obtenue est plus parallèle et ne possède que deux niveaux de logique au lieu de trois.

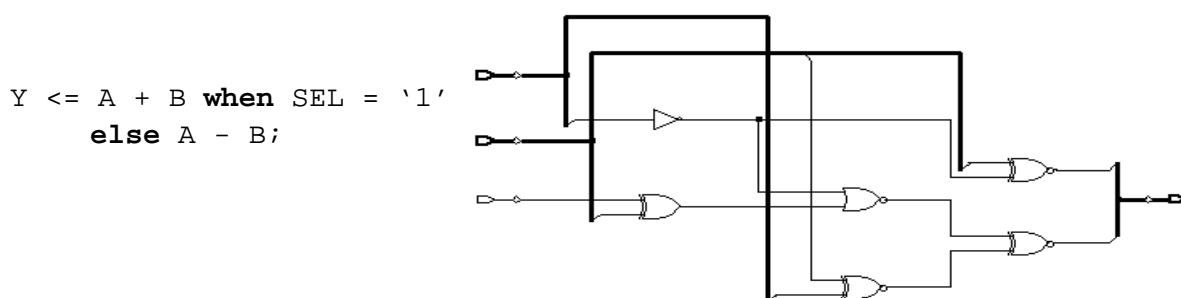
### Partage de ressources

Bien que ce soit plutôt une tâche pour des outils de synthèse architecturale, les outils de synthèse logique sont capables de minimiser les ressources (opérateurs) utilisées en les partageant sur plusieurs instructions. Ceci peut être réalisé par une technique de modélisation spécifique.

**NOTE:** Les opérateurs "+", "-", "\*" et "/" peuvent être potentiellement partagés. En pratique, seuls les opérateurs d'addition et de soustraction peuvent l'être de manière efficace avec un outil de synthèse logique.

La [Figure 27](#) illustre le partage d'un seul additionneur pour deux instructions. Ceci est possible car les opérandes ne sont pas les mêmes. En fait, il suffirait qu'un seul opérande soit différent pour permettre le partage. Des multiplexeurs sont inférés, ce qui augmente le délai des chemins de données, au bénéfice d'une surface plus petite cependant.

La [Figure 28](#) illustre un autre cas de partage de ressource pour lequel les opérandes sont les mêmes mais les opérateurs sont différents. Un seul additionneur-soustracteur est inféré dans ce cas, sans multiplexeurs.



**Figure 28.** Partage de ressource: un seul additionneur-soustracteur inféré sans multiplexeurs.

**NOTE:** L'usage d'opérateurs VHDL peut aboutir à un nombre très important de portes. Un exemple typique est l'opérateur "\*". D'autre part le modèle est simple, mais la réalisation n'est pas forcément efficace (circuit combinatoire inféré p. ex.).

## VHDL pour la synthèse

- Partage de ressources manuel

```
Y <= A + B when SEL = '1' else C + D;
```

```
-- instructions concurrentes
MUX1 <= A when SEL = '1' else C;
MUX2 <= B when SEL = '1' else D;
Y <= MUX1 + MUX2;

-- instructions séquentielles
if SEL = '1' then
  MUX1 := A;
  MUX2 := B;
else
  MUX1 := C;
  MUX2 := D;
end if;
Y <= MUX1 + MUX2;
```

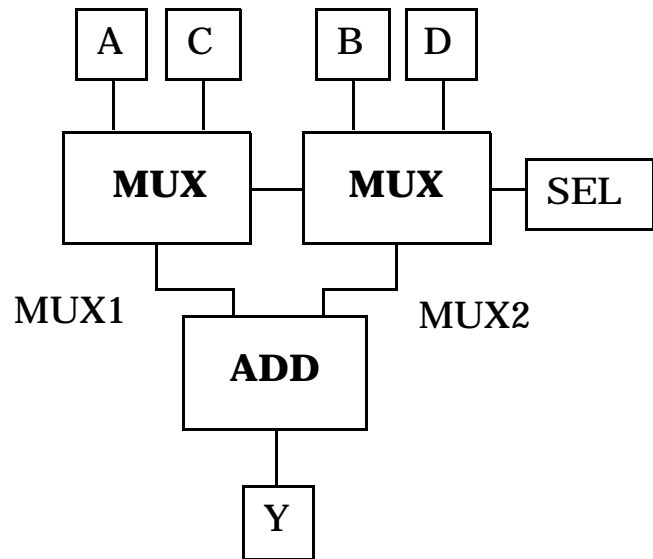


Figure 29. 1 additionneur inféré.

```
-- instructions concurrentes
SUM1 <= A + B;
SUM2 <= C + D;
Y <= SUM1 when SEL = '1' else SUM2;

-- instructions séquentielles
SUM1 := A + B;
SUM2 := C + D;
if SEL = '1' then
  Y <= SUM1;
else
  Y <= SUM2;
end if;
```

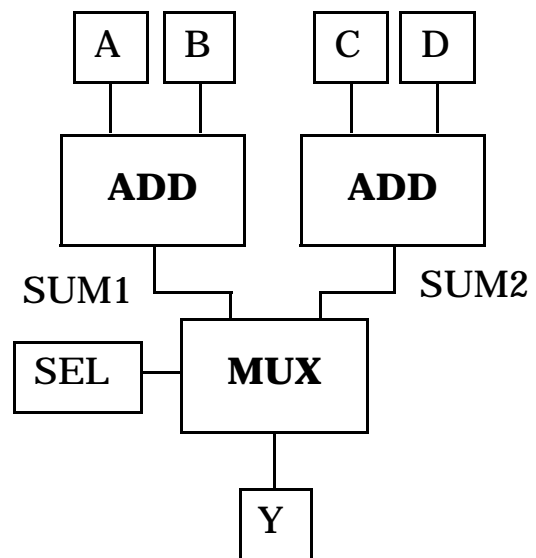


Figure 30. 2 additionneurs inférés.

Il est aussi possible de contrôler le partage des ressources manuellement. En reprenant l'exemple de la [Figure 27](#), on peut réécrire l'instruction VHDL concurrente pour faire apparaître explicitement les deux multiplexeurs et l'additionneur unique ([Figure 29](#)) ou au contraire deux additionneurs et un seul multiplexeur ([Figure 30](#)). On donne pour chaque exemple une version concurrente et séquentielle équivalentes.

**NOTE:** Le partage de ressource n'est possible que pour une instruction à la fois et dans le cas où l'instruction est contenue dans (ou est équivalente à) un seul processus.

Par exemple, le [Code 27](#) donne un exemple pour lequel le partage automatique de ressource n'est pas possible malgré le fait que les opérations soient mutuellement exclusives car les instructions concernées sont considérées comme indépendantes.

```
if SEL = '1' then
    Y <= A + B;
end if;
if SEL /= '1' then
    Y <= E + F;
end if;
```

**Code 27.** Pas de partage de ressource possible.

## VHDL pour la synthèse

- Un processus infère un circuit combinatoire ssi:
  - 1) Liste de sensibilité ou instruction wait équivalente
  - 2) Variables locales assignées avant d'être lues
  - 3) Processus sensible à tous les signaux lus
  - 4) Signaux assignés dans toutes les branches
- Si l'une des quatre règles ci-dessus n'est pas satisfaite, un circuit séquentiel est inféré (registres)
- L'horloge d'un circuit séquentiel est inférée par une forme particulière de l'instruction wait (le nom du signal n'a pas d'importance):

```
wait until CLK = '1';
wait until CLK = '1' and CLK'event;
wait until CLK = '1' and not CLK'stable;
```

- Inférence de latches ou de flip-flops

Latch	Flip-flop
<pre>process (CLK, D) begin   if CLK = '1' then     Q &lt;= D;   end if; end process;</pre>	<pre>process begin   wait until CLK = '1';   Q &lt;= D; end process;</pre>
<pre>-- Attention: différence -- entre simulation pré et -- post synthèse! process (CLK) begin   if CLK = '1' then     Q &lt;= D;   end if; end process;</pre>	<pre>process (CLK) begin   if CLK = '1' and CLK'Event then     Q &lt;= D;   end if; end process;</pre>

### Utilisation de processus et de l'instruction `wait`

L'usage de processus permet de contrôler le type de circuit inféré par la synthèse, à savoir un circuit combinatoire ou séquentiel. Le premier type de circuit est asynchrone et son comportement est uniquement dirigé par des événements sur des signaux. Le second type de circuit est synchronisé sur un signal d'horloge et implique des registres (latches ou flip-flops).

Il existe quatre conditions à remplir pour assurer qu'un processus va inférer un circuit purement combinatoire:

- 1) Le processus possède une liste de sensibilité ou une seule instruction `wait` équivalente.
- 2) Le processus ne déclare pas de variable locale ou ses variables locales sont toutes assignées avant d'être lues.
- 3) Tous les signaux lus dans le processus font partie de la liste de sensibilité ou de l'instruction `wait` équivalente.
- 4) Tous les signaux assignés dans le processus le sont dans chaque branche impliquée par une instruction conditionnelle (instruction `if`) ou de sélection (instruction `case`).

Si l'une quelconque de ces règles n'est pas satisfaite, un ou plusieurs éléments de mémoire sont inférés. Le circuit inféré est dans ce cas séquentiel synchrone. Le signal d'horloge est identifié par des instructions de formes particulières:

```
wait until CLK = '1';
wait until CLK = '1' and CLK'event;
wait until CLK = '1' and not CLK'stable;
```

**NOTE:** Le nom du signal considéré comme l'horloge n'a aucune importance.

**NOTE:** L'usage de plusieurs instructions `wait` infère une machine d'états dont les états sont implicites.

Dans le cas où des éléments de mémoire sont inférés, il est possible de contrôler le type de registre utilisé. Une latch est inférée si la synchronisation du processus est faite sur un état. Un flip-flop est inféré si la synchronisation du processus est faite sur un flanc.

L'usage de l'instruction `wait` est usuellement soumise aux contraintes suivantes:

- Il n'est pas admis de synchroniser un processus sur plusieurs flancs ou états d'horloge différents ([Code 28](#)). Il faut utiliser plusieurs processus dans ce cas.

<pre>process begin     wait until CLK = '1';     Q1 &lt;= D;     wait until CLK = '0';     Q2 &lt;= Q1; end process;</pre>	<pre>process begin     wait until CLK1 = '1';     Q1 &lt;= D;     wait until CLK2 = '1';     Q2 &lt;= Q1; end process;</pre>
--	--

**Code 28.** Types de synchronisations non supportées en synthèse.

- Toutes les formes possibles de l'instruction `wait` ne sont pas acceptées pour la synthèse. D'autres formes que celles données ci-dessus peuvent toutefois être acceptées par certains outils (p. ex. `wait on <liste-signaux>` ;).

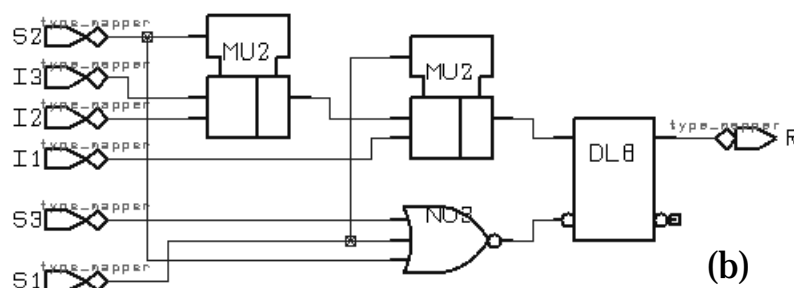
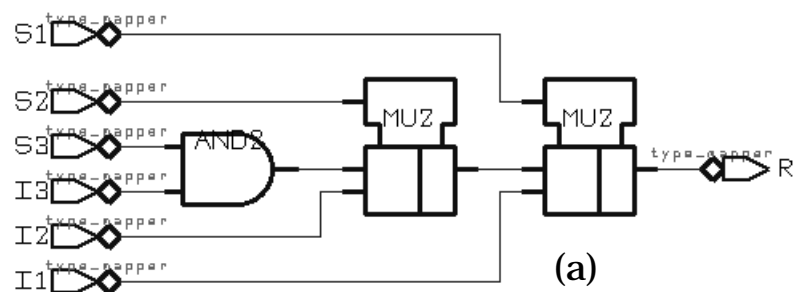
## VHDL pour la synthèse

- Instructions séquentielles supportées:
  - Instruction `wait`
  - Assignment de variable (`:=`) ou de signal (`<=`)
  - Instruction conditionnelle (`if`) et de sélection (`case`)
  - Instructions de boucle (`loop`, `for`, `while`)
  - Appels de sous-programme (procédure et fonction)
- Assignment de signal
  - Clause de délai non supportée
  - Formes d'onde à éléments multiples non supportée
- Instruction conditionnelle `if`

```

process (S1, S2, S3, I1, I2, I3)
begin
  if S1 = '1' then
    R <= I1;
  elsif S2 = '0' then
    R <= I2;
  elsif S3 = '1' then
    R <= I3;
  else
    R <= '0';
  end if;
end process;

```



**Figure 31.** Instruction conditionnelle `if`  
 (a) avec branche `else`; (b) sans branche `else`.



### Instructions séquentielles supportées

Toutes les instructions séquentielles de VHDL sont supportées pour la synthèse, avec toutefois quelques restrictions d'utilisation.

- L'instruction **wait** a déjà été discutée à la [page 71](#).
- Une clause de délai dans une assignation de signal est ignorée pour la synthèse, mais pas considérée comme un erreur:

```
S <= '0' after 10 ns; -- délai ignoré
```

L'assignation d'une forme d'onde à éléments multiples est par contre considérée comme une erreur:

```
S <= '1', '0' after 20 ns, '1' after 30 ns; -- erreur
```

- L'instruction conditionnelle **if** implique une priorité. La [Figure 31](#) donne un exemple d'usage de l'instruction conditionnelle et le circuit inféré par la synthèse. La priorité est réalisée par une des multiplexeurs en série. Dans le circuit (a) la branche **else** est présente et le circuit inféré est combinatoire. Dans le cas (b) la branche **else** est omise et le circuit inféré contient un registre (latch) pour mémoriser la valeur du signal R lorsqu'aucune des conditions n'est vraie.
- L'instruction de sélection **case** n'implique pas de priorité. Elle peut inférer un circuit combinatoire ou séquentiel selon le contexte. La [Figure 32](#) illustre un exemple d'instruction de sélection conduisant à un circuit combinatoire. L'usage d'une telle instruction pour générer un circuit séquentiel est utile pour synthétiser des machines d'états.

```
package casestmt_pkg is
  type sel_cmd is (S1, S2, S3, S4);
end casestmt_pkg;

use work.casestmt_pkg.all;
entity casestmt is
  port (SEL: in sel_cmd;
        I1, I2, I3: in bit;
        R: out bit);
end;

architecture comb of casestmt is
  process (SEL, I1, I2, I3)
  begin
    case SEL is
      when S1 => R <= I1;
      when S2 => R <= I2;
      when S3 => R <= I3;
      when others => R <= '0';
    end case;
  end process;
end comb;
```

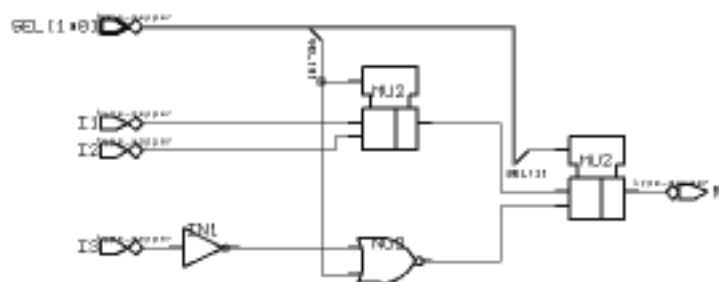


Figure 32. Synthèse d'une instruction de sélection **case**.

## VHDL pour la synthèse

- Instructions de boucle

```
-- X, Y: in bit_vector(0 to 2);
-- S: out bit_vector(0 to 2);
architecture a of e is
  subtype nat is
    natural range 0 to 2;
  constant N: nat := 1;
begin
  process (X, Y)
  begin
    for I in X'range loop
      S(I) <= X(I) and Y((I + 1) mod N);
      exit when I = N;
    end loop;
  end process;
end a;
```

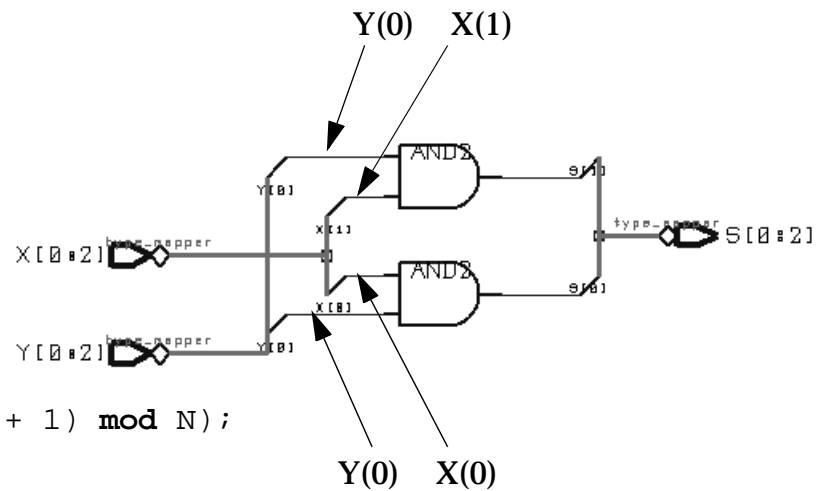


Figure 33. Instruction **for** avec limites statiques.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity whilestmt is
  port (A: in std_logic_vector(7 downto 0);
        CLK, EN : in std_logic;
        B: out std_logic_vector(7 downto 0));
end whilestmt;

architecture rtl of whilestmt is
begin
  process begin
    while (EN = '1') loop
      wait until CLK'event and CLK = '1';
      B <= A;
    end loop;
    wait until CLK'event and CLK = '1';
  end process;
end rtl;
```

Figure 34. Instruction **while**.

- Les instructions de boucle (**loop**, **for**, **while**) sont supportées en synthèse avec quelques restrictions. L’instruction **for** ou **while** avec des limites d’itération statiques se synthétise par duplication du circuit inféré par la boucle (*loop unrolling*). La Figure 33 donne un exemple d’utilisation de l’instruction **for** avec en plus une condition de sortie prématurée de la boucle (instruction **exit**).

**NOTE:** Les codes suivants sont équivalents:

```

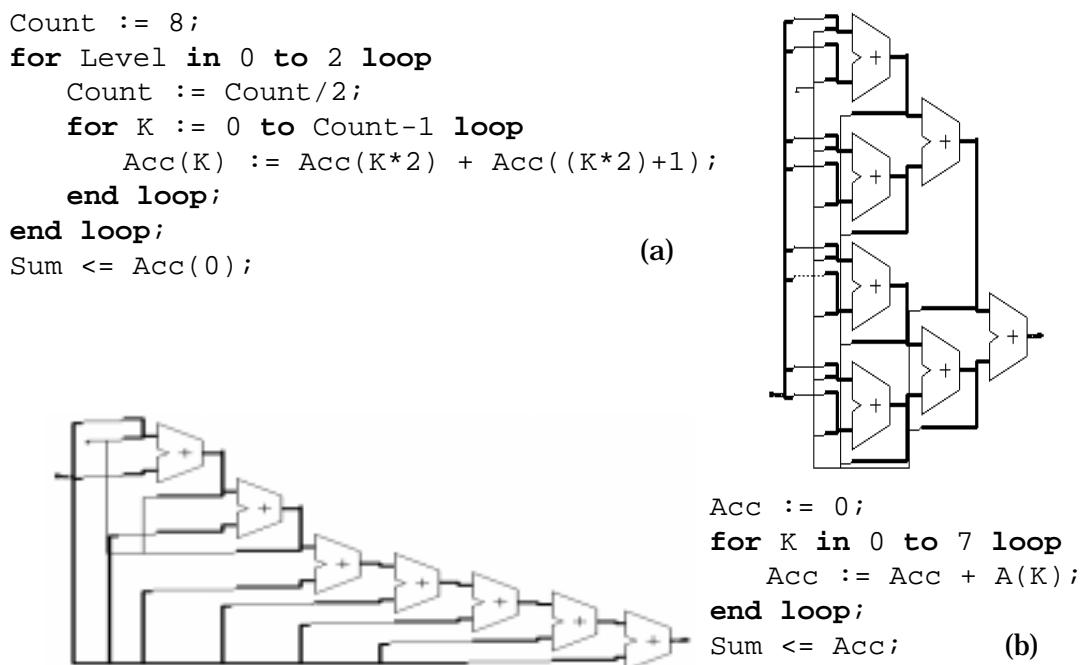
signal S1, S2: bit_vector(1 to 10);

for i in S1'range loop           S2 <= S1;
    S2(i) <= S1(i);
end loop;

```

Lorsque les limites ne sont pas statiques un ou plusieurs éléments de mémorisation (registres) sont inférés pour stocker l’indice de boucle et un circuit de contrôle est ajouté au circuit inféré par la boucle elle-même. Une instruction **wait** est alors requise dans le corps de la boucle. La Figure 34 donne un exemple d’utilisation de l’instruction **while**. Une seconde instruction **wait** est aussi requise pour le cas où EN = ‘0’. L’instruction de boucle infinie **loop ... end loop** peut être considérée comme une boucle **while** dont la condition est toujours vraie.

**NOTE:** Il est possible de contrôler la structure du circuit inféré par la boucle, en l’occurrence une structure série ou parallèle. La Figure 35 illustre les structures possibles pour une opération d’addition avec accumulation. La version parallèle est plus complexe est ainsi moins efficace en simulation, mais conduit à un circuit plus rapide.



**Figure 35.** Exemples de boucles inférant (a) une structure parallèle ou (b) une structure série.

## VHDL pour la synthèse

- Appel de sous-programme
  - Ne génère pas de niveau hiérarchique (! composant)
  - Fonction: infère un circuit combinatoire
  - Procédure: infère un circuit combinatoire ou séquentiel
    - Combinatoire ssi:
      - ses arguments sont de modes `in` et `out`
      - elle ne possède pas d'instruction `wait`
      - elle ne manipule que des objets locaux et ses arguments
    - Séquentiel autrement

```

-- type data is integer range 0 to 3;
-- type darray is array (1 to 3) of data;
-- ports: inar: in darray; outar: out darray
architecture a of sort is
begin
  process (inar)
    procedure swap (d: inout darray; l, h: in integer) is
      variable tmp: data;
    begin
      if d(l) > d(h) then
        tmp := d(l);
        d(l) := d(h);
        d(h) := tmp;
      end if;
    end swap;
    variable tmpar: darray;
  begin
    tmpar := inar;
    swap(tmpar,1,2);
    swap(tmpar,2,3);
    swap(tmpar,1,2);
    outar <= tmpar;
  end process;
end a;

```

**Code 29.** Exemple d'utilisation d'une procédure.

- Un appel de sous-programme (fonction ou procédure) ne crée pas de niveau hiérarchique dans la structure du circuit inféré.

Un appel de fonction infère un circuit combinatoire puisqu'il ne peut apparaître que dans une expression.

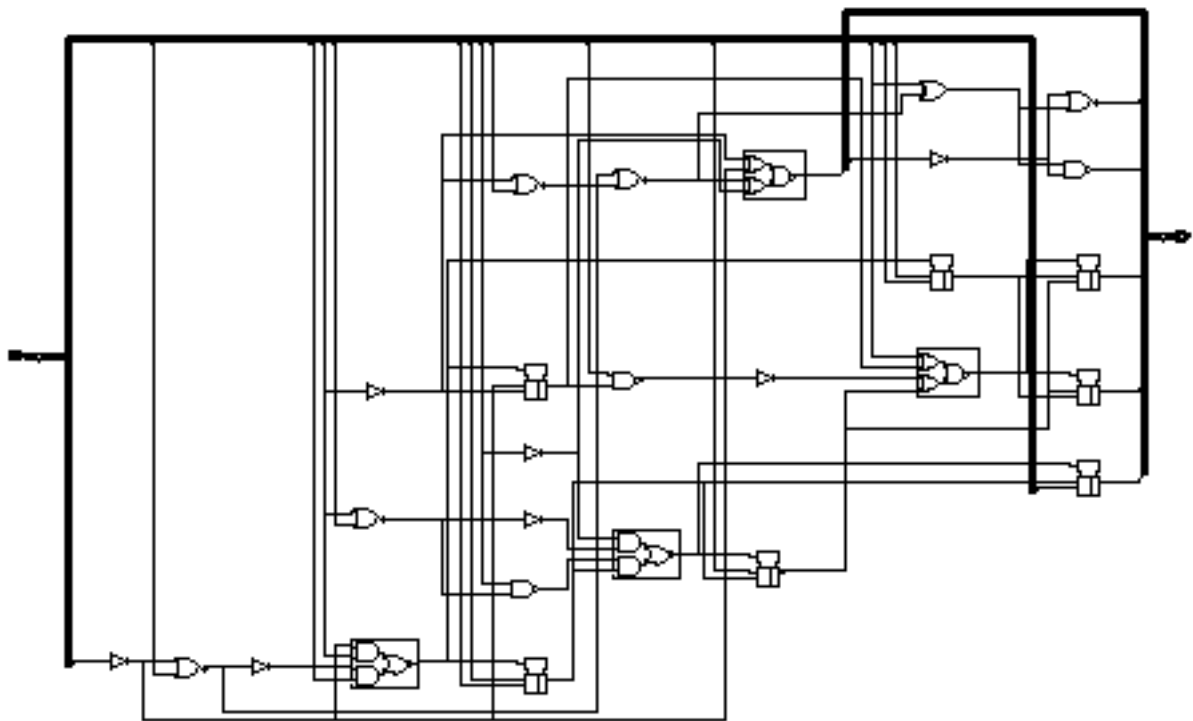
**NOTE:** Les fonctions de résolution et de conversion de types sont ignorées par la synthèse.

Un appel de procédure peut inférer un circuit combinatoire ou séquentiel. Le circuit est combinatoire ssi:

- ses arguments sont de modes **in** et **out**
- il n'y a pas d'instruction **wait** dans le corps de procédure
- la procédure ne manipule que des objets locaux et ses arguments (pas d'effet de bord).

Autrement, l'appel de procédure infère un circuit séquentiel.

Le [Code 29](#) donne un extrait de modèle réalisant la permutation des bits d'un mot de 3 bits à l'aide d'une procédure. La [Figure 36](#) illustre le circuit combinatoire inféré.



**Figure 36.** Circuit inféré par le modèle du [Code 29](#) réalisant une permutation des bits d'un mot de 3 bits.

## VHDL pour la synthèse

- Instructions concurrentes supportées:
  - Processus
  - Assignment de signal + versions conditionnelle et sélective
  - Instance de composant
  - Instruction **generate** (forme itérative + conditionnelle)

- Assignment concurrente de signal: circuit combinatoire

```
-- forme conditionnelle (else obligatoire)
S2 <= (S1 and B) when CMD = '0' else (C or D);

-- forme sélective
with CMD select
    S2 <= (S1 and B) when '0' else
          (C or D)   when others;
```

- Instance de composant
  - Modèle structurel (*netlist*)
  - Peut être nivelé (*flatten*, *ungroup*), synthétisé puis fixé (*don't touch*), dupliqué (*uniquify*)
  - Seule la configuration par défaut est supportée
- Paramètres génériques: limités aux types entier (et dérivés) et énumérés

### ***Instructions concurrentes supportées***

Toutes les instructions concurrentes de VHDL sont supportées pour la synthèse, mais toujours avec quelques restrictions.

- L'instruction **process** a déjà été présentée et discutée à la [page 71](#).
- L'assignation de signal simple a déjà été discutée à la [page 73](#). Les formes conditionnelle et sélective génèrent des circuits combinatoires. Pour s'en convaincre il suffit de dériver leurs formes séquentielles à base de processus équivalentes.
- La déclaration et l'instance de composant permet de définir une hiérarchie structurelle qui est conservée par la synthèse. Il est recommandé d'user de composants pour des modèles complexes au niveau RTL. Cela facilite aussi la spécification de contraintes temporelles aux ports des composants.

Un composant peut être nivelé (*flattened* ou *ungrouped*) après synthèse et son contenu fusionné avec l'environnement dans lequel le composant était instancié. Cette opération est utile pour des circuits combinatoires dans le but d'optimiser plus avant.

Un composant peut être optimisé séparément, puis inclus dans un circuit plus grand en figeant sa structure (*don't touch*).

Toutes les instances d'un composant font normalement référence à un seul circuit. Il s'agit dans certains cas de rendre les instances uniques (*uniquify*) pour pouvoir optimiser chaque instance séparément en fonction de son environnement.

**NOTE:** Seule la configuration par défaut est supportée en synthèse. Il faut donc que la déclaration de composant ait la même signature (ou vue) que la déclaration d'entité du modèle que l'on désire utiliser pour le composant.

### ***Paramètres génériques***

Les outils de synthèse logique actuels ne supportent en général que des paramètres génériques de type entier (ou dérivé) et énuméré.





## 3. Modélisation de circuits numériques

### 3.1. Circuits logiques combinatoires

Cette section illustre comment de la logique purement combinatoire peut être modélisée en VHDL et synthétisée. Un bloc logique combinatoire ne possède pas d'élément de mémorisation et donc ne présente pas d'état interne. Les blocs logiques combinatoires sont les principaux responsables des délais apparaissant dans un circuit. On parle aussi de *logique aléatoire* (*random logic*) car ces blocs ne possèdent pas de structure logique inhérente qu'il s'agit de préserver à tout prix et peuvent ainsi être optimisés (restructurés) pour satisfaire des contraintes de surface ou de délai.

Les instructions concurrentes d'assignation de signal et leurs processus équivalents sont les plus naturellement utilisées pour modéliser de la logique combinatoire. En guise de rappel, la liste de sensibilité d'un processus doit mentionner *tous* les signaux lus dans le corps du processus si l'on veut éviter d'inférer des éléments de mémoires lors de la synthèse (page 71).

#### Equations logiques

Le style de description flot de données (*dataflow*) est le plus approprié pour modéliser des équations logiques. Il fait usage des opérateurs logiques prédéfinis pour les types standard `bit`, `bit_vector`, `boolean`, `std_(u)logic` et `std_(u)logic_vector`.

Le [Code 30](#) donne le modèle d'une porte AND à deux entrées sous la forme d'une description flot de données et sous la forme d'un processus équivalent.

```

entity AND2 is
    generic (Tprop: time := 0 ns);
    port (A, B: in bit; Q: out bit);
end entity AND2;

architecture df11 of AND2 is
begin
    Q <= A and B after Tprop;
end df11;

architecture df12 of AND2 is
begin
    Q <= '1' after Tprop
        when A = '1' and B = '1'
        else '0' after Tprop;
end df12;

architecture proc of AND2 is
begin
    P_AND2: process (A, B)
    begin
        if A = '1' and B = '1' then
            Q <= '1' after Tprop;
        else
            Q <= '0' after Tprop;
        end if;
    end process P_AND2;
end proc;

```

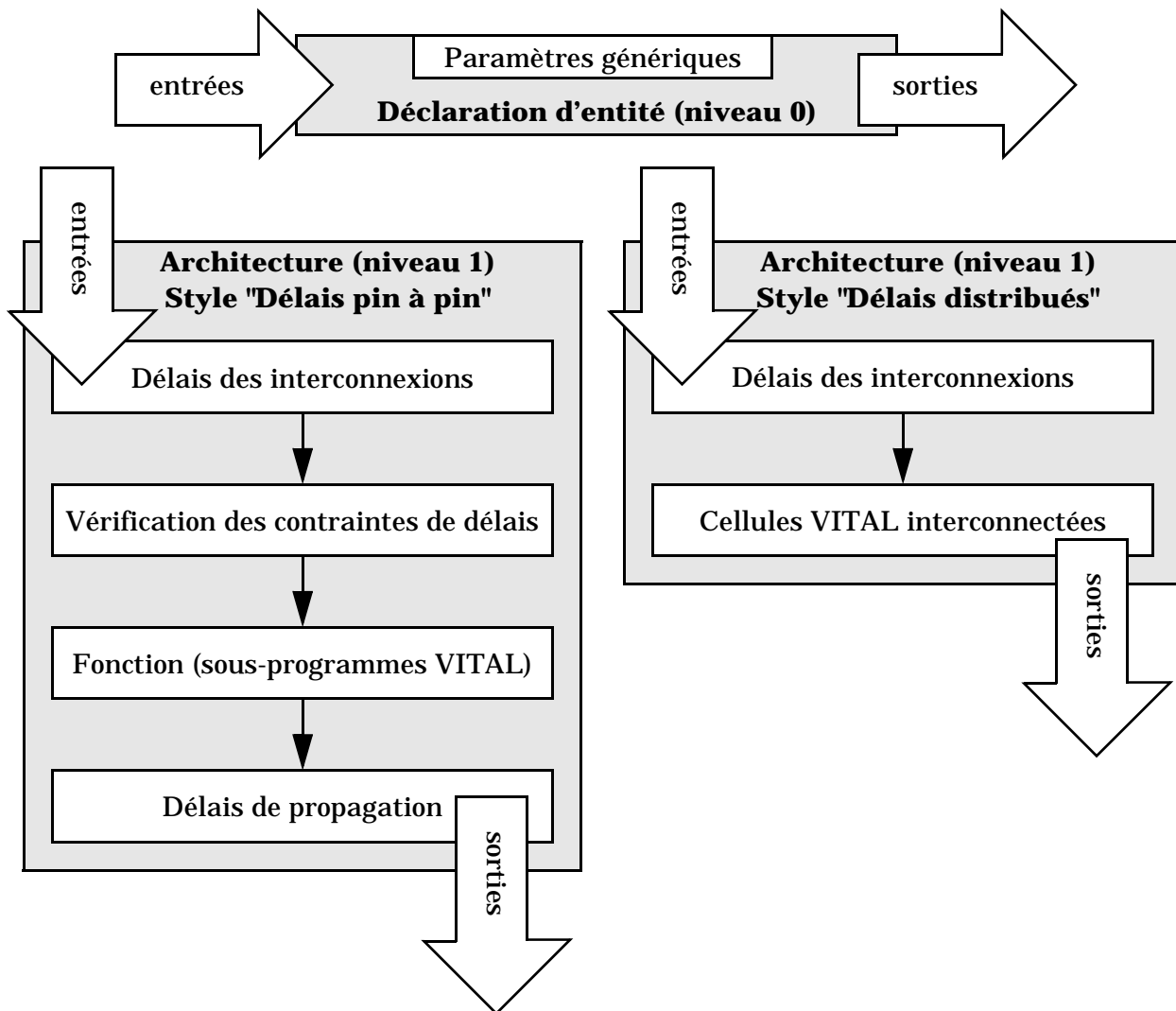
**Code 30.** Modélisation d'une porte AND à 2 entrées selon deux styles différents: flot de données et algorithmique.

La modélisation de portes logiques telles que la porte AND2 ci-dessus pose le problème de l'interopérabilité d'une bibliothèque de modèles de cellules standard, c'est-à-dire de sa capacité à être facilement réutilisée quelle soit la technologie. Le type `bit` standard n'est certainement pas suffisamment riche pour satisfaire tous les cas possibles et le langage VHDL permet a priori plusieurs manières, pas forcément compatibles, de modéliser les données technologiques telles que les délais. L'initiative VITAL (*VHDL Initiative Towards ASIC Libraries*), démarrée en 1992 par un consortium d'industriels, a permis de définir un ensemble de règles de modélisation VHDL pour le développement de bibliothèques ASIC. Ces règles font maintenant partie du stan-

dard IEEE 1076.4-1995 [STD1076.4]. Les principes derrière les règles de modélisation VITAL sont les suivants:

- Les modèles VHDL définissent les fonctionnalités des portes logiques ainsi que les noms et les significations des paramètres (génériques) représentant les délais associés aux portes.
- Les modèles VHDL utilisent les types du paquetage standard STD\_LOGIC\_1164 [STD1164].
- Les valeurs des délais dépendent de la technologie et sont toujours calculés en-dehors des modèles eux-mêmes. Les valeurs de délais sont stockées dans un fichier au format SDF [SDF] et utilisées dans les modèles VHDL par l'intermédiaire des paramètres génériques.
- La simulation des circuits avec les modèles VITAL permet de valider une conception pour la fabrication (*sign-off simulation*). Elle peut être en plus accélérée grâce au codage des modèles au coeur du simulateur.

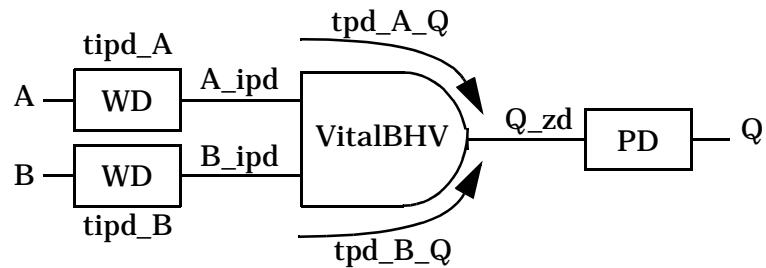
La norme VITAL définit deux niveaux de modélisation. Le *niveau 0* définit une déclaration d'entité standard qui inclut des noms et les types standard pour les paramètres génériques et les ports. Le *niveau 1* ajoute des règles de modélisation pour le corps d'architecture. Il propose deux styles de modélisation: le style "délais pin à pin" (*pin-to-pin delay style*) et le style "délais distribués" (*distributed delay style*). La Figure 37 illustre les niveaux et les styles de modélisation VITAL.



**Figure 37.** Niveaux et styles de modélisation VITAL.

Le Code 31 donne le modèle VITAL niveau 1 d'une porte logique AND à deux entrées en style "délais pin à pin". Il fait usage de deux paquetages standard définis dans la norme. Le paquetage VITAL\_TIMING définit

les types et les sous-programmes pour la sélection des délais, la vérification des délais, l'émission de messages et la détection de courses (*glitches*). Le paquetage VITAL\_PRIMITIVE définit les fonctions combinatoires et séquentielles communément utilisées et des fonctions utilitaires. Ces deux paquetages sont normalement compilés dans la bibliothèque IEEE. La Figure 38 illustre les paramètres utilisés et la décomposition du style "délais pin à pin".



**Figure 38.** Paramètres et décomposition du modèle VITAL pour le style "délais pin à pin".

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.vital_timing.all;

entity AND2 is
  generic (
    -- délais d'entrée et de propagation
    tipd_A: VitalDelayType01 := (tr01 => 0 ns, tr10 => 0 ns);
    tipd_B: VitalDelayType01 := (tr01 => 0 ns, tr10 => 0 ns);
    tpd_A_Q: VitalDelayType01 := (tr01 => 1.2 ns, tr10 => 1.1 ns);
    tpd_B_Q: VitalDelayType01 := (tr01 => 1.2 ns, tr10 => 1.1 ns);
    TimingChecksOn: Boolean := FALSE); -- vérifications actives
  port (
    Q: out std_logic;
    A, B: in std_logic);
  -- la déclaration d'entité est du niveau 0
  attribute VITAL_LEVEL0 of AND2: entity is TRUE;
end AND2;

use IEEE.vital_primitive.all;

architecture level1 of AND2 is
  -- l'architecture est de niveau 1
  attribute VITAL_LEVEL1 of bhv: architecture is TRUE;
  -- signaux locaux prenant en compte les délais d'interconnexion
  signal A_ipd, B_ipd: std_ulogic := 'U';
begin
  WireDelay: block -- application des délais d'interconnexion
  begin
    VitalWireDelay(A_ipd, A, tipd_A); -- procédures concurrentes
    VitalWireDelay(B_ipd, B, tipd_B);
  end block WireDelay;
  VITALBehavior: process (A_ipd, B_ipd)
    variable GlitchData_Q: GlitchDataType; -- détection de courses
    variable Q_zd: std_ulogic; -- valeur de sortie non retardée
  begin
    -- application de la fonction logique
    Q_zd := VitalAND2(A_ipd, B_ipd, DefaultResultMap);
    -- application des délais de propagation
    VitalPathDelay01(
      OutSignal => Q, OutSignalName => "Q", OutTemp => Q_zd,
      Paths => (
        0 => (InputChangeTime => A_ipd'last_event,
              PathDelay => tpd_A_Q,
              PathCondition => TRUE),
        1 => (InputChangeTime => B_ipd'last_event,
              PathDelay => tpd_B_Q,
              PathCondition => TRUE)),
      GlitchData => GlitchData_Q,
      GlitchMode => NoGlitch,
      GlitchKind => OnEvent);
  end process VITALBehavior;
end level1;

```

**Code 31.** Modèle VITAL niveau 1 d'une porte AND2 en style "délais pin à pin".

## Multiplexeurs

La fonction de base d'un multiplexeur est de sélectionner une entrée parmi plusieurs pour rendre sa valeur disponible en sortie. La sélection est effectuée au moyen d'un ou de plusieurs signaux de contrôle.

