Utilisation de macro blocs en synthèse VHDL

Marie-Claude Cebelieu

▶ To cite this version:

Marie-Claude Cebelieu. Utilisation de macro blocs en synthèse VHDL. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1995. Français. NNT : . tel-00346055

HAL Id: tel-00346055

https://theses.hal.science/tel-00346055

Submitted on 11 Dec 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

Présentée par

Marie-Claude CEBELIEU

Pour obtenir le grade de
Docteur de l'Institut National Polytechnique de Grenoble
(arrêté ministériel du 30 Mars 1992)
(Spécialité **Microélectronique**)

Utilisation de macro blocs en synthèse VHDL

Date de soutenance : 20 Décembre 1995

Composition du Jury :

Madame	Professeur Gabrièle SAUCIER	
Messieurs	Professeur Guy MAZARE	Président
	Professeur Peter MARWEDEL	Rapporteur
	Vincent OLIVE	Rapporteur
	Michel CRASTES	

Thèse préparée au laboratoire de Conception de Systèmes Intégrés à l'INPG.

Remerciements

-- ◊ - ◊◊◊ - ◊ --

Je tiens tout d'abord à remercier Madame Gabrièle Saucier, Professeur à l'ENSIMAG et directrice du Laboratoire de Conception de Systèmes Intégrés, pour m'avoir accueillie dans son laboratoire et pour avoir suivi mon travail durant ces années de recherche.

Je remercie Monsieur Guy Mazaré, Professeur et directeur de l'ENSIMAG, de me faire l'honneur de présider ce jury.

Je remercie Monsieur Peter Marwedel, Professeur à l'Université de Dortmund, pour avoir accepté d'être rapporteur de mon travail et pour ses remarques pertinentes. Je le remercie d'autant plus pour avoir eu à affronter la barrière de la langue Française lors de la lecture de ce rapport de thèse.

Je remercie Monsieur Vincent Olive, Ingénieur de Recherches au CNET, pour l'examen et la critique qu'il a apporté à ce travail en tant que rapporteur.

Je remercie Michel Crastes de Paulet, pour avoir accepter d'être mon directeur de thèse, pour le temps précieux et le soin qu'il a accordé à la lecture de ce rapport, et pour ses encouragements et son soutien.

Je tiens également à remercier tous les membres du laboratoire CSI pour leur collaboration et tous ceux qui ont contribué de près ou de loin à l'élaboration de ce travail. Je remercie tout particulièrement Alexandre, Hamid, Laurent, Jérôme, Pascale, Martine, Xavier, Jean-Pierre, Khalid, Anne, Régis, Raphaël, Kacem, Pierre, Patrick, Antoine, Olivier, Lionel, Thomas, Hervé, Arnaud, et tant d'autres, pour leur amitié, leurs conseils et leur soutien.

A mon père.

A mon mari.

Résumé

-- ◊ - ◊◊◊ - ◊ --

Le contexte général de cette thèse se situe dans le domaine de la synthèse RTL (Register Transfer Level). Une spécification initiale en termes de transferts de registres décrite dans un langage de haut niveau (VHDL, Verilog) définit l'ordre des opérations. A partir de cette spécification, le système de synthèse RTL génère une description structurelle fonctionnellement équivalente interconnectant des portes de base et des macro blocs de la cible technologique.

Le langage de description considéré ici est le langage VHDL standardisé par le groupe IEEE en 1987. Ce choix est justifié par une étude comparative entre différents langages. Les principales caractéristiques du langage VHDL ainsi que les améliorations apportées par la nouvelle norme de 1992 sont évoquées.

Dans une seconde partie, les limitations du langage VHDL pour son utilisation en synthèse et le flot de conception à partir d'une spécification RTL sont présentés. Plusieurs modèles VHDL d'éléments simples et de macro blocs sont décrits pour la synthèse. Le flot général de conception utilisant ces macro blocs est analysé et détaillé pour deux cas pratiques : l'utilisation des générateurs XBLOX de Xilinx et ACTgen d'Actel dans le logiciel de synthèse ASYL+.

La dernière partie s'attache plus précisément à la modélisation d'éléments de bibliothèques en vue de leur utilisation en synthèse. Un format de bibliothèque, permettant de décrire tout aussi bien des portes simples que des macro blocs, est défini. Le nouveau format de bibliothèque standard VITAL est analysé ainsi que ses perspectives d'utilisation en simulation et en synthèse. La norme LPM qui définit un ensemble d'éléments standards indépendants de la technologie est également présentée. Cette dernière partie a conduit à la définition d'un nouveau flot de synthèse unifié utilisant les macro blocs et à la mise en place de plusieurs optimisations basées sur la notion de dérivation.

Mots clés : Synthèse RTL, Modélisation VHDL, Macro blocs, Bibliothèques, VITAL, LPM, Flots de conception.

Abstract

-- ◊ - ◊◊◊ - ◊ --

The general context of this thesis belongs to the field of RTL (Register Transfer Level) synthesis. An initial specification in terms of register transfers described using a high level language (VHDL, Verilog) defines the operation ordering. From this specification, the RTL synthesis system generates a structural description functionally equivalent interconnecting basic gates and macro blocks of the target technology.

The description language considered here is the VHDL language standardised by the IEEE group in 1987. This choice is justified by a comparative study between different languages. The main characteristics of the VHDL language as well as the improvements resulting from the 1992 normalisation are discussed.

In a second part, the limitations of the VHDL language for synthesis and the RTL VHDL design flow are presented. Several VHDL synthesis models of simple elements and macro blocks are described. The general design flow using these macro blocks is analysed and detailed for two practical cases : the handling of Xilinx XBLOX and Actel ACTgen generators in the ASYL+ synthesis software.

The last part focuses on the modelling of library elements for synthesis. A library format allowing the description of simple gates as well as macro blocks is defined. The new standard VITAL library format and the prospects of its use in simulation and synthesis are analysed. The LPM standard defining a set of technology independent elements is also presented. This last part has led to the definition of a new unified synthesis flow using macro blocks and the proposal for optimisation techniques based on the notion of derivation.

Key words : RTL synthesis, VHDL modelling, Macro blocks, Libraries, VITAL, LPM, Design flows.

Table des matières

-- ◊ - ◊◊◊ - ◊ --

Résumé..**1**

Abstract..**3**

Table des matières..**5**

Introduction Générale..**13**

Chapitre I : Le langage VHDL..**17**

1. Généralités..**17**

1.1. Nécessité.. 17

1.2. Un peu d'historique.. 18

2. Les principales caractéristiques du langage VHDL..**20**

2.1. Un langage s'appliquant à plusieurs niveaux de descriptions.. 20

2.2. Simulation conduite par événements.. 20

2.3. Modularité.. 20

2.4. Portabilité.. 22

2.5. Entité, architecture et composant.. 23

2.5.1. Entité.. 23

2.5.2. Architecture.. 23

2.5.3. Composant et configuration 24
2.6. Objets 27
2.6.1. Les constantes 27
2.6.2. Les variables 27
2.6.3. Les signaux 28
2.6.4. Les fichiers 29
2.7. Types 30
2.7.1. Les types scalaires 31
2.7.2. Les types composés 31
2.7.3. Objets et types 32
2.8. Instructions concurrentes 33
2.8.1. Définition 33
2.8.2. Simulation 34
2.8.3. L'instruction concurrente d'affectation de signal 35
2.8.4. Fonction de résolution 35
2.8.5. Processus 36
2.8.6. Les autres instructions concurrentes 37
2.9. Instructions séquentielles 38
2.10. Les paquetages 39
2.11. Environnement préalablement défini du langage 39
2.12. Les trois styles de descriptions en VHDL 40
3. VHDL'92 : les différences par rapport à la norme de 87 43
3.1. Les grandes lignes qui ont guidé le processus de re-standardisation 43
3.2. Les nouveaux mécanismes de simulation 45
3.2.1. Activation lors du dernier delta délai 45
3.2.2. Les variables partagées (ou variables globales) 46
3.3. Les nouveaux mécanismes de structuration 47
3.3.1. Instanciation directe 47
3.3.2. Association incrémentale 48
3.3.3. Notion de groupe 49
3.4. Les nouveaux mécanismes d'interface 50
3.4.1. Architectures et sous-programmes étrangers 50
3.4.2. Lecture et écriture de fichiers 50
3.4.3. Fonctions impures 50
3.5. Environnement préalablement défini : les nouveaux opérateurs, fonctions et attributs 51
3.5.1. Opérateurs de décalage et de rotation et opérateur "xnor" 51

3.5.2. Attributs préalablement définis "'DRIVING_VALUE" et "'DRIVING"... 51
3.5.3. Attribut préalablement défini "'ASCENDING" 51
3.5.4. Attributs préalablement définis "'BEHAVIOR" et "'STRUCTURE" 52
3.5.5. Attributs préalablement définis "'IMAGE" et "'VALUE" 52
3.5.6. Attributs préalablement définis "'SIMPLE_NAME" , "'PATH_NAME" et "'INSTANCE_NAME" 52
3.6. Améliorations mineures 52
4. VHDL par rapport aux autres langages 54
4.1. Historique 54
4.2. Caractéristiques générales 55
4.3. Différences au niveau de la modélisation 55
4.3.1. Niveaux de descriptions 55
4.3.2. Unités de conception 56
4.3.3. Domaines concurrent et séquentiel 56
4.3.4. Objets et types 56
4.3.5. Environnement préalablement défini 57
4.3.6. Descriptions structurelles 57
4.4. Conclusion 57
5. Conclusion 58

