Jacques Weber
Sébastien Moutault
Maurice Meaudre

Le Langage VHDL

du langage au circuit, du circuit au langage

5e édition

DUNOD

© Dunod, Paris, 1987, 2001, 2007, 2011, 2016
11 rue Paul Bert 92240 Malakoff

ISBN 978-2-10-072688-2

Table des matières

PARTIE E

UN LANGAGE EN ÉVOLUTION

Avant-propos à la cinquieme édition

Le langage VHDL évolue pour prendre aujourd'hui en charge l'activité de vérification. L'apparition de la norme 2008 a marqué une première étape importante vers cet objectif. La seconde étape, la création de bibliothèques spécialisées dans la vérification, était déjà annoncée. En 2012, l'apparition de la bibliothèque OSVVM (*Open Source VHDL Verification Methodology*) a marqué le point de départ de cette seconde étape.

La principale nouveauté de cette cinquième édition est la présentation des principales fonctionnalités offertes par cette toute récente bibliothèque de vérification.

Les outils de simulation, et a fortiori de synthèse, acceptent encore aujourd'hui diversement VHDL-2008, nous avons donc conservé dans le corps de l'ouvrage le standard, universellement reconnu, de la norme VHDL-1993. La dernière partie de l'ouvrage se consacre aux nouveautés liées au standard 2008.

Cette cinquième édition est articulée en cinq parties qui correspondent à des approches complémentaires :

- Une introduction générale présente l'ensemble du sujet : la réalisation (synthèse) et le test des circuits numériques, en utilisant VHDL. La description du circuit à réaliser et la simulation – tant au niveau fonctionnel que post-synthèse – se fait en utilisant ce même langage évolué. Cette unicité dans les différents niveaux de description est l'une des raisons de la généralisation des langages évolués dans le domaine de la conception de circuits.
- La seconde partie permet au lecteur de découvrir l'ensemble des possibilités offertes par le langage pour la synthèse et le test des circuits numériques, en suivant une démarche qui conduit à une application facile à implanter dans un circuit programmable (FPGA). Moyennant l'emploi de cartes d'évaluation et d'outils de développement (Altera ou Xilinx) très répandus dans les établissements

d'enseignement et les entreprises, le lecteur peut immédiatement mettre en œuvre les exemples supports. La même partie présente un panorama des pièges les plus classiques rencontrés dans cette démarche d'élaboration d'un nouveau composant numérique.

- La troisième partie apporte des compléments d'information sur la sémantique du langage et sur les outils de modélisation. À partir d'exemples très simples, le lecteur y découvre les principes de la modélisation des limites de fonctionnement des circuits numériques. Cette partie est complétée par la présentation d'un ensemble d'outils standardisés de simulation post-synthèse, utilisés par tous les grands fabricants de circuits programmables et d'ASIC : la librairie VITAL.
- La quatrième partie est une récapitulation de la syntaxe du langage VHDL. Cette partie a vocation à servir de référence au programmeur qui y trouvera décrites la plupart des constructions utiles en conception de circuit, avec des exemples simples ou des renvois à des applications tirées de la seconde partie.
- La dernière partie est consacrée aux dernières évolutions de VHDL et son univers, c'est-à-dire le standard VHDL-2008 et la bibliothèque OSVVM.

Les exemples, pris comme colonne vertébrale de la seconde partie, ont été choisis suffisamment classiques pour que de nombreux lecteurs en connaissent les principes ; ils illustrent la grande majorité des constructions utilisées en synthèse des systèmes numériques.

L'ordre de lecture n'est pas impérativement celui de l'organisation du livre : le lecteur qui souhaite privilégier un aperçu général du langage, avant d'explorer en détail les exemples de la seconde partie, peut très bien établir un parcours différent (parties A, D, C puis B, par exemple).

De nombreuses ressources Internet sont liées à cet ouvrage : les exemples du livre bien sûr, mais aussi des liens vers des fabricants de circuits, d'outils logiciels ou des sites dédiés à VHDL. On trouvera également sur le site un chapitre sur les circuits programmables, un aperçu sur le monde des « systèmes complexes sur une puce », sur celui des modules *IP* (*Intellectual Property*), sur celui du standard *WISHBONE* d'interconnexion de tels modules et des exercices sont également mis à disposition du lecteur. Toutes ces ressources sont accessibles sur le site de l'éditeur : www.dunod.com.