Le [Code 32](#) donne quatre architectures équivalentes d'un multiplexeur 4-1. L'usage de l'instruction séquentielle case ou de son instruction concurrente équivalente donne un modèle plus lisible.

```

library IEEE;
use IEEE.std_logic_1164.all;

entity mux41 is
  port (SEL: in  std_logic_vector(1 downto 0);
        INP: in  std_logic_vector(3 downto 0);
        Q:  out std_logic);
end mux41;

architecture df11 of mux41 is
begin
  Q <= INP(3) when SEL = "00" else
        INP(2) when SEL = "01" else
        INP(1) when SEL = "10" else
        INP(0); -- when SEL = "11"
end df11;

architecture df12 of mux41 is
begin
  with SEL select
    Q <= INP(3) when "00",
        INP(2) when "01",
        INP(1) when "10",
        INP(0) when "11",
        INP(3) when others;
end df12;

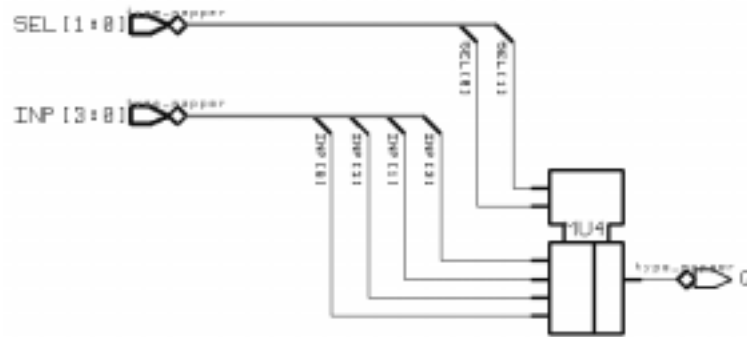
architecture proc1 of mux41 is
begin
  process (SEL, INP)
  begin
    if SEL = "00" then
      Q <= INP(3);
    elsif SEL = "01" then
      Q <= INP(2);
    elsif SEL = "10" then
      Q <= INP(1);
    else -- SEL = "00"
      Q <= INP(0);
    end if;
  end process;
end proc1;

architecture proc2 of mux41 is
begin
  process (SEL, INP)
  begin
    case SEL is
      when "00" => Q <= INP(3);
      when "01" => Q <= INP(2);
      when "10" => Q <= INP(1);
      when "00" => Q <= INP(0);
      when others => Q <= INP(3);
    end case;
  end process;
end proc2;

```

**Code 32.** Modélisation d'un multiplexeur à 4 entrées selon différents styles.

La [Figure 39](#) donne les circuits synthétisés par les architectures df11 ou proc1 et df12 ou proc2. Pour le synthétiseur utilisé (Synopsys) un seul multiplexeur est inféré. Il faut noter que en général les formes conditionnelles (df11, proc1) impliquent une priorité qui nécessite plus de logique et aboutissent ainsi à des circuits moins performants en vitesse. L'usage de la forme sélective (df12, proc2) est donc recommandée.



**Figure 39.** Circuit synthétisé pour les architectures du Code 32.

Une autre manière d'inférer un multiplexage est d'utiliser des tableaux indicés. Cette technique est utile lorsque les données à sélectionner sont structurées. Dans ce cas un arbre de multiplexeurs est inféré par la synthèse. Le Code 33 donne un exemple de sélection de données structurées au moyen d'un tableau à deux dimensions. La complexité du modèle réside dans ce cas dans les déclarations des types de données, ce qui requiert un paquetage séparé. La Figure 40 donne le circuit inféré en synthèse.

```

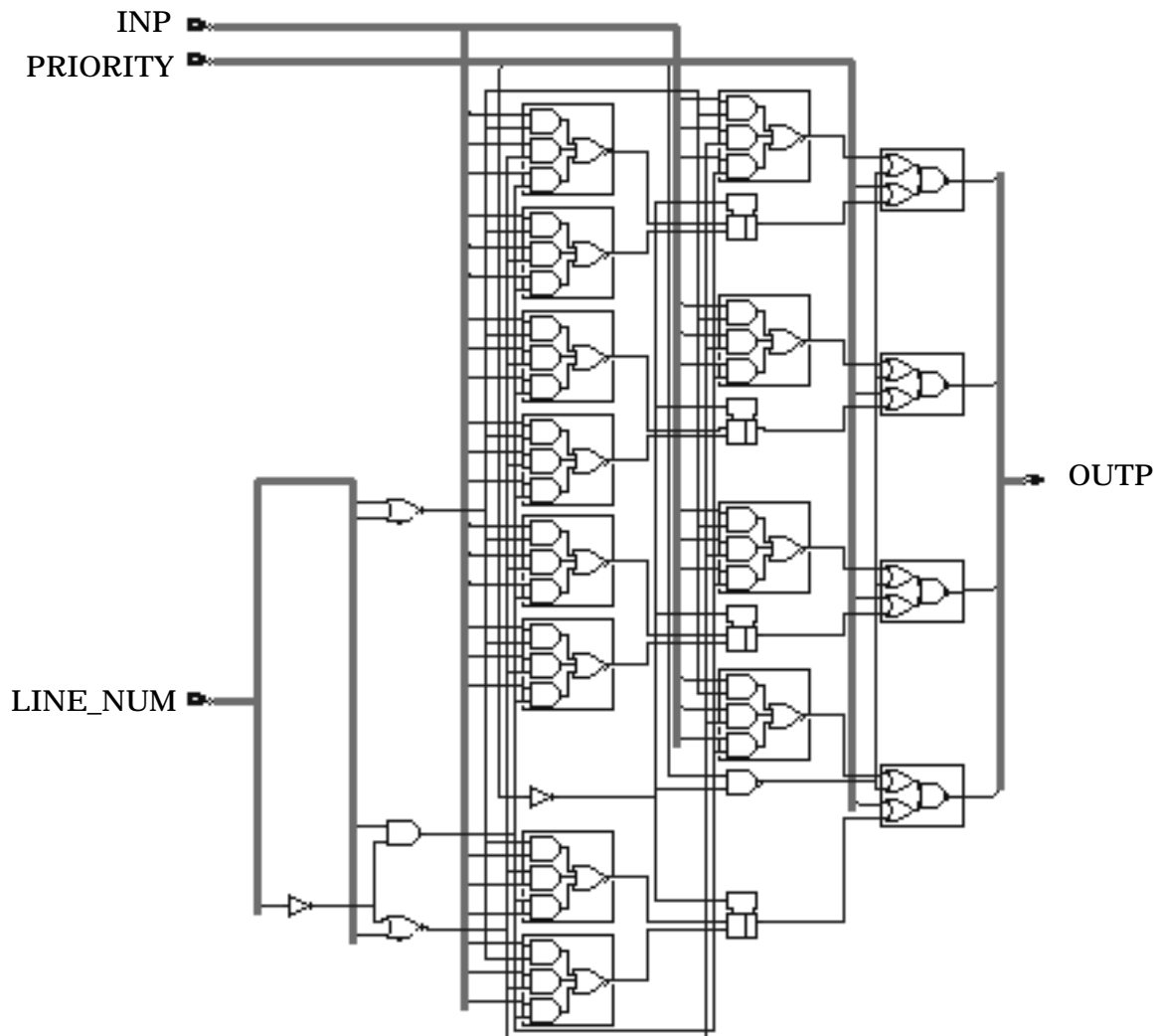
package sel_pkg is
  -- la donnée de base est un mot de 4 bits
  subtype bv4 is bit_vector(3 downto 0);
  -- une ligne du bus d'entrée est un tableau indexé
  -- par trois niveaux de priorité
  type t_priority is (LOW, MEDIUM, HIGH);
  type t_priority_arr is array (LOW to HIGH) of bv4;
  -- le bus d'entrée est a priori non contraint
  type t_input_arr is array (natural range <>) of t_priority_arr;
end sel_pkg;

use WORK.sel_pkg.all;
entity muxtree is
  port (INP:      in  t_input_arr(0 to 2);
        LINE_NUM: in  natural range 0 to 2;
        PRIORITY: in  t_priority_arr;
        OUTP:     out bv4);
end muxtree;

architecture index of muxtree is
begin
  OUTP <= INP(LINE_NUM)(PRIORITY);
end index;

```

**Code 33.** Exemple de sélection de données structurées et d'inférence d'un arbre de multiplexage.



**Figure 40.** Circuit inféré par le modèle du [Code 33](#).

### Encodeurs

Un encodeur permet de représenter (coder) un mot de  $2^n$  bits au moyen d'un mot de  $n$  bits. La [Table 3.1](#) donne

Entrées								Sorties		
A7	A6	A5	A4	A3	A2	A1	A0	Q2	Q1	Q0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

**Table 3.1.** Table de vérité d'un encodeur binaire 8-3.

la table de vérité d'un encodeur binaire 8-3. On suppose, pour simplifier, que le mot d'entrée ne possède qu'un

seul 1. Le [Code 34](#) donne le modèle VHDL d'un tel encodeur utilisant une instruction concurrente sélective d'assignation de signal. Le problème ici est de minimiser la logique inférée car seules 8 sur 256 valeurs d'entrée sont possibles. La clause **when others** permet de prendre en compte les valeurs d'entrée "non intéressantes" (pour éviter de générer des éléments de mémoire). De plus, l'assignation à la valeur "--" permet au synthétiseur de choisir les meilleures valeurs '0' ou '1' en fonction des contraintes d'optimisation. La [Figure 41](#) donne le circuit inféré en synthèse.

```

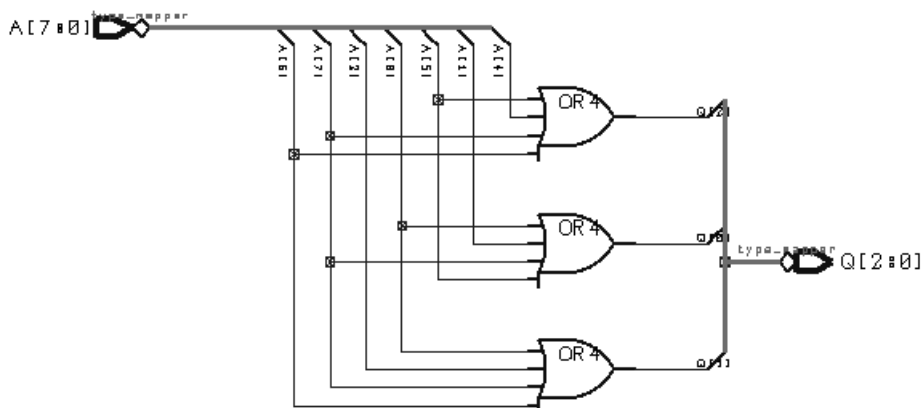
library IEEE;
use IEEE.std_logic_1164.all;

entity encoder83 is
  port (A: in  std_logic_vector(7 downto 0);
        Q: out std_logic_vector(2 downto 0));
end encoder83;

architecture csel of encoder83 is
begin
  with A select
    Q <= "000" when "00000001",
         "001" when "00000010",
         "010" when "00000100",
         "011" when "00001000",
         "100" when "00010000",
         "101" when "00100000",
         "110" when "01000000",
         "111" when "10000000",
         "---" when others;
end csel;

```

**Code 34.** Modèle d'un encodeur binaire 8-3.



**Figure 41.** Circuit inféré par le modèle du [Code 34](#).

Un modèle générique d'encodeur binaire  $M$ - $N$  (avec  $M = 2^N$ ) doit plutôt utiliser une instruction de boucle for car le nombre de cas n'est pas connu a priori. Le [Code 35](#) donne un modèle générique d'encodeur binaire. La valeur de  $M$  n'est pas un paramètre explicite, mais elle est dérivée de la valeur du paramètre  $N$  pour assurer la relation  $M = 2^N$ . La vérification se fera alors sur les tailles effectives des signaux d'interface. Le corps d'architecture compare systématiquement le mot d'entrée à un mot de test ne contenant qu'un seul '1' qui est déplacé d'un bit à gauche à chaque test. La boucle est stoppée dès que l'égalité est obtenue. La [Figure 42](#) donne le circuit inféré par le modèle générique pour  $N = 3$ .



```

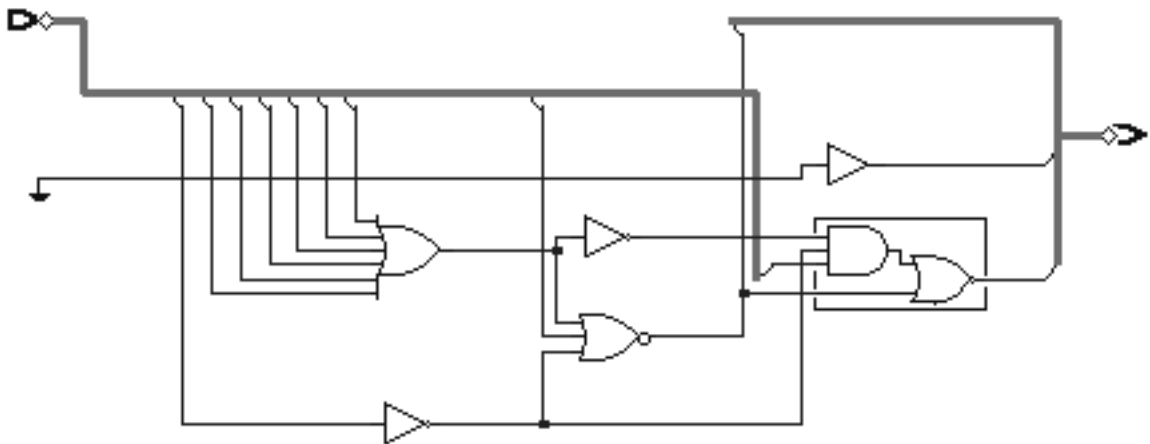
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity encoderMN is
  generic (N: positive);
  port (A: in  unsigned(2**N-1 downto 0);
        Q: out unsigned(N-1 downto 0));
end encoderMN;

architecture bhv of encoderMN is
begin
  process (A)
    variable test: unsigned(Q'range);
  begin
    test := (others => '0');
    test(test'low) := '1';
    Q <= (others => '-'); -- valeur par défaut
    for i in 0 to Q'length-1 loop
      if A = test then
        Q <= conv_unsigned(i, Q'length);
        exit;
      end if;
      test := shl(test, conv_unsigned(1, 1));
    end loop;
  end process;
end bhv;

```

**Code 35.** Modèle générique d'encodeur binaire.

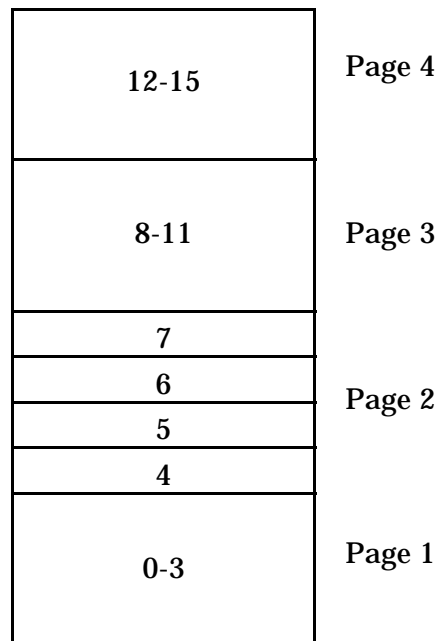


**Figure 42.** Circuit inféré par le modèle du Code 35 pour  $N = 3$ .

## Décodeurs

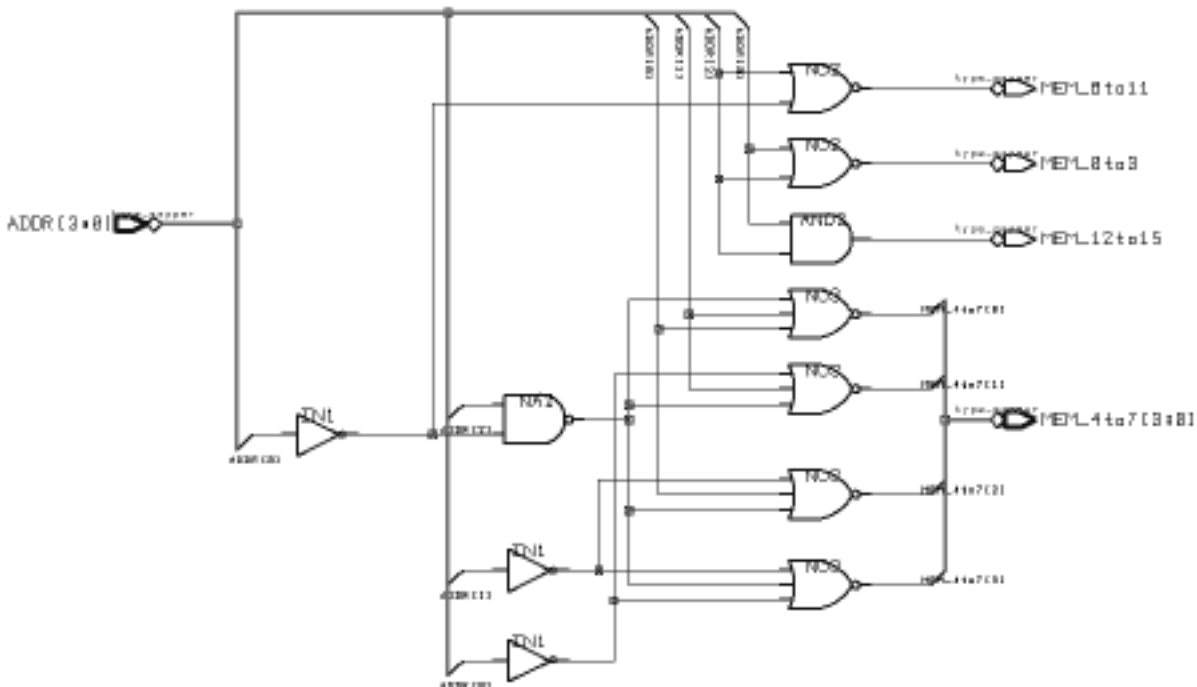
Un décodeur permet de décoder un mot codé sur  $n$  bits pour obtenir sa valeur sur  $2^n$  bits au maximum.

A titre d'exemple, considérons le modèle d'un décodeur d'adresses 4 bits. On suppose que la mémoire considérée est décomposée en quatre pages, la deuxième page étant elle-même décomposée en quatre segments (Figure 43).



**Figure 43.** Table des adresses de la mémoire.

Le Code 36 donne le modèle VHDL du décodeur d'adresses. Le circuit possède sept signaux de sortie pour activer le segment mémoire concerné. La Figure 44 donne le circuit inféré en synthèse.



**Figure 44.** Circuit inféré par le modèle du Code 36.

```

library IEEE;
use IEEE.std_logic_1164.all;

entity addr4b_decoder is
  port (ADDR: in natural range 0 to 15;
        MEM_0to3, MEM_8to11, MEM_12to15: out std_logic;
        MEM_4to7: out std_logic_vector(3 downto 0));
end addr4b_decoder;

architecture bhv of addr4b_decoder is
begin
  process (ADDR)
  begin
    -- initialisations
    MEM_0to3    <= '0';
    MEM_4to7    <= (others => '0');
    MEM_8to11   <= '0';
    MEM_12to15 <= '0';
    -- décodage
    case ADDR is
      when 0 to 3    => MEM_0to3    <= '1';
      when 4          => MEM_4to7(0) <= '1';
      when 5          => MEM_4to7(1) <= '1';
      when 6          => MEM_4to7(2) <= '1';
      when 7          => MEM_4to7(3) <= '1';
      when 8 to 11  => MEM_8to11   <= '1';
      when 12 to 15 => MEM_12to15  <= '1';
    end case;
  end process;
end bhv;

```

**Code 36.** Modèle du décodeur d'adresses 4 bits.

### Comparateurs

Un comparateur compare deux ou plusieurs entrées et produit un signal '0' ou '1' selon que la comparaison est vraie ou fausse.

Le [Code 37](#) donne le modèle générique d'un comparateur binaire produisant les signaux de contrôle EQ (égalité), GT (plus grand que), LT (plus petit que), NE (pas égal), GE (plus grand ou égal), et LE (plus petit ou égal). Une autre possibilité serait de mettre des bits d'un registre d'état à '1'. Les opérations de comparaison s'effectuent sur des valeurs entières converties à partir des mots d'entrée à comparer. La conversion n'est en fait qu'un changement temporaire de type (*type casting*) pour pouvoir appliquer des opérateurs de comparaison non disponibles en standard pour le type `std_logic_vector`. La [Figure 45](#) donne le circuit inféré en synthèse pour un comparateur deux bits.

```

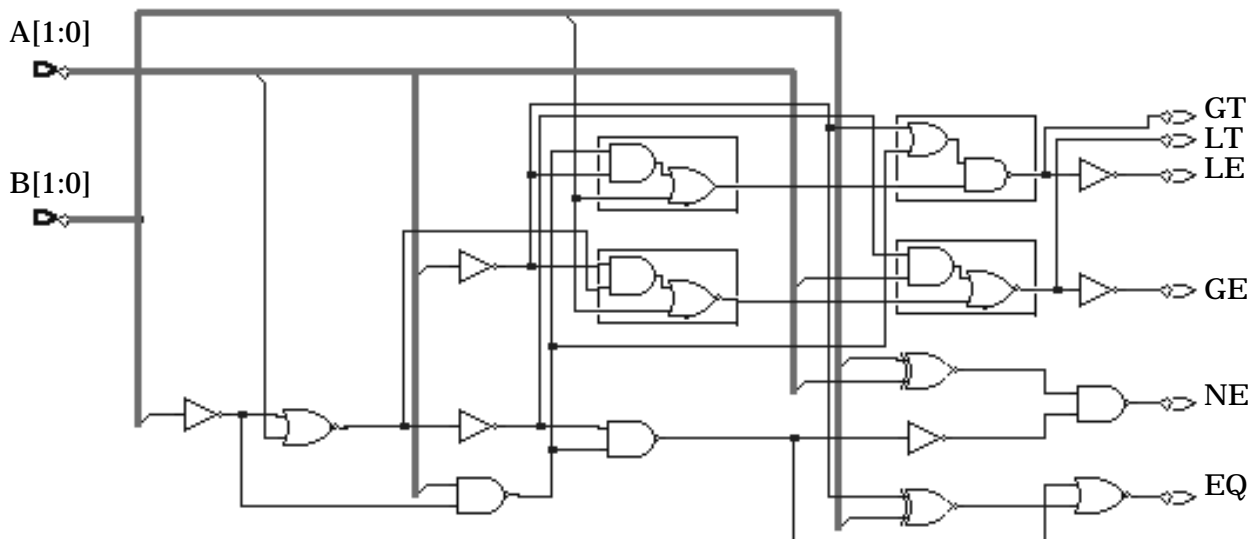
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity comparator is
  generic (NBits: positive := 2);
  port (A, B: in std_logic_vector(NBits-1 downto 0);
        EQ, GT, LT, NE, GE, LE: out std_logic);
end comparator;

architecture bhv of comparator is
begin
  process (A, B)
  begin
    if unsigned(A) > unsigned(B) then
      EQ <= '0'; LT <= '0'; LE <= '0';
      GT <= '1'; NE <= '1'; GE <= '1';
    elsif unsigned(A) < unsigned(B) then
      EQ <= '0'; GT <= '0'; GE <= '0';
      LT <= '1'; NE <= '1'; LE <= '1';
    else
      NE <= '0'; GT <= '0'; LT <= '0';
      EQ <= '1'; GE <= '1'; LE <= '1';
    end if;
  end process;
end bhv;

```

**Code 37.** Modèle VHDL générique d'un comparateur binaire.



**Figure 45.** Comparateur 2 bits inféré par le modèle du Code 37 avec NBits = 2.

Il est aussi possible d'effectuer des comparaisons logiques arbitrairement complexes. Le Code 38 donne un exemple d'une telle comparaison. Noter l'usage des parenthèses pour spécifier une précedence des opérations différente de celle par défaut. La Figure 46 donne le circuit inféré par la synthèse.

```

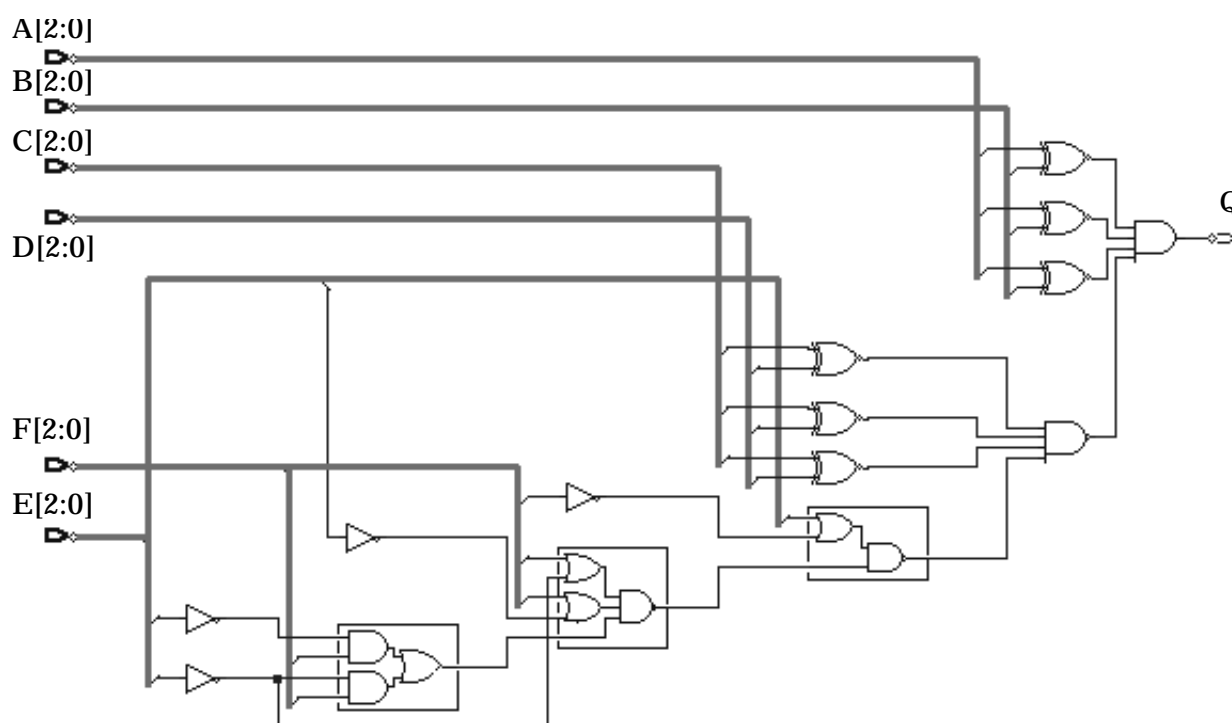
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity comparator2 is
  port (A, B, C, D, E, F: in unsigned(2 downto 0);
        Q: out std_logic);
end comparator2;

architecture dfl of comparator2 is
begin
  Q <= '1' when (A = B and (C /= D or E >= F)) else '0';
end dfl;

```

**Code 38.** Modèle de comparateur utilisant des comparaisons multiples.



**Figure 46.** Circuit inféré en synthèse par le modèle du [Code 38](#).

### Unité arithmétique et logique (ALU)

L'unité arithmétique et logique constitue le coeur d'une unité de calcul telle que l'on en trouve dans un microprocesseur. Elle est capable d'effectuer des opérations arithmétiques (addition, soustraction, incrémentation, décrémentation, décalages) et logiques (AND, OR, NOT, XOR, etc.) de base sur deux bus de données.

Le [Code 39](#) donne le modèle générique d'une unité arithmétique et logique simple opérant sur des arguments non signés. Noter les conversions de types nécessaires. La [Figure 47](#) donne le circuit inféré en synthèse avec  $N_{Bits} = 2$ . Noter l'usage du même opérateur pour plusieurs opérations arithmétiques (ADD, INC1, SUB, DEC1). Cette possibilité de partage de ressources ([page 67](#)) n'est pas forcément disponible pour tous les synthétiseurs. La modélisation et la synthèse d'opérateurs arithmétiques sera vue plus loin.

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

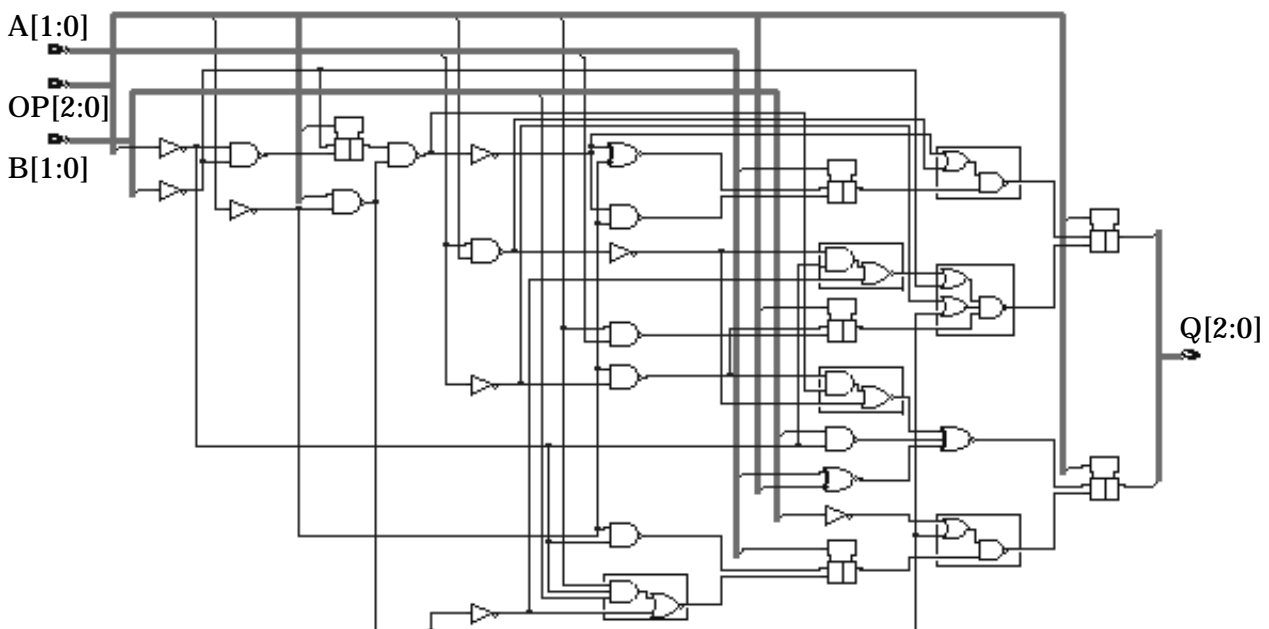
package ALU_pkg is
    type alu_op is (ADD, SUB, INC1, DEC1, LNOT, LAND, LOR, LSHR);
end ALU_pkg;

use WORK.ALU_pkg.all;
entity ALU is
    generic (NBits: positive);
    port (OP:    in  alu_op;
          A, B:  in  unsigned(NBits-1 downto 0);
          Q:     out unsigned(NBits-1 downto 0));
end ALU;

architecture dfl of ALU is
begin
    with OP select
        Q <= A + B when ADD,
            A - B when SUB,
            A + 1 when INC1,
            A - 1 when DEC1,
            unsigned(not std_logic_vector(A)) when LNOT,
            unsigned(std_logic_vector(A) and std_logic_vector(B)) when LAND,
            unsigned(std_logic_vector(A) or std_logic_vector(B)) when LOR,
            shr(A, conv_unsigned(1, 1)) when LSHR;
end dfl;

```

**Code 39.** Modèle d'une unité arithmétique et logique.



**Figure 47.** Circuit inféré en synthèse par le modèle du Code 39 avec NBits = 2.

## 3.2. Circuits logiques séquentiels

Cette section illustre comment de la logique séquentielle (ou synchrone) peut être modélisée en VHDL et synthétisée. Un bloc logique séquentiel possède un ou plusieurs éléments de mémorisation ainsi que potentiellement un bloc combinatoire. Les éléments de mémorisation sont des latches ou des flip-flops. Le comportement d'un bloc logique synchrone est fonction d'un signal d'horloge.

L'instruction de base pour modéliser des circuits logiques synchrones est le processus (page 71). Les instructions concurrentes d'assignation de signal sont aussi utilisables, mais en général moins bien supportées par les outils de synthèse. L'avantage du processus est d'offrir plus de flexibilité pour modéliser des blocs combinatoires dont certains signaux doivent être mémorisés.

### Latches

Un élément de mémorisation de type latch transmet la valeur du signal d'entrée à la sortie du composant lorsqu'un signal de commande est actif. Il existe plusieurs styles possibles de modélisation d'un composant latch. Le Code 40 donne le modèle d'une latch D avec signaux PR (preset) et CL (clear) asynchrones (page 65). L'usage de l'instruction `if` implique une priorité sur les signaux d'entrée: ici PR, CL, EN dans l'ordre décroissant.