Chapitre II : Modélisations VHDL pour la synthèse 59

1. Généralités 59
1.1. Définition 59
1.2. La synthèse automatique de circuits : dans quel but ? 59
1.2. Les différents niveaux de synthèse 61
1.2.1. Synthèse de haut niveau 62
1.2.2. Synthèse au niveau transferts de registres (ou synthèse RTL) 62
1.2.3. Synthèse de contrôleurs 63
1.2.4. Synthèse d'équations Booléennes 63
1.2.5. Synthèse topologique 64
1.3. Les limitations du langage VHDL pour la synthèse 64
1.4. VHDL, un langage pour la synthèse : oui ou non ? 65
1.5. Méthodologie de conception d'un circuit à partir d'une description VHDL....... 67
1.5.1. Les étapes du cycle de conception 68
1.5.2. La description du circuit pour la synthèse 69
1.5.3. Le processus de synthèse 69
1.5.4. La validation finale 71

1.6. Contrôle du processus de synthèse 72
1.6.1. Contrôle du processus de synthèse : de quoi s'agit-il ? 72
1.6.2. Contraintes sur les caractéristiques physiques du circuit 72
1.6.3. Formats d' interfaces 73
1.7. Environnement VHDL pour la synthèse 74
1.7.1. Paquetage "STD_LOGIC_1164" 74
1.7.2. Groupe IEEE travaillant sur la synthèse 75
2. Modélisations VHDL d'éléments de bas niveau 76
2.1. Définition 76
2.2. Modélisations VHDL de blocs combinatoires 76
2.2.1. Définition 76
2.2.2. Modélisation de portes logiques de base 76
2.2.3. Modélisations de portes logiques complexes par des équations Booléennes 77
2.2.4. Modélisations de portes complexes par des tables de vérité 78
2.2.5. Modélisation de portes trois-états 80
2.3. Modélisations VHDL de blocs séquentiels 81
2.3.1. Définitions 81
2.3.2. Modélisations de verrous 82
2.3.3. Modélisations de bascules 84
3. Modélisations VHDL de macro blocs 86
3.1. Définitions 86
3.2. Modélisations VHDL de multiplexeurs 87
3.3. Modélisations VHDL de comparateurs 89
3.4. Modélisations VHDL d'opérateurs arithmétiques 90
3.5. Modélisations VHDL de décaleurs 91
3.6. Modélisations VHDL de registres 93
3.7. Modélisations VHDL de compteurs 94
3.8. Modélisations VHDL de mémoires RAM 96
4. Synthèse utilisant des macro blocs 97
4.1. Introduction 97
4.2. Flot de synthèse 98
4.3. Génération de macro blocs 101
4.3.1. Additionneurs 101
4.3.2. Soustracteurs 102
4.3.3. Incrémenteurs et décrémenteurs 102
4.3.4. Comparateurs 103
4.3.5. Multiplieurs 103

4.4. Appel à des macro blocs de bibliothèques 103
4.4.1. Appel par inférence 103
4.4.2. Appel par fonctions ou procédures 106
4.4.3. Appel structurel par instanciation de composants 107
4.5. Cas particulier des bibliothèques de générateurs 108
4.5.1. Notion et spécification des paramètres d'un générateur 108
4.5.2. Exemple d'interface avec la bibliothèque de générateurs XBLOX de Xilinx111
4.5.3. Exemple d'interface avec la bibliothèque de générateurs ACTgen d'Actel. 112
4.6. Optimisations 114
4.6.1. Optimisations apportant un gain en surface 114
4.6.2. Optimisation apportant un gain en temps 117
4.6.3. Optimisation apportant un gain en surface et en temps 119
4.6.4. Autres optimisations possibles non intégrées dans ASYL+ 119
5. Conclusion 121

Chapitre III : Modélisation d'éléments de bibliothèques 123

1. Introduction 123
2. Modélisation des éléments de bibliothèques dans ASYL+ 124
2.1. Modélisation des cellules de bas niveau : format de bibliothèque de bas niveau 124
2.1.1. Les portes combinatoires 124
2.1.2. Les portes préalablement définies 125
2.1.2. Les portes inutilisables par la synthèse 127
2.2. Modélisation de macro blocs : format de bibliothèque de haut niveau 129
2.3. Limitations de ces formats 132
2.3.1. Limitations liées à l'existence de deux formats séparés 132
2.3.2. Limitations liées au format de bibliothèque de bas niveau 132
2.3.3. Limitations liées au format de bibliothèque de haut niveau 133
2.3.4. Insuffisances générales 134
2.4. Objectifs du nouveau format 134
2.5. Proposition de nouveau format 135
2.5.1. Remarques générales 135
2.5.2. Format du fichier décrivant les informations structurelles et fonctionnelles des éléments de bibliothèque 135
2.5.2.1. Les différentes méthodes de description d'un élément de bibliothèque135
2.5.2.2. Informations générales décrites par le nouveau format de bibliothèque136
2.5.2.3. Informations générales à toutes les cellules de la bibliothèque 137
2.5.2.4. Informations liées aux cellules de la bibliothèque 138

2.5.2.5. Informations liées aux ports des cellules de la bibliothèque 138
2.5.2.6. Informations liées à la fonctionnalité des cellules de la bibliothèque . 140
2.5.2.7. Exemples réels de description avec le nouveau format de bibliothèque142
2.5.3. Format du fichier décrivant les informations temporelles des éléments de bibliothèques 144
2.5.3.1. Informations générales décrites par ce fichier 144
2.5.3.2. Informations générales sur la bibliothèque 145
2.5.3.3. Informations temporelles sur les éléments de la bibliothèque 146
2.5.3.4. Informations sur les capacités des ports d'entrée et de sortie........ 146
2.5.3.5. Informations sur les temps de traversée........ 147
2.6. Conclusion........ 148
3. Les nouveaux standards : VITAL et LPM 149
3.1. Problèmes liés à la description et à l'utilisation des bibliothèques 149
3.1.1. Problèmes liés à la description des bibliothèques........ 149
3.1.2. Problèmes liés à l'utilisation des bibliothèques 150
3.2. VITAL........ 150
3.2.1. Motivations........ 150
3.2.2. Objectifs 152
3.2.3. Informations temporelles prise en compte par VITAL 153
3.2.4. Le niveau 0 de conformité........ 154
3.2.5. Le niveau 1 de conformité........ 156
3.2.6. Flot de conception VHDL utilisant VITAL 158
3.2.7. Mécanisme de rétro-annotation via SDF........ 160
3.2.8. VITAL et SDF : liens avec la synthèse........ 161
3.2.9. Limitations........ 164
3.3. LPM 165
3.3.1. Motivations........ 165
3.3.2. Objectifs 166
3.3.3. Flot général d'utilisation de LPM........ 167
3.3.4. Passage des éléments LPM à la technologie ciblée 168
3.3.5. Les règles suivies lors de la définition des éléments LPM........ 171
3.3.6. Caractéristiques générales des éléments LPM........ 172
3.3.7. Description structurelle EDIF-LPM 173
3.3.8. Autres fichiers associés à LPM 175
3.3.9. Utilisation au niveau d'ASYL+........ 177
3.3.10. Mapping LPM et optimisation par dérivation 179
4. Conclusion........ 181

Conclusion Générale **183**

Bibliographie et références bibliographiques **187**

Annexe I : Format de description en bibliothèque des éléments utilisés par la synthèse de bas niveau **195**

I.1. Grammaire du format **196**

I.2. Exemples **201**

Annexe II : Format de description en bibliothèque des éléments utilisés par la synthèse comportementale **205**

II.1. Grammaire du format **206**

II.2. Exemples **208**

Annexe III: Nouveau format de description en bibliothèque des éléments utilisés par tout type de synthèse **213**

III.1. Grammaire du format **214**

III.1.1. Grammaire du fichier décrivant les informations structurelles et fonctionnelles des éléments de bibliothèque 214

III.1.2. Liste des types des éléments, des ports et des opérations 217

III.1.3. Grammaire du fichier décrivant les informations temporelles des éléments de bibliothèque 218

II.2. Exemples **221**

III.2.1. Exemple de fichier décrivant les informations structurelles et fonctionnelles 221

III.2.1. Exemple de fichier décrivant les informations temporelles 225

Introduction Générale

-- ◊ - ◊◊◊ - ◊ --

D'énormes progrès ayant été faits au niveau de l'intégration technologique en particulier pour les technologies "sub-micronique", la complexité des circuits à réaliser n'est plus limitée aujourd'hui par leur fabrication, mais par leur conception. Le concepteur est contraint à utiliser de plus en plus les outils d'automatisation mis à sa disposition et ceci pour plusieurs raisons : la complexité du circuit devient trop élevée, le temps de conception trop long et le risque à l'erreur trop grand. L'automatisation de la réalisation des circuit intégrés est donc aujourd'hui une nécessité [Farl 88]. Le phénomène le plus récent ayant marqué l'évolution de la conception assistée par ordinateurs (CAO) est l'importance prise par les outils de synthèse automatique.

Les outils de synthèse peuvent être classés par niveau d'abstraction de la spécification du circuit à synthétiser. Selon ces niveaux, nous pouvons distinguer quatre types d'outils de synthèse : la synthèse de haut niveau, la synthèse architecturale, la synthèse logique et la synthèse topologique. La synthèse de haut niveau traite une spécification algorithmique décrivant les opérations à réaliser par le circuit ainsi que leur relation de dépendance. Le résultat de la synthèse de haut niveau est une spécification en termes de transferts de registres dans laquelle l'ordre des opérations est défini. Les opérations sont allouées à des opérateurs et les variables à des registres. C'est cette spécification qui constitue le point d'entrée de la synthèse architecturale également appelée synthèse RTL (pour "Register Transfer Level"). Le résultat obtenu après la synthèse RTL est une spécification en termes d'équations Booléennes et d'instanciations d'éléments de la cible technologique, tels que des portes de base (bascules, portes trois-états) et des macro blocs. Les équations Booléennes sont alors traitées par la

synthèse logique afin d'obtenir une spécification structurelle en termes de portes et de macro blocs de la cible technologique considérée. C'est cette dernière spécification qui est prise en compte par la synthèse topologique afin d'obtenir le dessin des masques, dans le cas des circuits intégrés à la demande (ASICs), ou les données utilisées pour programmer un boîtier, dans le cas des circuits programmables.