Jacques Weber
Sébastien Moutault

À la mémoire de Maurice Meaudre, à qui ce livre doit beaucoup, décédé avant la publication de la troisième édition de ce livre.

PARTIE **A**

MODÉLISATION ET SYNTHÈSE : LE MÊME LANGAGE

Représentation spatiale et analyse séquentielle[1] : la première privilégie le global, l'organisation, les relations, dans un traitement parallèle (et/ou simultané) ; la seconde décompose, analyse les successions d'événements, décrit les causalités.

Les deux visions sont nécessaires à la description des circuits électroniques. Elles produisent, dans les approches traditionnelles, deux catégories de langages, complémentaires mais disjoints : les schémas et les algorithmes[2]. Tenter de saisir le fonctionnement d'un microprocesseur à partir de son schéma électrique est illusoire, l'expliquer par un algorithme également[3].

Les langages de description du matériel, traduction littérale de *hardware description language* (HDL), se doivent de marier les deux approches. Le même langage permet, à tous les niveaux d'une représentation hiérarchique, d'associer les aspects structurels et les aspects fonctionnels, l'espace et le temps. VHDL est l'un de ces langages, il y en a d'autres, VERILOG, par exemple. Nous avons choisi, ici, d'explorer un peu le premier.

L'origine du langage suit une histoire un peu banale dans sa répétition[4] : le développement de l'informatique, la complexité croissante des circuits, ont fait fleurir une

1. Hémisphères droit et gauche du cerveau humain, diront certains. Mais c'est une autre histoire.
2. Peu importe, à ce niveau, le support physique utilisé, graphique ou textuel. Un schéma peut être décrit par un dessin, le monde graphique, ou par une *net list*, le monde textuel. De même, un algorithme peut être décrit par un diagramme (*flowchart*), le monde graphique, ou par un langage procédural, le monde textuel. Chaque représentation a ses avantages et ses inconvénients.
3. La deuxième tentative est, en outre, fausse. Elle ne prend pas en compte le parallélisme inhérent à tout système électronique.
4. Toute ressemblance avec ADA serait purement fortuite.

multitude de langages plus ou moins compatibles entre eux, plus ou moins cohérents entre synthèse (fabrication) et simulation (modélisation). Le programme VHSIC (*very high speed integrated circuits*), impulsé par le département de la défense des États-Unis dans les années 70-80, a donné naissance à un langage : VHSIC-HDL, ou VHDL.

Trois normes successives (IEEE[1]-1076-1987, 1993 et 2008) en ont stabilisé la définition, complétées par des retouches et quelques extensions. La norme la plus majoritairement supportée par les outils reste aujourd'hui celle de 1993. Nous verrons les principales modifications apportées par la dernière mouture du langage (2008) à la fin de cet ouvrage. La prochaine évolution majeure consistera à intégrer les outils nécessaires à l'élaboration de bancs de vérification abstraits, outils déjà supportés par le principal concurrent de VHDL (voir *Le langage SystemVerilog* des mêmes auteurs).

> *« Apprendre à communiquer en une langue nouvelle prend toujours l'allure d'un défi. Au début, il semble que tout est nouveau, et doit être assimilé avant de caresser l'espoir d'utiliser le langage. Puis, après une première expérience, il apparaît des points de ressemblance avec notre propre langage. Au bout d'un moment, nous réalisons que les deux langages ont de nombreuses racines communes. »*

Cette première phrase de *ADA une introduction*, par H. LEDGARD, résume on ne peut mieux la difficulté. Par où commencer ?

Nous commencerons par les principes, les idées directrices, avant de nous jeter à l'eau dans la partie suivante.

1 LE LANGAGE VHDL DANS LE FLOT DE CONCEPTION

Les différentes étapes de la conception d'un circuit intégré sont illustrées par le diagramme de la figure A.1.