```
entity dlatch is
  port (EN, PR, CL: in bit; -- commandes
        D:          in bit; -- entrée
        Q:          out bit); -- sortie
end dlatch;

architecture bhv of dlatch is
begin
  process (EN, PR, CL, D)
  begin
    if PR = '1' then
      Q <= '1';
    elsif CL = '1' then
      Q <= '0';
    elsif EN = '1' then
      Q <= D;
    end if;
  end process;
end bhv;
```

**Code 40.** Modèle d'une latch D avec preset et clear asynchrones.

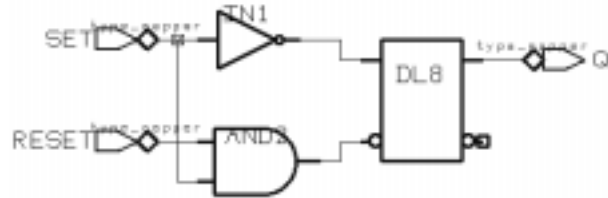
La modélisation d'une latch SR, c'est-à-dire sans entrée D, nécessite quelques artifices de modélisation dans la mesure où la bibliothèque de cellules standard ne possède que des cellules de type latch D. En effet, une modélisation telle que celle du Code 41 produit un circuit exhibant une course lorsque, à la fin d'une phase de SET le signal D de la latch D change en même temps que son signal ENABLE. Ceci a pour effet de violer le temps de setup de la latch D.

```

entity srlatch is
  port (SET, RESET: in bit; -- commandes
        Q: out bit); -- sortie
end srlatch;

architecture bad of srlatch is
begin
  process (SET, RESET)
  begin
    if SET = '0' then
      Q <= '1';
    elsif RESET = '0' then
      Q <= '0';
    end if;
  end process;
end bad;

```

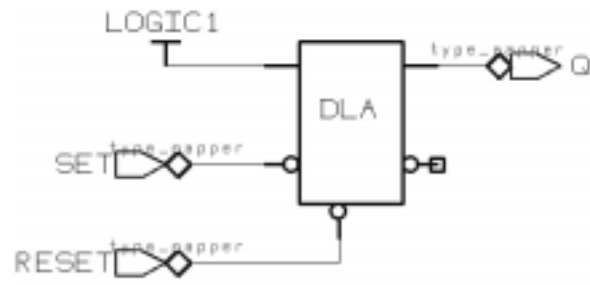


**Code 41.** Modélisation d'une latch SR avec course.

```

library SYNOPSIS;
use SYNOPSIS.attributes.all;
architecture good of srlatch is
  attribute async_set_reset of
    RESET: signal is "TRUE";
begin
  process (SET, RESET)
  begin
    if RESET = '0' then
      Q <= '0';
    elsif SET = '0' then
      Q <= '1';
    end if;
  end process;
end good;

```



**Code 42.** Modélisation d'une latch SR sans course.

### Flip-flops

Un élément de mémorisation de type flip-flop transmet la valeur du signal d'entrée à la sortie du composant sur un flanc de signal d'horloge. Comme pour la latch il existe plusieurs modèles possibles pour inférer un flip-flop. Le [Code 43](#) donne le modèle d'un flip-flop avec set et reset asynchrones. L'usage de l'instruction **if** implique une priorité sur les signaux d'entrée: ici RESET, SET, CLK dans l'ordre décroissant.



```

entity dff is
  port (CLK:      in bit; -- horloge
        SET, RST: in bit; -- commandes
        D:       in bit; -- entrée
        Q:       out bit); -- sortie
end dff;

architecture bhv of dff is
begin
  process (CLK, SET, RST)
  begin
    if RST = '1' then
      Q <= '0';
    elsif SET = '1' then
      Q <= '1';
    elsif CLK'event and CLK = '1' then
      Q <= D;
    end if;
  end process;
end bhv;

```

**Code 43.** Modèle d'un flip-flop D avec preset et clear asynchrones.

La détection d'un flanc de signal telle que celle effectuée dans le [Code 43](#) ne fonctionnerait pas toujours correctement pour un signal de type `std_logic`, du moins en simulation. En effet, si le signal CLK est du type `std_logic`:

```

if CLK'event and CLK = '1' then
  -- détecte aussi les transitions de 'X' à '1'

if CLK'event and CLK'last_value = '0' and CLK = '1' then
  -- ne détecte que les transitions de '0' à '1'

if rising_edge(CLK) then
  -- ne détecte que les transitions de '0' à '1'

```

La dernière solution est la meilleure car plus compacte. Les fonctions `rising_edge(S)` et `falling_edge(S)` sont toutes deux définies de manière standard dans le paquetage `STD_LOGIC_1164`.

Il faut noter que l'instruction `wait` est aussi applicable et même largement utilisée:

```

wait until CLK = '1';
-- ou
wait until rising_edge(CLK);1

```

L'usage d'un processus permet de modéliser de la logique combinatoire dont des signaux doivent être synchronisés. Le [Code 44](#) donne le modèle d'un circuit séquentiel inférant deux flips-flops. Le premier est dû à l'assignation du signal de sortie Q et le second est dû à l'assignation du signal local S et au fait que ce signal est utilisé dans l'expression assignée au signal Q. La variable V n'infère pas d'élément de mémorisation car sa nouvelle valeur est directement utilisée dans l'expression calculant la valeur du signal Q. La [Figure 48](#) donne le circuit séquentiel inféré par la synthèse.

---

1. Synopsys ne supporte pas cette forme de l'instruction `wait`.

```

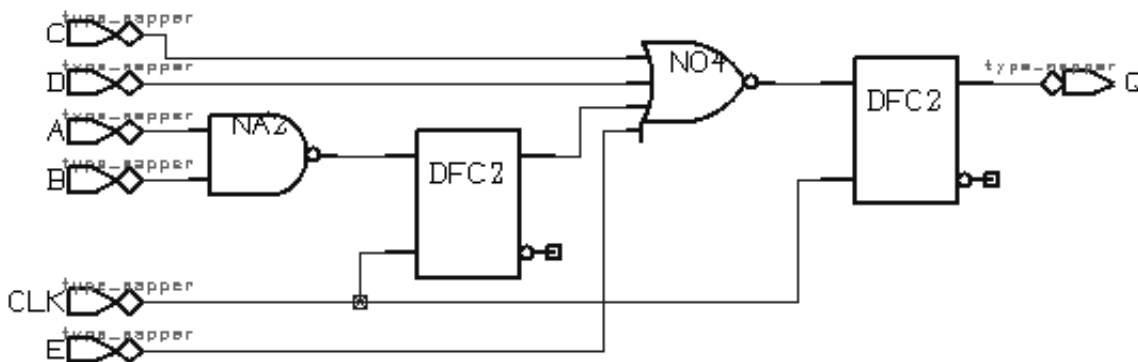
library IEEE;
use IEEE.std_logic_1164.all;

entity sync_logic is
  port (CLK, A, B, C, D, E: in std_logic;
        Q: out std_logic);
end sync_logic;

architecture bhv of sync_logic is
  signal S: std_logic;
begin
  process
    variable V: std_logic;
  begin
    wait until CLK = '1';
    S <= A nand B;           -- infère un flip-flop
    V := C or D;
    Q <= not (S or V or E); -- infère un flip-flop
  end process;
end bhv;

```

**Code 44.** Modélisation de logique combinatoire entre flip-flops.



**Figure 48.** Circuit inféré par la synthèse du modèle du [Code 44](#).

Dans le cas où il s'agit de rendre le comportement du circuit dépendant des deux flancs de l'horloge, il est nécessaire d'utiliser deux processus. Le [Code 45](#) donne un exemple simple d'utilisation des deux flancs d'horloge. La [Figure 49](#) donne le circuit inféré par la synthèse.

```

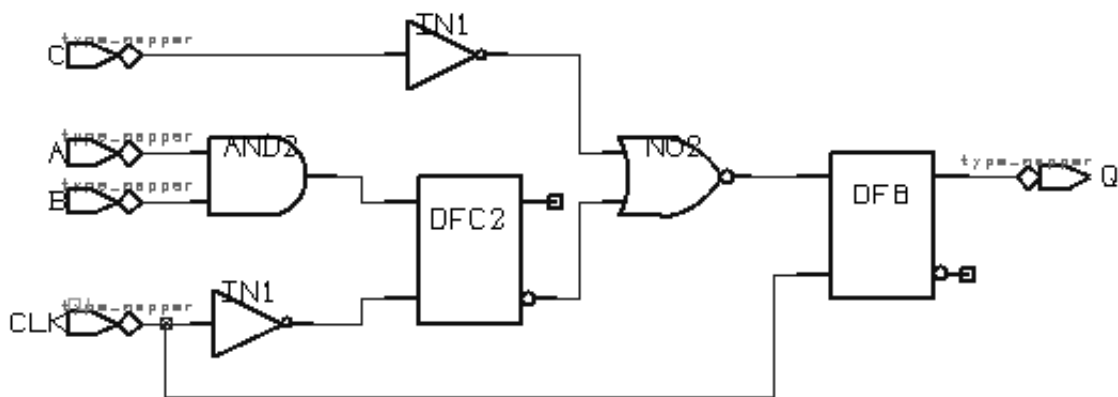
library IEEE;
use IEEE.std_logic_1164.all;

entity clk_edges is
  port (CLK, A, B, C: in std_logic;
        Q: out std_logic);
end clk_edges;

architecture bhv of clk_edges is
  signal S: std_logic;
begin
  P1: process
  begin
    wait until CLK = '1';
    Q <= S and C;
  end process P1;
  P2: process
  begin
    wait until CLK = '0';
    S <= A and B;
  end process P2;
end bhv;

```

**Code 45.** Utilisation des deux flancs d'horloge.



**Figure 49.** Circuit inféré par la synthèse du modèle du [Code 45](#).

Finalement, si plusieurs signaux d'horloges différents sont nécessaires, il faut utiliser un processus par signal d'horloge.

## Registres

Des exemples de modèles de registres ont déjà été donnés au Chapitre 2 (Code 6, Code 17). Le Code 46 donne le modèle d'un banc de registres de 32 mots de 8 bits. Le contrôle du circuit est assuré par le bus d'adresses ADDR, une commande de lecture RD et une commande d'écriture WR. Le bus de données D est bidirectionnel. Comme ce bus est normalement partagé avec d'autres bancs de registres il est nécessaire que le signal correspondant soit d'un type résolu. Le bus D est mis dans l'état haute-impédance lorsque le banc de registre n'est accédé ni en lecture, ni en écriture. Bien que le modèle soit assez compact, le circuit inféré par la synthèse est de grande taille.

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity reg_file is
  port (CLK, RD, WR: in    std_logic;
        ADDR:          in    unsigned(4 downto 0);
        D:             inout std_logic_vector(7 downto 0));
end reg_file;

architecture bhv of reg_file is
begin
  process
    type t_mem is array (natural range 0 to 2**ADDR'length-1)
                  of std_logic_vector(D'range);
    variable mem: t_mem; -- mémoire interne
  begin
    wait until CLK = '1';
    D <= (others => 'Z'); -- par défaut le bus est en haute impédance
    if WR = '1' then
      mem(conv_integer(ADDR)) := D; -- écriture
    elsif RD = '1' then
      D <= mem(conv_integer(ADDR)); -- lecture
    end if;
  end process;
end bhv;

```

**Code 46.** Modèle d'un banc de registres (32 mots de 8 bits).

Une autre classe intéressante de registres est celle des registres à décalage à rétroaction linéaire (*Linear Feedback Shift Register* ou LFSR, Annexe D). Les applications typiques des registres LFSR sont les compteurs, les circuits de test, la génération de valeurs pseudo-aléatoires, le codage et le décodage de données, la compression de données. Une boucle de rétroaction combinatoire force le registre à générer une suite pseudo-aléatoire de valeurs binaires. Les points de rétroaction (*taps*) sont usuellement "XORés" ou "XNORés" avant d'être réintroduits dans le registre.

Le choix des points de rétroaction détermine combien de valeurs d'une séquence donnée sont générées avant que cette séquence soit répétée. Certains choix de points de rétroaction produisent des séquences de longueurs maximum, soit  $2^n - 1$  pour un registre de  $n$  bits. Il est possible de générer toutes les  $2^n$  valeurs possibles moyennant une modification de la structure du circuit. Si moins de  $2^n - 1$  valeurs sont nécessaires, il faut prévoir de la logique supplémentaire pour éviter que le registre génère une valeur non permise et reste ainsi bloqué dans cet état.

Le Code 47 donne le modèle d'un registre LFSR 8 bits générant toutes les  $2^8$  valeurs possibles. Les points de rétroaction doivent être appliqués aux bits 1, 2, 3 et 7. Ces points sont définis dans le vecteur constant TAPS.

L'état interne du registre est mémorisé dans la variable `LFSR_reg`, ce qui permet de lire et de modifier cet état librement avant de l'assigner au bus de sortie `Q`. Le NOR de toutes les bits du registre moins le bit de poids fort, c.à.d. `Q[6:0]`, génère la logique supplémentaire pour couvrir toutes les  $2^8$  valeurs possibles. Ceci génère la variable `bits0_6_zero`, qui est elle-même ensuite XORée avec le bit de poids fort du registre pour générer la valeur à appliquer aux points de rétroaction. Finalement, chaque bit du registre est soit simplement décalé, soit sa valeur XORée avec la valeur de rétroaction avant d'être décalée. La [Figure 50](#) donne le circuit inféré par la synthèse.

```

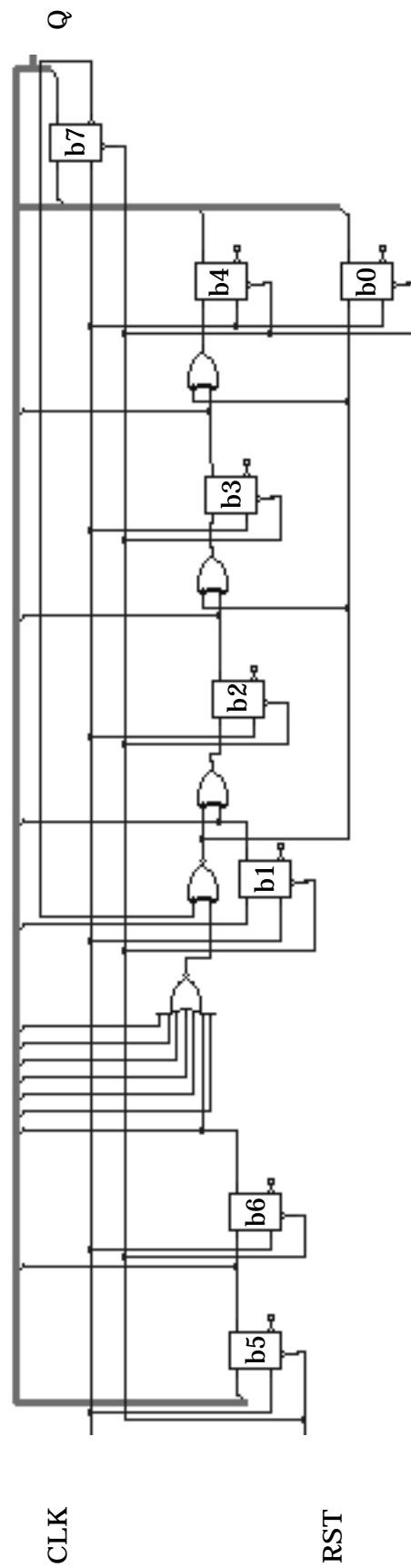
library IEEE;
use IEEE.std_logic_1164.all;

entity LFSR8 is
  port (CLK, RST: in std_logic;
         Q: out std_logic_vector(0 to 7));
end LFSR8;

architecture bhv of LFSR8 is
  -- points de rétroaction
  constant TAPS: std_logic_vector(Q'range) :=
    (1 | 2 | 3 | 7 => '1', others => '0');
  signal LFSR_reg: std_logic_vector(Q'range); -- registre interne
begin
  process (RST, CLK)
    variable bits0_6_zero, feedback: std_logic;
  begin
    if RST = '1' then
      LFSR_reg := (others => '0'); -- initialisation
    elsif CLK'event and CLK = '1' then
      -- logique permettant de générer les  $2^8 = 256$  valeurs possibles
      bits0_6_zero := '1';
      for i in 0 to 6 loop
        if LFSR_reg(i) = '1' then
          bits0_6_zero := '0'; exit;
        end if;
      end loop;
      -- valeur de rétroaction à appliquer
      feedback := LFSR_reg(7) xor bits0_6_zero;
      -- décalage des bits et XOR aux points de rétroaction
      for i in 7 downto 1 loop
        if TAPS(i-1) = '1' then
          LFSR_reg(i) <= LFSR_reg(i-1) xor feedback;
        else
          LFSR_reg(i) <= LFSR_reg(i-1); -- simple décalage
        end if;
      end loop;
      LFSR_reg(0) <= feedback;
    end if;
  end process;
  Q <= LFSR_reg;
end bhv;

```

**Code 47.** Modèle d'un registre LFSR 8 bits générant les  $2^8$  valeurs possibles.



**Figure 50.** Circuit inféré par la synthèse du modèle du [Code 47](#).

## Compteurs

Un compteur est un registre dont les états successifs définissent une séquence prédéfinie de valeurs. Le changement d'état dépend d'impulsions sur une ou plusieurs entrées, usuellement un signal d'horloge. Le [Code 48](#) donne le modèle d'un compteur/décompteur synchrone à  $N=2^P-1$ . Le compteur se remet automatiquement à zéro (resp.  $N$ ) après qu'il ait atteint la valeur  $N$  (resp. 0) en comptant (resp. en décomptant). La [Figure 51](#) donne le circuit inféré en synthèse avec  $P=2$ .

```

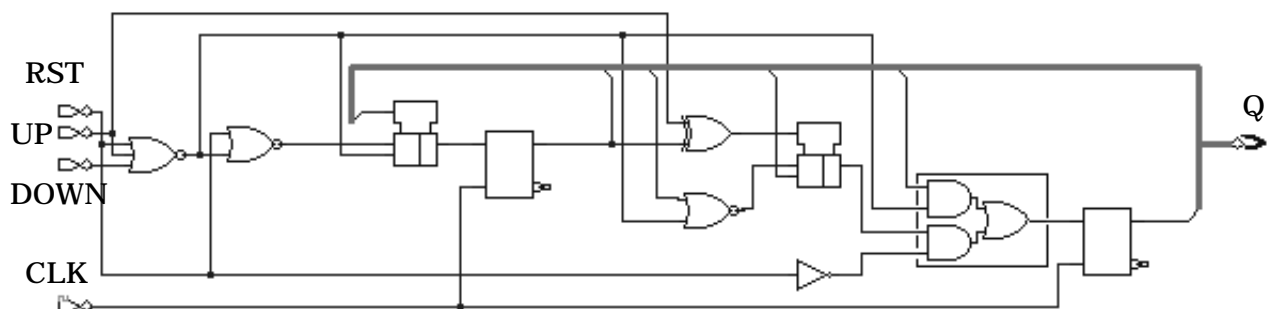
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity rpl_counter_ud is
  generic (NBits: positive);
  port (CLK, RST, UP, DOWN: in bit;
        Q: out std_logic_vector(NBits-1 downto 0));
end rpl_counter_ud;

architecture bhv of rpl_counter_ud is
  signal count_reg: unsigned(Q'range);
begin
  process
  begin
    wait until CLK = '1';
    if RST = '1' then
      count_reg <= (others => '0'); -- initialisation
    elsif UP = '1' then
      count_reg <= count_reg + 1; -- compteur
    elsif DOWN = '1' then
      count_reg <= count_reg - 1; -- décompteur
    end if;
  end process;
  Q <= std_logic_vector(count_reg);
end bhv;

```

**Code 48.** Compteur/décompteur synchrone à  $N = 2^{NBits}-1$ .



**Figure 51.** Circuit inféré en synthèse par le modèle du [Code 48](#) avec  $NBits=2$ .

Un compteur de Johnson est une forme de registre à décalage dont chaque valeur binaire successive ne change que d'un seul bit par rapport à la valeur précédente. L'avantage est que la logique de transition à un nouvel état est simplifiée et ne génère ainsi pas de course. Par exemple, la séquence d'un compteur de Johnson sur 3 bits est:

000 001 011 111 110 100 000 001 ...

Noter que certaines valeurs ne sont jamais générées ("010" et "101") et il s'agit d'éviter que de tels états apparaissent. Un premier modèle serait d'énumérer tous les états possibles au moyen d'une instruction sélective **case**. Le [Code 49](#) donne le modèle d'un tel compteur utilisant une autre technique basée sur un registre à décalage.

```

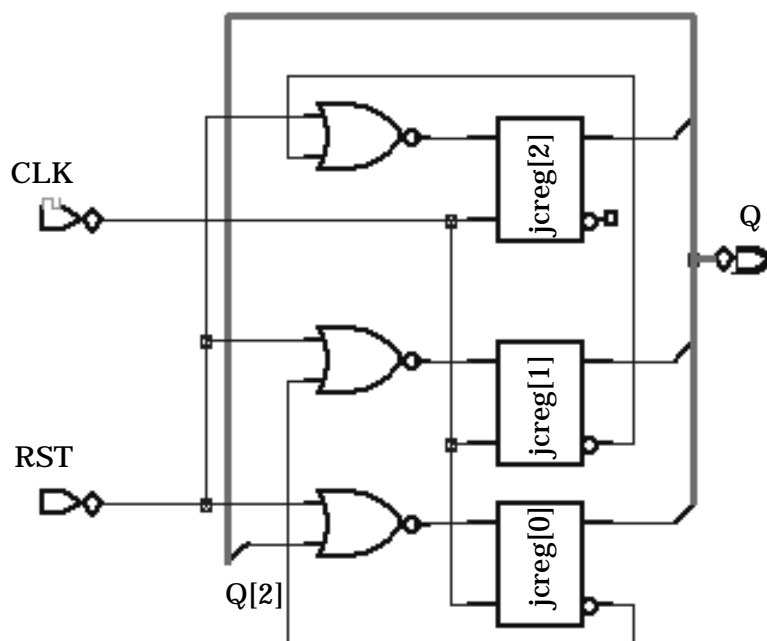
library IEEE;
use IEEE.std_logic_1164.all;

entity johnson_counter is
  generic (NBits: positive);
  port (CLK, RST: in bit;
        Q: out std_logic_vector(NBits-1 downto 0));
end johnson_counter;

architecture bhv of johnson_counter is
  signal jcreg: std_logic_vector(Q'range);
begin
  process
  begin
    wait until CLK = '1';
    if RST = '1' then
      jcreg <= (others => '0'); -- initialisation
    else
      jcreg <= jcreg(NBits-2 downto 0) & not jcreg(NBits-1);
    end if;
  end process;
  Q <= jcreg;
end bhv;

```

**Code 49.** Compteur de Johnson générique basé sur un registre à décalage.



**Figure 52.** Circuit inféré par le modèle du [Code 49](#) avec NBits=3.



### 3.3. Circuits avec état haute-impédance

Des circuits tampons (*buffers*) avec état haute-impédance sont nécessaires dans le cas où plusieurs blocs du circuit ont accès à un même bus. VHDL permet de gérer en simulation un signal multi-sources (le bus) au moyen de fonctions de résolution (page 57). Le type standard résolu `std_logic` définit l'état 'Z' ainsi qu'une fonction de résolution associée (§ B.3.). En synthèse, l'état 'Z' infère aussi des buffers à trois états.

Le Code 50 donne le modèle d'une latch D avec sortie à trois états. La Figure 53 donne le circuit synthétisé.

```

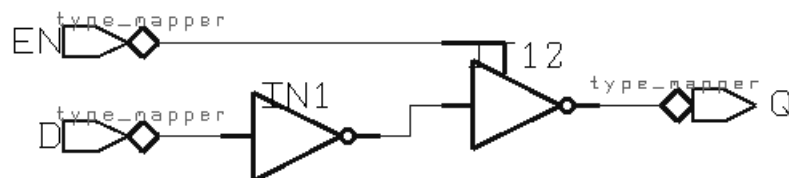
library IEEE;
use IEEE.std_logic_1164.all;

entity dlatchz is
  port (EN, D: in  std_logic;
        Q:      out std_logic);
end dlatchz;

architecture bhv of dlatchz is
begin
  process (EN, D)
  begin
    if EN = '1' then
      Q <= D;
    else
      Q <= 'Z';
    end if;
  end process;
end bhv;

```

**Code 50.** Modèle d'une latch D avec sortie à trois états.



**Figure 53.** Circuit synthétisé à partir du Code 50.

La modélisation d'un flip-flop avec sortie trois états nécessite quelques précautions si l'on ne veut pas inférer trop d'éléments de mémoire. Le [Code 51](#) donne le modèle que l'on serait tenté d'écrire. Le problème est que le signal de sélection du buffer 3 états inféré est aussi synchronisé sur le signal d'horloge. La [Figure 54](#) donne le circuit synthétisé.

```

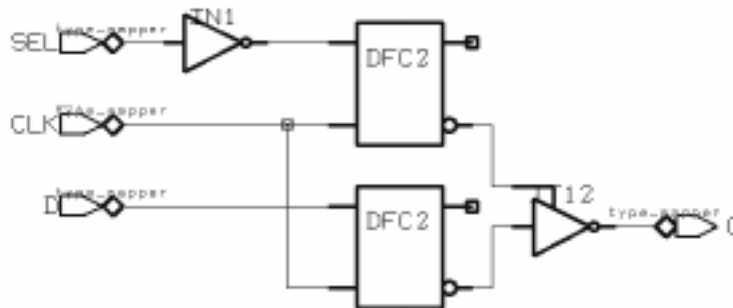
library IEEE;
use IEEE.std_logic_1164.all;

entity dffz is
  port (CLK, SEL, D: in  std_logic;
        Q:          out std_logic);
end dffz;

architecture bhv of dffz is
begin
  FF: process (CLK, SEL, D)
  begin
    if CLK'event and CLK = '1' then
      if SEL = '1' then
        Q <= D;
      else
        Q <= 'Z';
      end if;
    end if;
  end process FF;
end bhv;

```

**Code 51.** Modèle d'un flip-flop avec sortie à trois états.



**Figure 54.** Circuit synthétisé à partir du [Code 52](#).

Le [Code 52](#) donne le modèle d'un flip-flop possédant un contrôle asynchrone du buffer de sortie 3 états. L'usage de deux processus permet de séparer la partie séquentielle du circuit de la partie inférant un buffer à trois états. Le signal de contrôle TS a été ajouté. La [Figure 55](#) donne le circuit synthétisé.

```

library IEEE;
use IEEE.std_logic_1164.all;

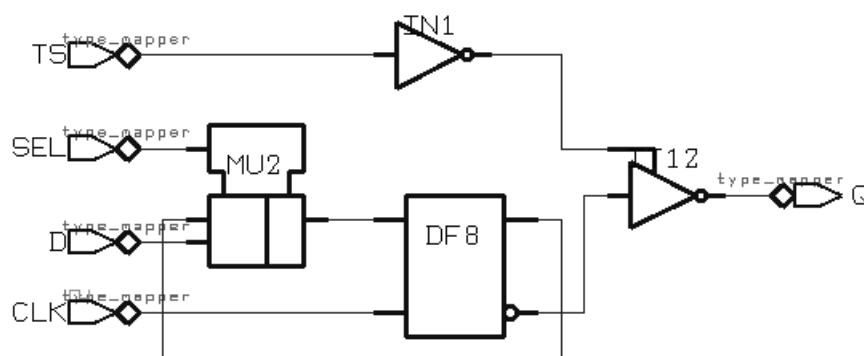
entity dffz is
  port (CLK, TS, SEL, D: in  std_logic;
        Q:                   out std_logic);
end dffz;

architecture bhv of dffz is
  signal S: std_logic;
begin
  FF: process (CLK, SEL, D)
  begin
    if CLK'event and CLK = '1' then
      if SEL = '1' then
        S <= D;
      end if;
    end if;
  end process FF;

  BUF3S: process (TS, S)
  begin
    if TS = '0' then
      Q <= S;
    else
      Q <= 'Z';
    end if;
  end process BUF3S;
end bhv;

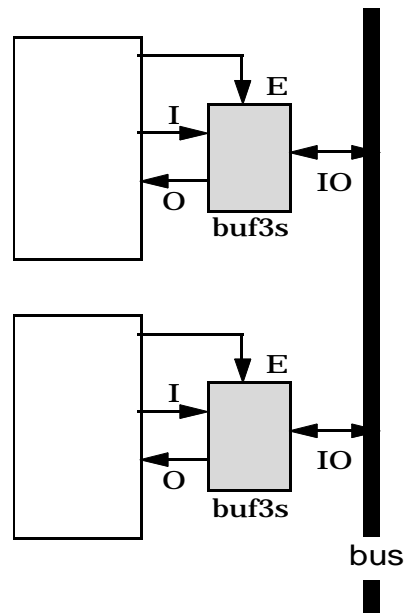
```

**Code 52.** Modèle d'un flip-flop avec sortie à trois états.



**Figure 55.** Circuit synthétisé à partir du [Code 52](#).

Finalement, le Code 53 donne le modèle d'un buffer 3 états générique permettant d'interfacer des blocs quelconques avec un bus (Figure 56). Le modèle utilise une instruction concurrente conditionnelle d'assignation de signal pour définir un comportement à trois états. La Figure 57 donne le circuit synthétisé.



**Figure 56.** Buffer 3 états générique.

```

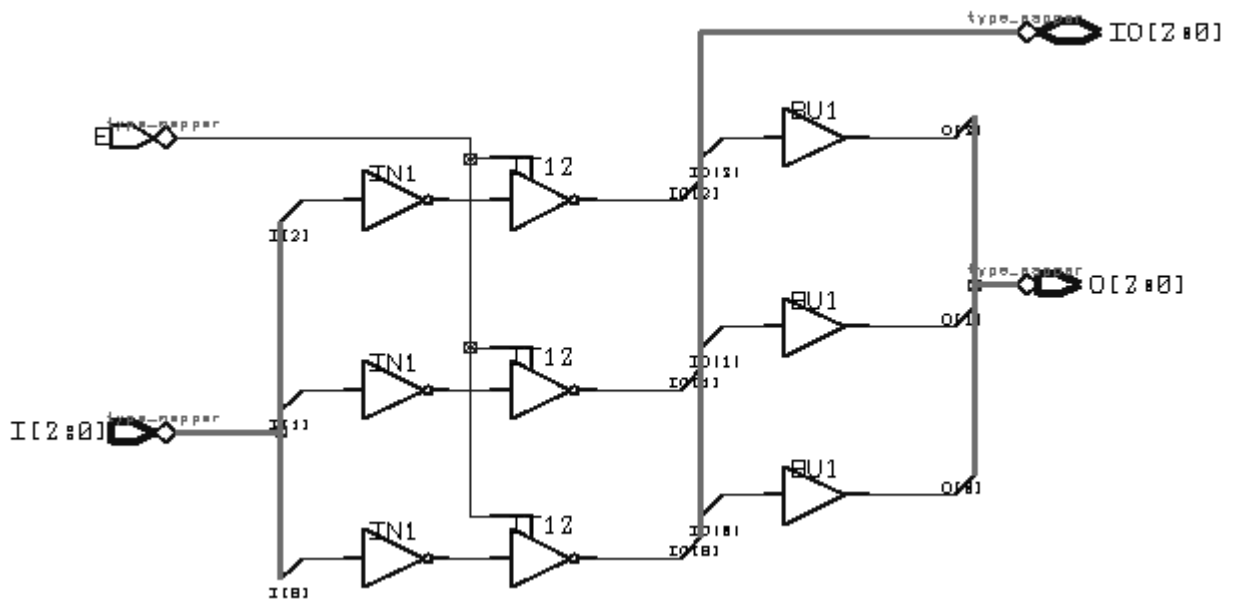
library IEEE;
use IEEE.std_logic_1164.all;

entity buf3s_bidir is
  generic (N: integer);
  port (I: in    std_ulogic_vector(N-1 downto 0);
        E: in    std_logic;
        O: out   std_ulogic_vector(N-1 downto 0);
        IO: inout std_logic_vector(N-1 downto 0));
end buf3s_bidir;

architecture bhv of buf3s_bidir is
begin
  IO <= std_logic_vector(I) when E = '1' else
        (others => 'Z');
  O <= std_ulogic_vector(IO);
end bhv;

```

**Code 53.** Modèle du buffer 3 états générique.



**Figure 57.** Circuit synthétisé à partir du [Code 53](#) avec  $N = 3$ .

Des inverseurs sont inférés sur le bus I parce que les buffers à trois états sont de type inverseur. Des buffers sont inférés sur le bus O pour découpler la sortie et pour forcer le mode **out** du bus. L'état haut-impédance imposé lorsque le signal E est inactif permet l'écriture de l'état du bus dans le module périphérique par l'intermédiaire du port O.



### 3.4. Machines à états finis

Les machines à états finis (*Finite State Machine*, FSM) sont couramment utilisées pour modéliser des séquenceurs ou des blocs de contrôle. De tels circuits contrôlent le fonctionnement de parties opératives (registres, ALUs, accumulateurs, etc.) selon des schémas qui peuvent être arbitrairement complexes.

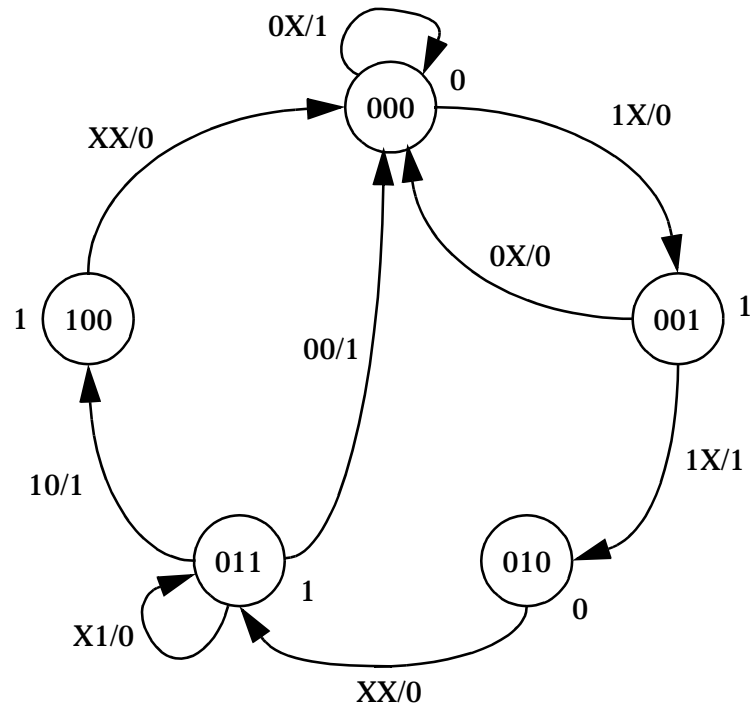
#### Table des états et diagramme d'états

La fonction d'une machine à états finis, ou plus simplement machine d'états, est usuellement représentée de manière équivalente soit par une table d'états soit par un diagramme d'états. La [Table 3.1](#) présente une table d'états définissant pour chaque entrée primaire utile (couple AB; la valeur X dénotant une valeur quelconque 0 ou 1): l'état courant du système avant l'application des entrées, l'état suivant du système après application des entrées et la valeur des sorties du système. Deux types de sorties sont considérés: la sortie de Mealy  $Z\_Me$  dépend de l'état courant et des entrées primaires, la sortie de Moore  $Z\_Mo$  ne dépend que de l'état courant. Les états sont codés sur 3 bits et des identificateurs STx leur sont attribués, avec  $x = 0 \dots 4$ .