Le contexte général de cette thèse s'inscrit dans le domaine de la synthèse RTL. La spécification de départ d'un tel système de synthèse peut être décrite selon plusieurs langages de haut niveau, mais celui que nous considérerons dans cette thèse sera le langage VHDL.

Le premier chapitre de cette thèse sera consacré au langage VHDL. Nous évoquerons les principales caractéristiques de ce langage standardisé par le groupe IEEE en 1987. Ensuite nous décrirons les améliorations qui y ont été apportées lors du vote de la nouvelle norme en 1992. Enfin, nous tâcherons de justifier notre choix qui s'est porté sur le langage VHDL en établissant une comparaison avec d'autres langages de description de haut niveau.

Après nous être intéressés au langage lui-même, le second chapitre sera consacré à son utilisation pour la synthèse. Dans une première partie nous présenterons les limitations engendrées par l'utilisation de ce langage pour la synthèse, puis nous exposerons la méthodologie de conception à suivre pour réaliser un circuit à partir d'une description VHDL. C'est également à ce niveau que nous évoquerons le passage des contraintes spécifiées par le concepteur que devra prendre en compte le système de synthèse. La seconde et la troisième parties de ce chapitre concerneront plus particulièrement la modélisation en VHDL. Nous donnerons d'abord des modèles d'éléments combinatoires et séquentiels simples, puis des modèles d'éléments plus complexes tels que des comparateurs ou des compteurs. C'est dans la quatrième partie de ce chapitre que nous évoquerons un aspect nouveau de la synthèse comportementale : l'utilisation de macro blocs. Deux cas pourront être distingués selon l'existence ou la non existence d'une bibliothèque de macro blocs. Les connexions aux générateurs XBLOX de Xilinx et ACTgen d'Actel permettront d'expérimenter la méthode proposée. Un certain nombre d'optimisations mises en place dans le système de synthèse ASYL+ utilisant des macro blocs, seront également abordées.

Enfin, pour que le système de synthèse puisse utiliser des macro blocs de bibliothèques, un certain nombre d'informations les concernant doivent pouvoir être décrites et connues par l'outil de synthèse. La liste de ces informations, qui a été déduite de la connaissance d'un grand nombre de bibliothèques, sera donnée dans le troisième et dernier chapitre. Celui-ci proposera un nouveau format de description des éléments en bibliothèque permettant de décrire toutes ces données. Ce format bien particulier est celui utilisé dans le système ASYL+. Actuellement, afin

que les descriptions de bibliothèques puissent être plus facilement utilisées et certifiées, un effort de standardisation se porte sur le format VITAL ("VHDL Initiative Towards ASIC Libraries") décrit au cours du troisième chapitre. Toujours afin de faciliter les échanges entre les divers outils utilisés dans le cadre de la CAO, car les bibliothèques sont tout aussi bien indispensables à la synthèse qu'à la simulation et qu'aux outils de placement et de routage, une bibliothèque d'éléments virtuels génériques a été mise au point. Cette bibliothèque dénommée LPM pour "Library of Parameterized Modules" a été récemment standardisée et suscite un vif intérêt chez de nombreux fabriquants tels que ATT et Actel. Les éléments contenus dans cette bibliothèque seront décrits en fin de ce chapitre.

La nouveauté introduite par ce travail est donc l'utilisation des macro blocs en synthèse. Nous montrerons comment il est possible de faire appel à ces blocs à partir d'une spécification VHDL et nous verrons comment décrire ces blocs en bibliothèque.

Notons enfin que cette thèse peut être recommandée à tout débutant en conception de circuits à partir de VHDL. De nombreuses informations sur ce langage et sur son utilisation pour la synthèse y sont données. De plus, à l'image de l'incontournable livre intitulé "Circuit Synthesis with VHDL" [Airi 94], de nombreux exemples de modélisation en VHDL sont décrits, ce qui peut réconforter un concepteur débutant en VHDL.

Chapitre I

-- ◊ - ◊◊◊ - ◊ --

Le langage VHDL

VHDL sont les initiales de VHSIC Hardware Description Langage, VHSIC étant celles de Very High Speed Integrated Circuits. Autrement dit, VHDL signifie : langage de description de matériel s'appliquant aux circuits intégrés à très grande vitesse.

1. Généralités

1.1. Nécessité

L'évolution des technologies induit une complexité croissante des circuits intégrés qui ressemblent de plus en plus aux systèmes complets d'hier. Est intégré dans une puce ce qui occupait une carte entière quelques années plus tôt. La simulation logique globale du système au niveau "porte" n'est plus envisageable en terme de temps de simulation. C'est donc tout naturellement que des simulateurs fonctionnels ont commencé à être utilisés en micro-électronique.

Dans les années 70, une grande variété de langages (HILO, CAP/DSDL, MODLAN, CONLAN, CADOC pour n'en citer que quelques uns [Cras 85]) étaient utilisés, et par là même une grande diversité de simulateurs. Cette diversité avait pour conséquence une non portabilité

des modèles et donc une impossibilité d'échange entre les sociétés. Un des rôles de VHDL est de permettre l'échange de descriptions entre concepteurs. Ainsi peuvent être mises en place des méthodologies de modélisation et de description de bibliothèques en langage VHDL.

L'effort de standardisation d'un langage tel que VHDL était nécessaire par le fait qu'il ne s'agissait pas de construire un seul simulateur VHDL (contrairement à la quasi totalité des autres langages), mais de permettre l'apparition d'une multitude d'outils (de simulation, de vérification, de synthèse, ...) de constructeurs différents utilisant la même norme, ce qui garantit ainsi une bonne qualité (la concurrence) et l'indépendance vis à vis des constructeurs de ces outils.

1.2. Un peu d'historique

Les langages de description de matériel (HDLs) ont été inventés à la fin des années 60. Ils s'appuyaient sur les langages de programmation et devaient permettre la description et la simulation de circuits.

Entre les années 1968 et 1975, une grande diversité de langages et de simulateurs ont vu le jour [Cras 85]. Cependant, leur syntaxe et leur sémantique étaient incompatibles et les niveaux de descriptions étaient variés.

En 1973, le besoin d'un effort de standardisation s'est fait ressentir et c'est ainsi que le projet CONLAN (pour CONsensus LANguage) a été mis en place. Les principaux objectifs de ce projet étaient de définir un langage de matériel permettant de décrire un système à plusieurs niveaux d'abstractions, ayant une syntaxe unique et une sémantique formelle et non ambiguë. Un groupe de travail (essentiellement académique) s'est donc formé en 1976 et a donné lieu à un rapport en janvier 1983. C'est alors que la portabilité des descriptions s'est avérée être de plus en plus indispensable.

En mars 1980, le département de la défense des États Unis d'Amérique (le DoD) lançait le programme VHSIC. En 1981, des demandes pour un nouveau langage de description de systèmes matériels, indépendant de toute technologie et permettant de couvrir tous les besoins de l'industrie micro-électronique, ont été formulées. En 1982, ces requêtes ont été mises par écrit et classifiées. C'est en 1983, que d'importantes sociétés telles que IBM, Intermetrics ou encore Texas Instruments se sont investies dans ce projet et à la fin de l'année 1984, un premier manuel de référence du langage ainsi qu'un manuel utilisateur ont été rédigés.

En 1985, des remises en cause et des évaluations ont donné naissance à la version 7.2 du langage VHDL, et en juillet 1986, Intermetrics a développé un premier compilateur et un premier simulateur.

C'est en mars 1986, qu'un groupe chargé de la standardisation du langage VHDL a été créé. Il s'agit du groupe américain VASG (VHDL Analysis and Standardization Group) qui est un sous comité des DASS (Design Automation Standard Subcommittees), eux-mêmes émanant de l'IEEE.

La norme VHDL IEEE 1076 a été approuvée le 10 décembre 1987. Cette norme est parfois notée VHDL'87 en référence à l'année de vote. La syntaxe et la sémantique de ce langage ont été publiées dans le manuel de référence du langage VHDL [IEEE 87].

En tant que standard IEEE, le langage VHDL est revoté tous les cinq ans afin de le remettre à jour, d'améliorer certaines caractéristiques ou encore d'ajouter de nouveaux concepts. Ainsi en 1991 a débuté le processus de re-standardisation : regroupement et analyse des requêtes, définition des nouveaux objectifs du langage, spécifications des changements à apporter au langage.

En 1992 est planifié l'approbation du nouveau standard par l'évaluation des transformations apportées au langage et par un vote. La nouvelle norme du langage VHDL a été votée en septembre 1993. La syntaxe et la sémantique de cette nouvelle norme ont donné lieu à un nouveau manuel de référence du langage VHDL [IEEE 93].

Il est à souligner que la nouvelle version du langage VHDL généralement notée VHDL'92 est, au moins théoriquement, la seule et unique version légale du langage. L'approbation de la nouvelle norme VHDL en 1993 rend le standard précédent (IEEE Std 1076 voté en 1987) désuet. Il n'y a pas deux standards (VHDL'87 et VHDL'92) mais uniquement un seul. Néanmoins, et heureusement, VHDL'92 reste compatible avec VHDL'87.

Il est également à noter qu'en 1992 se sont formés, autour du langage VHDL, de nombreux groupes de travail ayant des intérêts particuliers. Pour ne citer qu'un exemple, puisque VHDL'92 ne couvre pas plus le domaine analogique (ou celui de la simulation en mode mixte analogique numérique) que VHDL'87, un groupe de travail s'intéressant aux extensions analogiques du langage VHDL a été créé et doit donné le jour à un nouveau standard : le VHDL Std 1076.1 qui doit être voté au courant de l'année 1996.