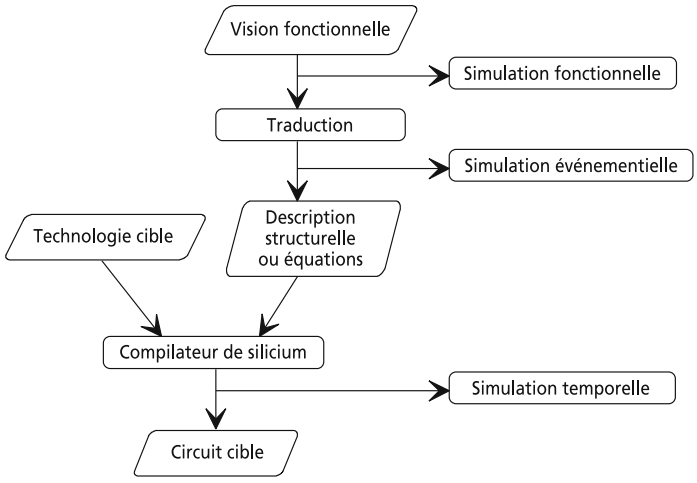

Figure A.1 Étapes de conception d'un circuit.

1. *Institute of Electrical and Electronics Engineers*.

L'ensemble du processus qui permet de passer de la vision fonctionnelle au circuit constitue la synthèse.

➤ Un langage commun

Le principe majeur de VHDL est de fournir un langage commun, depuis la description au niveau système jusqu'au circuit. À tous les niveaux de description le langage est le même. Le découpage en étapes, illustré par la figure A.1, n'est pas toujours aussi schématique, il peut y en avoir plus comme il peut en manquer.

➤ Tout n'est pas synthétisable

Créer un langage, qui permette à la fois de valider une conception au niveau système, de construire les programmes de test utiles à tous les niveaux et de générer automatiquement des schémas d'implantation sur le silicium, est évidemment un propos ambitieux. VHDL est donc beaucoup plus qu'une simple traduction textuelle de la structure interne d'un circuit.

Les constructions légales ne sont pas toutes synthétisables, seul un sous-ensemble des instructions permises par le langage fait sens en synthèse. L'aboutissement – heureux cela va sans dire – d'un projet implique une vision précise de la différence entre la réalité du silicium et le monde virtuel qui sert à le concevoir. Comme tout langage de haut niveau, VHDL est modulaire. Cette modularité permet de voir un schéma complexe comme l'assemblage d'unités plus petites, elle contribue également à séparer le réel du virtuel.

Sans rentrer ici dans des exemples techniques – nous les aborderons plus avant – indiquons simplement qu'un modèle VHDL peut détecter des incohérences dans les signaux qui lui sont transmis, des fautes de *timing*, par exemple. Face à une telle situation le modèle peut interagir avec le simulateur, modifier les conditions du test, avertir l'utilisateur par des messages ou consigner des résultats dans un fichier. De tels modules de programme ne sont évidemment pas synthétisables. Parfois la distance entre synthèse et modélisation est plus ténue : générer un retard de 13 nanosecondes, par exemple, n'est pas, dans l'absolu impensable. Mais comment créer ce retard dans un circuit cadencé par une horloge à 100 MHz ?

➤ Simulation fonctionnelle

Tester un circuit revient à lui appliquer des stimuli, des vecteurs de test, et à analyser sa réponse. À un deuxième niveau, le test inclut l'environnement dans lequel le circuit est immergé, la carte sur laquelle il est implanté. Les conditions expérimentales reproduiront, dans un premier temps, les conditions de fonctionnement normales. Dans un deuxième temps elles chercheront à cerner les limites, une fréquence maximum, par exemple.

Tester un module logiciel revient à écrire un programme qui lui transmet des données, reçoit les résultats et les analyse. Dans une première étape les données transmises sont celles attendues, elles servent à identifier d'éventuelles erreurs

grossières. Dans une deuxième étape, les données sortent de l'ordinaire prévu, provoquent une situation de faute. Les exemples informatiques abondent, en voici un : quand on réalise un programme de tracé de courbes ou une calculatrice graphique, demander le tracé de $y = x^2$ est un test de la première catégorie, celui de $y = log\ [sin\ (1/x^2)\ ]$ de la seconde [1].

VHDL est un langage informatique qui décrit des circuits. Les programmes de test simulent l'environnement d'un montage, ils bénéficient de la souplesse de l'informatique pour recréer virtuellement toutes les situations expérimentales possibles et imaginables, même celles qui seraient difficiles à réaliser en pratique.

Le test est une opération psychologique difficile à mener : nous avons une fâcheuse tendance à vouloir nous convaincre nous-mêmes de la justesse de notre réalisation, à construire des tests qui nous conforteront dans cette idée. Or le but du test est de chercher la faute, même, et surtout, la faute maligne, celle qui ne se révèle pas au premier abord [2].