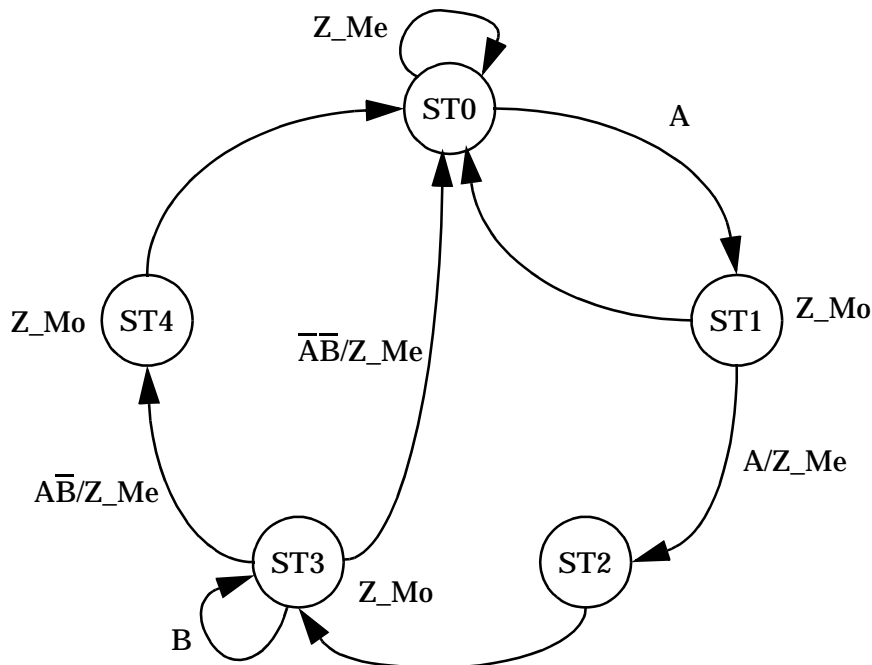
Entrées				Sorties	
A	B	Etat courant		Etat suivant	
		Z_Me	Z_Mo		
0	X	000 (ST0)	000 (ST0)	1	0
1	X	000 (ST0)	001 (ST1)	0	0
0	X	001 (ST1)	000 (ST0)	0	1
1	X	001 (ST1)	010 (ST2)	1	1
X	X	010 (ST2)	011 (ST3)	0	0
X	1	011 (ST3)	011 (ST3)	1	1
0	0	011 (ST3)	000 (ST0)	1	1
1	0	011 (ST3)	100 (ST4)	0	1
X	X	100 (ST4)	000 (ST0)	0	1

**Table 3.1.** Exemple de table d'états avec sorties de Mealy et de Moore.

La [Figure 58](#) et la [Figure 59](#) illustrent deux manières de définir un diagramme d'état correspondant à la [Table 3.1](#). Les cercles représentent les états et les flèches représentent les transitions entre les états. La transition d'un état à l'autre est synchronisée sur signal d'horloge implicite qui n'est représenté ni dans la tables d'états ni dans les diagrammes d'états. Les valeurs des états sont dénotées soit par leurs valeurs binaires ([Figure 58](#)), soit par leurs noms ([Figure 59](#)). Les valeurs des entrées primaires régissant les transitions sont dénotées par des couples AB et les valeurs des sorties primaires de Mealy sont dénotées juste après, séparées par un slash (/). Les valeurs des sorties primaires de Moore sont dénotées à côté du cercle correspondant à l'état qui les génère. Le diagramme d'états de la [Figure 59](#) n'utilise que les valeurs symboliques actives. Les valeurs non mentionnées sont supposées être à 0 (inactives) ou X. Une exception est faite pour distinguer les transitions ST3 à ST4 et ST3 à ST0. Une expression AB dénote un ET logique entre A et B.



**Figure 58.** Diagramme d'états correspondant à la [Table 3.1](#).



**Figure 59.** Diagramme d'états simplifié correspondant à la [Table 3.1](#).

### Structure d'une machine à états finis

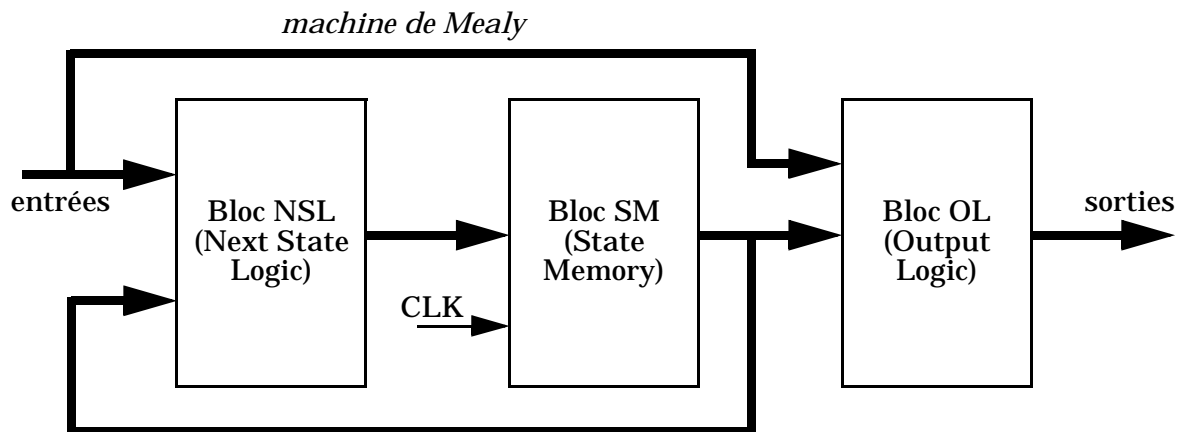
Une machine à états finis possède une structure logique en trois parties ([Figure 60](#)):

- Un registre (appelé SM à la [Figure 60](#)), séquencé par un signal d'horloge, mémorise l'état courant de la machine.



- Un bloc logique (appelé NSL à la Figure 60), usuellement combinatoire, effectue le calcul du prochain état de la séquence.
- Un autre bloc logique (appelé OL à la Figure 60), également usuellement combinatoire, génère les signaux de sortie.

On parle d'une *machine de Moore* lorsque les signaux de sortie ne dépendent que de l'état courant. On parle de *machine de Mealy* lorsque les signaux de sortie dépendent en plus des signaux d'entrée.



**Figure 60.** Structure logique d'une machine à état finis.

La modélisation en VHDL d'une machine à états finis doit considérer les aspects suivants:

- L'utilisation d'un registre d'état explicite (déclaré dans le modèle) ou implicite (utilisation d'instructions `wait` multiples).
- L'utilisation de un, deux ou trois processus.
- Des sorties primaires de Mealy ou de Moore, asynchrones ou synchrones.
- Une initialisation synchrone ou asynchrone de la machine d'états.
- Un encodage approprié des états.

### *Modélisation avec un processus et registre d'état explicite*

Le Code 54 donne le modèle de la machine à états finis dont le comportement est décrit dans la Table 3.1 et dans les diagrammes d'états des Figures 58 et 59. Ce modèle déclare le registre d'état et n'utilise qu'un seul processus. La machine d'états est ainsi définie par une seule instruction sélective `case`. Le registre d'état est déclaré comme un signal, mais il pourrait aussi bien être déclaré comme une variable locale au processus. La machine est initialisée au moyen d'un reset synchrone. Les sorties primaires sont en plus synchrones. L'encodage des états est effectué par défaut de manière binaire dans l'ordre de déclaration des états énumérés: ST0 ("000"), ST1 ("001"), ST2 ("010"), ST3 ("011") et ST4 ("100").

```

library IEEE;
use IEEE.std_logic_1164.all;

entity FSM is
  port (CLK, RST:   in  std_logic;  -- signaux de contrôle
        A, B:      in  std_logic;  -- entrées primaires
        Z_Me, Z_Mo: out std_logic); -- sorties primaires
end FSM;
```

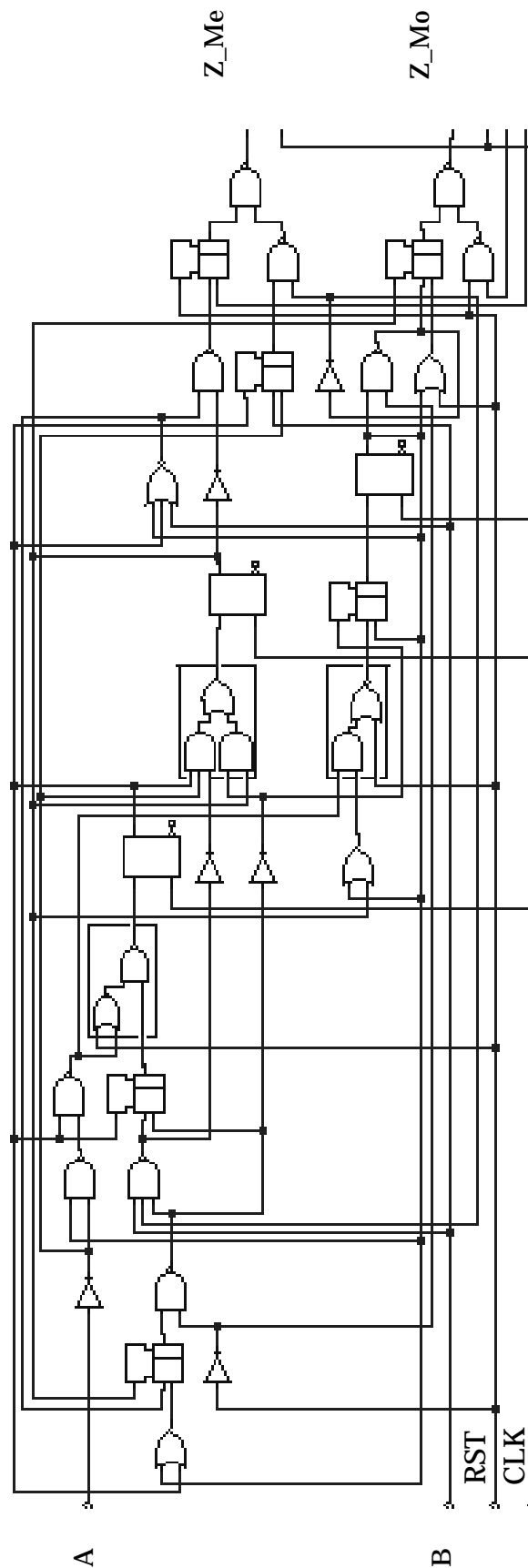
```

architecture oneproc of FSM is
  type fsm_states is (ST0, ST1, ST2, ST3, ST4);
  signal state: fsm_states;
begin
  process
  begin
    wait until CLK = '1';
    if RST = '1' then          -- reset synchrone
      state <= ST0;
    else
      case state is
        when ST0 => -----
          Z_Mo <= '0';        -- sortie de Moore
          if A = '0' then
            Z_Me <= '1';     -- sortie de Mealy
          else
            Z_Me <= '0';
            state <= ST1;
          end if;
        when ST1 => -----
          Z_Mo <= '1';
          if A = '1' then
            Z_Me <= '1';
            state <= ST2;
          else
            Z_Me <= '0';
            state <= ST0;
          end if;
        when ST2 => -----
          Z_Mo <= '0';
          Z_Me <= '0';
          state <= ST3;
        when ST3 => -----
          Z_Mo <= '1';
          if B = '1' then
            Z_Me <= '0';
            state <= ST3;
          else
            Z_Me <= '1';
            if A = '1' then
              state <= ST4;
            else
              state <= ST0;
            end if;
          end if;
        when ST4 => -----
          Z_Mo <= '1';
          Z_Me <= '0';
          state <= ST0;
      end case;
    end if;
  end process;
end oneproc;

```

**Code 54.** Modèle de la machine à états finis de la [Table 3.1](#) avec un processus.

La [Figure 61](#) donne le circuit synthétisé pour la machine à états finis du [Code 54](#).



**Figure 61.** Circuit synthétisé à partir du modèle du [Code 54](#).

### Modélisation avec trois processus et registre d'état explicite

L'usage de trois processus permet d'implémenter la structure de la [Figure 60](#) et de faire apparaître explicitement les trois blocs NSL, SM et OL. Il permet aussi de mieux contrôler la génération de circuits combinatoires et séquentiels. L'usage de deux signaux locaux `current_state` et `next_state` devient nécessaire, mais un seul registre d'état est effectivement inféré en synthèse. Les signaux de sortie `Z_Mo` et `Z_Me` ne sont plus synchrones.

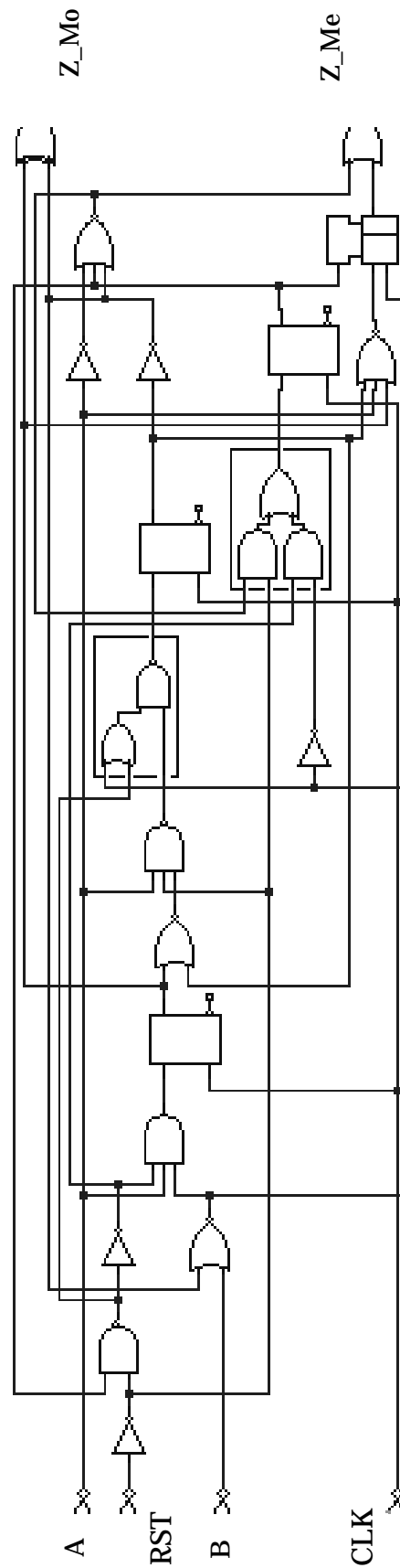
```

architecture threeproc of FSM is
  type fsm_states is (ST0, ST1, ST2, ST3, ST4);
  signal current_state, next_state: fsm_states;
begin
  -----
  NSL: process (current_state) -- processus combinatoire
  begin
    next_state <= current_state; -- pour éviter un registre
    case current_state is
      when ST0 => if A = '1' then next_state <= ST1; end if;
      when ST1 => if A = '1' then next_state <= ST2;
                    else next_state <= ST0; end if;
      when ST2 => next_state <= ST3;
      when ST3 => if B = '1' then next_state <= ST3;
                    elsif A = '1' then next_state <= ST4;
                    else next_state <= ST0; end if;
      when ST4 => next_state <= ST0;
    end case;
  end process NSL;
  -----
  SM: process -- processus séquentiel avec reset synchrone
  begin
    wait until CLK = '1';
    if RST = '1' then current_state <= ST0;
    else current_state <= next_state; end if;
  end process SM;
  -----
  OL: process (current_state) -- processus combinatoire
  begin
    case current_state is
      when ST0 => Z_Mo <= '0';
                    if A = '0' then Z_Me <= '1';
                    else Z_Me <= '0'; end if;
      when ST1 => Z_Mo <= '1';
                    if A = '1' then Z_Me <= '1';
                    else Z_Me <= '0'; end if;
      when ST2 => Z_Mo <= '0'; Z_Me <= '0';
      when ST3 => Z_Mo <= '1';
                    if B = '1' then Z_Me <= '0';
                    else Z_Me <= '1'; end if;
      when ST4 => Z_Mo <= '1'; Z_Me <= '0';
    end case;
  end process OL;
end threeproc;

```

**Code 55.** Modèle de la machine à états finis de la [Table 3.1](#) avec trois processus.

La [Figure 62](#) donne le circuit synthétisé pour la machine à états finis du [Code 55](#).



**Figure 62.** Circuit synthétisé à partir du modèle du [Code 55](#).

### Modélisation avec registre d'état implicite

L'usage d'un processus avec plusieurs instructions **wait** définit une machine à états finis dont les états ne sont pas explicitement nommés. Chaque instruction **wait** définit un point de synchronisation représentant chacun un état du système (Code 56).

```
process
begin
  wait until CLK = '1';
  -- traitement de l'état 1
  wait until CLK = '1';
  -- traitement de l'état 2
  ...
  wait until CLK = '1';
  -- traitement de l'état N
end process;
```

**Code 56.** Machine à états finis avec états implicites.

Le mécanisme d'initialisation devient un peu plus lourd que dans le cas où les états sont explicites. Le Code 57 illustre le mécanisme de reset synchrone. Le code relatif au traitement de l'état 1 peut être factorisé dans une procédure pour réduire la taille du code, mais la redondance du traitement de l'initialisation reste inévitable.

```
process
begin
  loop
    wait until CLK = '1';
    if RST = '1' then
      -- traitement de l'état 1
      exit;
    end if;
    -- traitement de l'état 2
    ...
    wait until CLK = '1';
    if RST = '1' then
      -- traitement de l'état 1
      exit;
    end if;
    -- traitement de l'état N
    wait until CLK = '1';
    loop
      -- traitement de l'état 1
      exit when RST = '0';
    end loop;
  end loop;
end process;
```

**Code 57.** Machine à états finis avec états implicites et reset synchrone.

Le [Code 58](#) illustre le mécanisme de reset asynchrone. Il faut noter que la forme de l'instruction `wait` requise n'est pas forcément supportée par tous les outils de synthèse actuels.

```

process
begin
  RST_LABEL: loop
    if RST = '1' then
      -- initialisation
    end if;
    wait until (CLK'event and CLK = '1') or RST = '1';
    next RST_LABEL when RST = '1';
    -- traitement de l'état 1
    wait until (CLK'event and CLK = '1') or RST = '1';
    next RST_LABEL when RST = '1';
    -- traitement de l'état 2
    ...
    wait until (CLK'event and CLK = '1') or RST = '1';
    next RST_LABEL when RST = '1';
    -- traitement de l'état N
  end loop RST_LABEL;
end process;

```

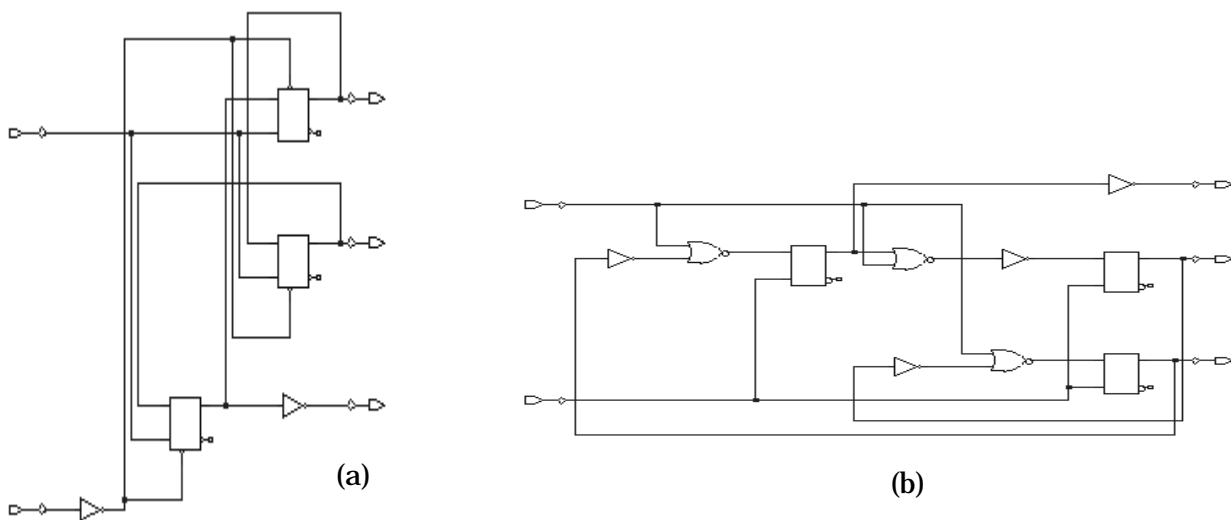
**Code 58.** Machine à états finis avec états implicites et reset asynchrone.

### *Reset synchrone/asynchrone et comportement sain*

Une machine à états finis à  $n$  flip-flops possède  $2^n$  états binaires possibles, mais souvent seul un sous-ensemble de ces états est effectivement utile (p.ex. une machine à cinq états nécessite trois flip-flops au minimum et possède ainsi  $2^3 - 5 = 3$  états invalides). Il s'agit dès lors d'éviter que la machine se retrouve par erreur dans un état invalide, ce qui modifierait son comportement de manière imprévisible.

Une première solution est d'assurer l'initialisation de la machine à un état valide. Le reset asynchrone a l'avantage de mettre la machine dans un état valide quel que soit l'état du signal d'horloge. Il n'infère pas en plus de logique supplémentaire car il utilise directement les entrées de reset asynchrone des flip-flops de la bibliothèque de cellules standard (si ils existent). Le reset synchrone requiert quant-à-lui plus de logique combinatoire. De plus, l'état initial du circuit au démarrage et avant le reset synchrone est imprévisible et peut bloquer le système de manière permanente. Par contre, les flip-flops avec reset asynchrone sont généralement plus grands en surface que ceux avec reset synchrone.

La [Figure 63](#) illustre la synthèse d'une machine à états finis utilisant un encodage des états "one-hot". Le circuit de gauche utilise un reset asynchrone et celui de droite utilise un reset synchrone. La version avec reset asynchrone donne un circuit environ 15% moins grand que la version avec reset synchrone.



**Figure 63.** Circuit inféré en synthèse (a) avec un reset asynchrone (surface: 9.62) et (b) avec un reset synchrone (surface: 11.53).

Une deuxième solution, complémentaire, est d'assurer que le calcul du prochain état ne peut qu'aboutir à un état valide. Ceci est possible en ajoutant par exemple une clause **when others** au processus NSL du [Code 55](#) en assignant le prochain état `next_state` à un état valide (ST0 p.ex.).

### Encodage des états

L'encodage des états consiste à attribuer à chaque état un nombre binaire unique. Il existe plusieurs formats possibles:

- L'encodage *séquentiel* est l'encodage par défaut pour les outils de synthèse. A chaque état est assigné un nombre binaire croissant à partir d'une chaîne de  $n$  zéros.
- Les encodages *de Gray* et *de Johnson* assignent des valeurs binaires successives qui ont la particularité de ne changer que d'un seul bit d'un nombre à l'autre. L'avantage est de réduire les erreurs de transitions possibles dues à des changements sur des entrées asynchrones. L'encodage de Gray utilise tous les  $2^n$  états possibles alors que l'encodage de Johnson revient à implémenter un registre d'état à décalage ([page 103](#)) et requiert ainsi plus de flip-flops.
- L'encodage "*one-hot*" n'utilise qu'un seul flip-flop par état, ce qui conduit à des circuits plus rapides, mais aussi plus grands en surface.
- Les outils de synthèse permettent aussi de définir un encodage personnalisé.

La [Table 3.2](#) donne les valeurs d'encodage pour  $n = 16$  et pour chacun des styles d'encodage ci-dessus.



No.	Séquentiel	Gray	Johnson	One-hot
0	0000	0000	00000000	0000000000000001
1	0001	0001	00000001	0000000000000010
2	0010	0011	00000011	0000000000000100
3	0011	0010	00000111	00000000000001000
4	0100	0110	00001111	0000000000010000
5	0101	0111	00011111	0000000001000000
6	0110	0101	00111111	0000000010000000
7	0111	0100	01111111	0000000010000000
8	1000	1100	11111111	0000000100000000
9	1001	1101	11111110	0000001000000000
10	1010	1111	11111100	0000010000000000
11	1011	1110	11111000	0000100000000000
12	1100	1010	11110000	0001000000000000
13	1101	1011	11100000	0010000000000000
14	1110	1001	11000000	0100000000000000
15	1111	1000	10000000	1000000000000000

**Table 3.2.** Formats d'encodages standard de machines à états finis pour  $n = 16$ .

Le contrôle de l'encodage utilise une approche qui dépend de l'outil de synthèse. Dans le cas de Synopsys, il est possible d'utiliser un attribut spécifique, nommé ENUM\_ENCODING. Par exemple, si l'on reprend le [Code 55](#), il s'agirait de le modifier comme indiqué dans l'extrait du [Code 59](#) pour spécifier un codage de Gray. La [Figure 64](#) donne le circuit synthétisé. A comparer avec celui obtenu à la [Figure 62](#) qui utilisait un encodage binaire par défaut.

```

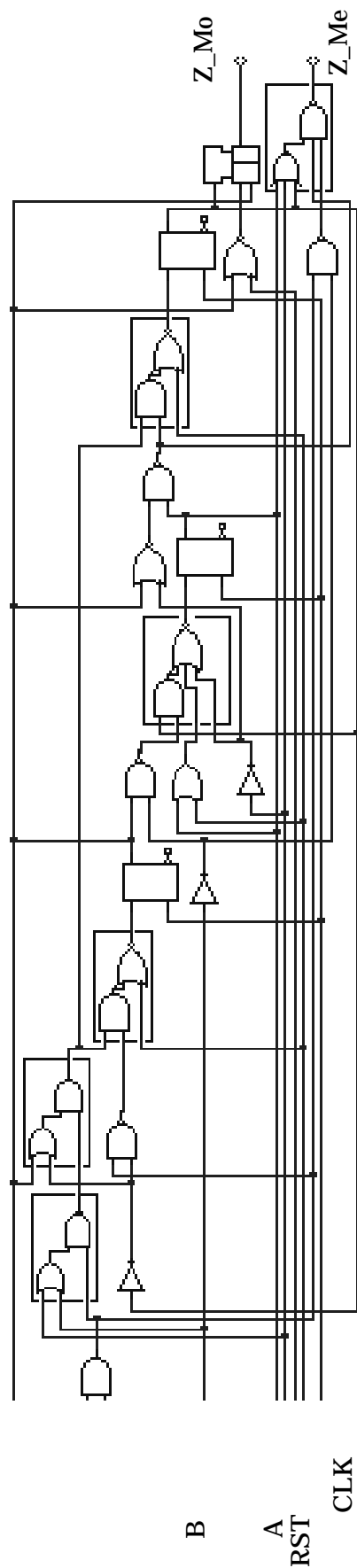
library SYNOPSYS;
use SYNOPSYS.attributes.all;

architecture threeproc of FSM is
  type fsm_states is (ST0, ST1, ST2, ST3, ST4);
  attribute ENUM_ENCODING of fsm_states: type is
    "000 001 010 110 111";
  signal current_state, next_state: fsm_states;
begin
  ...
end threeproc;

```

**Code 59.** Spécification d'un encodage de Gray.

Il faut noter que l'outil Synopsys fournit d'autres moyens plus puissants pour la synthèse de machines à états finis sous la forme de commandes de synthèse (identification du registre d'état, définition du style d'encodage, minimisation des états, etc.). Ceci sort toutefois du cadre de ce document car le code VHDL lui-même n'est plus concerné.

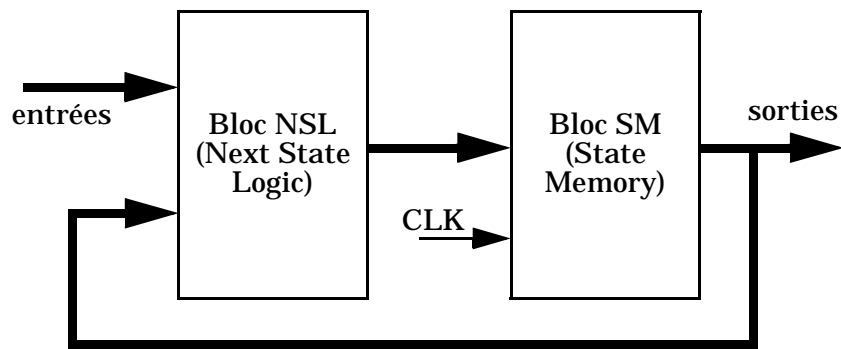


**Figure 64.** Synthèse de la machine à états finis du [Code 59](#).

### Variantes possibles de machines d'états

En plus des structures traditionnelles de Moore ou de Mealy, il existe des variantes qui permettent principalement d'éviter de produire des signaux de sortie ayant des valeurs temporairement invalides dues à la propagation des signaux dans la partie combinatoire du bloc OL (Figure 60).

Une première solution est d'utiliser les valeurs de sortie possibles pour l'encodage des états. Le bloc OL est ainsi éliminé et les sorties s'obtiennent directement du bloc SM. La Figure 65 illustre la structure d'une telle machine d'états.



**Figure 65.** Machine de Moore avec encodage d'états par les valeurs de sortie.

Le Code 60 donne le modèle d'une telle machine de Moore pour l'exemple de la Table 3.1 et en ne considérant que la colonne Z\_Mo. On suppose que la valeur de la sortie Z\_Mo est encodée dans le bit de poids faible du vecteur d'états. Les vecteurs d'états sont déclarés comme des constantes de type `std_logic_vector` pour pouvoir accéder au bit de poids faible. Le corps de l'architecture utilise les deux processus NSL et SM du Code 55 tels quels, seul le processus OL étant remplacé par une simple assignation concurrente de signal.

Si l'on conserve la structure en trois processus, il est possible de synchroniser les signaux de sortie sur le signal d'horloge de la machine ou sur une autre horloge (p. ex. en opposition de phase). Il est aussi possible d'inférer des latches plutôt que des flaps-flops en rendant les sorties disponibles au moyen d'un signal de contrôle asynchrone (p. ex. ENABLE). Il est aisé de modifier le processus OL pour réaliser ces différents cas.

```

library IEEE;
use IEEE.std_logic_1164.all;

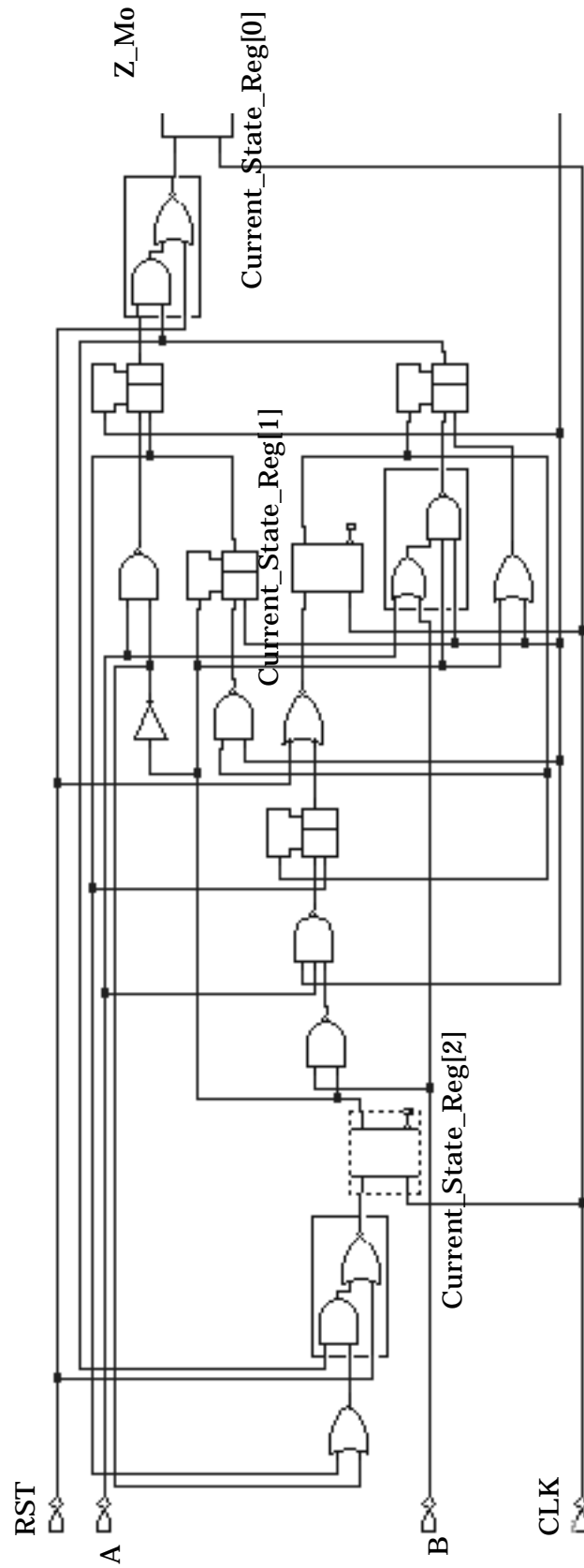
entity FSM is
  port (CLK, RST: in std_logic; -- signaux de contrôle
        A, B: in std_logic; -- entrées primaires
        Z_Mo: out std_logic); -- sortie de Moore
end FSM;

architecture MooreEnc of FSM is
  constant ST0: std_logic_vector := "000"; -- encodage des états
  constant ST1: std_logic_vector := "001";
  constant ST2: std_logic_vector := "010";
  constant ST3: std_logic_vector := "101";
  constant ST4: std_logic_vector := "111";
  signal current_state, next_state: std_logic_vector(2 downto 0);
begin
  -----
  NSL: process (current_state, A, B) -- processus combinatoire
  begin
    next_state <= current_state; -- pour éviter un registre
    case current_state is
      when ST0 => if A = '1' then next_state <= ST1; end if;
      when ST1 => if A = '1' then next_state <= ST2;
                    else next_state <= ST0; end if;
      when ST2 => next_state <= ST3;
      when ST3 => if B = '1' then next_state <= ST3;
                    elsif A = '1' then next_state <= ST4;
                    else next_state <= ST0; end if;
      when ST4 => next_state <= ST0;
      when others => next_state <= current_state;
                    -- pour couvrir tous les cas possibles

    end case;
  end process NSL;
  -----
  SM: process -- processus séquentiel avec reset synchrone
  begin
    wait until CLK = '1';
    if RST = '1' then
      current_state <= ST0;
    else
      current_state <= next_state;
    end if;
  end process SM;
  -----
  Z_Mo <= current_state(0);
end MooreEnc;

```

**Code 60.** Modèle d'une machine de Moore avec valeurs de sortie encodées dans les états de la machine.



**Figure 66.** Circuit synthétisé à partir du modèle du Code 60.

### 3.5. Opérateurs arithmétiques

Les opérateurs arithmétiques font partie de l'unité arithmétique et logique (ALU) d'un processeur. En général, les opérateurs effectivement réalisés sur le circuit sont des additionneurs/soustracteurs et les autres opérations (multiplication et division) sont réalisées avec ces circuits plus des registres et de la logique de contrôle supplémentaires.

Des opérations complexes de traitement du signal nécessitent principalement des opérateurs d'addition et de multiplication rapides. Il s'agit donc de disposer de plusieurs structures d'implémentation possibles en fonction des contraintes surface/délai imposées.