2. Les principales caractéristiques du langage VHDL

2.1. Un langage s'appliquant à plusieurs niveaux de descriptions

Le langage VHDL couvre tous les niveaux partant du niveau des portes logiques de base jusqu'au niveau des systèmes complets (plusieurs cartes, chacune comprenant plusieurs circuits). Il s'applique tout aussi bien au niveau structurel qu'au niveau comportemental, en passant par le niveau transferts de registres. Par contre, VHDL ne couvre pas le domaine purement électrique : il est très difficile de donner des descriptions VHDL de circuits au niveau transistors. L'utilisation de VHDL dans un tel domaine ne semble pas efficace et ne paraît pas pouvoir se justifier, d'autres langages étant plus adaptés à ce genre de description (comme le langage M développé par Mentor Graphics [Magi 92]). De nombreux modèles utilisant le langage VHDL sont disponibles dans [Airi 90].

2.2. Simulation conduite par événements

La sémantique de l'évolution dynamique des signaux dans la description d'un circuit est définie en termes d'événements. Un modèle ne peut s'observer qu'à des intervalles d'unités de temps de taille variable; il ne peut pas être exprimé par une fonction continue du temps (couvrant le domaine des nombres réels). Un simulateur VHDL est donc un simulateur événementiel : l'accès aux informations se fait chaque fois qu'un événement apparaît sur un signal. Ce type de simulateur est différent du simulateur synchrone qui est guidé par une horloge maître : dans ce cas les informations sont accédées à des intervalles réguliers de temps correspondant aux cycles de l'horloge.

2.3. Modularité

La modularité est une des caractéristiques essentielles de VHDL. Toute description VHDL peut se décomposer en un ensemble de modules appelés unités de conception.

La notion de fichier source VHDL disparaît tout à fait après l'étape de compilation du dit fichier. Celui-ci peut contenir plusieurs unités de conception, mais en aucun cas une unité de conception ne peut être découpée en plusieurs fichiers. Après compilation, chaque unité de conception est stockée dans une bibliothèque VHDL. Une bibliothèque VHDL correspond donc à un ensemble de modules VHDL ayant passé avec succès la phase d'analyse syntaxique et sémantique. Chaque module peut alors servir de base à une simulation, une synthèse, etc. La figure I.1 illustre les étapes d'analyse et d'exploitation d'un fichier VHDL contenant plusieurs unités de conception.

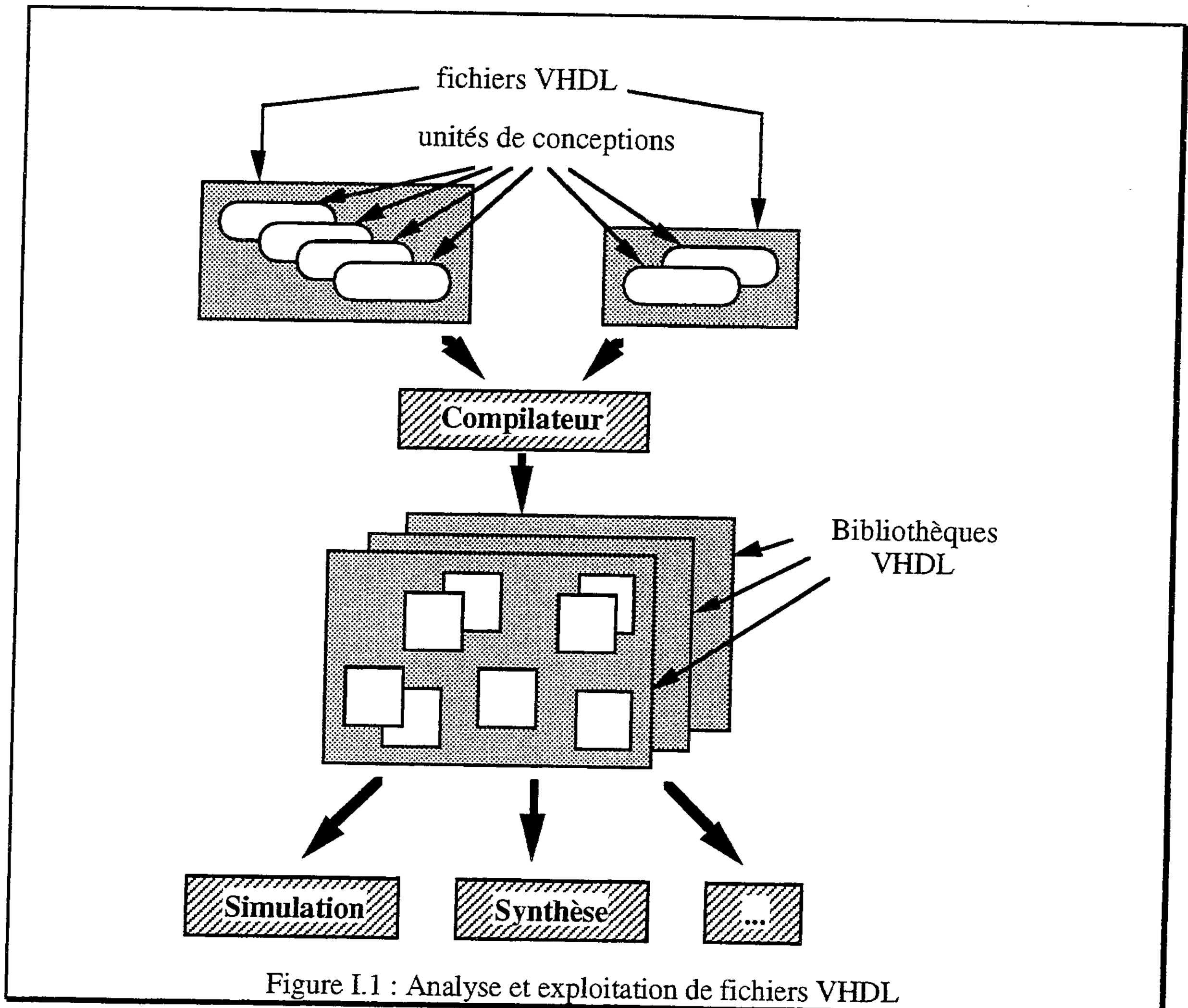

Figure I.1 : Analyse et exploitation de fichiers VHDL

Il existe cinq types différents d'unité de conception pouvant être classés en deux parties : les unités de conception dites primaires et celles dites secondaires. Les unités primaires décrivent l'objet vu de l'extérieur : son interface avec le monde extérieur, alors que les unités secondaires le décrivent vu de l'intérieur et donnent la façon dont il est constitué. Comme le montre la figure I.2, ces unités peuvent encore se classifier en deux groupes : le premier étant plutôt orienté matériel et le second plutôt logiciel.

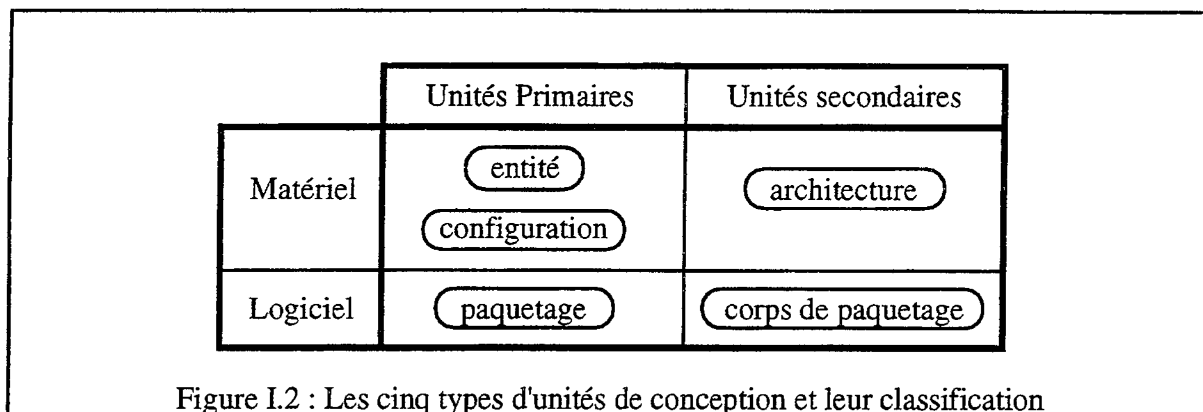

Figure I.2 : Les cinq types d'unités de conception et leur classification

Chaque unité de conception peut faire référence à une (ou plusieurs) unité(s) de conception se trouvant dans la même bibliothèque VHDL qu'elle même ou dans d'autres bibliothèques VHDL. Cependant au moment de la compilation d'une unité de conception faisant référence à d'autres unités, quelle que soit la bibliothèque à laquelle elles appartiennent, il est vérifié que ces unités aient été analysées avec succès au préalable. Egalement lors de la compilation, VHDL étant un langage fortement typé, une vérification statique des interfaces s'opère par compatibilité des types VHDL employés.

2.4. Portabilité

La portabilité constituait un des objectifs principaux pendant la phase de définition du langage VHDL. L'utilisation largement répandue de ce langage est essentiellement due au fait qu'il soit un standard. Un standard offre beaucoup plus de garanties (stabilité, fiabilité, etc.) vis-à-vis des utilisateurs que ne le permettrait un langage potentiellement précaire développé par une société privée.

Pour qu'un langage soit portable, sa syntaxe et sa sémantique doivent être aussi claires et non-ambiguës que possible. Un travail méticuleux a été fait pour décrire la syntaxe et la sémantique par rapport à la simulation du langage VHDL. Le manuel de référence [IEEE 93] est le résultat de ce travail. Bien que la clarté de ce document, surtout pour un débutant, soit discutable, ce document offre un réel intérêt : il constitue une référence complète pour les constructeurs d'outils VHDL, assurant ainsi un haut niveau de portabilité. Cependant, ce document ne défini pas une sémantique formelle du langage et ceci est plutôt gênant pour les personnes travaillant sur les outils de preuves formelles partant de spécifications VHDL. C'est pourquoi ces personnes ont chercher tout d'abord à définir une sémantique formelle du langage VHDL [Borr 87], [Sale 92].