➤ Du langage au circuit : la synthèse

Le synthétiseur interprète un programme et en déduit une structure de schéma. Dans une deuxième étape, il cherche à reproduire ce schéma dans une technologie, au moindre coût (surface de silicium), avec l'efficacité la plus grande (vitesse maximum).

Considérons un exemple simple : on souhaite réaliser la somme de deux nombres entiers, et stocker le résultat dans un registre synchrone, qui dispose d'une remise à zéro synchrone.

La partie active du code VHDL, traduisant ces opérations élémentaires, est indiquée ci-dessous :

```
-- ceci est un commentaire,
-- il va jusqu'à la fin de la ligne.
somme : PROCESS
BEGIN
   WAIT UNTIL RISING_EDGE(hor); -- attente du front montant
   IF raz = '1' THEN            -- commande de remise à zéro
      reg <= 0;
   ELSE                         -- addition
      reg <= e1 + e2;
   END IF;
END PROCESS somme;
```

De ce programme le synthétiseur infère la présence d'un registre synchrone (des bascules D) pour stocker le résultat *reg*, grâce à l'instruction d'attente d'un front d'horloge. Il infère un bloc de calcul qui renvoie la somme des deux opérandes et l'affecte au contenu du registre.

---

1. Dans une première étape on peut supprimer le logarithme. L'intervalle de variation intéressant est évidemment centré autour de 0, comme {– 1..1}.
2. Une situation de débordement dans un calcul numérique, par exemple.

D'où la structure de la figure A.2.

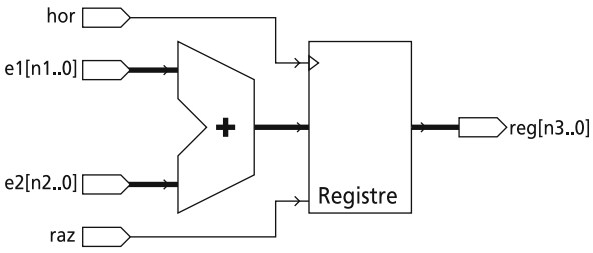

Figure A.2 Un additionneur.

Dans cette architecture, de nombreux points restent à préciser : quels sont les types des opérandes et du résultat ? Quelles sont leurs tailles, en nombre de chiffres binaires ? Comment réaliser l'addition compte tenu du circuit cible ?

Les réponses aux premières de ces questions se trouvent dans des zones déclaratives :

```
PORT (hor, raz : IN STD_LOGIC;                -- signaux binaires
      e1, e2   : IN INTEGER RANGE 0 TO 15;    -- entiers 4 bits
      reg      : OUT INTEGER RANGE 0 TO 31); -- entiers 5 bits
```

Le schéma comportera donc cinq bascules. À propos de ces premières instructions VHDL, nous découvrons l'importance des déclarations, l'omission de la restriction du domaine de variation des nombres conduirait, par défaut, à générer systématiquement un additionneur 32 bits.

La réponse à la dernière des questions précédentes ne se trouve pas dans le programme source. Elle fait partie intégrante de l'outil de synthèse, et dépend du circuit cible. Si ce dernier ne comporte comme opérateurs combinatoires élémentaires que des portes logiques, le synthétiseur devra être capable de traduire l'addition binaire en équations logiques, chiffre binaire par chiffre binaire. Si le circuit cible comporte des blocs arithmétiques, le synthétiseur tentera de les utiliser. Souvent la réalité se situe à mi-chemin entre logiciel et matériel : le circuit ne comporte que des opérateurs logiques, mais le compilateur de silicium possède des bibliothèques fournies de fonctions standard précompilées et optimisées. La traduction[1] du code source revient alors à reconnaître lesquelles de ces fonctions peuvent être utiles.

Le programmeur attendrait, d'un outil de CAO idéal, que toutes les réponses non inscrites dans le programme source soient fournies automatiquement, au mieux de ses intérêts, sans qu'il ait à s'en préoccuper. Ne nous leurrons pas, une bonne connaissance des circuits utilisés reste nécessaire, quelle que soit la qualité des logiciels employés.

1. Avec un certain humour, les auteurs d'un tel traducteur, qui s'adapte aux familles de circuits cibles, l'avaient baptisé *metamor* (utilisé il y a une dizaine d'années par SYNARIO, de DATA IO).