#### *Additionneurs/Soustracteurs*

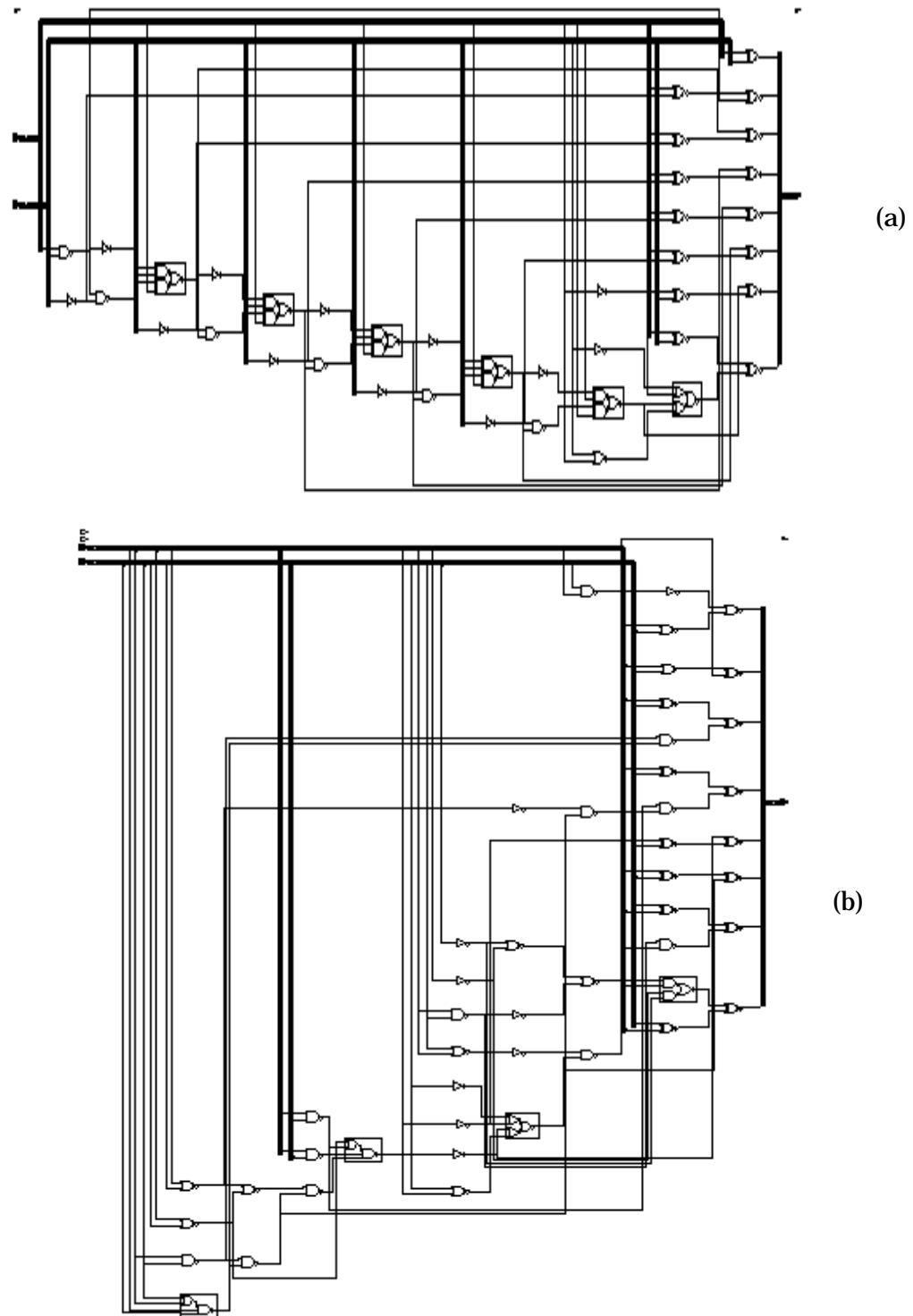
La manière la plus simple d'inférer un additionneur est d'utiliser l'opérateur "+" dans un modèle. L'outil de synthèse Synopsys dispose d'une bibliothèque de modèles d'additionneurs prédéfinis (appelée DesignWare). Un soustracteur utilise usuellement un additionneur dont l'un des opérandes est représenté en complément à deux:  $A - B$  est équivalent à  $A + \text{cpl2}(B)$  où  $\text{cpl2}(B) = \text{cpl}(B) + 1$  (p. ex. si  $B = "010"$ ,  $\text{cpl2}(B) = "101" + "001" = "110"$ ).

Le [Code 61](#) donne le modèle générique d'un additionneur  $N$  bits réalisé par une architecture flot de données. Les opérandes et le résultat sont du type entier signé et le modèle tient compte d'une saturation à 0 et à  $2^N - 1$  (processus P\_SUM). Le processus P\_SUM peut inférer différentes structures d'additionneurs selon les contraintes de synthèse imposées. La [Figure 67](#) illustre l'inférence d'un additionneur à propagation de retenue (*ripple carry adder*) ou d'un additionneur à anticipation de retenue (*carry look-ahead adder*), ce dernier offrant une vitesse de calcul plus rapide au détriment d'une surface plus grande.

```
entity addern is
  generic (N: positive);
  port (A, B: in natural range 2**N-1 downto 0;
        S: out natural range 2**N-1 downto 0);
end;

architecture dfl of addern is
  signal SUM: integer range -2**(N+1) to 2**(N+1)-1;
begin
  P_SUM: SUM <= A + B;
  P_SAT:
    S <= 0      when SUM < 0      else -- saturation à 0
          2**N-1 when SUM >= 2**N else -- saturation à 2**N-1
          SUM;  -- résultat correct
end dfl;
```

**Code 61.** Modèle flot de données d'un additionneur générique  $N$  bits.



**Figure 67.** Inférence d'un additionneur (a) à propagation de retenue ou (b) à anticipation de retenue en fonction des contraintes de synthèse.

Il reste toutefois nécessaire de modéliser la structure d'un additionneur de manière plus fine si l'on veut mieux contrôler les performances obtenues ou si l'on désire implémenter une structure spécifique (p. ex. un additionneur série).

Un additionneur de petite taille ( $N < 8$  bits) est généralement du type à propagation de retenue (*ripple carry*) dont le temps de calcul est proportionnel à  $N$ . La cellule de base est l'additionneur complet 1 bit dont les équations logiques sont:

$$\begin{aligned} \text{SUM} &= A \oplus B \oplus \text{CIN} \\ \text{COUT} &= (A \cdot B) + (A \cdot \text{CIN}) + (B \cdot \text{CIN}) \end{aligned}$$

Le [Code 62](#) donne le modèle de l'additionneur complet 1 bit et la [Figure 68](#) donne le circuit synthétisé.

```

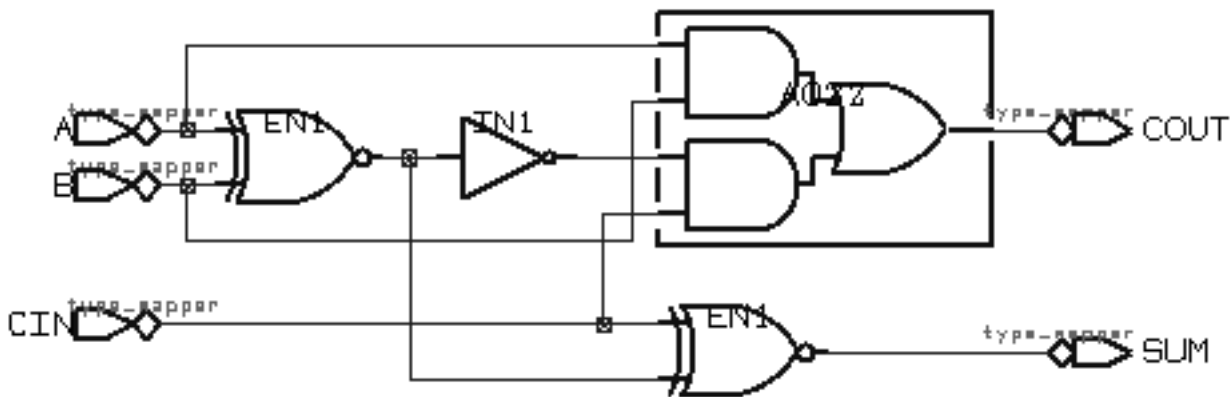
library IEEE;
use IEEE.std_logic_1164.all;

entity fadd1b is
  port (A, B, CIN: in std_logic;
         SUM, COUT: out std_logic);
end fadd1b;

architecture dfl of fadd1b is
begin
  SUM <= A xor B xor CIN;
  COUT <= (A and B) or (A and CIN) or (B and CIN);
end;

```

**Code 62.** Modèle d'un additionneur 1 bit complet.



**Figure 68.** Circuit synthétisé par le modèle du [Code 62](#).



On se propose de réaliser un additionneur/soustracteur  $A \pm B$  dont l'argument A est un mot de 4 bits et l'argument B est un mot de 2 bits (Figure 69). Un signal de contrôle SubAddBar définit si l'on doit effectuer une soustraction (SubAddBar = '1') ou une addition (SubAddBar = '0'). Le XOR des bits de l'entrée B avec SubAddBar fournit le complément à 1 de B. Le complément à 2 de B est obtenu en connectant le signal SubAddBar au signal CIN du premier additionneur 1 bit.

Si une addition provoque un dépassement de capacité vers la haut (*overflow*), ou si une soustraction provoque un dépassement de capacité vers la bas (*underflow*), il s'agit de forcer un "1111", respectivement un "0000", à la sortie. Ces cas sont détectés lorsque SubAddBar = '0' et COUT[3] = '1', respectivement lorsque SubAddBar = '1' et COUT[3] = '0'. Cette tâche est effectuée par un bloc, ou un processus, séparé.

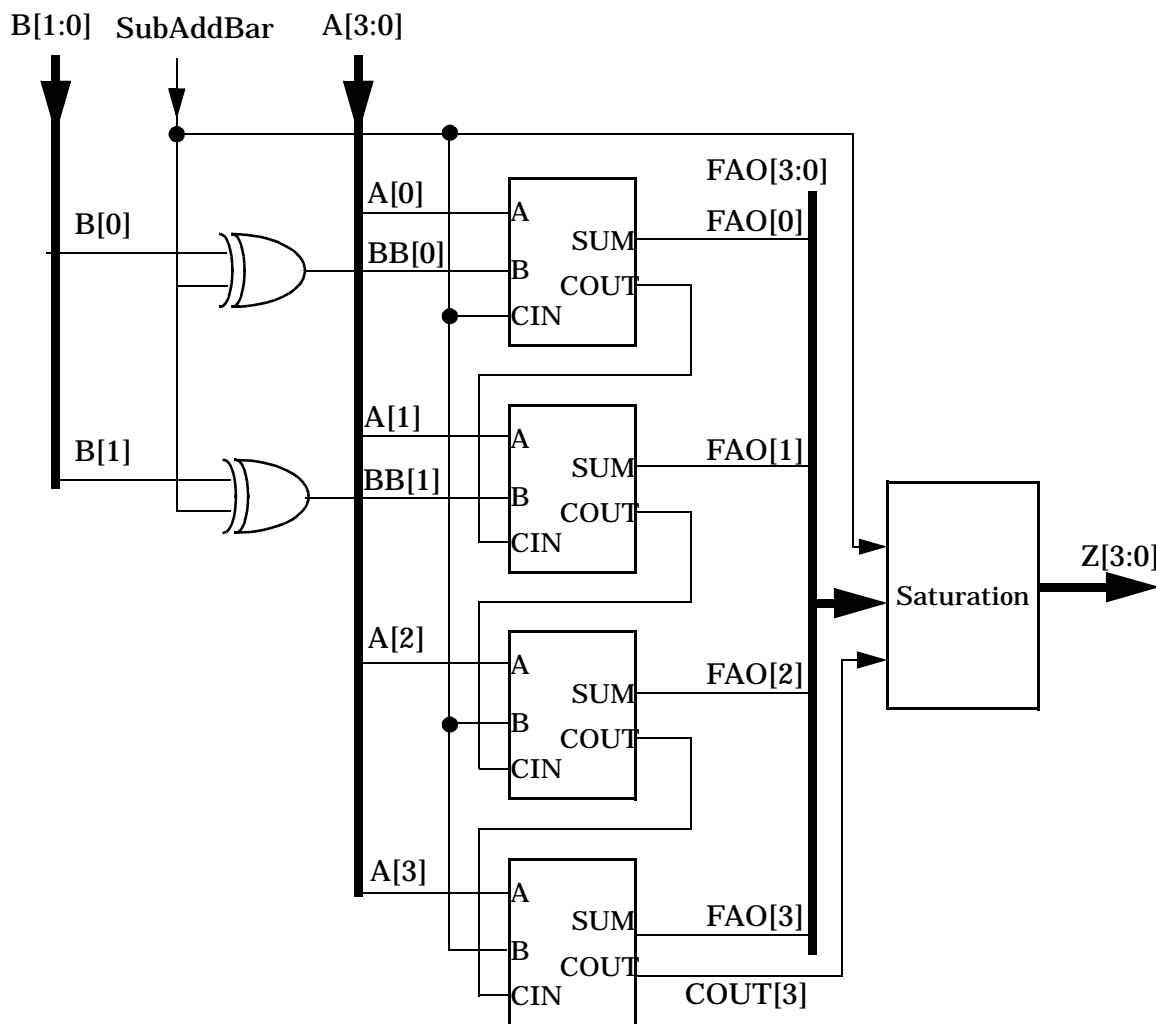


Figure 69. Structure d'un additionneur/soustracteur.

Le Code 63 donne le modèle d'un tel additionneur/soustracteur. Le modèle utilise l'instruction **generate** pour être indépendant de la taille des mots d'entrée. Le modèle tel qu'il est donné n'est pas pour autant complètement générique, mais il pourrait le devenir si les tailles des bus de données étaient passées comme paramètres.

Il faut noter que la propagation de la retenue constitue le chemin critique du circuit, quelque soit la taille des opérandes.

```

library IEEE;
use IEEE.std_logic_1164.all;

entity addsub42 is
  port (SubAddBar: in std_logic; -- commande addition/soustraction
        A:          in std_logic_vector(3 downto 0);
        B:          in std_logic_vector(1 downto 0);
        Z:          out std_logic_vector(3 downto 0));
end addsub42;

architecture RTL of addsub42 is
  component fadd1b
    port (A, B, CIN: in std_logic; SUM, COUT: out std_logic);
  end component;
  signal BB: std_logic_vector(B'range); -- complément à 1 de B
  signal FAO, COUT: std_logic_vector(A'range);
  signal COUT_MSB: std_logic;
begin

  P_INVB: process (SubAddBar, B) -- calcul du complément à 1 de B
  begin
    for i in B'range loop
      BB(i) <= B(i) xor SubAddBar;
    end loop;
  end process P_INVB;

  -- Génération des instances d'additionneurs
  G_ALL_FA: for i in 0 to A'length-1 generate
    -- première cellule FA
    G_FIRST_FA: if i = 0 generate
      FAO: fadd1b port map (A    => A(i),
                           B     => BB(i),
                           CIN  => SubAddBar,
                           SUM   => FAO(i)
                           COUT  => COUT(i));
    end generate G_FIRST_FA;

    -- autres cellules FA avec entrées BB
    G_BB_FA: if i > 0 and i < B'length generate
      FABB: fadd1b port map (A    => A(i),
                           B     => BB(i),
                           CIN  => COUT(i-1),
                           SUM   => FAO(i)
                           COUT  => COUT(i));
    end generate G_BB_FA;

    -- reste des cellules FA avec entrées SubAddBar
    G_SA_FA: if i >= B'length generate
      FASA: fadd1b port map (A    => A(i),
                           B     => SubAddBar,
                           CIN  => COUT(i-1),
                           SUM   => FAO(i)
                           COUT  => COUT(i));
    end generate G_SA_FA;

```

```

end generate G_ALL_FA;

-- détection de la saturation
COUT_MSB <= COUT(A'length-1);
P_SAT: process (COUT_MSB, SubAddBar, FAO)
begin
  if SubAddBar = '0' and COUT_MSB = '1' then
    Z <= (others => '1'); -- overflow
  elsif SubAddBar = '1' and COUT_MSB = '0' then
    Z <= (others => '0'); -- underflow
  else
    Z <= FAO;
  end if;
end process P_SAT;
end RTL;

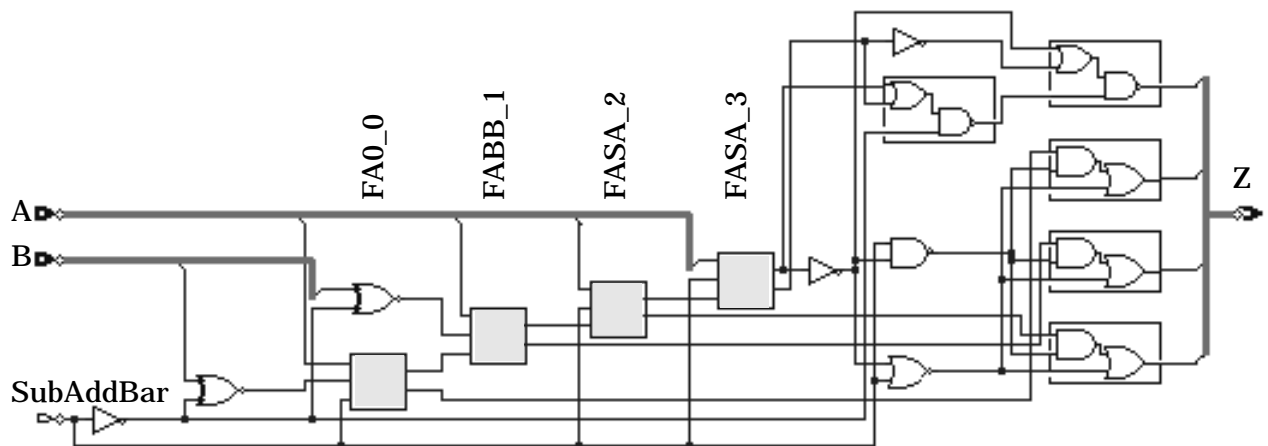
```

**Code 63.** Modèle de l'additionneur/soustracteur de la Figure 69.

Quelques remarques liées au synthétiseur Synopsys:

- Le synthétiseur impose que les noms des signaux apparaissant dans une liste de sensibilité soient simples, c'est-à-dire que des composants de tableaux ne sont pas acceptés. Il a donc fallu déclarer un signal COUT\_MSB pour le processus P\_SAT qui réalise la saturation à "1111" ou à "0000".
- Synopsys ne reconnaît en effet que la configuration par défaut. Il faut donc que la déclaration du composant additionneur complet 1 bit (fadd1b) ait une signature (noms, modes, et types) rigoureusement identiques à celle de l'entité qui sera effectivement utilisée.
- Les quatre instances de l'additionneur 1 bit doivent être préalablement rendues uniques (*uniquify*) avant l'optimisation et l'assignation des portes logiques.

La Figure 70 illustre le circuit synthétisé. On retrouve bien la structure prévue à la Figure 69.



**Figure 70.** Circuit synthétisé à partir du Code 63.

Une structure d'additionneur plus rapide peut être déterminée si l'on considère que lors l'addition de deux nombres  $A$  et  $B$  en notation binaire, si  $A_i$  est égal à  $B_i$  (l'indice  $i$  dénotant le bit de poids  $i$ ) il n'est pas nécessaire de connaître la retenue entrante  $C_i$  (et donc inutile d'attendre qu'elle soit calculée) pour calculer la retenue sortante  $C_{i+1}$  et la somme  $S_{i+1}$ . En effet:

- Si  $A_i = B_i = 0$ ,  $C_{i+1}$  sera forcément nul.
- Si  $A_i = B_i = 1$ ,  $C_{i+1}$  sera forcément égal à 1.

Donc, si  $A_i = B_i$  on peut directement additionner les bits de rang  $i+1$  sans attendre l'arrivée de la retenue  $C_{i+1}$ . Ceci amène aux principes de **propagation** et de **génération**:

- Si  $A_i \neq B_i$ , alors  $C_{i+1} = C_i$  (propagation). On note  $P_i = A_i \oplus B_i$  le signal de propagation du bit  $i$ .
- Si  $A_i = B_i$ , alors il y a génération de  $C_{i+1} = 0$  ou 1. On note  $G_i = A_i \cdot B_i$  le signal de génération du bit  $i$ .

L'étage  $i$  de l'additionneur reçoit ainsi une retenue entrante  $C_i$  si et seulement si:

- l'étage  $i-1$  a généré une retenue ( $G_{i-1} = 1$ ), ou si
- l'étage  $i-1$  a propagé une retenue générée à l'étage  $i-2$  ( $P_{i-1} = G_{i-2} = 1$ ), ou si
- les étages  $i-1$  et  $i-2$  ont propagé une retenue générée à l'étage  $i-3$  ( $P_{i-1} = P_{i-2} = G_{i-3} = 1$ ), ou si
- les étages  $i-1, i-2, i-3, \dots, 1, 0$  ont propagé une retenue entrant dans l'additionneur ( $P_{i-1} = P_{i-2} = \dots = P_1 = P_0 = C_0 = 1$ ).

Il faut noter que ces propriétés s'appliquent à tout bloc (tranche) de bits de taille quelconque  $k$  et on peut ainsi considérer  $C_{-1}$  et  $C_{k-1}$  comme les retenues entrante et sortante de ce bloc.

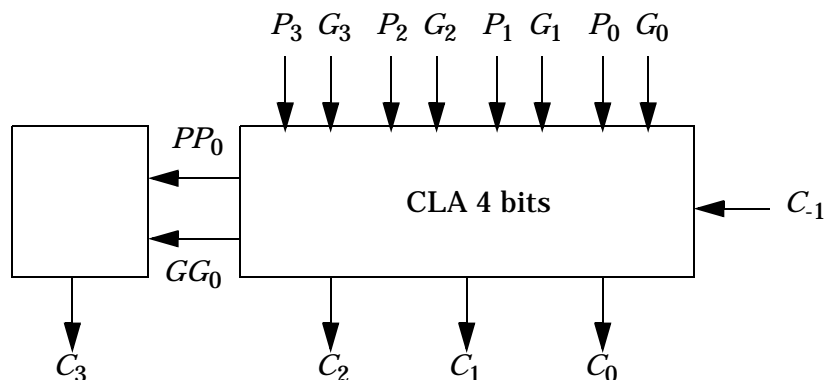
Un **additionneur à anticipation de retenue** (*carry look-ahead adder*, CLA) exploite ces propriétés pour accélérer l'opération d'addition. Un additionneur CLA est usuellement réalisé à l'aide de blocs de  $k = 4$  bits pour minimiser le fanin sur le calcul des retenues. On a en effet les relations suivantes:

$$\begin{aligned} C_0 &= G_0 + P_0 C_{-1} \\ C_1 &= G_1 + P_1 G_0 + P_1 P_0 C_{-1} \\ C_2 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1} \end{aligned}$$

$$\begin{aligned} PP_0 &= P_0 P_1 P_2 P_3 \\ GG_0 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 P_0 G_0 \end{aligned}$$

$$C_3 = GG_0 + PP_0 C_{-1}$$

$PP_0$  et  $GG_0$  représentent respectivement le signal de propagation et de génération **du bloc** de 4 bits. Un bloc CLA de 4 bits a ainsi la forme de la [Figure 71](#).



**Figure 71.** Bloc CLA de 4 bits.

La somme est alors obtenue par les relations suivantes:

$$S_0 = A_0 \oplus B_0 \oplus C_{-1} = P_0 \oplus C_{-1}$$

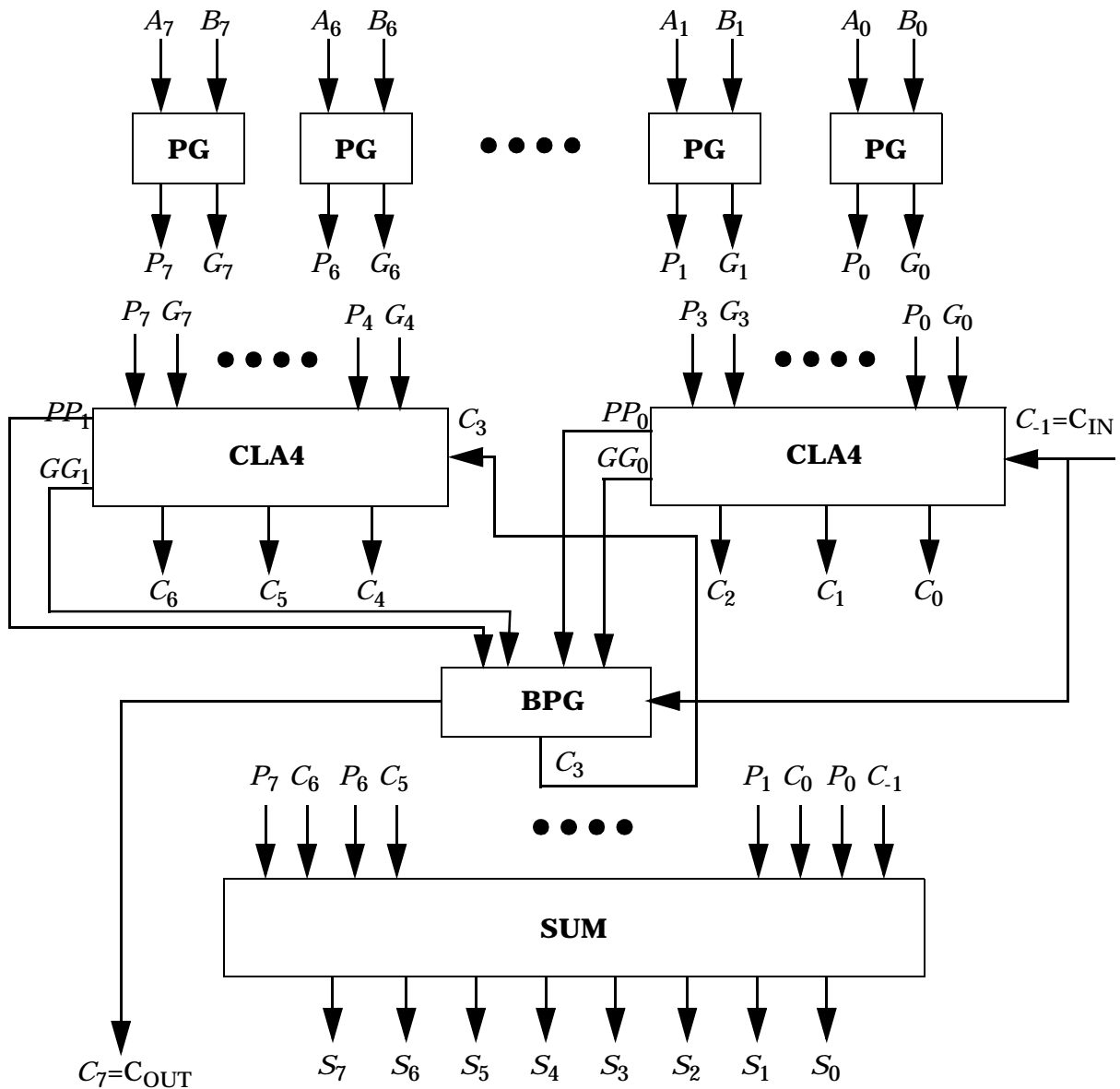
$$S_1 = A_1 \oplus B_1 \oplus C_0 = P_1 \oplus C_0$$

$$S_2 = A_2 \oplus B_2 \oplus C_2 = P_2 \oplus C_2$$

...

$$S_{n-1} = A_{n-1} \oplus B_{n-1} \oplus C_{n-2} = P_{n-1} \oplus C_{n-2}$$

On se propose de développer maintenant le modèle d'un additionneur CLA 8 bits. La structure d'un tel additionneur est donnée à la [Figure 72](#). Les blocs PG déterminent en parallèle s'il y aura propagation ou génération de la retenue. Deux blocs CLA4 calculent les retenues en parallèle. Le bloc BPG détermine s'il y a propagation ou génération de retenue d'un bloc CLA4 à l'autre et calcule la retenue de sortie. Finalement le bloc SUM calcule le vecteur somme.



**Figure 72.** Structure d'un additionneur CLA 8 bits.

Le modèle VHDL d'un tel additionneur est donné ci-dessous. Il s'agit tout d'abord de définir un paquetage contenant des procédures qui seront utilisées comme procédures concurrentes dans le modèle du composant ([Code 64](#)).

```

library IEEE;
use IEEE.std_logic_1164.all;

package CLA_pkg is
  -- Fonction d'un bloc PG
  procedure PG (signal A, B: in std_logic_vector(3 downto 0);
               signal P, G: out std_logic_vector(3 downto 0));
  -- Fonction d'un bloc CLA 4 bits
  procedure CLA4 (signal P, G: in std_logic_vector(3 downto 0);
                 signal CIN: in std_logic;
                 signal COUT: out std_logic_vector(2 downto 0);
                 signal PP, GG: out std_logic);
  -- Calcul de la propagation/génération sur le bloc CLA4 suivant
  procedure BPG (signal PP, GG, CIN: in std_logic;
                signal COUT: out std_logic);
end CLA_pkg;

package body CLA_pkg is

  -- Fonction d'un bloc PG
  procedure PG (signal A, B: in std_logic_vector(3 downto 0);
               signal P, G: out std_logic_vector(3 downto 0)) is
  begin
    P := A xor B;
    G := A and B;
  end PG;

  -- Fonction d'un bloc CLA 4 bits
  procedure CLA4 (signal P, G: in std_logic_vector(3 downto 0);
                 signal CIN: in std_logic;
                 signal COUT: out std_logic_vector(2 downto 0);
                 signal PP, GG: out std_logic) is
    variable pptmp, ggtmp, clast: std_logic;
  begin
    pptmp := P(0); ggtmp := G(0); clast := CIN;
    for i in 1 to COUT'length loop
      pptmp := pptmp and P(i);
      ggtmp := (ggtmp and P(i)) or G(i);
      clast := (clast and P(i-1)) or G(i-1);
      COUT(i-1) := clast;
    end loop;
    PP <= pptmp; GG <= ggtmp;
  end CLA4;

  -- Calcul de la propagation/génération sur le bloc CLA4 suivant
  procedure BPG (signal PP, GG, CIN: in std_logic;
                signal COUT: out std_logic) is
  begin
    COUT <= GG or (PP and CIN);
  end BPG;

end CLA_pkg;

```

**Code 64.** Paquetage pour l'additionneur CLA.

Le [Code 65](#) donne ensuite le modèle de l'additionneur CLA 8 bits. Chaque appel de procédure concurrente est équivalent à un processus sensible sur les signaux de mode **in** de la procédure qui a pour seule instruction séquentielle l'appel de cette procédure. Une procédure concurrente ne génère pas de structure en synthèse.

```

library IEEE;
use IEEE.std_logic_1164.all;
use WORK.CLA_pkg.all;

entity add8b is
  port (A, B: in std_logic_vector(7 downto 0);
         CIN: in std_logic;
         SUM: out std_logic_vector(7 downto 0);
         COUT: out std_logic);
end add8b;

architecture CLA of add8b is

  signal P, G, C, CC: std_logic_vector(7 downto 0);
  signal PP, GG:      std_logic_vector(1 downto 0);

begin

  -- calcul des retenues CC(2:0)
  PG(A => A(3 downto 0), B => B(3 downto 0),
     P => P(3 downto 0), G => G(3 downto 0));
  CLA4(P => P(3 downto 0), G => G(3 downto 0), CIN => CIN,
      COUT => C(2 downto 0), PP => PP(0), GG => GG(0));

  -- calcul des retenues CC(6:4)
  PG(A => A(7 downto 4), B => B(7 downto 4),
     P => P(7 downto 4), G => G(7 downto 4));
  CLA4(P => P(7 downto 4), G => G(7 downto 4), CIN => C(3),
      COUT => C(6 downto 4), PP => PP(1), GG => GG(1));

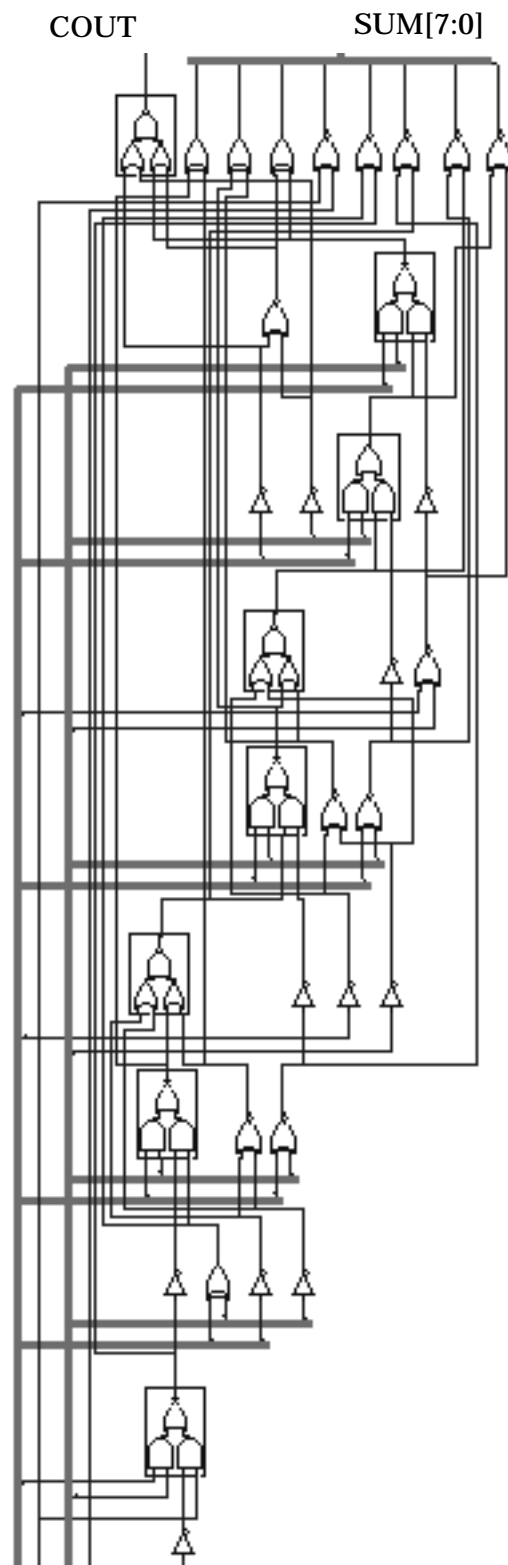
  -- calcul des retenues restantes C(3) et C(7)
  BPG(PP => PP(0), GG => GG(0), CIN => CIN, COUT => C(3));
  BPG(PP => PP(1), GG => GG(1), CIN => C(3), COUT => C(7));

  -- calcul de la somme et de la retenue de sortie
  CC <= C(6 downto 0) & CIN;
  SUM <= P xor CC;
  COUT <= C(7);

end CLA;

```

**Code 65.** Modèle de l'additionneur CLA 8 bits.



CIN  
A[7:0] B[7:0]

**Figure 73.** Additionneur CLA 8 bits synthétisé à partir du [Code 65](#).



## Multiplieurs

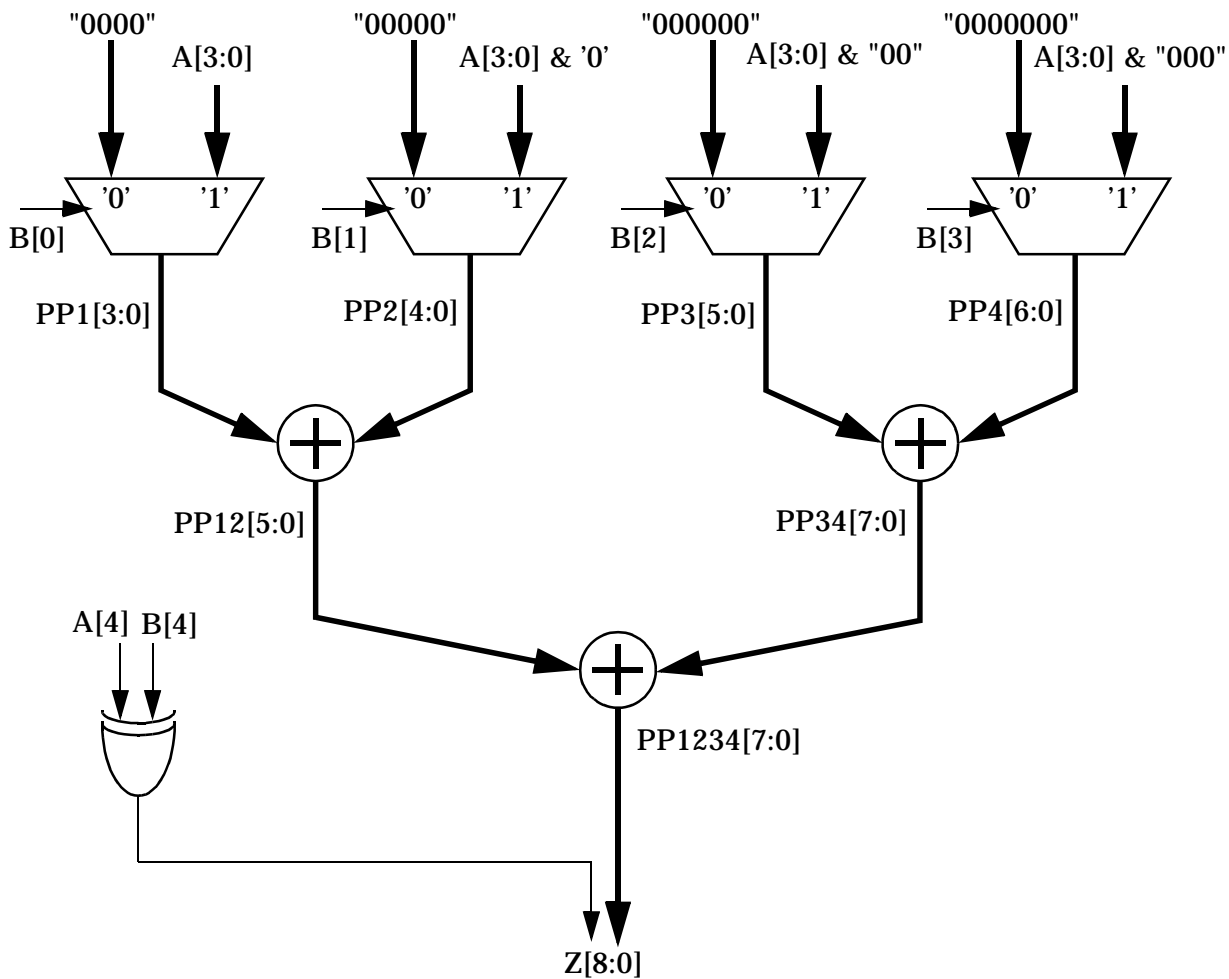
Les outils de synthèse logique acceptent en général l'opérateur '\*' et sont capable d'inférer un multiplieur. Comme pour les additionneurs, l'outil Synopsys possède une bibliothèque de modèles VHDL de multiplieurs paramétrés prédéfinis (DesignWare). Les multiplieurs inférés de cette manière sont combinatoires, ce qui donne des circuits performants pour des tailles d'opérandes petites (moins de 4 ou 5 bits). Pour des opérandes de plus de 4 ou 5 bits, il devient préférable d'opter pour une réalisation séquentielle. La dépendance surface/taille des opérandes est exponentielle pour une réalisation combinatoire, alors qu'elle est linéaire pour une réalisation séquentielle. Par contre, une réalisation combinatoire est plus rapide qu'une réalisation séquentielle qui nécessite plusieurs cycles d'horloge pour effectuer l'opération de multiplication. La variété de choix possibles requiert ainsi de développer le modèle du multiplieur à synthétiser plutôt que de se baser sur un mécanisme d'inférence automatique. D'autre part, les multiplieurs font hautement usage d'additionneurs et il est possible de bénéficier de la large gamme d'architectures d'additionneurs pour optimiser les performances des multiplieurs.