VHDL est donc un langage portable. Cependant, l'utilisation de certaines constructions VHDL peuvent mener à des descriptions non portables. Ces constructions potentiellement non

portables sont tout de même peu nombreuses ; la liste en est donnée à l'annexe C du manuel de référence du langage VHDL [LRM 93].

2.5. Entité, architecture et composant

2.5.1. Entité

L'entité décrit la vue externe du modèle : elle permet de définir les ports par où sont véhiculées les informations dynamiques (signaux) et les paramètres génériques rendant compte des informations statiques. La figure I.3 donne un exemple d'entité définie pour un additionneur N bits. Il est possible de définir des valeurs par défaut pour les paramètres ; dans l'exemple qui est donné, le paramètres N correspondant au nombre de bits de l'additionneur a la valeur huit par défaut.

```
entity ADD_N is
generic (  N :       INTEGER := 8  );
port (     CIN :     in BIT;
           A,B :     in BIT_VECTOR (N downto 1);
           S :       out BIT_VECTOR (N downto 1);
           COUT :    out BIT       );
end ADD_N;
```

Figure I.3 : Exemple de définition d'entité

2.5.2. Architecture

L'architecture définit la vue interne du modèle. Cette description peut être structurelle, comportementale ou bien un mélange des deux. A chaque entité peut être associée une ou plusieurs architectures, mais, au moment de l'exécution (simulation, synthèse, ...), seulement une architecture et une seule est utilisée. Cette dernière est spécifiée par le mécanisme de configuration expliqué plus loin. Le fait de permettre plusieurs architectures pour la même entité présente un intérêt majeur pour les comparaisons, comme par exemple la comparaison entre une architecture décrite par l'utilisateur et celle générée à partir de cette description par un outil de synthèse.

Une architecture dépend implicitement de l'entité à laquelle elle est associée : tous les objets définis dans l'entité sont connus par l'architecture. Ainsi les ports définis au niveau de l'entité sont vus comme des signaux à l'intérieur de l'architecture et peuvent ainsi être lus ou écrits à cet endroit. Ceci se fait par le biais d'instructions concurrentes énumérées au niveau du corps de

l'architecture. L'architecture comprend aussi une partie déclarative où peuvent figurer un certain nombre de déclarations (de signaux, de composants, etc.) internes à l'architecture. A titre d'exemple, la figure I.4 donne une description possible d'architecture; associée à l'entité décrite précédemment pour un additionneur N bits.

```
architecture ARCH of ADD_N is
    signal C : BIT_VECTOR (N downto 0);
begin
    C(0) <= CIN;
    for I in 1 to N generate
        S(I) <= A(I) xor B(I) xor C(I-1);
        C(I) <= (A(I) and C(I-1)) or (B(I) and C(I-1)) or
                (A(I) and B(I));
    end generate;
    COUT <= C(N);
end ARCH;
```

Figure I.4 : Exemple de définition d'architecture

2.5.3. Composant et configuration

La notion de composant est une notion puissante de VHDL. Un modèle (un couple entité-architecture) peut être utilisé plusieurs fois par le biais de l'instanciation. Intancier un modèle signifie prendre une copie de ce modèle. Cette instanciation peut se faire directement (cela est devenu possible en VHDL'92) ou indirectement par l'intermédiaire d'un objet appelé composant. L'instanciation indirecte est un mécanisme plus général et plus flexible que l'instanciation directe car elle permet non pas d'associer directement l'instanciation au modèle lui-même mais à une idée que l'on a de ce modèle.

Le composant représente la vue externe du modèle. Aucun comportement n'est rattaché à la notion de composant, mais son gros avantage est de permettre la compilation. Donc beaucoup de tests statiques peuvent être faits même si le couple entité-architecture qui sera finalement utilisé n'est pas connu, ou même, n'existe pas encore. Ce mécanisme permet donc une méthodologie de conception dite descendante.

Utiliser la notion de composant implique trois opérations fondamentales : la déclaration, l'instanciation et la configuration du composant.

Lorsque le concepteur décrit à la fois le composant et l'entité correspondante, la déclaration de composant est très similaire à celle de l'entité. A titre d'exemple la déclaration du composant correspondant à l'entité décrite précédemment est donnée à la figure I.5.

```
component ADD_N
generic (   N :         INTEGER := 8   );
port (      CIN :       in BIT;
            A,B :       in BIT_VECTOR (N downto 1);
            S :         out BIT_VECTOR (N downto 1);
            COUT :      out BIT      );
end component;
```

Figure I.5 : Exemple de déclaration de composant

Cependant, considérer que la déclaration de composant est une simple redondance de la déclaration d'entité est une erreur. Lors de l'utilisation de bibliothèques ou d'unités de conception déjà existantes écrites par quelqu'un d'autre, la puissance de la notion de composant apparaît évidente : l'adaptation est possible. Dans l'exemple donné à la figure I.6, certains ajustements doivent être faits : le modèle a un paramètre en plus (TP, pouvant représenter le temps de propagation de l'inverseur), des noms de ports différents et ces ports ne sont pas donnés dans le même ordre. La configuration de composant permet de faire ces ajustements.

```
architecture A of E is
    component INVERSEUR
    port (  ENTREE :  in BIT;
            SORTIE :  out BIT      );
    end component;
begin
    ...
end A;

entity INV is
generic (  TP :     TIME := 2 ns     );
port (     S :      out BIT;
           E :      in BIT  );
end INV;
```

Figure I.6 : Exemple de déclaration de composant nécessitant des ajustements

Instancier un composant consiste à lui donner un nom et à indiquer comment sont connectés les ports du composant avec les signaux. Un exemple d'instanciation de composant est donné à la figure I.7 ; il s'agit de l'instanciation de l'additionneur N bits donné précédemment.

```
architecture A of E is
    component ADD_N
    generic (   N :       INTEGER   );
    port (  CIN :         in BIT;
            A,B :         in BIT_VECTOR (N downto 1);
            S :           out BIT_VECTOR (N downto 1);
            COUT :        out BIT       );
    end component;
    signal RE,RS : BIT;
    signal E1,E2,S : BIT_VECTOR (10 downto 1);
begin
    ADD_10 : ADD_N generic map (10) port map (RE,E1,E2,S,RS);
end A;
```

Figure I.7 : Exemple d'instanciation de composant

A ce niveau de description aucune information n'est donnée sur la façon dont sera configuré le composant (c'est-à-dire quel sera le couple entité-architecture associé à ce composant). En pratique, ce couple peut même ne pas exister à cette étape. Si une unité de conception séparée appelée unité de configuration est utilisée pour la configuration des composants, le code précédent peut être compilé mais non exécuté.

La configuration de composant consiste à associer une paire entité-architecture à une instance de composant. Elle peut être faite au niveau de l'architecture où les composants sont instanciés ou bien à l'intérieur d'une unité de conception spécifique appelée unité de configuration. Dans les deux cas, elle permet l'adaptation du composant au modèle. Cette adaptation peut consister à changer les noms des ports et des paramètres, modifier leur ordre, faire en sorte que certains d'entre eux disparaissent, changer leur type par utilisation d'une fonction de conversion, et même associer des tranches à des éléments ou des éléments à des tranches. Dans l'exemple donné à la figure I.8, l'instance ADD_10 est configurée à l'entité ADD_N et à son architecture associée ARCH décrites précédemment.

```
architecture A of E is
    component ADD_N
    generic (   N :      INTEGER   );
    port (  CIN :        in BIT;
            A,B :        in BIT_VECTOR (N downto 1);
            S :          out BIT_VECTOR (N downto 1);
            COUT :       out BIT     );
    end component;
    for ADD_10 : ADD_N use entity work.ADD_N(ARCH);
    signal RE,RS : BIT;
    signal E1,E2,S : BIT_VECTOR (10 downto 1);
begin
    ADD_10 : ADD_N generic map (10) port map (RE,E1,E2,S,RS);
end A;
```

Figure I.8 : Exemple de configuration de composant donnée au niveau de l'architecture

La flexibilité donnée par la notion de composant est très puissante. Sélectionner un couple entité-architecture est possible jusqu'à très tard dans le cycle de conception (juste avant la simulation ou la synthèse) et changer une bibliothèque par une autre pour remplacer un modèle par un autre est une opération instantanée.

2.6. Objets

2.6.1. Les constantes

Les constantes sont des objets bien connus des langages de programmation. Une fois que leur valeur est définie pendant l'initialisation, cette valeur reste inchangée par l'exécution. Cependant, lorsqu'elles sont déclarées dans un sous-programme, les constantes sont recalculées à chaque appel du sous-programme mais restent constantes pendant l'exécution du sous-programme. Il s'agit là de constantes localement statiques dont la valeur n'est connue qu'au moment de la compilation de l'unité de conception.

2.6.2. Les variables

Les variables, comme dans les langages de programmation, ont leur valeur immédiatement mise à jour lorsqu'elles sont assignées. Le mécanisme d'assignation des signaux diffère de celui des variables.

Il existe deux sortes de variables : les variables régulières et les variables partagées. Les variables régulières peuvent être déclarées soit à l'intérieur de sous-programme soit à l'intérieur

de processus ; elles sont locales à leur bloc de déclaration. Les variables partagées ont été introduites par VHDL'92. Nous y reviendrons ultérieurement.

2.6.3. Les signaux

Dans les circuits intégrés, l'information est propagée à travers des fils. En VHDL, ces fils sont appelés signaux. Les signaux définissent donc un réseau d'interconnexions.