➤ Du circuit au langage : la modélisation

La réalité du monde physique introduit des différences de comportement entre le modèle fonctionnel et sa réalisation dans un circuit. Les principales de ces différences proviennent des temps de calcul des opérateurs. Une fois le programme compilé, traduit en un schéma de portes et de bascules interconnectées, il est intéressant de prévoir, compte tenu de la technologie employée, les limites de son fonctionnement. Cette opération de modélisation post-synthèse est connue sous le terme de *rétro-annotation*.

Un modèle rétro-annoté réalise – c'est du moins souhaitable – la même fonction que le modèle synthétisable dont il est issu, mais il apporte des renseignements complémentaires concernant les limites de fonctionnement que l'on peut attendre du circuit. On peut remarquer que le modèle rétro-annoté n'est, lui, pas synthétisable ! Il représente une réalité qu'il est incapable de créer[1].

2 QUELS QUE SOIENT L'OUTIL ET LE CIRCUIT : LA PORTABILITÉ

La *portabilité*[2] est le maître mot. Dans le monde des circuits ce mot a un double sens : portabilité vis-à-vis des circuits et portabilité vis-à-vis des logiciels de conception.

➤ Indépendance vis-à-vis du circuit cible

Le même langage peut servir à programmer un circuit de quelques centaines de portes équivalentes, et à concevoir un circuit intégré spécifique qui en contient plusieurs millions. Les compilateurs utilisés n'ont, bien sûr, pas grand-chose en commun ; le nombre de personnes impliquées dans un projet non plus. Mais le langage est le même[3]. Cette indépendance vis-à-vis du circuit cible a des applications industrielles : elle permet, par exemple, de commencer une réalisation avec des circuits programmables, dont le cycle de réalisation est simple et rapide, pour passer à un circuit spécifique dans une deuxième étape. La première version sert de prototype à la seconde.

---

1. Le rêve de tout électronicien : diminuer les temps de retard du modèle rétro-annoté pour augmenter la fréquence de travail du circuit réel.
2. Nous pouvons compléter, à la lumière de ce paragraphe, une remarque précédente que nous avions faite à propos des représentations graphique et textuelle. La représentation graphique a des aspects séduisants de lisibilité, c'est évident. Elle pose cependant un problème non trivial de portabilité (rares sont les outils graphiques universels) et de rigueur. Une représentation textuelle est moins intuitive, mais ne pose pas de problème de portabilité et permet une formalisation plus aisée, qui lui assure un caractère non ambigu. De nombreux systèmes proposent les deux modes ; le mode graphique reste local, le passage d'un système à un autre se fait par l'intermédiaire du langage textuel.
3. Un langage n'est qu'un outil. Mais les langages de haut niveau ne sont pas neutres au niveau des méthodes de travail qu'ils induisent. Dans un contexte d'apprentissage, il est intéressant de disposer d'outils de développement petits et bon marché, qui permettent de développer des applications suffisamment simples pour être menées à terme par une seule personne en quelques heures.

➤ Indépendance vis-à-vis du compilateur

Le deuxième aspect de la portabilité concerne les outils de développement. Le même programme est accepté par des logiciels de CAO très divers.

Historiquement, cette portabilité était excellente en simulation, c'est la moindre des choses, mais présentait des limitations sévères en synthèse. Pour un synthétiseur, certaines constructions sont un peu « magiques » : elles orientent le compilateur vers des opérations spécifiques, grâce à un sens caché qu'il est le seul à connaître. Ces particularismes locaux sont souvent liés à des définitions de types de données[1], qui figurent dans des librairies spécifiques d'un outil. La règle en VHDL est de fournir les codes sources des librairies, règle respectée par tous les fournisseurs ; mais cette (bonne) habitude, si elle assure une portabilité sans faille des outils de synthèse aux outils de simulation, ne peut évidemment pas assurer le portage des aspects magiques cachés. L'exemple caractéristique de données à caractère caché est la modélisation des bus. Le langage n'avait pas, volonté d'ouverture ou omission, prévu les logiques *trois états* dans ses types de base. En simulation, rien n'est plus simple que de construire de nouveaux types : tout est prévu. En synthèse, l'apparition d'un signal qui peut prendre deux états logiques et un état haute impédance, provoque la génération d'opérateurs très particuliers, qui échappent au monde des équations booléennes. Il y a un sens caché attaché à un type trois états. Tous les fournisseurs de logiciel ont défini leurs propres librairies synthétisables, toutes sensiblement équivalentes… mais strictement incompatibles.