L'algorithme classique de multiplication utilise une suite d'additions et de décalages. Par exemple:

Chiffre décimal	Chiffre binaire	
23	1 0 1 1 1	
x 25	x 1 1 0 0 1	
----	-----	
	1 0 1 1 1	produit partiel 1
	0 0 0 0 0	produit partiel 2
	0 0 0 0 0	produit partiel 3
	1 0 1 1 1	produit partiel 4
	1 0 1 1 1	produit partiel 5
---	-----	
575	1 0 0 0 1 1 1 1 1 1	somme des produits partiels

Il est assez immédiat de réaliser cet algorithme sous la forme d'un circuit combinatoire. L'exemple ci-dessus porte sur des nombres non signés, mais il est possible d'appliquer le même algorithme pour des nombres signés en considérant le bit de poids fort comme le bit de signe et les autres bits comme l'opérande. Le signe du résultat est alors déterminé par un simple ou exclusif des bits de signe des opérandes et l'algorithme traite les opérandes diminués de leur bit de signe.

On se propose tout d'abord de modéliser un multiplieur combinatoire pour nombres signés de 5 bits A[4:0] et B[4:0] (A[4] et B[4] sont les bits de signe, A[3:0] et B[3:0] représentent les valeurs absolues des opérandes). La [Figure 74](#) illustre la structure d'un tel multiplieur. Les quatre produits partiels sont tout d'abord calculés: selon le bit correspondant de l'opérande B, le produit vaut zéro si ce bit vaut zéro ou une version décalée de l'opérande A si ce bit vaut 1. La somme des produits partiels est effectuée au moyen d'un arbre de trois additionneurs pour équilibrer et minimiser les délais des chemins. Le signe du résultat est calculé de manière indépendante par un ou exclusif des bits de signe des opérandes.



**Figure 74.** Structure d'un multiplieur combinatoire de nombres signés 5 bits.

Le [Code 66](#) donne le modèle VHDL du multiplieur combinatoire de nombres signés 5 bits. Bien que les opérandes A et B et le résultat Z soient signés, le type `unsigned` est utilisé. La raison en est que le type `signed` est réservé pour une représentation en complément à deux, alors que la représentation utilisée est du type signe-magnitude. Les produits partiels et les sommes partielles doivent opérer sur des opérandes de même taille. C'est pourquoi il s'agit de compléter certains opérandes par des zéros.

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE std_logic_arith.all;

entity mult5x5_signed is
  port (A, B: in  unsigned(4 downto 0);
        Z:      out unsigned(8 downto 0));
end mult5x5_signed;

architecture dfl of mult5x5_signed is

  signal Am, Bm: unsigned(3 downto 0); -- valeurs absolues de A et B
  signal PP1:   unsigned(3 downto 0); -- produits partiels
  signal PP2:   unsigned(4 downto 0);
  signal PP3:   unsigned(5 downto 0);
  signal PP4:   unsigned(6 downto 0);
  signal PP12:  unsigned(5 downto 0);
  signal PP34:  unsigned(7 downto 0);
  signal PP1234: unsigned(7 downto 0);

begin
  -- Calcul des produits partiels
  Am <= A(3 downto 0); Bm <= B(3 downto 0);
  PP1 <= Am          when Bm(0) = '1' else (others => '0');
  PP2 <= Am & '0'    when Bm(1) = '1' else (others => '0');
  PP3 <= Am & "00"   when Bm(2) = '1' else (others => '0');
  PP4 <= Am & "000"  when Bm(3) = '1' else (others => '0');

  -- Calcul des sommes partielles
  PP12 <= ("00" & PP1) + ('0' & PP2);
  PP34 <= ("00" & PP3) + ('0' & PP4);
  PP1234 <= ("00" & PP12) + PP34;

  Z(7 downto 0) <= PP1234;
  Z(8) <= A(4) xor B(4); -- bit de signe
end dfl;

```

**Code 66.** Modèle VHDL d'un multiplieur combinatoire de nombres signés 5 bits.

La réalisation d'un multiplieur basé sur l'algorithme additions-décalages mais au moyen d'un circuit séquentiel permet de réduire la surface du circuit. Il est ainsi possible de n'utiliser qu'un seul additionneur avec accumulation de la somme partielle et de décaler les produits partiels vers la droite (au lieu de décaler l'opérande A vers la gauche). Aussi, lorsqu'un bit de l'opérande B est égal à zéro, il n'est pas nécessaire d'ajouter un mot de zéros au produit partiel.

L'algorithme de multiplication séquentielle pour des nombres signés en représentation signe-magnitude est donné dans le pseudo-code suivant (l'architecture correspondante est donnée à la Figure 75):

```

-- soient A[m-1:0] et B[n-1:0] et Z[m+n-1:0] = A * B
RegA := A[m-2:0];           -- sans le bit de signe
RegB := B[n-2:0];           -- sans le bit de signe
Acc := 0;                   -- accumulateur
E := 0;                     -- retenue pour la prochaine addition
PS := A[m-1] xor B[n-1];    -- signe du produit
SC := n-1;                  -- compteur
loop
  exit when SC = 0;
  if RegB[0] = '1' then     -- addition
    E & Acc := Acc + RegA;
  end if;
  shr(E & Acc & RegB);      -- décalage à droite
  SC := SC - 1;
end loop;
Z := Acc & RegB;

```

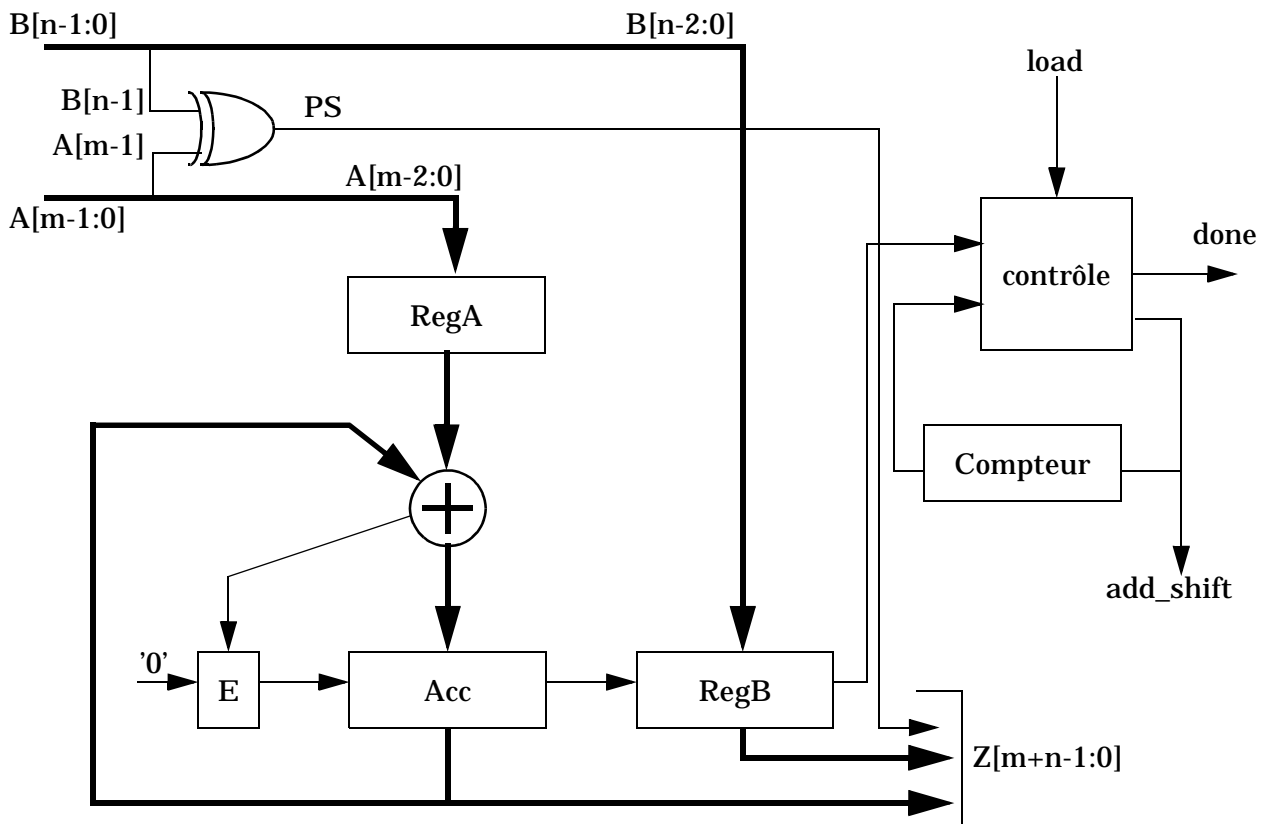


Figure 75. Structure du multiplieur séquentiel.

Le fonctionnement du multiplieur est le suivant. Lorsque le signal de contrôle `load` est actif, les opérandes A et B, moins leur bit de signe, sont chargés dans les registres `RegA` et `RegB`, respectivement. Le bit de poids faible de B donne la première valeur du signal de contrôle `add_shift`. Ce signal de contrôle détermine si la prochaine opération sera une addition (`add_shift = '1'`) ou un décalage (`add_shift = '0'`).

Lorsque `add_shift = '1'` la somme des registres `RegA` et `Acc` forme un produit partiel qui est transféré dans le registre (`E & Acc`). Lorsque `add_shift = '0'` le registre (`E & Acc & RegB`) est décalé d'un bit vers la droite et le compteur est décrémenté. Les opérations de chargement, d'addition et de décalage nécessitent chacune un cycle d'horloge.

Une opération de multiplication nécessite entre  $(n-1)$  et  $2*(n-1)$  cycles d'horloge avant de pouvoir produire un résultat correct. chaque bit '1' de B requièrera 2 cycles d'horloge et chaque bit '0' de B requièrera un cycle d'horloge. Lorsque le compteur atteint zéro, la multiplication est terminée et le signal `done` est asserté.

Par exemple, un multiplieur 9 x 6 peut être réalisé de deux manières différentes:

- Soit `A[8:0]` et `B[5:0]`: la multiplication nécessitera entre 5 et 10 cycles d'horloge et aura besoin d'un registre total (`E & Acc & RegB & RegA`) de 22 bits et d'un additionneur 8 bits.
- Soit `A[5:0]` et `B[8:0]`: la multiplication nécessitera entre 8 et 16 cycles d'horloge et aura besoin d'un registre total (`E & Acc & RegB & RegA`) de 19 bits et d'un additionneur 5 bits.

Le [Code 67](#) donne le modèle générique d'un multiplieur `MxN`.

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE std_logic_arith.all;

entity mult_mxn is
  generic (M, N: natural);
  port (CLK, RST, LOAD: in std_logic;
        A:    in unsigned(M-1 downto 0);
        B:    in unsigned(N-1 downto 0);
        Z:    out unsigned(M+N-1 downto 0);
        DONE: out std_logic);
end mult_mxn;

architecture seq of mult_mxn is

  const MAX_COUNT: natural := N - 1;

  signal PS:          std_logic;          -- signe du produit
  signal RegA:       unsigned(M-2 downto 0); -- registre A
  signal RegB:       unsigned(N-2 downto 0); -- registre B
  signal ACC:        unsigned(M-2 downto 0); -- accumulateur
  signal E:          std_logic;          -- retenue de l'acc.
  signal Add_Shift:  std_logic;          -- addition ou décalage
  signal SeqCount:   integer range 0 to MAX_COUNT; -- compteur

```

```

begin
  process
    variable E_ACC:      unsigned(M downto 0);
    variable E_ACC_RegB: unsigned(M+N-1 downto 0);
    variable Finished:   std_logic;
  begin
    wait until CLK = '1';
    if RST = '1' then -- reset synchrone
      PS <= '0';
      RegA := (others => '0');
      RegB := (others => '0');
      ACC  := (others => '0');
      E <= '0';
      Add_Shift <= '0';
      SeqCount <= MAX_COUNT;
      Finished <= '0';
    elsif LOAD = '1' then -- charge nouveaux opérandes
      PS <= A(M-1) xor B(N-1);
      RegA := A(M-2 downto 0);
      RegB := A(N-2 downto 0);
      ACC  := (others => '0');
      E <= '0';
      Add_Shift <= RegB(0);
      SeqCount <= MAX_COUNT;
      Finished := '0';
    elsif Add_Shift = '1' then -- addition
      E_ACC := ('0' & RegA) + ACC;
      E <= E_ACC(M-1);
      ACC <= E_ACC(M-2 downto 0);
      Add_Shift <= '0';
    elsif Finished = '0' then -- décalage
      E_ACC_RegB := E & ACC & RegB;
      E_ACC_RegB := '0' & E_ACC_RegB(M+N-1 downto 1);
      E <= E_ACC(M-1);
      ACC <= E_ACC(M-2 downto 0);
      RegB <= E_ACC_RegB(N-2 downto 0);
      if RegB(0) = '1' then
        Add_Shift <= '1'; -- addition au cycle suivant
      else
        Add_Shift <= '0'; -- décalage au cycle suivant
      end if;
      if SeqCount = 0 then
        Finished := '1';
      else
        SeqCount <= SeqCount - 1;
      end if;
    end if;
    Done <= Finished;
  end process;
  Z <= PS & ACC & RegB;
end seq;

```

**Code 67.** Modèle générique d'un multiplieur MxN séquentiel.

# Annexe A: Syntaxe VHDL

Cette annexe décrit les éléments de base et la syntaxe du langage VHDL dans sa version 1993 (dénomé VHDL-93). On indique aussi les restrictions imposées par la version 1987 du langage (dénomé VHDL-97).

Pour plus de détails, consulter le manuel officiel de référence du langage [LRM93] ou l'un des nombreux ouvrages disponibles sur le sujet [Berg92] [Berg93] [Bhas95a] [Bhas95b] [Cohe95] [Ashe96] [Pick96] [Airi98] [Nava98] [Perr98].

## A.1. Format de description de la syntaxe

- Les mots-clés réservés sont indiqués en gras (exemple: **begin**). Les mots-clés non supportés en VHDL-87 sont indiqués en italique gras (exemple: ***entity***).
- Les termes en italique doivent être remplacés par un identificateur ou du code VHDL légal.
- Les termes entre crochets ( [ ... ] ) sont optionnels.
- Les termes entre accolades ( { ... } ) peuvent être répétés zéro ou plusieurs fois.
- Une liste de termes séparés par des barres verticales ( | ) indique que l'un des termes doit être sélectionné.
- Toute autre ponctuation (par ex.: parenthèses, virgules, point-virgules) doit être utilisée telle quelle.
- Le format est libre: les espaces multiples, les tabulations et les fins de lignes sont ignorés.

## A.2. Éléments de base

### A.2.1. Identificateurs et mots réservés

Les identificateurs et les mots réservés en VHDL ont la même structure:

- Ils doivent débiter par un caractère alphabétique.
- Les caractères suivants peuvent être alphabétiques ou numériques.
- Le caractère souligné "\_" peut apparaître dans un identificateur, mais ni au début ni à la fin.
- Plusieurs caractères soulignés successifs sont interdits.
- Les identificateurs peuvent être de longueur quelconque et tous les caractères sont significatifs.
- VHDL ne fait pas de distinction entre caractères minuscules et majuscules.

Exemples d'identificateurs légaux et distincts:

```
X   F1234   VHDL1076   VHDL_1076   Un_long_identificateur
```

Exemples d'identificateurs légaux, mais non distincts:

```
Alert   ALERT   alert
```

Exemples d'identificateurs illégaux:

```
74LS00  -- débute avec un caractère non alphabétique
Q_      -- se termine avec un caractère souligné
A_Z     -- contient deux caractères soulignés successifs
```

Un certain nombre d'identificateurs sont *réservés* en VHDL: ils ne peuvent pas être utilisés ailleurs que comme éléments de la syntaxe du langage. La [Table A.1](#) donne les mots réservés en VHDL.

abs	else	linkage	procedure	then
access	elsif	<i>literal</i>	process	to
after	end	loop	<i>pure</i>	transport
alias	entity			type
all	exit	map	range	<i>unaffected</i>
and	file	mod	record	units
architecture	for	nand	register	until
array	function	new	<i>reject</i>	use
assert		next	rem	
attribute	generate	nor	report	variable
begin	generic	not	return	
block	<i>group</i>	null	<i>rol</i>	wait
body	guarded		<i>ror</i>	when
buffer		of	select	while
bus	if	on	severity	with
	<i>impure</i>	open	signal	
case	in	or	<i>shared</i>	<i>xnor</i>
component	<i>inertial</i>	others	<i>sla</i>	xor
configuration	inout	out	<i>sll</i>	
constant	is		<i>sra</i>	
		package	<i>srl</i>	
disconnect	label	port	subtype	
downto	library	<i>postponed</i>		

**Table A.1.** Mots réservés en VHDL (Les mots en italique ne sont pas supportés en VHDL-87).

<b>NOTE</b>	VHDL-93 permet la réutilisation de mots réservés comme identificateurs moyennant leur encadrement à l'aide de caractères "\". Exemple: \begin\.
-------------	--

### A.2.2. Littéraux caractère

Un littéral caractère (*character literal*) représente un caractère imprimable unique. Il est représenté entre deux apostrophes. Par exemple 'A' représente le caractère littéral A majuscule et est différent de 'a' qui représente le caractère littéral a minuscule. L'usage est de prononcer le littéral 'A' comme "tick-A-tick". Le caractère espace est représenté comme ' ' ("tick-espace-tick") et le caractère apostrophe est représenté par ''' (trois apostrophes successives). Les littéraux caractères sont des littéraux de type énuméré contenant des caractères. Ils incluent les types prédéfinis Bit et Character.



### A.2.3. Littéraux chaînes de caractères et chaînes de bits

Un littéral chaîne de caractères (*string literal*) est une séquence de caractères imprimables encadrée par des guillemets ("). Les espaces et les tabulations sont admis et les caractères majuscules et minuscules sont considérés comme distincts. Les guillemets devant apparaître à l'intérieur du littéral doivent être doublés ("").

#### NOTE

Le littéral chaîne de caractère avec un seul caractère (ex: "A") est différent du littéral caractère de ce même caractère (ex: 'A').

Les chaînes de caractères peuvent représenter n'importe quel tableau mono-dimensionnel dont les éléments sont des caractères. En particulier elles peuvent représenter des valeurs de types prédéfinis `String` et `Bit_Vector`.

Un littéral chaîne de bits (*bit string literal*) peut commencer par un spécificateur de base:

- Le caractère B (ou b) dénote un mot binaire et seuls les caractères '0', '1' et '\_' sont admis dans le littéral. C'est la base par défaut.
- Le caractère O (ou o) dénote un mot octal et seuls les caractères '0', '1', '2', '3', '4', '5', '6', '7' et '\_' sont admis dans le littéral.
- Le caractère X (ou x) dénote un mot hexadécimal et seuls les caractères '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c', 'd', 'e', 'f' et '\_' sont admis dans le littéral.

Le caractère souligné permet de grouper les bits sans changer la valeur du mot.

Par exemple tous les littéraux suivants représentent la même valeur:

```
"001110100000"  B"001110100000"  b"001110100000"  B"001_110_100_000"
O"1640"          x"3A0"          X"3a0"          X"3_A_0"
```

Les littéraux chaînes de bits représentent des valeurs de n'importe quel type tableau mono-dimensionnel dont les éléments sont du type prédéfini `Bit`; notamment le type prédéfini `Bit_Vector`.

### A.2.4. Littéraux numériques

Les littéraux numériques (*numeric literals*) représentent des valeurs entières ou réelles. Ils peuvent être représentés dans n'importe quelle base entre 2 et 16 incluses.

Les nombres en base 10 sont représentés comme une séquence de chiffres de 0 à 9 inclus. Le caractère souligné est admis pour séparer des groupes de chiffres. Un nombre réel doit posséder une partie entière, un point et une partie décimale (avec un zéro explicite si nécessaire). Un exposant optionnel de la forme E (ou e) suivi d'un nombre entier positif ou négatif peut être spécifié à la suite.

Exemples de nombres décimaux:

```
-- nombres entiers:      -- nombres réels:
    12                    0.0
    0                     -4.56
    1e6                   1.076E+3
    123_456              3.141_593
                        10e-02
```

Les littéraux numériques dans d'autres bases (*based literals*) commencent par un spécificateur de base: un entier entre 2 et 16 inclus, puis sont suivis par un nombre entre caractères "#".

Exemples de littéraux numériques basés:

```
2#110_1010#
16#CA#
16#f.ff#e+2  -- l'exposant est relatif à la base, ici 16**2
10#1076#
```

### A.2.5. Agrégats

Un agrégat (*aggregate*) représente la valeur d'un tableau ou d'un enregistrement sous la forme d'une association de valeurs aux éléments du tableau (indices) ou de l'enregistrement (noms de champs). Chaque association a une *partie formelle* (*formal part*) optionnelle, qui est l'indice de l'élément du tableau ou le nom du champ de l'enregistrement à associer ou le mot réservé **others**, suivie de la flèche d'association "=>". Vient ensuite la *partie actuelle* (*actual part*) qui représente la valeur à associer.

Une association sans partie formelle est dite *association par position* (*positional association*). Une association avec partie formelle est dite *association par nom* (*named association*). Les deux formes peuvent être mélangées dans un même agrégat, sous réserve que toutes les associations par position doivent précéder toutes les associations par nom. L'ordre des associations par nom peut être quelconque, à l'exception de celle utilisant le mot clé **others** qui doit impérativement être la dernière de l'agrégat. Elle permet d'associer tous les éléments non encore associés.

Exemples d'agrégats:

```
(1, 2, 3, 4, 5)
```

Un tableau de cinq éléments entiers ou un enregistrement de cinq champs de valeurs entières. Association par position.

```
(Jour => 21, Mois => Sep, An => 1998)
```

```
(Mois => Sep, Jour => 21, An => 1998)
```

Une valeur d'enregistrement à trois champs, dont deux d'entre eux, "Jour" et "An", sont de types entier et le troisième, "Mois", de type énuméré. Association par nom. Les deux formulations sont équivalentes.

```
('1', '0', '0', '1', others => '0')
```

Un vecteur de bits. Les quatre premiers bits sont "1001", tous les autres bits valent '0', quel que soit leur nombre. Association mixte.

```
(( 'X', '0', 'X'), ('0', '0', '0'), ('X', '0', '1'))
```

```
("X0X", "000", "X01")
```

Un tableau (à deux dimensions) de mots de trois bits. Les deux formulations sont équivalentes. Association par position.

### A.2.6. Commentaires

Tout texte d'une ligne situé au-delà de la chaîne "--" est considéré comme un commentaire et est ignoré par le compilateur. Exemples:

```
-- commentaire sur une ligne
```

```
-- commentaire
-- sur plusieurs
-- lignes
```

### A.2.7. Types et sous-types

Un type (*type*) est caractérisé par un ensemble de valeurs et un ensemble d'opérations. Il existe quatre classes de types:

- Les types *scalaires* (*scalar*): entier, réel, énuméré, physique.
- Les types *composites* (*composite*): tableau, enregistrement.
- Le type *accès* (*access*): accès (pointeur) à des objets d'un type donné.
- Le type *fichier* (*file*): fichier, séquence de valeurs d'un type donné.

Un sous-type (*subtype*) n'est pas un nouveau type, mais plutôt une restriction (contrainte) sur les valeurs d'un type de base. Ces restrictions peuvent être statiques (fixes) ou dynamiques (connues seulement lors de la simulation).

Une déclaration de type (sous-type) définit un type (sous-type):

```
type nom-type is définition-type ;
subtype nom-sous-type is [ fonction-résolution ]
                           nom-type [ contrainte ] ;
```

Le paquetage STANDARD déclare un certain nombre de types prédéfinis. Il est possible de déclarer de nouveaux types pour l'une quelconque des quatre classes ci-dessus.

### Types numériques

Le type entier Integer et le type flottant Real sont des types numériques prédéfinis dont la dynamique dépend de l'implémentation:

```
type Integer is range dépend de l'implémentation;
type Real is range dépend de l'implémentation;
```

La norme VHDL impose une dynamique minimum: [-2147483647,+2147483647] pour le type Integer et [-1.0E38,+1.0E38] pour le type Real.

Exemples de types non prédéfinis:

```
-- types entiers:
type Byte is range 0 to 255; -- intervalle ascendant
type Bit_Index is range 31 downto 0; -- intervalle descendant

-- types réels:
type Signal_Level is range -15.0 to +15.0; -- intervalle ascendant
type Probability is range 0.0 to 1.0; -- intervalle ascendant
```

Noter les intervalles *ascendants* (*ascending range*) et *descendants* (*descending range*).

Les sous-types Natural et Positive sont prédéfinis comme dérivés du type Integer avec une dynamique limitée:

```
subtype Natural is Integer range 0 to Integer'High;
subtype Positive is Integer range 1 to Integer'High;
```

Noter l'usage de l'*attribut* prédéfini 'High (prononcer "tic high") pour désigner le plus grand entier représentable dans l'implémentation utilisée.

### Types énumérés

Un type énuméré (*enumerated type*) déclare un ensemble ordonné d'identificateurs ou de caractères distincts. Différentes déclarations de types énumérés peuvent toutefois utiliser le même identificateur ou le même caractère. Dans ce cas, l'identificateur ou le caractère est dit *surchargé* (*overloaded*).

Le paquetage STANDARD déclare un certain nombre de types énumérés prédéfinis dont:

```
type Bit is ('0', '1');           -- ensemble de caractères
type Boolean is (False, True);  -- ensemble d'identificateurs
```

D'autres types énumérés peuvent être définis par l'utilisateur, par exemple:

```
type State is (Idle, Init, Check, Shift, Add);
type Mixed is (False, 'A', 'B', Idle);
    -- avec surcharge et mélange d'identificateurs et de caractères
```

### Types physiques

Un type physique (*physical type*) permet de représenter une quantité physique telle qu'une masse, une longueur, un temps, etc. Un type physique est caractérisé par son *unité de base* (*base unit*), l'intervalle de ses valeurs autorisées et une éventuelle collection de sous-unités ainsi que leur correspondance entre elles:

```
type Time is range dépend de la machine -- 32 bits garantis
units
  fs;           -- unité de base (femtoseconde)
  ps = 1000 fs; -- picoseconde
  ns = 1000 ps; -- nanoseconde
  us = 1000 ns; -- microseconde
  ms = 1000 us; -- milliseconde
  sec = 1000 ms; -- seconde
  min = 60 sec; -- minute
  hr = 60 min;  -- heure
end units;
```

#### NOTE

Le type physique prédéfini Time est un type entier essentiel pour la simulation. L'unité de base (1 femtoseconde) définit la *limite de résolution* (*resolution limit*) maximum d'une simulation. Il est toutefois possible d'utiliser pour la simulation une *unité secondaire* (*secondary unit*), multiple de l'unité de base, pour permettre d'atteindre un temps final plus grand (avec toutefois une perte de précision, puisque les temps sont toujours exprimés comme multiples de la limite de résolution). Le type Time est principalement utilisé pour spécifier des délais pour la simulation.

Pour écrire une valeur d'un type physique, il suffit d'écrire la valeur numérique, suivie d'un espace et de l'unité. Par exemple: 10 ns, 2 hr, 30 min.

<b>NOTE</b>	<p>VHDL-93 définit un nouveau sous-type <code>Delay_Length</code> basé sur le type <code>Time</code>, mais n'ayant que des valeurs positives ou nulles, ce qui évite la spécification d'un délai négatif:</p> <pre style="text-align: center;"> <b>subtype</b> Delay_Length <b>is</b> Time                                 <b>range</b> 0 <b>to</b> Time'High; </pre>
-------------	---

## Types tableaux

Un type tableau (*array type*) est un type composite déclarant une collection de valeurs de même type de base. Un type tableau peut être mono-dimensionnel (avec un indice) ou multi-dimensionnel (avec plusieurs indices). Un type tableau peut être *non contraint* (*unconstrained*, avec des limites d'indice(s) non connues à la déclaration) ou *contraint* (*constrained*, avec des limites d'indice(s) connues à la déclaration).

Le paquetage STANDARD déclare un certain nombre de types tableaux prédéfinis dont:

```

type Bit_Vector is array (Natural range <>) of Bit;
type String is array (Positive range <>) of Character;

```

Noter la spécification "<>", appelée *box*, pour dénoter un type tableau non contraint. D'autres types tableaux peuvent être définis par l'utilisateur, par exemple:

```

type Word is array (31 downto 0) of Bit; -- intervalle descendant
type Address is range 0 to 255;
type Memory is array (Address) of Word; -- intervalle ascendant
type Truth_Table is array (Bit, Bit) of Bit;

```

Noter les intervalles *ascendants* (*ascending range*) et *descendants* (*descending range*). Le type de base d'un indice peut être de n'importe quel type énuméré.

L'accès à un élément d'un objet de type tableau est possible par indexation du nom de l'objet. Soient par exemple un tableau mono-dimensionnel A et un tableau bi-dimensionnel B, les premiers éléments de ces tableaux sont référencés par A(1) et B(1,1) respectivement. Soit TT un objet de type Truth\_Table, alors TT('0', '1') référence l'élément de la première ligne et de la deuxième colonne de la matrice TT.

Une *tranche* (*slice*) d'éléments contigus peut être spécifiée grâce à la même notation d'intervalle que celle utilisée pour la déclaration du type:

```

-- Soit W un objet de type Word:
W(31 downto 16) -- spécifie les 16 premiers bits
W(15 downto 0) -- spécifie les 16 derniers bits

```

<b>NOTE</b>	<p>Le type d'intervalle (ascendant ou descendant) d'une tranche doit être le même que celui du type de base. Il faut déclarer un alias (§ A.2.10) si ce n'est pas le cas</p>
-------------	--

L'utilisation d'agrégats offre encore une autre manière d'accéder aux éléments d'un tableau:

```
type CharStr is array (1 to 4) of Character;
-- Accès positionnel: ('T', 'O', 'T', 'O')
-- Accès par nom: (1 => 'T', 2 => 'O', 3 => 'T', 4 => 'O')
-- Accès par défaut: (1 | 3 => 'T', others => 'O')
```

Noter le mot-clé **others** pour accéder aux éléments d'indices non explicitement mentionnés.

### **Types enregistrements**

Un type enregistrement (*record type*) est un type composite déclarant une collection de valeurs dont les types de base peuvent être différents. Il n'existe pas de type enregistrement prédéfini. L'exemple suivant donne un exemple de type enregistrement qui pourrait être utile pour l'écriture d'un modèle de micro-processeur:

```
type Instruction is record
  Op_Code: Processor_Operation; -- p. ex. (OP_LOAD, OP_ADD, ...)
  Address_Mode: Mode;           -- p. ex. (NONE, IND, DIR, ...)
  Operand1, Operand2: Integer range 0 to 15;
end record Instruction;
```

L'accès à un *champ* (*field*) d'un enregistrement est effectué par une notation sélective. Si *Inst* et le nom de l'objet de type *Instruction*, la notation *Inst.Op\_Code* fait par exemple référence au premier champ de l'enregistrement. Comme pour les tableaux, il est aussi possible d'utiliser des agrégats, en notation positionnelle ou par noms. Dans ce dernier cas, les identificateurs des champs remplacent les indices:

```
-- association par nom:
(Op_Code => OP_ADD, Address_Mode => NONE, Operand1 => 2,
 Operand2 => 15)
-- association par position:
(OP_LOAD, IND, 7, 8)
```

### **Types accès**

Un type accès (*access type*) permet de déclarer un pointeur. Le paquetage *TEXTIO* déclare un type accès prédéfini:

```
type Line is access String;
```

Les types accès sont surtout utiles pour déclarer des structures de données dynamiques telles que des listes:

```
type Cell; -- déclaration incomplète de type
type Link is access Cell;
type Cell is record
  Value: Integer;
  Succ: Link;
end record Cell;
```

Noter la déclaration incomplète de type nécessaire dans ce cas.