Les signaux existent du début jusqu'à la fin du processus de simulation. Aucune création ou destruction de signal n'est permise pendant la simulation ; par contre, il est possible de déconnecter ou de connecter un signal.

La notion d'horloge en tant que signal spécifique n'existe pas en VHDL. Ceci implique des problèmes de reconnaissance du signal d'horloge par les outils de synthèse au moment de la lecture de la description VHDL.

Il est important de noter que l'échange d'information entre composants transite uniquement au travers de signaux. Une description structurelle VHDL, c'est-à-dire un ensemble complet de composants interconnectés par des signaux, existe à travers le processus de simulation. Même si les signaux semblent signifier physiquement des fils, pour des fins d'optimisation, certains d'entre eux peuvent être fusionnés pendant le processus de simulation et de synthèse.

Les signaux ne sont pas des contenants comme les variables. Les signaux sont des objets permanents et ont des liens fixés avec d'autres signaux. Les signaux ont une histoire : ils ont un passé accessible par l'utilisation d'attributs (qui sont des sortes de fonctions), un présent c'est-à-dire une valeur courante, et un futur consistant à un ensemble de couples (valeur, temps) projetés gérés par le noyau du simulateur. La figure I.9 donne la représentation usuelle du pilote d'un signal.

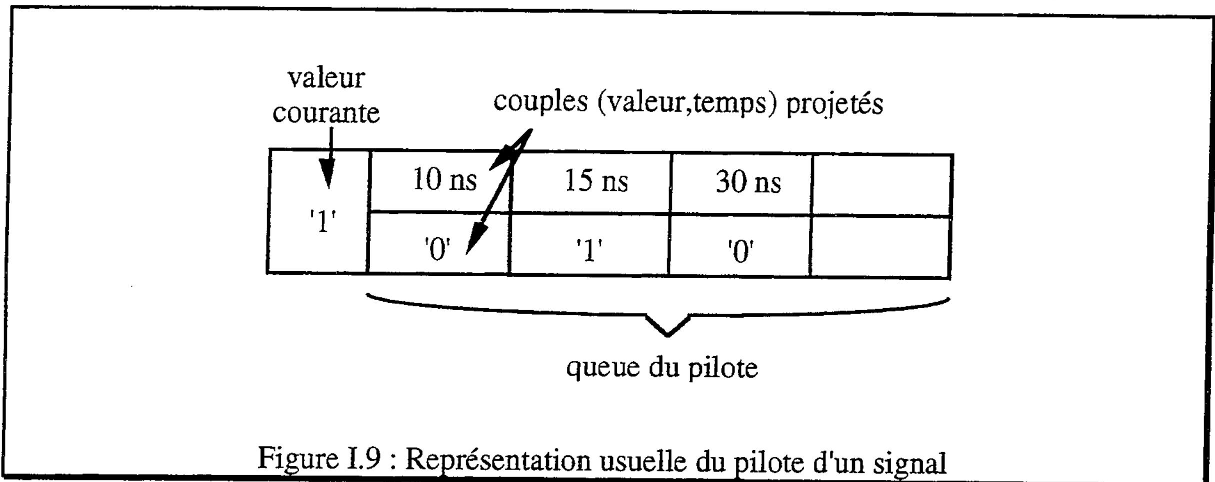

Figure I.9 : Représentation usuelle du pilote d'un signal

La caractéristique principale du signal provient de la sémantique de son assignation. Un signal peut être affecté par le résultat d'une expression pouvant utiliser les valeurs courantes d'autres signaux (comme par exemple *"A <= B and C;"* A,B et C étant des signaux).

Mais l'assignation d'un signal n'implique pas que la valeur courante de ce signal soit mise à jour immédiatement. En fait, la valeur de l'expression est stockée dans le pilote du signal affecté, en tant que valeur projetée de ce signal. Elle ne deviendra seulement la valeur courante du signal que lorsque sera rencontrée une instruction de synchronisation (instruction *"wait"*). Le concept tout entier de concurrence en VHDL repose sur ce mécanisme d'assignation de signaux.

De plus ce mécanisme peut utiliser des retards comme le montre l'exemple suivant : *"A <= B and C after 5 ns;"* . Une clause telle que *"after"* est en général ignorée par les outils de synthèse mais elle contient une sémantique de simulation très précise. Cette clause est souvent utilisée des concepteurs pour décrire les stimuli au niveau des descriptions de tests.

2.6.4. Les fichiers

Les fichiers ne peuvent pas être assignés, mais les opérations de lecture et d'écriture peuvent s'opérer sur cette dernière classe d'objets. Les fichiers n'ont pas de sens pour la synthèse et sont donc réservés pour les parties non-synthétisable des descriptions : les tests, les traces par exemple.

2.7. Types

Une des principales caractéristiques de VHDL est qu'il s'agit d'un langage fortement typé. Tous les objets définis en VHDL doivent appartenir à un type avant d'être utilisés. Deux objets sont compatibles si ils ont la même définition de type. Un type définit l'ensemble des valeurs que peut prendre un objet ainsi que l'ensemble des opérations disponibles sur cet objet. VHDL utilise le typage par nom par opposition au typage structurel.

Puisque VHDL s'applique au domaine logique, les valeurs '0' et '1' peuvent être considérées. Ceci constitue une première approche de la notion de type : l'ensemble de ces deux valeurs définit le type BIT. Il est possible de définir d'autres valeurs comme par exemple la valeur 'Z' désignant la valeur trois-états. Un type plus complet, englobant neuf valeurs logiques différentes, a été spécialement défini pour les aspects de simulation et de synthèse et sera décrit plus en détail dans le chapitre II.

Structurer l'information est une manière de fournir plus d'abstraction. Par exemple, il est plus commun de parler de mots que de vecteurs de bits. Cette possibilité est offerte en VHDL, et donc un simple signal peut être utilisé pour représenter un bus complet.

La puissance de description de VHDL provient essentiellement du fait que le concepteur peut définir et utiliser ses propres types. De plus, un grand nombre de vérifications au niveau de la compilation assure la cohérence entre les valeurs et les opérations effectuées sur ces valeurs. Le typage permet de découvrir très tôt, avant la simulation, des erreurs au niveau du modèle VHDL.

Il existe quatre familles de types VHDL :
- les types scalaires, dont la valeur est composé d'un seul élément,
- les types composés, dont la valeur comprend plusieurs éléments,
- les types accès, qui sont les pointeurs bien connus des langages de programmation,
- les types fichiers, qui ne sont utilisés que pour les objets fichiers.

Les deux dernières familles ne seront pas décrites plus en détails dans ce qui suit. Les types accès sont principalement utilisés au niveau système et n'ont pas de relation directe avec les descriptions de matériel. Les fichiers servent surtout à stocker des stimuli ou des traces mais ne sont pas utilisés en synthèse. Ils pourraient cependant être utilisés au moment de l'élaboration pour stocker le contenu d'une mémoire ROM ou bien des valeurs de coefficients lors de la modélisation de filtres.

2.7.1. Les types scalaires

Il existent quatre sortes de types scalaires :

• Les types énumérés, dont les valeurs possibles sont explicitement listées :

```
type BIT is ('0','1');
type BOOLEAN is (FALSE, TRUE);
type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
```

Figure I.10 : Exemples de types énumérés

• Les entiers, dont le rang des valeurs peut être spécifié :

```
type MOT is range 0 to 15;
type MEMOIRE is range 1 to 2048;
```

Figure I.11 : Exemples de types entiers

Pour les types entiers, définir une contrainte sur le rang implique que des vérifications seront effectuées pendant la simulation. Toute violation de la contrainte générera une erreur. De plus, cette indication de contrainte permet aux outils de synthèse de calculer le nombre de bits nécessaire pour représenter les objets de ce type. En effet, il est d'usage de représenter un entier non contraint par un vecteur de taille maximale de 32 bits.

• Les types réels, qui peuvent aussi avoir un rang explicite :

```
type NOTE is range 0.0 to 20.0;
type MEMOIRE is range 1 to 2048;
```

Figure I.12 : Exemples de types réels

• Le types physiques, qui peuvent être considérés comme des entiers associés à une unité physique permettant de faciliter les notations et les vérifications d'analyse des dimensions.

2.7.2. Les types composés

Les types composés sont pratiques pour grouper des informations. Il en existe deux catégories : les tableaux et les enregistrements.

Les tableaux sont utilisés pour structurer des éléments de même type. Par exemple, un bus peut être vu comme un tableau à une dimension de bits.

```
type BUS_32 is array (0 to 31) of BIT;  -- bus de 32 bits
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

Figure I.13 : Exemples de types tableaux

Cependant, pour la synthèse, cette information a besoin d'être complétée par des conventions dans le but de fournir plus de détails sur la structure : Existe-t-il un bit de signe ? Où est-il ? Où se trouve le bit de poids le plus fort ?

La désignation d'un élément de tableau se fait par l'utilisation d'un index. Soit un objet A (constante, variable ou signal) de type BUS_32, A(10) désigne le onzième élément de A.

Les enregistrements constituent la seconde catégorie de types composés. Au niveau de la synthèse, les enregistrements s'utilisent bien moins que les tableaux mais ils peuvent parfois être pratiques.