Un effort de standardisation a été entrepris, entre 1993 et 1995, pour assurer une portabilité des programmes tant en synthèse qu'en simulation. Entre 1995 et 1997, tous les outils de conception de circuits programmables, par exemple, ont rejoint les nouveaux standards (IEEE-1076.3).

Le même type d'évolution a eu lieu dans le domaine de la rétro-annotation. Des modèles maison portables, puisque fournis sous forme de sources, au prix d'échanges de librairies volumineuses, on est passé, avec le standard IEEE-1076.4 (ou VITAL-95), à une compatibilité entre les bibliothèques des différents fondeurs. Cette compatibilité assure, de plus, une passerelle avec le monde VERILOG, riche en compétences et en expérience dans ce domaine.

3 DEUX VISIONS COMPLÉMENTAIRES : HIÉRARCHIQUE ET FONCTIONNELLE

Représentation spatiale et analyse séquentielle : VHDL favorise une approche hiérarchique pour la description d'un circuit complexe. Du niveau le plus élémentaire (les portes) au niveau le plus élevé (le système) le programmeur peut, définir à sa guise des sous-ensembles en relations, décrire un sous-ensemble par une représentation

1. Les seuls types que nous avons rencontrés, pour l'instant, sont les types entier et bit ; il y en a d'autres.

comportementale (analytique) ou par une nouvelle hiérarchie locale. Les deux approches peuvent coexister dans un même programme source, et l'utilisateur peut spécifier à tout moment quelle configuration il choisit d'analyser.

➤ Construction hiérarchique

Reprenons l'exemple de notre additionneur. Un lecteur pointilleux pourra nous objecter que nous en avons fait une représentation graphique structurelle, et que nous l'avons décrit par un programme comportemental. C'est on ne peut plus exact. Il était parfaitement possible de créer un programme VHDL qui reproduise la structure ; compte tenu de la simplicité du problème, nous avons considéré que l'approche analytique était plus lisible.

La vision hiérarchique du même additionneur consiste à traiter les blocs, l'additionneur et le registre, comme assemblage d'objets plus élémentaires. La figure A.3 en donne une image.

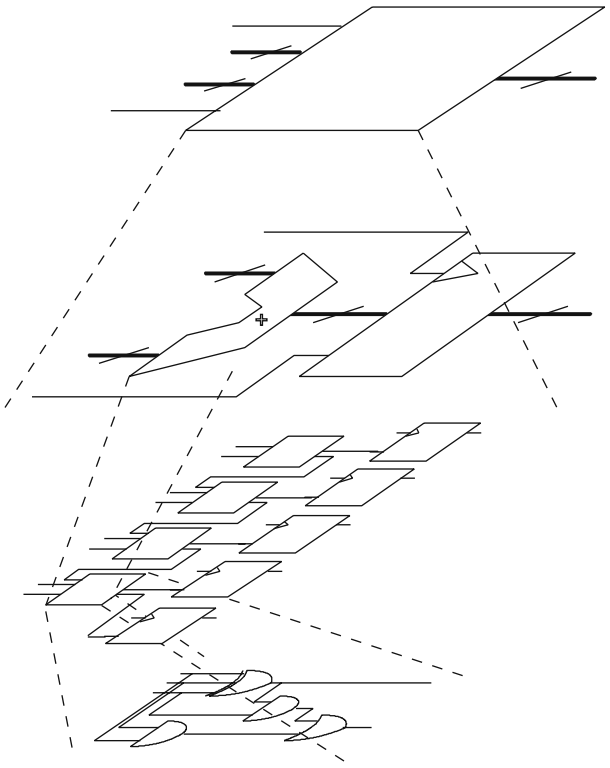

Figure A.3 Vision hiérarchique de l'additionneur.

La construction hiérarchique n'est, bien sûr, pas propre à VHDL. Tous les systèmes de CAO dignes de ce nom la comprennent.

Nous ne donnerons pas ici la façon de créer une hiérarchie dans le langage, chaque chose en son temps. Contentons-nous d'insister sur l'importance du concept et sur le fait que les instructions correspondantes sont extrêmement puissantes ; beaucoup plus qu'il n'est possible de le figurer avec des images. VHDL inclut toute une algorithmique spatiale, qui permet de créer des structures paramétrables et évolutives.