L'opérateur prédéfini **new** permet d'allouer une zone mémoire suffisante pour stocker un objet du type référencé:

```

new Link    -- alloue un nouveau pointeur sur un objet de type Cell
              -- valeur initiale (Integer'Left,null)
new Link'(15,null) -- initialisation explicite
                  -- l'expression est qualifiée (Link')
new Link'(1, new Link'(2,null)) -- allocation chaînée

```

L'utilisation d'agrégats permet en plus de spécifier une valeur initiale de l'objet pointé. Le littéral **null** est utilisé pour spécifier un objet de type accès ne pointant sur aucun objet. La procédure prédéfinie `Deallocate` libère la place mémoire pointée par un objet de type accès et remet sa valeur à **null**:

```

procedure Deallocate (ptr: inout type-accès);

```

Soit `CellPtr` un objet de type `Link`. La notation `CellPtr.all` représente le nom de l'objet référencé par `CellPtr`. Si l'objet référencé est de type enregistrement, la notation sélective permet d'accéder à un champ particulier: par exemple `CellPtr.Value`, `CellPtr.Succ`.

## Types fichiers

Un type fichier (*file type*) déclare une séquence de valeurs contenues dans un fichier du système d'exploitation utilisé. Le paquetage `TEXTIO` déclare un type fichier prédéfini `TEXT`:

```

type Text is file of String;

```

D'autres types fichier peuvent être définis par l'utilisateur:

```

-- Fichier de chaînes de caractères de longueurs quelconques
type Str_File is file of String;
-- Fichier de valeurs entières non négatives
type Nat_File is file of Natural;

```

## A.2.8. Objets

VHDL reconnaît quatre classes d'objets: les constantes, les variables, les fichiers et les signaux. Tout objet peut prendre des valeurs d'un certain type, prédéfini ou non. Les trois premières classes se retrouvent dans les langages de programmation classiques, tandis que la dernière classe est propre à VHDL.

## Constantes

Une constante (*constant*) a par définition une valeur fixe définie une fois pour toute:

```

constant PI: Real := 3.1416;
constant INDEX_MAX: Integer := 10*N; -- N doit être déclaré avant
constant Delay: Delay_Length := 5 ns;
-- initialisation avec agrégat:
constant Null_BitVector: Bit_Vector(15 downto 0) := (others => '0');
constant TT: Truth_Table: (others => (others => '0'));
constant Inst: Instruction :=
    (Op_Code => OP_ADD, Address_Mode => NONE,
     Operand1 => 2, Operand2 => 15);
-- constante à valeur différée
constant MAX_SIZE: Natural;

```

Une constante dont la valeur n'est pas spécifiée est une dite *à valeur différée* (*deferred constant*). Une telle déclaration ne peut apparaître que dans une déclaration de paquetage et sa valeur doit être définie dans le corps de paquetage correspondant.

## Variables

Une variable (*variable*) est un objet dont la valeur est modifiable par affectation. Une déclaration de variable définit son nom, son type et éventuellement sa valeur initiale:

```
variable count: Natural;
  -- valeur initiale: count = 0 (= Natural'Left)
variable isHigh: Boolean;
  -- valeur initiale: isHigh = False (= Boolean'Left)
variable currentState: State := Idle; -- initialisation explicite
variable Memory: Bit_Matrix(0 to 7, 0 to 1023);
```

### NOTE

L'utilisation des variables était initialement (c.à.d. pour VHDL-87) limitée au domaine des instructions séquentielles (processus, sous-programmes). VHDL-93 autorise la déclaration et l'utilisation de *variables globales* ou *partagées* (*shared variables*) au niveau du corps d'architecture. Cette possibilité permet l'écriture de modèles à un niveau abstrait qui ne fait plus guère référence au matériel (par exemple: un gestionnaire de pile LIFO). Elle permet aussi d'éviter d'utiliser des signaux et tout ce qu'ils imposent (pilote, synchronisation). Elle introduit cependant un non déterminisme qui doit être géré par l'utilisateur.

## Fichiers

Un fichier (*file*) est un objet de type fichier faisant référence à un stockage sur une mémoire de masse (disque) externe et dépendante du système d'exploitation. Une déclaration de fichier définit son nom logique, son type et éventuellement son mode d'utilisation et son nom externe:

```
type Integer_File is file of Integer;
file File1: IntegerFile;
  -- fichier local en mode lecture (Read_Mode, défaut)
  -- ouverture explicite requise (avec la procédure File_Open)

file File2: IntegerFile is "test.dat";
  -- mode lecture (défaut) et lien sur un nom externe
  -- appel implicite de la procédure
  --     File_Open(F => File2, External_Name => "test.dat",
  --             Open_Kind => Read_Mode)

file File3: IntegerFile open Write_Mode is "test.dat";
  -- mode écriture avec lien sur nom externe
  -- appel implicite de la procédure
  --     File_Open(F => File3, External_Name => "test.dat",
  --             Open_Kind => Write_Mode)
```



**NOTE**

La syntaxe ci-dessus n'est valable que pour VHDL-93. La syntaxe VHDL-87 pour l'usage de fichiers n'est pas un sous-ensemble de la syntaxe VHDL-93 et n'est donc pas acceptée par un analyseur VHDL-93.

La déclaration de fichier en VHDL-87 est:

```
file nom-fichier : (sous-)type is
                    [ in | out ] nom_externe ;
```

où **in** ouvre le fichier en lecture (mode par défaut), **out** ouvre le fichier en écriture et *nom-externe* est une chaîne de caractères

Soit *type-élément* le type des éléments contenus dans un fichier de type *type-fichier*.

Deux sous-programmes prédéfinis `read` et `endfile` permettent la lecture de fichiers:

```
procedure read (file f: type-fichier; value: out type-élément);
function endfile (file f: type-fichier) return Boolean;
-- retourne TRUE lorsque la fin du fichier est atteinte, FALSE sinon
```

La procédure prédéfinie `write` permet d'écrire dans un fichier:

```
procedure write (file f: type-fichier; value: in type-élément);
```

Un fichier peut être ouvert explicitement par la procédure prédéfinie `file_open`:

```
procedure file_open (file f: type-fichier; external_name: in String;
                    open_kind: in File_Open_Kind := Read_Mode);
procedure file_open (status: out File_Open_Status;
                    file f: type-fichier;
                    external_name: in String;
                    open_kind: in File_Open_Kind := Read_Mode);
```

La seconde forme retourne l'état de l'opération: `Open_OK`, `Status_Error`, `Name_Error`, ou `Mode_Error`.

Un fichier peut être explicitement fermé au moyen de la procédure prédéfinie `file_close`:

```
procedure file_close (file f: type-fichier);
```

**NOTE**

Les procédures `file_open` et `file_close` ne sont pas disponibles en VHDL-87. L'ouverture d'un fichier est faite implicitement dans sa déclaration et sa fermeture est implicitement faite lorsque la simulation quitte la zone de visibilité de la déclaration de fichier.

Un fichier ne peut pas être l'objet d'une assignation. Il peut seulement être passé comme paramètre d'un sous-programme.

**Signaux**

Un signal (*signal*) est un objet représentant une forme d'onde temporelle logique (une suite discrète de paires temps/valeurs). Les valeurs prises par le signal dépendent du type associé au signal.

La déclaration de signal de base définit le nom du signal, son (sous-)type et éventuellement sa valeur initiale:

```
signal nom: (sous-)type [ := valeur-initiale ];
```

Une déclaration de signal ne peut apparaître que dans le domaine des instructions concurrentes. Un signal ne peut pas être d'un type fichier ou d'un type accès.

Exemples de déclarations de signaux:

```
signal S: Bit_Vector(15 downto 0);
  -- valeur initiale par défaut: (others => '0')
signal CLK: Bit := '0'; -- valeur initiale explicite
```

### A.2.9. Attributs

Un attribut (*attribute*) permet d'accéder aux propriétés associées aux types et aux objets VHDL. Un attribut sur un objet Obj est référencé par la notation Obj'Attribut. Il existe un certain nombre d'attributs prédéfinis, dont les plus utilisés sont donnés ici: attributs associés aux types scalaires (Table A.2), aux types discrets ou physiques (Table A.3), aux tableaux (Table A.4) et aux signaux (Table A.5). L'utilisation d'attributs permet d'écrire des modèles portables.

T'Left	limite à gauche de T
T'Right	limite à droite de T
T'Low	limite inférieure de T
T'High	limite supérieure de T

**Table A.2.** Quelques attributs de type; T est un type scalaire.

Exemples:

```
type Address is Integer range 7 downto 0;
-- Address'Low = 0, Address'High = 7
-- Adress'Left = 7, Address'Right = 0
```

T'Pos(X)	position de X dans la déclaration de T
T'Val(N)	valeur de l'élément de T à la position N
T'Succ(X)	prochaine valeur après X (dans l'ordre)
T'Pred(X)	valeur précédant X (dans l'ordre)

**Table A.3.** Quelques attributs de type; T est un type discret ou physique, X est un objet de type T et N est un entier.

Exemples:

```
type Level is ('U', '0', '1', 'Z');
-- Level'Pos('U') = 0, Level'Val(2) = '1'
-- Level'Succ('1') = 'Z', Level'Pred('0') = 'U'
```

A'Left	limite à gauche de l'intervalle du tableau
A'Right	limite à droite de l'intervalle du tableau

**Table A.4.** Quelques attributs de tableau; A est un objet de type tableau.

A'Low	limite inférieure de l'intervalle du tableau
A'High	limite supérieure de l'intervalle du tableau
A'Range	intervalle du tableau
A'Reverse_Range	intervalle inverse du tableau
A'Length	longueur du tableau (nombre d'éléments)

**Table A.4.** Quelques attributs de tableau; A est un objet de type tableau.

Exemples:

```

type Word is array (31 downto 0) of Bit;
type Memory is array (7 downto 0) of Word;
variable Mem: Memory;
-- Mem'Low = 0, Mem'High = 7, Mem'Left = 7, Mem'Right = 0
-- Mem'Range = 7 downto 0, Mem'Reverse_Range = 0 to 7
-- Mem'Length = 8, Mem(0)'Length = 32
-- Mem(1)'Range = 31 downto 0, Mem(2)'High = 31

```

S'Stable(T)	signal booléen = True si aucun événement ne s'est produit depuis T unités de temps
S'Delayed(T)	signal équivalent à S, mais retardé de T unités de temps
S'Event	signal booléen = True si un événement s'est produit sur S, False sinon
S'Transaction	signal booléen = True si S a subi une transaction, False sinon
S'Last_Event	temps écoulé depuis le dernier événement

**Table A.5.** Quelques attributs de signal.  
S est un signal et T une expression de type Time.

### A.2.10. Alias

Une *déclaration d'alias* (*alias declaration*) permet de définir des noms supplémentaires aux objets déclarés ou à des parties de ceux-ci. Le but principal d'un alias est d'améliorer la lisibilité de la description. Par exemple:

```

variable Real_Number: Bit_Vector(0 to 31);
alias Sign: Bit is Real_Number(0);
alias Mantissa: Bit_Vector(23 downto 0) is Real_Number(8 to 31);
-- Mantissa est une variable de 24 bits
-- Mantissa(23 downto 18) est équivalent à Real_Number(8 to 13)

```

Un autre usage intéressant des alias permet d'écrire du code générique portable. Par exemple, les opérations arithmétiques ne sont pas définies pour le type prédéfini Bit\_Vector. Une possibilité serait de *surcharger* les opérations arithmétiques prédéfinies pour qu'elles acceptent de travailler sur des opérandes de type Bit\_Vector. Le code suivant définit une fonction pouvant opérer sur des arguments tableaux de tailles quelconques:

```

function "*" (A, B: Bit_Vector) return Bit_Vector is
  alias AA: Bit_Vector(A'Length-1 downto 0) is A;
  alias AB: Bit_Vector(B'Length-1 downto 0) is B;
  autres déclarations...
begin
  corps de la fonction avec usage de AA et AB
end "*";

```

Il s'agit en particulier de faire attention la façon dont les indices des tableaux sont définis (ascendants ou descendants). Les deux déclarations d'alias permettent d'écrire le corps de la fonction en supposant que les deux arguments ont des indices descendants. Le fonctionnement sera correct même si les arguments effectifs ne satisfont pas cette contrainte.

### A.2.11. Expressions et opérateurs

Une expression définit une relation entre un certain nombre de termes par l'intermédiaire d'opérateurs. La [Table A.6](#) donne la liste des opérateurs prédéfinis en VHDL.

**	<b>abs</b>	<b>not</b>			
*	/	<b>mod</b>	<b>rem</b>		
+ (signe)	- (signe)				
+	-	&			
<i>sll</i>	<i>srl</i>	<i>sla</i>	<i>sra</i>	<i>rol</i>	<i>ror</i>
=	/=	<	<=	>	>=
<b>and</b>	<b>or</b>	<b>nand</b>	<b>nor</b>	<b>xor</b>	<b>xnor</b>

**Table A.6.** Opérateurs prédéfinis dans l'ordre décroissant de leur niveau de précédance.  
(Les mots en italique ne sont pas supportés en VHDL-87)

Les opérateurs logiques **and**, **or**, **nand**, **nor**, **xor** et **xnor** peuvent s'appliquer à des valeurs et à des tableaux mono-dimensionnels de types Bit ou Boolean. Dans le cas de tableaux, les opérations logiques sont effectuées élément par élément. Pour des valeurs de types Bit ou Boolean, les opérateurs **and**, **or**, **nand** et **nor** utilisent un *court-circuit* (*short-circuit operations*): l'opérande de droite n'est évalué que si celui de gauche ne permet pas de déterminer le résultat. Les opérateurs **and** et **nand** n'évaluent l'opérande de droite que si celui de gauche vaut '1' ou True. Les opérateurs **or** et **nor** n'évaluent l'opérande de droite que si celui de gauche vaut '0' ou False.

Les opérateurs relationnels "=", "/=", "<", "<=", ">" et ">=" doivent avoir deux opérandes de mêmes types et ils retournent une valeur de type Boolean. Les opérateurs "=" et "/=" peuvent avoir des opérandes de n'im-

porte quel type, sauf du type fichier. Les autres opérateurs doivent avoir des opérandes de type scalaire ou de type tableau mono-dimensionnel à base de type discret (type entier ou énuméré).

<b>NOTE</b>	<p>Les opérateurs binaires de décalage <b>sll</b>, <b>srl</b>, <b>sla</b> et <b>sra</b> et de rotation <b>rol</b> et <b>ror</b> sont introduits en VHDL-93. Ils s'appliquent à des tableaux mono-dimensionnels de types <code>Bit</code> ou <code>Boolean</code> (opérande de gauche). L'opérande de droite doit être de type entier:</p> <ul style="list-style-type: none"> <li>• s'il est nul, aucune opération n'est effectuée (exemple: <code>A sll 0</code> laisse le vecteur A tel quel);</li> <li>• s'il est positif, l'opération est répétée un certain nombre de fois (exemple: <code>A rol 5</code> effectue une rotation de 5 bits vers la gauche du vecteur A);</li> <li>• s'il est négatif, l'opération opposée est effectuée un certain nombre de fois (exemple: <code>A sll -4</code> est équivalent à <code>A slr 4</code>).</li> </ul> <p>Les opérations de décalage impliquent une perte de la valeur d'un élément du vecteur. La valeur injectée est '0' si le vecteur est du type <code>Bit_Vector</code> et <code>False</code> si le type de base du vecteur est <code>Boolean</code>.</p>
-------------	---

Les opérateurs de signe (+ et -), ainsi que les opérateurs d'addition (+) et de soustraction (-) conservent leur sémantique usuelle.

L'opérateur de concaténation (&) n'est applicable qu'aux tableaux mono-dimensionnels. Le résultat de la concaténation est un nouveau tableau mono-dimensionnel constitué du contenu de l'opérande de gauche suivi du contenu de l'opérande de droite:

```

type Byte is Bit_Vector(7 downto 0);
constant ZERO: Byte := "0000" & "0000";
constant C: Bit_Vector := ZERO & ZERO; -- C'Length = 8
variable A: Byte;
A := '0' & A(A'Left downto 1); -- équivalent à l'opérateur srl

```

<b>NOTE</b>	<p>Un problème avec VHDL-87 est que l'intervalle du vecteur concaténé peut provoquer une erreur de contrainte. En effet, la limite de gauche du vecteur concaténé est celle de l'opérande de gauche et l'intervalle du vecteur résultat est calculé à partir de sa déclaration. Exemple:</p> <pre> <b>variable</b> A, B: Bit_Vector(15 <b>downto</b> 0); B := A(7 <b>downto</b> 0) &amp; A(15 <b>downto</b> 8); -- Erreur de contrainte: -- B'Range = 7 <b>downto</b> -8! </pre> <p>VHDL-93 corrige le problème: pour un intervalle ascendant (descendant), la limite de gauche (droite) du vecteur concaténé est celle de l'opérande de gauche (droite).</p>
-------------	---

Les opérateurs de multiplication ("\*") et de division ("/") acceptent des opérandes de type entier, réel et physique. Les opérateurs modulo (**mod**) et de reste (**rem**) n'acceptent que des arguments de type entier. L'opérateur de valeur absolue (**abs**) accepte n'importe quel type numérique. L'opérateur de mise à la puissance ("\*\*") accepte un opérande de gauche de type entier ou réel, mais seulement un opérande de droite de type entier. Si ce dernier est négatif, l'opérande de gauche doit être de type réel.

## A.3. Unités de conception

L'unité de conception (*design unit*) est le plus petit module compilable séparément. VHDL offre cinq types d'unités de conception: la déclaration d'entité, le corps d'architecture, la déclaration de configuration, la déclaration de paquetage et le corps de paquetage.

### A.3.1. Clause de contexte

Une clause de contexte (*context clause*) permet de définir une ou plusieurs bibliothèques par une ou plusieurs déclarations de bibliothèques et un ou plusieurs chemins d'accès aux noms disponibles dans la ou les bibliothèques déclarées. La clause de contexte précède une unité de conception.

Une *déclaration de bibliothèque* (*library clause*) définit le(s) nom(s) logique(s) de la (des) bibliothèque(s):

```
library nom-bibliothèque { , nom-bibliothèque };
```

```
-- Exemple:
```

```
library IEEE, CMOS_LIB;
```

L'association d'un nom logique de bibliothèque avec son emplacement physique, un répertoire Unix p. ex., est dépendant de l'outil VHDL utilisé. Deux bibliothèques logiques sont prédéfinies en VHDL: la bibliothèque WORK est la seule bibliothèque dans laquelle il est possible de placer des unités de conception compilées. Toutes les autres bibliothèques sont en mode lecture. La bibliothèque STD ne contient que deux unités de conception: le paquetage STANDARD et le paquetage TEXTIO.

L'accès aux noms d'une unité de conception doit normalement spécifier le chemin sous la forme:

```
nom-bibliothèque.nom-unité-conception.nom-simple
```

où *nom-simple* peut être un nom de type, d'objet, de sous-programme, etc. Une clause **use** (*use clause*) permet d'éviter de devoir spécifier le chemin complet pour utiliser un nom simple. Une clause **use** peut avoir plusieurs formes, chacune d'elles offrant un niveau de visibilité différent:

```
use nom-bibliothèque.all;    -- tous les noms de toutes les unités
```

```
use nom-bibliothèque.nom-unité-conception;    -- l'unité seule
```

```
use nom-bibliothèque.nom-unité-conception.all;
```

```
-- tous les noms de l'unité
```

```
use nom-de-bibliothèque.nom-unité-conception.nom-simple;
```

```
-- le nom simple seul
```

```
-- Exemples:
```

```
use IEEE.all;
```

```
use IEEE.std_logic_1164;
```

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

```
use IEEE.std_logic_1164.std_logic;
```

La première forme est potentiellement dangereuse car il peut y avoir conflit de noms entre plusieurs unités de la bibliothèque. La deuxième forme ne rend visible que le nom de l'unité de conception. La troisième forme est la plus courante. La quatrième forme est du même type que la deuxième, mais appliquée à un nom de type, d'objet, de sous-programme, etc. particulier.

Toute unité de conception est implicitement précédée des déclarations suivantes:

```
library STD, WORK;
```

```
use STD.STANDARD.all;
```

### A.3.2. Déclaration d'entité

Une déclaration d'entité (*entity declaration*) définit l'interface d'un modèle avec son environnement. Elle déclare aussi des caractéristiques et des instructions communes à toute architecture relative.

Syntaxe:

```
[ clause-de-contexte ]
entity nom-entité is
  [ generic ( constante-interface { , constante-interface } ); ]
  [ port ( port-interface { , port-interface } ); ]
  { déclaration-locale }
  [ begin
    { instruction-concurrente }
  end [ entity ] [ nom-entité ] ;
```

Les déclarations locales suivantes sont admises: type, sous-type, constante, signal, variable partagée, fichier, alias, attribut, sous-programme (en-tête et corps), clause **use**.

Les seules instructions qui peuvent apparaître dans une déclaration d'entité sont l'instruction concurrente d'assertion, l'appel concurrent de procédure et le processus. Ces deux dernières instructions doivent être *passives*, c.à.d. qu'elles ne peuvent qu'accéder aux objets (signaux, variables globales) sans modifier leurs valeurs.

Exemple:

```
entity ENT is
  generic (N: positive := 1);
  port (S1, S2: in bit; S3: out bit_vector(0 to N-1));
begin
  assert S1 /= S2 report "Erreur: S1 = S2" severity ERROR;
end entity ENT;
```

### A.3.3. Corps d'architecture

Un corps d'architecture (*architecture body*) définit une réalisation d'une déclaration d'entité. Une déclaration d'entité peut avoir zéro, une ou plusieurs architectures associées.

Syntaxe:

```
[ clause-de-contexte ]
architecture nom-architecture of nom-entité is
  { déclaration-locale }
begin
  { instruction-concurrente }
end [ architecture ] [ nom-architecture ] ;
```

Les déclarations locales suivantes sont admises: type, sous-type, constante, signal, variable partagée, fichier, alias, attribut, sous-programme (en-tête et corps), composant, clause **use**.

Toutes les instructions concurrentes sont admises dans le corps d'architecture.

Exemple:

```

architecture ARCH of ENT is
  constant delay: time := 10 ns;
  signal S: bit;
begin
  S <= S1 and S2;
  S3 <= (0 to N-2 => '0') & S after delay;
end architecture ARCH;

```

### A.3.4. Déclaration de paquetage

Une déclaration de paquetage (*package declaration*) groupe des déclarations qui peuvent être utilisées par d'autres unités de conception.

Syntaxe:

```

[ clause-de-contexte ]
package nom-paquetage is
  { déclaration-locale }
end [ package ] [ nom-paquetage ] ;

```

Les déclarations locales suivantes sont admises: type, sous-type, constante, signal, variable partagée, fichier, alias, attribut, sous-programme (en-tête seulement), composant, clause **use**.

Exemple:

```

package PKG is
  constant MAX: integer := 10;
  subtype BV10 is bit_vector(MAX-1 downto 0);
  procedure PROC (A: in BV10; B: out BV10);
  function FUNC (A, B: in BV10) return BV10;
end package PKG;

```

### A.3.5. Corps de paquetage

Un corps de paquetage (*package body*) contient les définitions des déclarations incomplètes de la déclaration de paquetage correspondante.

Syntaxe:

```

[ clause-de-contexte ]
package body nom-paquetage is
  { déclaration-locale }
end [ package body ] [ nom-paquetage ] ;

```

Les déclarations locales suivantes sont admises: type, sous-type, constante, variable partagée, fichier, alias, sous-programme (en-tête et corps), clause **use**.



Exemple:

```

package body PKG is
  procedure PROC (A: in BV10; B: out BV10) is
  begin
    B := abs(A);
  end procedure PROC;
  function FUNC (A, B: in BV10) return BV10 is
    variable V: BV10;
  begin
    V := A and B;
    return (not(V));
  end function FUNC;
end package body PKG;

```

### A.3.6. Déclaration de configuration

Une déclaration de configuration (*configuration declaration*) définit les associations (*binding*) entre les instances de composants d'un modèle et les entités de conception (paires entité/architecture) actuelles. La déclaration de configuration peut prendre une forme assez complexe. On se limite ici à la version simplifiée la plus courante.

Syntaxe:

```

[ clause-de-contexte ]
configuration nom-configuration of nom-entité is
  for nom-architecture
    { for spécification-composant
      indication-association ;
    }
  end for; }
end for;
end [ configuration ] [ nom-configuration ] ;

```

La spécification de composant identifie l'instance de composant à traiter. Syntaxe:

```

nom-instance { , nom-instance } | others | all : nom-composant

```

Le mot-clé **others** dénote toutes les instances de composants non encore traitées. Le mot-clé **all** dénote toutes les instances d'un même composant.

L'indication d'association définit les correspondances entre des instances de composants et les entités de conception actuelles. Elle définit aussi les correspondances entre paramètres génériques formels et actuels et entre ports formels et actuels. Syntaxe:

```

use entity nom-entité ( nom-architecture )
  [ generic map ( liste-association-générique ) ]
  [ port map ( liste-association-port ) ]

```

Exemple:

```

configuration CONF of ENT2 is
  for ARCH2
    for C: COMP use entity WORK.DFF(A3)
      port map (CLK, D, Q, QB);
    end for;
    for all: COMP2 use entity WORK.DFF(A2)
      port map (CLK, D, Q, QB);
    end for;
  end for;
end configuration CONF;

```

### A.3.7. Déclarations d'interfaces

Une *déclaration d'interface* (*interface declaration*) est utilisée dans une déclaration d'entité, une déclaration de composant et une déclaration de sous-programme.

Une *constante d'interface* (*interface constant*) peut être un paramètre générique ou un argument constant d'un sous-programme. Syntaxe:

```

[ constant ] nom-constante { , nom-constante } : [ in ] (sous-)type
[ := expression ]

```

Un *signal d'interface* (*interface signal*) peut être un port ou un argument d'un sous-programme. Syntaxe:

```

[ signal ] nom-signal { , nom-signal } : [ mode ] (sous-)type
[ := expression ]

```

Le mode d'un signal d'interface peut être:

- **in**: le signal ne peut qu'être lu dans l'architecture ou le corps de sous-programme. C'est le seul mode admis pour un argument de fonction.
- **out**: le signal ne peut qu'être écrit (une valeur lui est assignée) dans l'architecture ou le corps de sous-programme.
- **inout**: le signal peut être lu et écrit dans l'architecture ou le corps de sous-programme. Le signal peut être assigné de plusieurs sources.
- **buffer**: le signal peut être lu et écrit dans l'architecture ou le corps de sous-programme. Le signal ne peut être assigné que par une seule source. Ce mode n'est pas admis pour un argument de procédure.

Une *variable d'interface* (*interface variable*) ne peut être qu'un argument de sous-programme. Syntaxe:

```

[ variable ] nom-variable { , nom-variable } : [ mode ] (sous-)type
[ := expression ]

```

Le mode d'une variable d'interface peut être:

- **in**: la variable ne peut qu'être lue dans le corps de sous-programme. C'est le seul mode admis pour un argument de fonction.
- **out**: la variable ne peut qu'être écrite (une valeur lui est assignée) dans le corps de sous-programme.
- **inout**: la variable peut être lue et écrite dans le corps de sous-programme. Elle peut être assignée de plusieurs sources.

Un *fichier d'interface* (*interface file*) ne peut être qu'un argument de sous-programme. Syntaxe:

```
file nom-fichier { , nom-fichier } : (sous-)type
```

**NOTE**

VHDL-87 traite les fichiers comme des variables. Un fichier d'interface est ainsi déclaré comme une variable:

```
variable nom-fichier : mode (sous-)type
```

où le mode ne peut être que **in** (le fichier est lu) ou **out** (le fichier est écrit).

### A.3.8. Déclaration de composant

Une déclaration de composant (*component declaration*) ne définit pas une nouvelle unité de conception, mais définit plutôt *l'empreinte* (*template, socket*) d'une entité de conception qui doit être instanciée dans le modèle. Cette empreinte correspond à l'interface de l'entité de conception (paramètres génériques et ports).

Syntaxe:

```
component nom-composant [ is ]
  [ generic ( constante-interface { , constante-interface } ) ; ]
  [ port ( port-interface { , port-interface } ) ; ]
end component [ nom-composant ] ;
```

Exemple:

```
component flipflop is
  generic (Tprop, Tsetup, Thold: Time := 0 ns);
  port (clk, d: in bit; q: out bit);
end component flipflop;
```

### A.3.9. Association

Une *indication d'association* (*binding indication*) établit une correspondance entre une *partie formelle* (*formal part*) et une *partie actuelle* (*actual part*). Syntaxe:

```
[ partie-formelle => ] partie-actuelle
```

La partie formelle est le nom d'une déclaration d'interface. La partie actuelle peut être une expression, un nom de signal, un nom de variable ou le mot-clé **open**.

L'*association par nom* (*named association*) utilise une partie formelle explicite. L'*association par position* (*positional association*) n'utilise que la partie actuelle. La partie formelle correspondante est déduite de la position de l'élément associé dans la liste d'interface.

## A.4. Instructions concurrentes

Une instruction concurrente ne peut apparaître que dans une déclaration d'entité ou un corps d'architecture.

### A.4.1. Processus

Un processus (*process*) définit une portion de code dont les instructions sont exécutées en séquence dans l'ordre donné. Chaque processus s'exécute de manière asynchrone par rapport aux autres processus et aux instances de composants.

Syntaxe:

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

La liste optionnelle de signaux entre parenthèses après le mot-clé **process** définit la *liste de sensibilité* (*sensitivity list*) du processus. Un événement sur l'un de ces signaux a pour conséquence une activation du processus et une exécution de ses instructions. Un processus ayant une liste de sensibilité ne peut pas contenir d'instructions **wait**.

Les déclarations locales suivantes sont admises: type, sous-type, constante, variable, fichier, alias, sous-programme (en-tête et corps), clause **use**. Les variables locales conservent leur valeur d'une activation du processus à une autre.

Les instructions séquentielles admises sont: assertion, conditionnelle (**if**), de sélection (**case**), de boucle (**loop**, **next**, **exit**), assignation de signal et de variable, de synchronisation (**wait**), appel de procédure.

#### A.4.2. Assignation concurrente de signal

L'assignation d'une valeur à un signal est dénotée par "<=". Trois types d'assignation concurrente de signal existe: l'assignation simple, l'assignation conditionnelle et l'assignation sélective.

##### **Assignation concurrente simple**

L'assignation concurrente simple assigne une valeur ou une forme d'onde à un signal.

Syntaxe:

```
[ étiquette : ] nom-signal <= [ mode-délai ] forme-onde ;
```

Le mode de délai permet de spécifier un mode inertiel ou un mode transport. Le mode inertiel est le mode par défaut. Syntaxe:

```
transport | [ reject temps-rejection ] inertial
```

Le temps de rejection doit être positif et plus petit ou égal au délai inertiel.

<b>NOTE</b>	VHDL-87 ne permet qu'une spécification limitée du type de délai: <i>nom-signal</i> <= [ <b>transport</b> ] <i>forme-onde</i> ;
-------------	---

Il est possible d'assigner une seule valeur ou une forme d'onde à un signal. Syntaxe:

```
valeur | expression [ after expression-temps ]
  { , valeur | expression [ after expression-temps ] }
```

L'expression temps est par défaut égale à 0 ns. Les expressions temps d'une forme d'onde doivent être spécifiées par valeurs croissantes. Un signal ne prend jamais sa nouvelle valeur immédiatement, mais seulement après un certain délai. Ce délai est celui spécifié par l'expression temps. C'est un *délai delta* (*delta delay*) si l'expression temps est égale à 0 ns.

L'instruction d'assignation concurrente de signal possède une forme équivalente utilisant un processus:

```
[ étiquette : ] process ( signaux de la forme d'onde )
begin
  nom-signal <= [ mode-délai ] forme-onde ;
end process [ étiquette ] ;
```

Exemples:

```
S <= A xor B after 5 ns;
-- processus équivalent:
process (A, B)
begin
  S <= A xor B after 5 ns;
end process;

S <= '0', '1' after 10 ns, '0' after 20 ns;
-- processus équivalent:
process
begin
  S <= '0', '1' after 10 ns, '0' after 20 ns;
  wait; -- le processus est stoppé indéfiniment
end process;
```

### **Assignment concurrente conditionnelle**

L'assignation concurrente conditionnelle (*conditional signal assignment statement*) permet d'assigner différentes valeurs ou formes d'ondes en fonction d'une condition.

Syntaxe:

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

L'instruction d'assignation concurrente conditionnelle de signal possède une forme équivalente utilisant un processus:

```
[ étiquette : ]
process ( signaux des formes d'ondes + signaux des conditions )
begin
  if expression-booléenne then
    assignation-signal-simple ;
  { elsif expression-booléenne then
    assignation-signal-simple ; }
  [ else
    assignation-signal-simple ; ]
  end if;
end process [ étiquette ] ;
```

Exemple:

```

A <= B after 10 ns when Z = '1' else C after 15 ns;
-- processus équivalent:
process (B, C, Z)
begin
  if Z = '1' then
    A <= B after 10 ns;
  else
    A <= C after 15 ns;
  end if;
end process;

```

### Assignment concurrente sélective

L'assignation concurrente de signal sélective (*selective signal assignment statement*) permet d'assigner différentes valeurs ou formes d'ondes en fonction de la valeur d'une expression de sélection.

Syntaxe:

```

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

```

L'expression de sélection est de type discret ou d'un type tableau monodimensionnel. Les choix peuvent prendre l'une des formes suivantes:

- Des littéraux chaînes de caractères, des littéraux chaînes de bits ou des expressions constantes du même type.
- Des intervalles de valeurs discrètes.
- Le mot-clé **others** spécifie tous les choix possibles non spécifiés dans les choix précédants.

L'instruction d'assignation concurrente sélective de signal possède une forme équivalente utilisant un processus:

```

[ étiquette : ]
process ( signaux de l'expression de sélection )
begin
  case expression is
    when choix { | choix } => forme-onde ;
    { when choix { | choix } => forme-onde ; }
  end case;
end process [ étiquette ] ;

```

Exemple:

```

with muxval select
  S <= A after 5 ns when "00",
    B after 10 ns when "01" | "10",
    C after 15 ns when others;
-- processus équivalent:
process (A, B, C, muxval)
begin
  case muxval is
    when "00"          => S <= A after 5 ns;
    when "01" | "10" => S <= B after 10 ns;
    when others        => S <= C after 15 ns;
  end case;
end process;

```

### A.4.3. Instance de composant

Une instance de composant (*component instantiation statement*) crée une copie d'un composant préalablement déclaré (§ A.3.8) et définit les connexions de ce composant avec le reste du modèle. Syntaxe:

```

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

```

Exemple:

```

C_ADD: ADDN generic map (N => 8)
          port map (A8, B8, sum => S8, cout => open);

```

L'*instanciation directe* (*direct instantiation*) n'est supportée qu'en VHDL-93. Elle ne requiert pas de déclaration de composant préalable. Syntaxe:

```

nom-instance : entity nom-entité [ ( nom-architecture ) ]
  [ generic map ( liste-association-paramètres-génériques ) ]
  [ port map ( liste-association-ports ) ] ;

```

Exemple:

```

C_ADD: entity WORK.ADDN(str)
      generic map (N => 8)
      port map (A8, B8, sum => S8, cout => open);

```

### A.4.4. Génération d'instructions

La génération d'instructions concurrentes peut être faite de manière itérative ou conditionnelle. La génération a lieu à l'élaboration, avant le début de la simulation.

Syntaxe:

```

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

```

Les déclarations locales admises sont les mêmes que pour le corps d'architecture. Elles sont dupliquées comme les instructions concurrentes.

**NOTE**

VHDL-87 ne permet pas les déclarations locales dans une instruction **generate**.

L'instruction de génération itérative (**for**) génère autant d'instances des instructions spécifiées que de valeurs prises par l'identificateur. L'identificateur n'a pas besoin d'être déclaré. L'instruction de génération conditionnelle (**if**) ne génère des instances des instructions spécifiées que si l'expression booléenne a une valeur True.

Exemple:

```
Gen: for i in 1 to N generate
  First: if i = 1 generate
    C1: COMP port map (CLK, D => A, Q => S(i));
    S2(i) <= S(i) after 10 ns;
  end generate First;
  Int: if i > 1 and i < N generate
    CI: COMP port map (CLK, D => S(i-1), Q => S(i));
    S2(i) <= S(i-1) after 10 ns;
  end generate Int;
  Last: if i = N generate
    CN: COMP port map (CLK, D => S(i-1), Q => B);
    S2(i) <= S(i-1) after 10 ns;
  end generate Last;
end generate Gen;
```

## A.5. Instructions séquentielles

Une instruction séquentielle ne peut apparaître que dans un processus ou dans un corps de sous-programme.