```
type DATE is record
    JOUR : INTEGER;
    MOIS : STRING (1 to 10);
end record;
```

Figure I.14 : Exemple de type enregistrement

L'accès à un élément de type enregistrement se fait par l'intermédiaire de son nom. Soit un objet B de type DATE, B.JOUR désigne l'élément entier de B. Une constante, tout comme une variable ou un signal peut être de type enregistrement.

2.7.3. Objets et types

La section précédente a permis d'énumérer les quatre classes différentes d'objets existant en VHDL : les constantes, les variables, les signaux et les fichiers. La notion de classe et de type est conceptuellement différente. Le type décrit la structure de l'information contenue par l'objet alors que la classe en indique la nature.

D'autre part, l'information contenue par ces objets peut être de nature différente : statique pour les constantes, locale et dynamique pour les variables non partagées ou globale et dynamique pour les signaux et les variables partagées.

2.8. Instructions concurrentes

2.8.1. Définition

Le comportement d'un circuit peut être décrit par un ensemble d'actions s'exécutant en parallèle. C'est pourquoi VHDL offre un jeu d'instructions dites concurrentes. Une instruction concurrente est une instruction dont l'exécution est indépendante de son ordre d'apparition dans le code VHDL. Par exemple, prenons le cas d'un simple verrou, comme le montre la figure I.15.

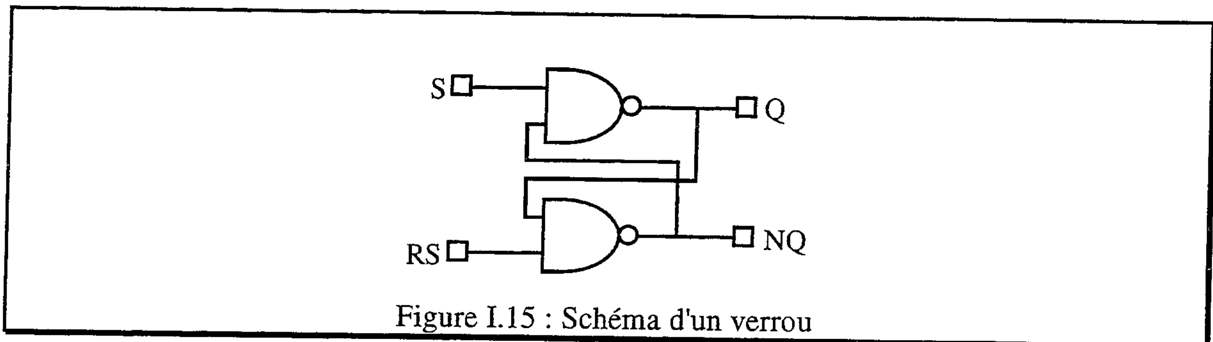

Figure I.15 : Schéma d'un verrou

Les deux portes constituant ce verrou fonctionnent en parallèle. Une description possible de ce circuit est donnée à la figure I.16 (seule l'architecture est donnée).

```
architecture ARCH of VERROU is
begin
    Q <= S nand NQ;
    NQ <= R nand Q;
end ARCH;
```

Figure I.16 : Description d'un verrou

Ces deux instructions s'exécutent en même temps. Elles sont concurrentes; leur ordre d'écriture n'est pas significatif ; quelle que soit l'ordre de ces instructions, la description reste inchangée. Les instructions concurrentes apparaissent entre les deux mots clés *"begin"* et *"end"* d'une architecture. Elles sont utilisées pour décrire la structure ou le comportement du circuit. La structure du circuit est spécifiée par les instructions d'instanciation de composants et les instructions *"block"* et *"generate"* . Par contre, le comportement du circuit est donné par les instructions concurrentes d'affectation de signaux, les processus, les appels concurrents de procédures et les instructions concurrentes d'assertion.

2.8.2. Simulation

En VHDL, un événement est un changement de valeur d'un signal. La simulation VHDL est conduite par événements. Une instruction VHDL est donc exécutée chaque fois qu'un événement apparaît sur un des signaux commandant cette instruction. Dans l'exemple ci-dessus, l'affectation du signal Q est commandée par les signaux S et NQ; donc si la valeur de l'un de ces deux signaux change, le signal Q est affecté. D'une manière plus générale, une instruction consiste à affecter une cible par une source. Au cours de la simulation d'une instruction, la source est évaluée dans un premier temps, puis la cible est mise à jour après un délai unitaire noté delta. Ce temps de simulation est un temps virtuel. La figure I.17 illustre la différence entre le temps de simulation virtuel décomposé en délais unitaires delta pour le simulateur et le temps réel tel que le perçoit l'utilisateur. Les chronogrammes qui sont donnés s'appliquent au verrou décrit à la figure I.16.

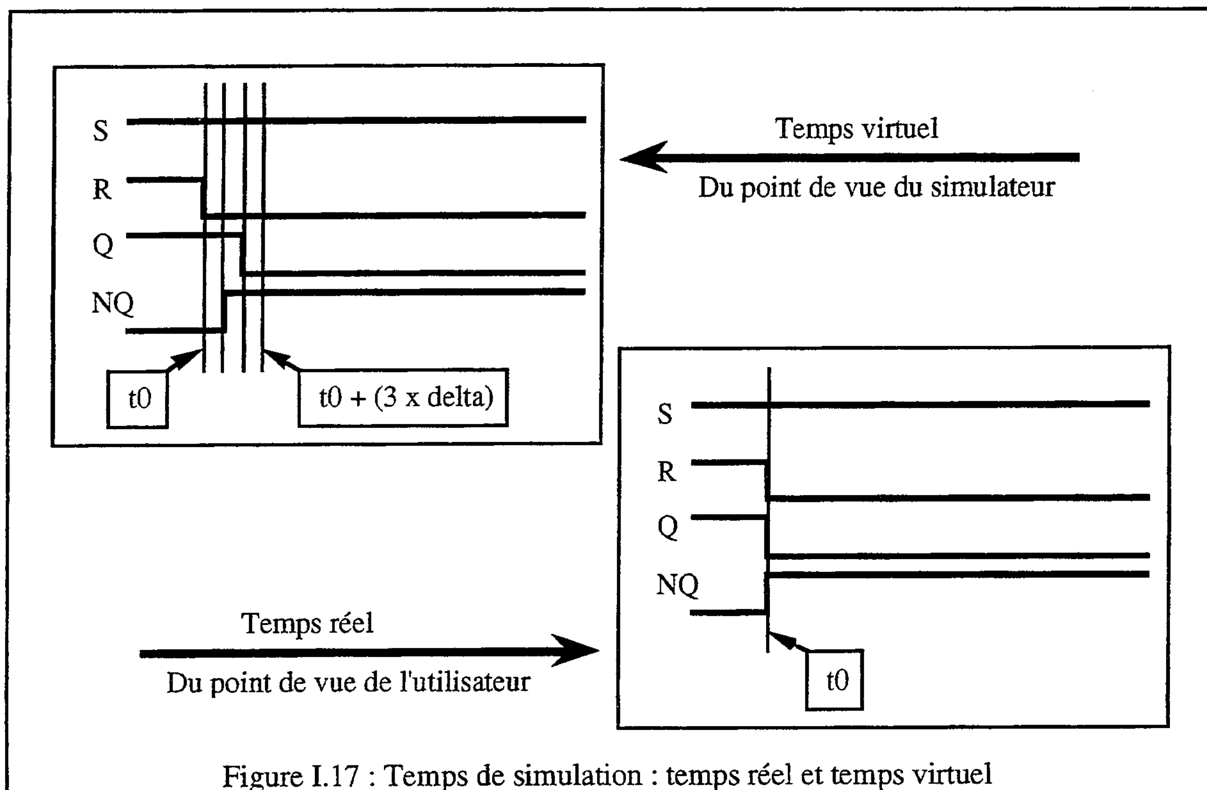

Figure I.17 : Temps de simulation : temps réel et temps virtuel

2.8.3. L'instruction concurrente d'affectation de signal

L'instruction concurrente d'affectation de signal peut prendre une forme conditionnelle ou sélective. Quelques exemples d'affectations concurrentes de signaux sont données à la figure I.18.

```
-- affectations inconditionnelles de signaux
S0 <= E1 after 5 ns;
S1 <= E1 + E2;
S2 <= FONCTION(E3,E4);

-- affectations conditionnelles de signaux
S3 <=   "001" when E5 = 1 else
        "010" when E5 = 2 else
        "100" when E5 = 3 else
        "000";
with E6 select
S4 <=   E4 when '1',
        E5 when others;
```

Figure I.18 : Exemples d'affectations concurrentes de signaux

2.8.4. Fonction de résolution

Sur une carte tout comme à l'intérieur d'un circuit intégré, il est possible, mais aussi délicat, de connecter plusieurs sorties ensemble. Que se passe-t-il lorsque ces sorties n'ont pas la même valeur ? Selon la technologie cible utilisée, les valeurs résultantes de ces conflits sont bien connues et donc de telles connexions peuvent être établies. En VHDL, lorsque plusieurs sorties sont connectées ensemble, le compilateur exige que le conflit soit résolu par une fonction particulière appelée fonction de résolution. Lorsqu'il y a un conflit, une telle fonction est systématiquement appelée pour résoudre le conflit en question. Elle permet de calculer la valeur résultante à partir des valeurs en conflit. Le concepteur à la possibilité d'écrire ses propres fonctions de résolution. Cependant, il peut aussi utiliser des types ou sous-types résolus, proposés dans des paquetages standards, pour lesquels les fonctions de résolution correspondantes sont déjà définies et sont automatiquement et implicitement appelées par le simulateur en cas de conflits. Notons que pour être portable, il est préférable que la fonction de résolution traite toutes ses entrées symétriquement car l'ordre dans lequel sont données les entrées peut varier d'un appel à un autre, pour une telle fonction. Une fonction de résolution est portable si elle est associative et commutative.

2.8.5. Processus

L'instruction VHDL *"process"* définit un processus séquentiel indépendant représentant le comportement d'une partie du modèle. Un processus est caractérisé par les signaux auxquels il est sensible et les opérations séquentielles qu'il exécute pour mettre à jour des signaux de sortie.

Toute instruction concurrente peut toujours être traduite par un processus (ou éventuellement plusieurs dans le cas d'instanciation de composant) appelé processus équivalent. Le processus est l'objet fondamental manipulé par le simulateur, une description VHDL se ramenant pour lui à un ensemble de processus.

Pendant la simulation, un processus ne peut être que dans l'un de ces deux états :

- soit il s'exécute comme un sous-programme,
- soit il attend qu'un signal change de valeur (c'est-à-dire un événement :*"event"*) et reste bloqué sur une instruction de synchronisation : l'instruction *"wait"*.

Un processus est décomposé en deux parties : une première partie déclarative, comprise entre les mots clés *"process"* et *"begin"* , où peuvent être déclarées entre autres des variables, et une seconde partie, délimitée par les mots clés *"begin"* et *"end process"* , où figurent les instructions séquentielles. Un processus est appelé seulement une fois au début de la simulation puis il boucle. Il s'exécute jusqu'à la prochaine instruction *"wait"* , attend les événements correspondants, puis s'exécute à nouveau jusqu'à la prochaine instruction *"wait"* . Lorsqu'il arrive aux mots réservés finaux *"end process"* , il s'exécute à nouveau instantanément à partir du mot-clé *"begin"* .

L'instruction *"wait"* offre plusieurs formes de synchronisation :

- attendre un événement sur un signal qui se trouve dans une liste (liste de sensibilité d'un processus) (*"wait on CLOCK, RESET;"*),
- attendre qu'une condition devienne vraie (*"wait until CLOCK = '1';"*),
- attendre une certaine période de temps (*"wait for 10 ns;"*),
- attendre n'importe quelle combinaison des trois formes précédentes (*"wait on S1,S2 until CLOCK = '1' for 5 ns;"*).

La figure I.19 donne les deux processus équivalents aux deux instructions concurrentes de la figure I.16 décrivant un verrou.