➤ Description fonctionnelle

Complémentaire de la précédente, la vision fonctionnelle apporte la puissance des langages de programmation. Au premier abord elle est assez classique, le programmeur habitué aux langages procéduraux se retrouve en terrain connu.

Le point important, qui rend VHDL très différent de langages comme C ou PASCAL, malgré les ressemblances syntaxiques fortes avec ce dernier, est que tout algorithme est la description interne d'un bloc situé quelque part dans la hiérarchie du schéma complet.

La vision structurelle est concurrente, la vision algorithmique est séquentielle, au sens informatique du terme. Un programme VHDL doit être compris comme l'assemblage, en parallèle, de tâches indépendantes qui s'exécutent concurremment. Les signaux sont, en réalité, les véhicules des informations électriques dans le schéma ; vus sous l'angle quelque peu informatique précédent, ils jouent le rôle de canaux de communications interprocessus.

Avant de rentrer dans le vif du sujet, autorisons-nous un conseil : n'oubliez jamais que vous décrivez un circuit. Les langages concurrents sont parfois déconcertants pour les novices, nous avons ici la chance de traduire dans un tel langage le monde des circuits ; la vision physique des choses permet de franchir sans difficulté bien des écueils.

PARTIE **B**

CONCEPTION ET TEST DES CIRCUITS NUMÉRIQUES

VHDL est un langage de haut niveau qui permet de décrire, d'un point de vue purement fonctionnel, le comportement de systèmes numériques. Il permet également de synthétiser des circuits numériques, c'est-à-dire de traduire en schéma logique la description de ces mêmes systèmes. Certaines constructions du langage sont synthétisables et sont a fortiori simulables. D'autres sont de purs outils de modélisation, utilisables pour simuler le fonctionnement d'un système mais pas pour en faire une synthèse. C'est ici la source d'erreurs principale du débutant qui, se laissant bercer par des constructions algorithmiques parfaitement simulables, ne voit pas qu'elles ne correspondent à aucun circuit réalisable. Bien que cette frontière entre ce qui est synthétisable et ce qui ne l'est pas soit floue et se déplace à mesure que les performances d'analyse des outils de synthèse s'améliorent, la description des circuits numériques obéit à des schémas d'écriture récurrents. Ce sont quelques unes de ces structures de code synthétisables que le lecteur est invité à explorer à travers cette deuxième partie.

Dans le premier chapitre, nous explorerons les principales constructions synthétisables du langage à travers la réalisation concrète de modules classiques : décodeurs, multiplexeurs, registres, compteurs, automates, etc. Puis nous aborderons les problèmes du test unitaire et de la simulation post-synthèse. Enfin, un chapitre sera entièrement consacré aux pièges classiques qu'est susceptible de rencontrer tout concepteur débutant de circuits numériques.

Chapitre **1**

Décrire le circuit

La description de circuits numériques en VHDL s'oriente classiquement selon trois axes complémentaires. Le premier, « comportemental » (*behavioral*), décrit un algorithme séquentiel, c'est-à-dire un enchaînement d'actions effectuées sur les signaux de sortie conséquemment à des observations faites sur les signaux d'entrées. La nature séquentielle de l'algorithme n'a pas de rapport avec la nature séquentielle ou combinatoire du composant décrit. Le second, « flot de données » (*data flow*), décrit le même algorithme par des relations causales entre signaux d'entrée et de sortie. Enfin le troisième, « structurel » (*structural*), correspond à l'élaboration d'un schéma où les signaux d'entrées/sorties de modules indépendants s'interconnectent pour former une architecture.

On favorisera ce dernier axe lorsqu'il s'agira de décrire un circuit complexe ; on préféra les deux premiers pour les petits sous-ensembles de ce même circuit.

Après un premier aperçu de ces trois approches, nous partirons de l'algorithme (« flot de données » et « comportemental ») pour aller vers le schéma (« structurel »). Nous verrons d'abord comment décrire des opérateurs combinatoires. Puis nous traiterons des opérateurs séquentiels et des automates. Ensuite nous aborderons la description structurelle.