### A.5.1. Assignment de signal

La syntaxe d'assignation de signal a déjà été présentée au § A.4.2 (assignation simple de signal).

### A.5.2. Assignment de variable

L'assignation d'une valeur à une variable est dénotée par ":=". Syntaxe:

```
[ étiquette : ] nom-variable := expression ;
```

La variable prend sa nouvelle valeur immédiatement.

Exemples:

```
count := (count + 1) / 2;
currentState := Init;
```

### A.5.3. Instruction conditionnelle

Syntaxe:



```

[ étiquette : ] if expression-booléenne then
  { instruction-séquentielle }
{ elsif expression-booléenne then
  { instruction-séquentielle } }
[ else
  { instruction-séquentielle } ]
end if [ étiquette ] ;

```

Exemples:

```

if A > B then
  Max := A;
elsif A < B then
  Max := B;
else
  Max := Integer'Low;
end if;

```

#### A.5.4. Instruction sélective

Syntaxe:

```

[ étiquette : ]
case expression is
  when choix { | choix } => { instruction-séquentielle }
  { when choix { | choix } => { instruction-séquentielle } }
end case [ étiquette ] ;

```

Voir l'assignation concurrente de signal sélective au sujet de l'expression de sélection et des formes possibles de choix.

Exemple:

```

case int is -- int est du type integer
  when 0 =>
    V := 4; S <= '1' after 5 ns;
  when 1 | 2 | 7 =>
    V := 6; S <= '1' after 10 ns;
  when 3 to 6 =>
    V := 8; S <= '1' after 15 ns;
  when 9 => null; -- pas d'opération
  when others =>
    V := 0; S <= '0'; -- tous les autres cas possibles
end case;

```

### A.5.5. Instructions de boucle

Plusieurs formes d'instructions de boucle sont disponibles.

Syntaxe:

```
[ étiquette : ]
[ while condition | for identificateur in intervalle ] loop
  { instruction-séquentielle }
end loop [ étiquette ] ;
```

L'indice d'une boucle **for** n'a pas besoin d'être déclaré et il est visible dans le corps de la boucle.

Les deux instructions **exit** et **next** permettent de contrôler le comportement de la boucle. L'instruction **next** stoppe l'itération courante et démarre la boucle à l'itération suivante. Syntaxe:

```
[ étiquette : ]
next [ étiquette-boucle ] [ when expression-booléenne ] ;
```

L'instruction **exit** stoppe la boucle et continue l'exécution à la première instruction après la boucle. Syntaxe:

```
[ étiquette : ]
exit [ étiquette-boucle ] [ when expression-booléenne ] ;
```

Exemples:

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

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

-- boucle while:
while i < str'length and str(i) /= ' ' loop
  next when i = 5; -- saut à l'indice suivant
  i := i + 1;
end loop;

-- boucle for:
L1: for i in 15 downto 0 loop
  L2: for j in 0 to 7 loop
    exit L1 when i = j; -- saut à l'indice suivant
    -- dans la boucle externe L1
    tab(i,j,) := i*j + 5;
  end loop L2;
end loop L1;
```

### A.5.6. Instruction wait

L'instruction **wait** permet de synchroniser un processus avec son environnement. Elle peut prendre plusieurs formes.

Syntaxe:

```
[ étiquette : ] wait
  [ on nom-signal { , nom-signal } ]
  [ until expression-booléenne ]
  [ for expression-temps ] ;
```

L'instruction **wait** seule stoppe un processus indéfiniment.

La forme **wait on** synchronise un processus sur un événement sur un ou plusieurs signaux. Ces signaux définissent une *liste de sensibilité* (*sensitivity list*). Exemple:

```
wait on S1, S2;
```

La forme **wait until** synchronise un processus sur une condition. Exemple:

```
wait until clk = '1';
-- équivalent à:
wait on clk until clk = '1';
```

<b>NOTE</b>	<p>Si l'instruction contient une liste de sensibilité explicite, la condition est seulement testée lorsqu'un événement survient sur l'un des signaux de la liste.</p> <p>Exemple:</p> <pre>wait on reset until clk = '1'; -- le signal clk ne fait pas partie de la -- liste de sensibilité</pre>
-------------	---

<b>NOTE</b>	<p>Si l'expression booléenne ne contient pas de signal, l'instruction est équivalente à un <b>wait</b> seul, donc le processus est suspendu indéfiniment!</p> <p>Exemple (où stop est une variable):</p> <pre>wait until stop = True; -- équivalent à wait</pre>
-------------	--

La forme **wait for** permet de spécifier un temps de suspension pour un processus. Exemple:

```
wait for 15 ns;
```

Si l'instruction contient en plus une liste de sensibilité ou une condition, le processus peut être réactivé plus tôt. Exemple:

```
wait until trigger = '1' for 20 ns;
-- est équivalent à:
wait on trigger until trigger = '1' for 20 ns;
```

## A.6. Sous-programmes

VHDL supporte deux types de sous-programmes: les procédures et les fonctions. Chacune d'elles encapsule une série d'instructions séquentielles. Une procédure est une instruction à part entière, alors qu'une fonction est une expression retournant un résultat.

### A.6.1. Procédure

Syntaxe:

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

Les déclarations locales peuvent inclure des types, des sous-types, des constantes, des variables et des sous-programmes. Contrairement à un processus, les déclarations sont élaborées à nouveau (p. ex. les variables sont recréées) à chaque appel de procédure.

Les paramètres d'interface admis sont les constantes, les variables, les signaux et les fichiers. Le § A.3.7 donne la syntaxe de ces paramètres d'interface.

L'*appel de procédure* (*procedure call*) est une instruction séquentielle ou concurrente selon l'endroit où il est effectué. Syntaxe:

```
[ étiquette : ] nom-procédure [ ( liste-association-paramètres ) ] ;
```

La syntaxe de la liste d'association des paramètres a été donnée au § A.3.9. Exemple:

```
procedure p (f1: in t1; f2: in t2; f3: out t3; f4: in t4 := v4) is
begin
  ...
end procedure p;
-- appels possibles (a = paramètre actuel):
p(a1, a2, a3, a4);
p(f1 => a1, f2 => a2, f4 => a4, f3 => a3);
p(a1, a2, f4 => open, f3 => a3);
p(a1, a2, a3);
```

L'*appel concurrent de procédure* (*concurrent procedure call*) est équivalent à un processus contenant le même appel de procédure et sensible aux paramètres signaux actuels de mode **in** ou **inout**. **Exemple:**

```
-- déclaration:
procedure proc (signal S1, S2: in bit; constant C1: integer := 5) is
begin
  ...
end procedure proc;

-- appel concurrent:
Appel_Proc: proc (S1, S2, C1);
```

```
-- processus équivalent:
Appel_Proc: process
begin
  proc (S1, S2, C1);
    wait on S1, S2;
end process Appel_Proc;
```

## A.6.2. Fonction

Une fonction calcule et retourne un résultat qui peut être directement utilisé dans une expression.

Syntaxe:

```
[ pure | impure ]
function nom-fonction
  [ ( liste-paramètres-interface ) ] return (sous-)type is
  { déclaration-locale }
begin
  { instruction-séquentielle }
end [ function ] [ nom-fonction ] ;
```

Les déclarations locales peuvent inclure des types, des sous-types, des constantes, des variables et des sous-programmes. Contrairement à un processus, les déclarations sont élaborées à nouveau (p. ex. les variables sont recrées) à chaque appel de fonction.

Les paramètres d'interface admis sont les constantes, les variables, les signaux et les fichiers. Le § A.3.7 donne la syntaxe de ces paramètres d'interface. Une fonction n'admet que des paramètres en mode **in** (c'est le mode par défaut).

Une fonction retourne la valeur calculée au moyen d'une instruction **return**. L'exécution d'une instruction **return** termine l'appel de la fonction. Syntaxe:

```
[ étiquette : ] return expression ;
```

Une fonction est dite **pure** si elle ne fait aucune référence à une variable ou à un signal déclaré dans l'environnement dans lequel la fonction est appelée. En d'autres mots, une fonction pure n'a pas d'effets de bord. C'est le type de fonction par défaut. Le mot-clé **impure** permet de définir des fonctions avec effets de bord.

<b>NOTE</b>	VHDL-87 n'admet que des fonctions pures.
-------------	--

L'**appel de fonction** (*function call*) n'est pas une instruction en elle-même, mais fait partie d'une expression. Syntaxe:

```
[ étiquette : ] nom-fonction [ ( liste-association-paramètres ) ] ;
```

La syntaxe de la liste d'association des paramètres a été donnée au § A.3.9.

Exemple:

```

function limit (value, min, max, gain: integer) return integer is
  variable val: integer
begin
  if value > max then
    val := max;
  elsif value < min then
    val := min;
  else
    val := value;
  end if;
  return gain * val;
end function limit;

-- Usages:
new_value := limit(value, min => 10, max => 100, gain => 2);
new_speed := old_speed + scale * limit(error, -10, +10, 2);

```

### A.6.3. Surcharge

La *surcharge* (*overloading*) est un mécanisme permettant de définir plusieurs sous-programmes ayant la même fonction et le même nom, mais agissant sur des paramètres de nombres et de types différents.

Exemples:

```

-- trois procédures de conversion vers un nombre entier:
procedure convert (r : in real;      result: out integer) is ...
procedure convert (b : in bit;      result: out integer) is ...
procedure convert (bv: in bit_vector; result: out integer) is ...

-- trois fonctions de calcul du minimum:
function min (a, b: in integer) return integer is ...
function min (a: in real; b: in real) return real is ...
function min (a, b: in bit) return bit is ...

```

L'appel de la procédure ou de la fonction définira quel sous-programme il faut exécuter en fonction des paramètres actuels, plus précisément en fonction de leur nombre, leurs types, et l'ordre dans lequel ils sont déclarés.

Il est aussi possible de surcharger des opérateurs prédéfinis tels que "+", "-", **and**, **or**, etc. Exemple:

```

type logic4 is ('0', '1', 'X', 'Z');
function "and" (a, b: in logic4) return logic4 is ...
function "or" (a, b: in logic4) return logic4 is ...

```

### A.6.4. Sous-programmes dans un paquetage

Un sous-programme placé dans un paquetage peut être décomposé en deux parties. Une partie, la déclaration de sous-programme, se trouve dans la déclaration de paquetage. L'autre partie, le corps de sous-programme se trouve dans le corps de paquetage.

Exemple d'un paquetage définissant un sous-type vecteur de 32 bits et ses opérations associées:

```
package bv32_pkg is
  subtype word32 is bit_vector(31 downto 0);
  procedure add (a, b: in word32;
                result: out word32; overflow: out boolean);
  function "<" (a, b: in word32 ) return boolean;
  ... -- autres déclarations de sous-programmes
end package bv32_pkg;

package body bv32_pkg is
  procedure add (a, b: in word32;
                result: out word32; overflow: out boolean) is
    ... -- déclarations locales
  begin
    ... -- corps de la procédure
  end procedure add;
  function "<" (a, b: in word32 ) return boolean is
    ... -- déclarations locales
  begin
    ... -- corps de la fonction
  end function "<";
  ... -- autres corps de sous-programmes
end package body bv32_pkg;
```





## Annexe B: Paquetages VHDL standard

Cette annexe décrit les contenus des paquetages standard STANDARD et TEXTIO qui font partie de la norme [LRM93] et du paquetage STD\_LOGIC\_1164 [STD1164].

### B.1. Paquetage STANDARD

Le paquetage STANDARD contient les types prédéfinis du langage VHDL. Le paquetage STANDARD est implicitement visible pour toute unité de conception.

Les déclarations du paquetage sont les suivantes:

```
-- types énumérés prédéfinis
type boolean is (false, true);
type bit is ('0', '1');
type character is ( ...256 caractères... );

    NOTE: VHDL-87 ne supporte que le jeu de caractères sur 7 bits, donc sans les caractères
    accentués.

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

-- types numériques prédéfinis
type integer is range dépend de l'implémentation ;
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
type real is range dépend de l'implémentation ;

-- traitement du temps
type time is range dépend de l'implémentation
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 1000 sec;
        hr = 1000 min;
    end units;
subtype delay_length is time range 0 to time'high;
impure function now return delay_length; -- retourne le temps courant

-- types tableau prédéfinis
type string is array (positive range <>) of character;
type bit_vector is array (natural range <>) of bit;

-- types pour traitement de fichiers
type file_open_kind is (read_mode, write_mode, append_mode);
type file_open_status is (read_mode, write_mode, append_mode);

-- attribut prédéfini
attribute foreign: string;
```

## B.2. Paquetage TEXTIO

Le paquetage TEXTIO contient des définitions de types et d'opérations utiles pour la lecture et l'écriture de fichiers textes. Le paquetage TEXTIO n'est pas implicitement visible. Toute unité de conception faisant usage de ce paquetage doit spécifier la clause de contexte:

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

Les déclarations du paquetage sont les suivantes:

```
-- types prédéfinis
type line is access string; -- une ligne est une chaîne dynamique
type text is file of string;
type side is (right, left);
subtype width is natural;

-- fichiers standard
file input: text open read_mode is "std_input";
file output: text open write_mode is "std_output";

-- opérations de lecture sur un fichier texte

procedure readline (file f: texte; l: out line);

-- pour chaque opération deux procédures surchargées sont définies:
-- l'une retourne l'état de l'opération en plus de la valeur lue

-- lecture d'un bit
procedure read (l: inout line; value: out bit; good: out boolean);
procedure read (l: inout line; value: out bit);

-- lecture d'un bit_vector
procedure read (l: inout line; value: out bit_vector; good: out boolean);
procedure read (l: inout line; value: out bit_vector);

-- lecture d'un booléen
procedure read (l: inout line; value: out boolean; good: out boolean);
procedure read (l: inout line; value: out boolean);

-- lecture d'un caractère
procedure read (l: inout line; value: out character; good: out boolean);
procedure read (l: inout line; value: out character);

-- lecture d'un entier
procedure read (l: inout line; value: out integer; good: out boolean);
procedure read (l: inout line; value: out integer);

-- lecture d'un réel
procedure read (l: inout line; value: out real; good: out boolean);
procedure read (l: inout line; value: out real);

-- lecture d'une chaîne de caractères
procedure read (l: inout line; value: out string; good: out boolean);
procedure read (l: inout line; value: out string);

-- lecture d'un temps
procedure read (l: inout line; value: out time; good: out boolean);
procedure read (l: inout line; value: out time);

-- opérations d'écriture sur un fichier texte
```

```
procedure writeline (file f: texte; l: inout line);  
  
-- écriture d'un bit  
procedure write (l: inout line; value: in bit;  
                justified: in side := right; field: in width := 0);  
  
-- écriture d'un bit_vector  
procedure write (l: inout line; value: in bit_vector;  
                justified: in side := right; field: in width := 0);  
  
-- écriture d'un booléen  
procedure write (l: inout line; value: in boolean;  
                justified: in side := right; field: in width := 0);  
  
-- écriture d'un caractère  
procedure write (l: inout line; value: in character;  
                justified: in side := right; field: in width := 0);  
  
-- écriture d'un entier  
procedure write (l: inout line; value: in integer;  
                justified: in side := right; field: in width := 0);  
  
-- écriture d'un réel  
procedure write (l: inout line; value: in real;  
                justified: in side := right; field: in width := 0;  
                digits: in natural := 0);  
  
-- écriture d'une chaîne de caractères  
procedure write (l: inout line; value: in string;  
                justified: in side := right; field: in width := 0);  
  
-- écriture d'un temps  
procedure write (l: inout line; value: in real;  
                justified: in side := right; field: in width := 0;  
                unit: in time := ns);
```

**NOTE:** VHDL-87 définit en plus la fonction **endline**:

```
function endline (l: in line) return boolean;
```

Cete fonction retourne la valeur **true** si la chaîne pointée par **l** est vide et la valeur **false** autrement. En VHDL-93 la condition peut être testée par l'expression **l'length = 0**.

### B.3. Paquetage STD\_LOGIC\_1164

Le paquetage STD\_LOGIC\_1164 est un standard IEEE définissant les types, les opérateurs et les fonctions nécessaires à une modélisation détaillée au niveau du transistor. Il est basé sur un système de valeurs logiques à 3 états (zéro, un et inconnu (*unknown*)) et 3 niveaux de forces (fort, faible et haute impédance). Un niveau fort représente l'effet d'une source active telle qu'une source de tension; un niveau faible représente l'effet d'une source résistive telle qu'une résistance pull-up ou un transistor de transmission; un niveau haute impédance représente l'effet d'une source désactivée. Un niveau fort domine un niveau faible qui domine lui-même un niveau haute impédance. Les 3 états sont complétés par l'état non initialisé (représentant le fait qu'une valeur n'a jamais été assignée à un signal) et l'état *dont-care* ou indéfini.

L'utilisation du paquetage nécessite les déclarations de contexte suivantes:

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

Les déclarations du paquetage sont les suivantes:

```
-----
-- 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;

```

Le standard définit une fonction de résolution `resolved` qui utilise une table de résolution réalisant les règles suivantes:

- S'il existe une seule source, le signal résolu prends la valeur de cette source.
- Si le tableau de sources est vide, le signal résolu vaut 'Z'.
- Une valeur de source forte ('X', '0' ou '1') domine une valeur de source faible ('W', 'L' ou 'H').
- Deux sources de même forces mais de valeurs différentes produisent une valeur résolue inconnue de même force ('X' ou 'W').
- La valeur haute impédance 'Z' est toujours dominée par des valeurs fortes et faibles.
- La résolution d'une valeur *don't care* '-' avec n'importe quelle autre valeur donne la valeur 'X'.
- La résolution d'une valeur 'U' avec n'importe quelle autre valeur donne la valeur 'U'. Ceci permet de détecter les signaux qui n'ont pas été initialisés correctement.

La fonction de résolution `resolved` est définie de la manière suivante:

```

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

function resolved ( s : std_ulogic_vector ) return std_ulogic is
  variable result : std_ulogic := 'Z'; -- weakest state default
begin
  -- the test for a single driver is essential otherwise the
  -- loop would return 'X' for a single driver of '-' and that
  -- would conflict with the value of a single driver unresolved
  -- signal.
  if s'length = 1) then return s(s'low);
  else
    for i in s'range loop
      result := resolution_table(result, s(i));
    end loop;
  end if;
  return result;
end resolved;

```





## Annexe C: Autres paquetages VHDL

Cette annexe décrit les contenus de paquetages VHDL qui ne sont pas standard mais largement utilisés en pratique.

### C.1. Paquetage STD\_LOGIC\_ARITH

Le paquetage STD\_LOGIC\_ARITH définit des types tableau représentant des entiers non signés ou signés, leur opérations arithmétiques et relationnelles correspondantes et des fonctions de conversion de types. Le paquetage utilise les définitions du paquetage standard STD\_LOGIC\_1164. La version du paquetage présentée ici est celle supportée par les outils Synopsys. Une version standard IEEE est disponible depuis peu [STD1076.3].

Même si le paquetage n'est pas standard, il est usuellement (et abusivement) compilé dans la bibliothèque IEEE<sup>1</sup>. Pour l'utiliser il faut donc déclarer les clauses de contextes suivantes:

```
library ieee;
use ieee.std_logic_1164.all;    -- optionnel
use ieee.std_logic_arith.all;
```

La déclaration du paquetage STD\_LOGIC\_ARITH est la suivante:

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

package std_logic_arith is

    -- types "entier" non signé et signé
    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;

    -- surcharge de l'opérateur binaire "+"
    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;
```

1. Le contenu du paquetage peut être légèrement différent selon l'environnement VHDL utilisé (p. ex. Synopsys, Mentor, etc.).

```

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;

-- surcharge de l'opérateur binaire "-"
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;

-- surcharge des opérateurs unaires "+", "-" et abs
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;

-- surcharge de l'opérateur "*"
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;

```

```
-- surcharge de l'opérateur "<"
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;

-- surcharge de l'opérateur "<="
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;

-- surcharge de l'opérateur ">"
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;

-- surcharge de l'opérateur ">="
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;

-- surcharge de l'opérateur "="
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;
```

```

-- surcharge de l'opérateur "/"=
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;

-- opérations de décalage1
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;

-- fonctions de conversion en type integer
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;

-- fonctions de conversion en type unsigned
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;

-- fonctions de conversion en type signed
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;

-- fonctions de conversion en type std_logic_vector
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;

-- retourne un vecteur std_logic_vector(size-1 downto 0) complété par
-- des zéros; size < 0 est équivalent à size = 0
function ext(arg: std_logic_vector; size: integer)
return std_logic_vector;

-- retourne un vecteur std_logic_vector(size-1 downto 0) complété par
-- des zéros en conservant le signe; size < 0 est équivalent à size = 0
function sxt(arg: std_logic_vector; size: integer)
return std_logic_vector;

end std_logic_arith;

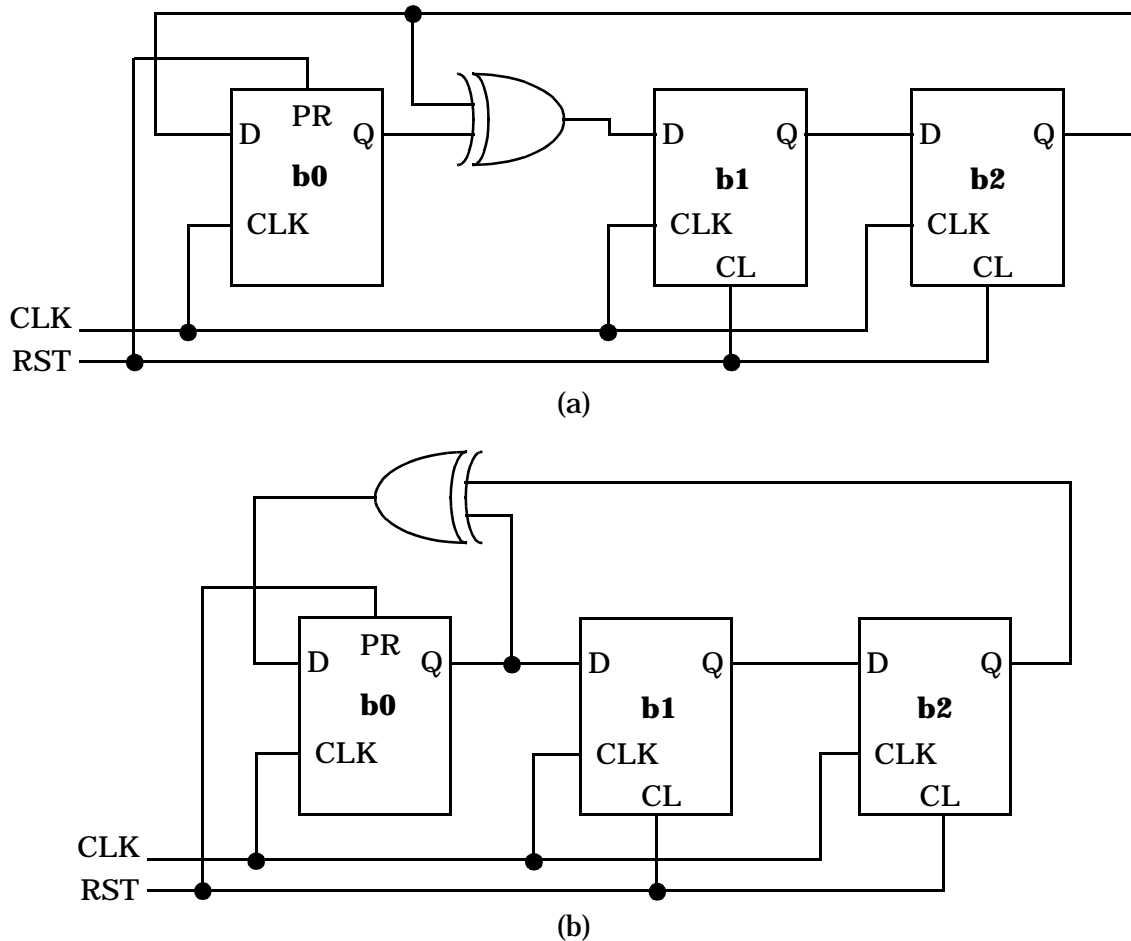
```

1. VHDL-93 définit les opérateurs **shl** et **shr** en standard.

## Annexe D: LFSR

Un circuit LFSR (*Linear Feedback Shift Register*) est un registre à décalage séquentiel possédant une rétroaction combinatoire dont l'effet est de générer des séquences binaires pseudo-aléatoires de différentes longueurs. La boucle de rétroaction effectue le XOR ou le XNOR de différents bits du registre. Le choix des bits participant à la boucle permet de générer des séquences de longueurs différentes. Certains choix permettent une séquence de longueur maximum  $2^n - 1$  pour un registre de  $n$  bits.

La Figure D.1 illustre deux structures possibles d'un LFSR 3 bits. La structure (a), appelée *one-to-many*, a l'avantage de posséder un délai combinatoire minimum entre les registres. La structure (b), appelée *many-to-one*, peut comporter un arbre de portes XOR ou XNOR complexe pour  $n > 3$ .



**Figure D.1.** Structures d'un LFSR 3 bits: (a) *one-to-many*, (b) *many-to-one*.

Les points de rétroaction sont appelés *taps*. Les deux structures LFSR de la Figure D.1 génèrent des séquences différentes de longueur maximum  $2^n - 1 = 7$  (Table D.1). Il est aussi possible de remplacer les XOR par des XNOR pour produire encore d'autres séquences.

L'usage de portes XOR (XNOR) interdit l'état "000" ("111") car dans ce cas le registre reste bloqué dans cet état. L'initialisation du registre permet d'éviter cet état. Une autre possibilité serait de permettre le chargement en parallèle d'une valeur légale.

one-to-many				many-to-one			
b2	b1	b0	$b2 \oplus b0$	b2	b1	b0	$b2 \oplus b0$
0	0	1	1	0	0	1	1
0	1	0	0	0	1	1	1
1	0	0	1	1	1	1	0
0	1	1	1	1	1	0	1
1	1	0	1	1	0	1	0
1	1	1	0	0	1	0	0
1	0	1	0	1	0	0	1

**Table D.1.** Séquences de longueur maximum générées par les structures LFSR de la Figure D.1.

Le Code 68 donne le modèle d'un LFSR 3 bits avec une structure *one-to-many*. Le Code 69 donne le modèle d'un LFSR 3 bits avec une structure *many-to-one*. Les taps [0,2] permettent de générer les séquences de longueurs maximales de la Table D.1.

```

library IEEE;
use IEEE.std_logic_1164.all;

entity LFSR3 is
  port (CLK, RST: in std_logic;
        Q:          out std_logic_vector(0 to 2));
end LFSR3;

architecture onetomany of LFSR3 is
  -- points de rétroaction
  constant TAPS: std_logic_vector(Q'range)
    := (0 | 2 => '1',others => '0');
  signal LFSR_reg: std_logic_vector(Q'range);
begin
  process (RST, CLK)
  begin
    if RST = '1' then
      LFSR_reg <= (0 => '1',others => '0'); -- initialisation
    elsif CLK'event and CLK = '1' then
      -- décalage des bits et XOR aux points de rétroaction
      for i in 2 downto 1 loop
        if TAPS(i-1) = '1' then
          LFSR_reg(i) <= LFSR_reg(i-1) xor LFSR_reg(2);
        else
          LFSR_reg(i) <= LFSR_reg(i-1); -- simple décalage
        end if;
      end loop;
      LFSR_reg(0) <= LFSR_reg(2);
    end if;
  end process;
  Q <= LFSR_reg;
end onetomany;

```

**Code 68.** .Modèle d'un LFSR 3 bits à structure *one-to-many*.

```

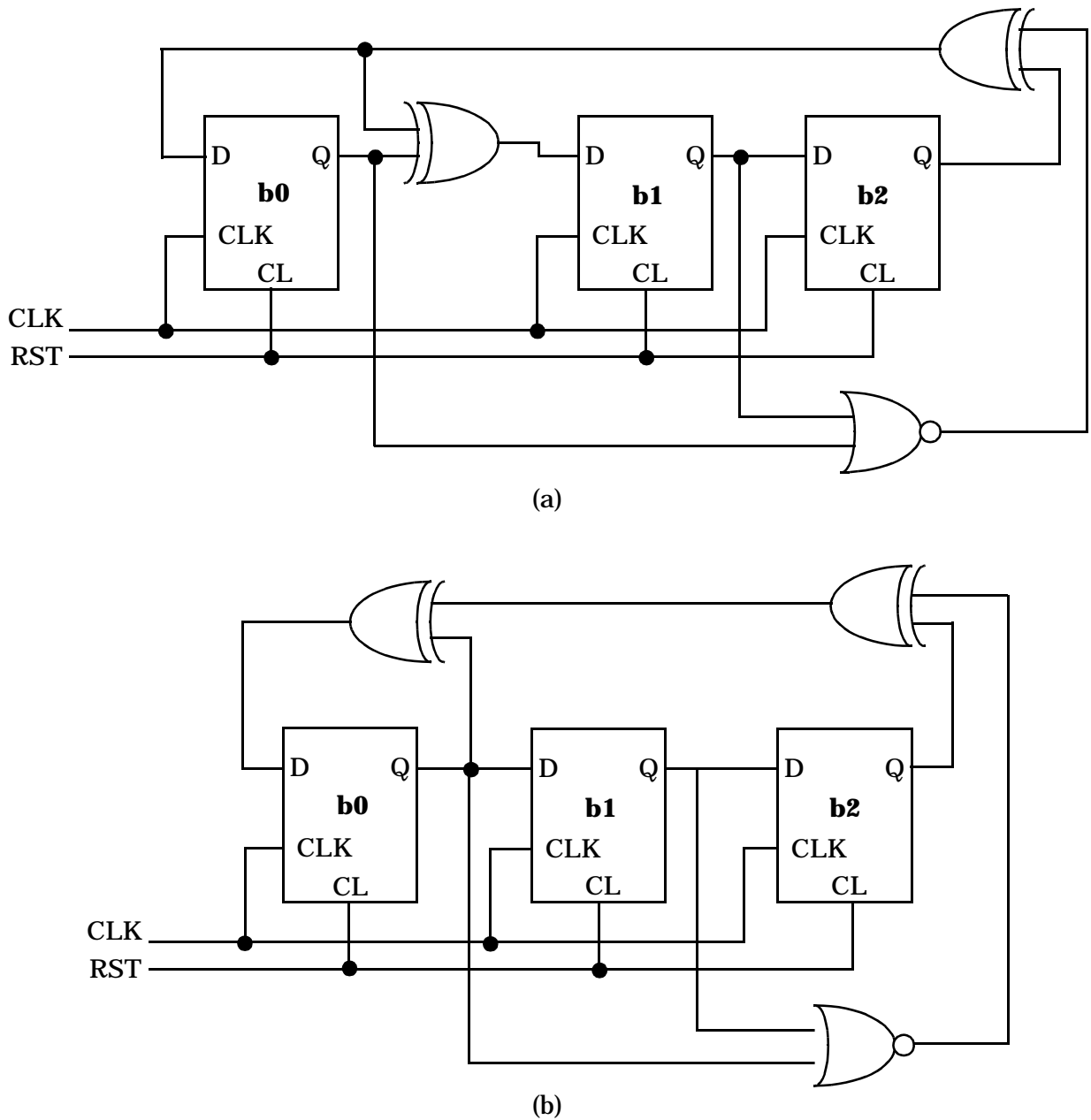
architecture manytoone of LFSR3 is
  -- points de rétroaction
  constant TAPS: std_logic_vector(Q'range)
    := (0 | 2 => '1',others => '0');
  signal LFSR_reg: std_logic_vector(Q'range);
begin
  process (RST, CLK)
    variable feedback: std_logic;
  begin
    if RST = '1' then
      LFSR_reg <= (0 => '1',others => '0'); -- initialisation
    elsif CLK'event and CLK = '1' then
      -- calcul du feedback
      feedback := LFSR_reg(2);
      for i in 0 to 1 loop
        if TAPS(i) = '1' then
          feedback := feedback xor LFSR_reg(i);
        end if;
      end loop;
      -- décalage des bits et XOR aux points de rétroaction
      for i in 2 downto 1 loop
        LFSR_reg(i) <= LFSR_reg(i-1); -- simple décalage
      end loop;
      LFSR_reg(0) <= feedback;
    end if;
  end process;
  Q <= LFSR_reg;
end manytoone;

```

**Code 69.** Modèle d'un LFSR 3 bits à structure *many-to-one*.



La génération de séquences incluant l'état interdit "000" (pour une rétroaction à base de XOR) ou "111" (pour une rétroaction à base de XNOR) est possible moyennant de la logique supplémentaire. Dans le premier cas (XOR), il s'agit de détecter que tous les bits, sauf le bit de poids fort, sont à zéro et d'inverser le bit de poids fort, nécessairement à un. Ceci force un état à zéro. L'inversion sur le bit de poids fort va le remettre à un et le cycle continuera comme avant. La [Figure D.2](#) illustre les structures de LFSR modifiées pour accepter l'état interdit.



**Figure D.2.** Structures de LFSR 3 bits générant les 8 états possibles, y compris l'état "000":  
(a) *one-to-many*, (b) *many-to-one*.



## Références

- [Airi94] R. Airiau, J.-M. Bergé, V. Olive, Circuit Synthesis with VHDL, Kluwer Academic Publishers, 1994.
- [Airi98] R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard, VHDL: du langage à la modélisation, Presses Polytechniques et Universitaires Romandes, 2ème éd., 1998.
- [Ashe96] P. J. Ashenden, The Designer's Guide to VHDL, Morgan Kaufmann, 1996.
- [Berg92] J.-M. Bergé, A. Fonkoua, S. Maginot, J. Rouillard, VHDL Designer's Reference, Kluwer Academic Publishers, 1992.
- [Berg93] J.-M. Bergé, A. Fonkoua, S. Maginot, J. Rouillard, VHDL'92, Kluwer Academic Publishers, 1993.
- [Bhas95a] J. Bhasker, A VHDL Primer: Revised Edition, Prentice Hall, 1995.
- [Bhas95b] J. Bhasker, A Guide to VHD Syntax, Prentice Hall Series in Innovative Technology, 1995.
- [Bhas96] J. Bhasker, A VHDL Synthesis Primer, Jarayam Bhasker, Star Galaxy Publishing, 1996.
- [Cohe95] B. Cohen, VHDL Coding Styles and Methodologies, Kluwer Academic Publishers, 1995.
- [LRM93] IEEE Standard VHDL Reference Manual, IEEE Std 1076-1993, IEEE Press, Ref. SH16840, June 1994.
- [Nava98] Z. Navabi, VHDL: Analysis and Modeling of Digital Systems, Second Edition, McGraw-Hill, 1998.
- [Ott94] D. E. Ott, T. J. Wilderotter, A Designer's Guide to VHDL Synthesis, Kluwer Academic Publishers, 1994.
- [Perr98] D. L. Perry, VHDL, Third Edition, McGraw-Hill, 1998.
- [Pick96] J. Pick, VHDL Techniques, Experiments, and Caveats, McGraw-Hill, 1996.
- [Rush96] A. Rushton, VHDL for Logic Synthesis: An Introductory Guide for Achieving Design Requirements, McGraw-Hill, 1996.
- [Rush98] A. Rushton, VHDL for Logic Synthesis, 2nd Ed., Wiley, 1998.
- [SDF] Standard Delay File Format Manual, version 2.1 or 3.0, Open Verilog International, <http://www.o vi.org/pubs.html>.
- [STD1076.3] IEEE Standard VHDL Synthesis Packages, IEEE Std 1076.3-1997, IEEE Press, Ref. SH94531, June 1997.
- [STD1076.6] IEEE Draft Standard For VHDL Register Transfer Level Synthesis, IEEE DASC VHDL Synthesis Interoperability Working Group, December 1997.
- [STD1164] IEEE Standard Multivalued Logic System for VHDL, IEEE Std 1164-1993, IEEE Press, Ref. SH16097, May 1993.
- [STD1076.4] IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification, IEEE Std 1076.4-1995, IEEE Press, Ref. SH94382, May 1996.