1.1 LA BOÎTE NOIRE ET SON CONTENU

Un opérateur élémentaire, un circuit intégré, une carte électronique ou un système complet sont intégralement définis par des signaux d'entrées et de sorties et par la

fonction qu'ils réalisent de façon interne. Ce double aspect – signaux de communication et fonction – se retrouve à tous les niveaux de la hiérarchie d'un projet. L'élément essentiel de toute description en VHDL, nommé *design entity* dans le langage, est formé par le couple *ENTITY ARCHITECTURE*. Le premier élément de ce couple décrit l'apparence externe d'une unité de conception, le second décrit son fonctionnement interne.

1.1.1 La boîte noire : une entité

La déclaration d'entité décrit l'interface entre le monde extérieur et une unité de conception, c'est-à-dire principalement ses signaux d'entrées et de sorties[1]. Une même entité peut être associée à plusieurs architectures différentes. Elle décrit alors une classe d'unités de conception qui présentent au monde extérieur le même aspect, avec des fonctionnements internes éventuellement différents. Le même circuit peut, par exemple, être décrit de façon purement fonctionnelle lors de la conception, et de façon structurelle lors de la synthèse pour contrôler le bon respect des règles temporelles de ses constituants physiques.

a) Déclaration d'entité

La déclaration d'entité d'un générateur de parité 7 bits s'écrit comme suit[2] :

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY parityGenerator IS
   PORT (data   : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
         even   : IN STD_LOGIC;
         parity : OUT STD_LOGIC);
END ENTITY parityGenerator;
```

Le signal *data* constitue les données d'entrée. L'état du signal de sortie *parity* dépendra de la polarité choisie par l'intermédiaire du signal *even*.

➤ Spécification de contexte

Les deux premières lignes constituent une spécification de contexte. Elles indiquent que, dans ce module, il est fait usage de la bibliothèque *IEEE*, dont le contenu est traité en détail au chapitre 5 page 135. Ces deux lignes seront systématiquement présentes devant toutes les déclarations d'entités.

1. La déclaration d'entité décrit aussi les éventuels paramètres génériques de l'unité de conception (voir pages 57 et 178).
2. Les mots du langage peuvent être écrits indifféremment en majuscules ou en minuscules. Nous avons choisi les majuscules, malgré leur aspect inesthétique, pour les distinguer visuellement des mots définis par l'auteur d'une description.

➤ Déclaration d'entité et spécification d'interface

Ensuite se trouve la déclaration d'entité proprement dite. Le nom de l'unité de conception figure dans l'en-tête de la déclaration (*parityGenerator*) et se trouve éventuellement répété à la fin.

Dans le corps de la déclaration, figure la description de l'interface de l'unité de conception : sa déclaration de port d'accès. Cette déclaration de port nous apprend que even est un signal d'entrée et que *parity* est un signal de sortie. Tout deux sont de type *STD_LOGIC*, qui est le type standard le plus courant pour désigner une information logique[1]. Le signal *data* –de type *STD_LOGIC_VECTOR*– est également déclaré en entrée, mais représente un vecteur de sept signaux logiques numérotés de 6 à 0, dans cet ordre, autrement dit un bus.

Nous pouvons d'ores et déjà dessiner la boite noire de notre première unité de conception.

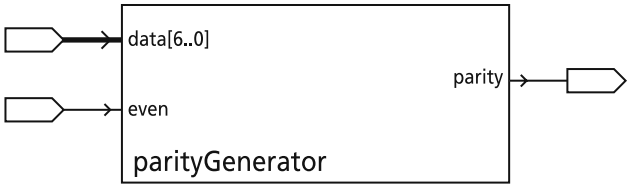

Figure 1.1 La boîte noire du générateur de parité.

b) Déclaration de port et modes d'accès

La déclaration de port d'accès énumère l'ensemble des signaux d'interface – les ports – qui permettent la communication entre le monde extérieur et l'unité de conception décrite. À chaque signal de communication sont attachés un nom, un mode d'accès et un type.

Un tampon bidirectionnel d'interfaçage avec un bus externe illustre parfaitement la variété des modes d'accès classiquement employés.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY biDirBuffer IS
   PORT (outEn  : IN STD_LOGIC;
         datIn  : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
         datOut : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
         datIO  : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END ENTITY biDirBuffer;
```

1. Nous verrons (section C.5.2 page 138) que le type *STD_LOGIC* modélise bien plus qu'une simple information logique. En toute rigueur, c'est le type *STD_ULOGIC* qui devrait être employé ici, mais l'usage fait que ce n'est pas le cas.