```
architecture ARCH of VERROU is
begin
    process
    begin
    Q <= S nand NQ;  -- affectation séquentielle
    wait on S, NQ;
    end process;
    process
    begin
    NQ <= R nand Q;  -- affectation séquentielle
    wait on R, Q;
    end process;
end ARCH;
```

Figure I.19 : Processus équivalents (description du verrou)

2.8.6. Les autres instructions concurrentes

Hormis les instanciations de composants qui ont déjà été vues précédemment et les affectations de signaux et les processus qui viennent d'être examinés, il existe d'autres instructions concurrentes : les appels de procédures, l'instruction d'assertion, l'instruction *"generate"* et l'instruction *"block"* .

Les appels concurrents de procédures sont à rapprocher des instanciations de composants et peuvent être utilisés dans le même but. Il est cependant à noter que la flexibilité offerte par le mécanisme de configuration dans le cas d'instanciations de composants est perdue lors de l'utilisation d'appels concurrents de procédures. Les appels concurrents de procédures peuvent également être utilisés pour effectuer automatiquement et dynamiquement des vérifications de contraintes de temps. Dans ce cas ces procédures sont dites passives : elles n'affectent aucun signal et ne possèdent que des paramètres en entrée.

L'instruction d'assertion est aussi une instruction passive : elle n'affecte aucun signal. Cette instruction est utilisée pour faire des vérifications et peut permettre de stopper une simulation selon la gravité affectée au test en question. Un exemple d'instruction d'assertion est donné à la figure I.20.

```
assert RESET = '1'
report "Reset is active!"
severity WARNING;
```

Figure I.20 : Exemple d'instruction d'assertion

L'instruction *"generate"* est un moyen de décrire des structures potentielles (en utilisant la forme conditionnelle) ou bien des structures fortement régulières (en utilisant la forme itérative). Cette instruction n'ajoute aucune fonctionnalité au comportement du modèle mais permet à la description d'être plus générale et plus flexible. Un exemple d'utilisation de la forme itérative de l'instruction *"generate"* a été donné à la figure I.4. modélisant les équations d'un additionneur N bits.

L'instruction *"block"* est le moyen de réunir des instructions concurrentes pour leur faire partager certaines déclarations locales invisibles du reste de la description. Mais l'utilisation première du bloc en modélisation est le bloc gardé qui permet de regrouper un ensemble d'instructions concurrentes et de ne les exécuter que lorsqu'une condition, la condition de garde, est vraie. Un exemple de bloc gardé modélisant un verrou est décrit à la figure I.21.

```
LATCH : block (CLOCK = '1')
begin
   Q <= guarded D after 1 ns;
end block;
```

Figure I.21 : Exemple de bloc gardé utilisé pour modéliser un verrou

2.9. Instructions séquentielles

Les instructions concurrentes qui viennent d'être présentées pourraient suffire à définir un langage de description de matériel. Cependant, certaines parties de systèmes sont plus faciles à décrire en utilisant des instructions séquentielles plus proches des instructions de langages classiques de programmation.

En VHDL, les instructions séquentielles ne s'utilisent qu'à l'intérieur des processus et des sous-programmes (fonctions et procédures). A ces endroits les instructions sont exécutées en séquence et donc l'ordre dans lequel elles sont données joue un rôle primordial au niveau du comportement obtenu. Ces instructions sont pour la plus part inspirées des langages classiques de programmation. Parmi ces instructions, la plupart sont similaires à celles des langages de programmation de haut niveau ; il s'agit de :

- l'instruction d'affectation de variable (*"VAR := 0;"*),
- l'appel de procédure;
- l'instruction *"if"* (*"if A='1' then B:=0; else B:=1; end if;"*),
- l'instruction *"case"* (*"case A is when '1' => B:=0; when others => B:=1; end case;"*),
- les instructions *"loop"*, *"for...loop"*, *"while...loop"* (*"for I in 0 to 5 loop VAR:=VAR+3; end loop;"*),

- les instructions *"next"* ,*"exit"* ,*"return"* ,
- l'instruction *"null"* .

D'autres sont spécifiques au langage VHDL ; il s'agit de :
- l'instruction d'affectation de signaux,
- l'instruction *"wait"* ,
- l'instruction d'assertion.

Toutes ces instructions ne seront pas davantage détaillées, certaines l'ayant été précédemment, les autres pouvant se comprendre facilement par analogie aux instructions classiques de programmation.

2.10. Les paquetages

Comme il a été dit à la section 2.3, il existe cinq unités de conception en VHDL. Les entités, les architectures et les configurations ayant été vues à la section 2.5, nous allons exposer ici les deux unités restantes : les paquetages et les corps de paquetage.

Les paquetages permettent de regrouper des déclarations de sous-programmes, de types, d'objets, etc. afin d'en faciliter leur réutilisation. Ils constituent également un moyen efficace de standardisation d'un environnement VHDL sans pour autant rajouter quoique ce soit au manuel de référence du langage.

Les objets du langage (sous-programmes, types, constantes, signaux, attributs, etc.) qui sont déclarés dans un paquetage peuvent ainsi être exportés. En effet, n'importe quelle unité de conception faisant référence à ce paquetage par l'intermédiaire de la clause VHDL *"use"* , a accès à ces objets et peut les utiliser. Un paquetage peut avoir un corps associé ; c'est même obligatoire lorsque des sous-programmes (procédures, fonctions) sont déclarés dans le paquetage. Dans ce cas, ce corps est unique et contient les corps séquentiels des sous-programmes.

2.11. Environnement préalablement défini du langage

Cet environnement est défini dans le manuel de référence du langage [IEEE 93] et constitue donc une partie du langage. Il consiste en :

• un ensemble de types préalablement définis à l'intérieur du paquetage *"STANDARD"* . Les types *"BIT"* , *"BOOLEAN"* et *"BIT_VECTOR"* en sont quelques exemples. Ce paquetage est implicitement référencé et par conséquent aucune clause particulière n'a besoin d'être donnée.

• des primitives d'entrée-sorties définies dans le paquetage *"TEXTIO"* . Ces primitives permettent de lire ou d'écrire du texte. Ces primitives ne sont pas synthétisables, mais par contre, dans un environnement de simulation, elles permettent de générer des vecteurs de tests et de conserver des traces.

• plusieurs familles d'opérateurs préalablement définis en VHDL : des opérateurs logiques (ET, OU, etc.), des opérateurs relationnels (=, /=, <, etc.), des opérateurs arithmétiques (+,-, etc.), etc.

• quelques attributs préalablement définis. Ils peuvent être classifiés selon ce qu'ils retournent (valeur ou signal) et selon les objets auxquels ils s'appliquent (tableaux, signaux, types, etc.). Pour ne donner que quelques exemples d'utilisation d'attributs préalablement définis, *"A'LENGTH"* qui est une expression, renvoie une valeur qui est la taille du tableau A. Utilisé à l'intérieur d'un sous-programme, cet attribut permet une grande flexibilité au niveau des paramètres. L'attribut *"S'DELAYED(10 ns)"* retourne un signal qui est le signal S retardé de 10 ns. Ceci peut être utilisé pour faire des vérifications temporelles.

Hormis les paquetages qui viennent d'être cités, d'autres paquetages ont été définis ultérieurement et ne font pas partie intégrante du langage. Nous y reviendrons plus tard au cours du chapitre II.

2.12. Les trois styles de descriptions en VHDL

Il existe trois styles de descriptions en VHDL :

- la description structurelle : le circuit est décrit par sa structure, sous forme d'une liste de composants instanciés et des interconnexions les reliant.

- la description flot de données : le circuit est décrit par une liste d'instructions concurrentes d'affectations de signaux du type "*<signal> <= <expression>* " où <expression> peut représenter un simple signal, ou une expression utilisant des opérateurs logiques, arithétiques ou relationnels, ou une expression conditionnelle.

- la description comportementale : le circuit est décrit par un ou plusieurs processus renfermant une liste d'instructions séquentielles. Ce style de description utilisée par la synthèse de haut niveau, permet en particulier de spécifier le circuit sous forme d'un algorithme et dans ce cas, nous pourrons même parler de description algorithmique.

Toute description peut également être un mélange de ces trois styles. C'est ce que montre la figure I.22.

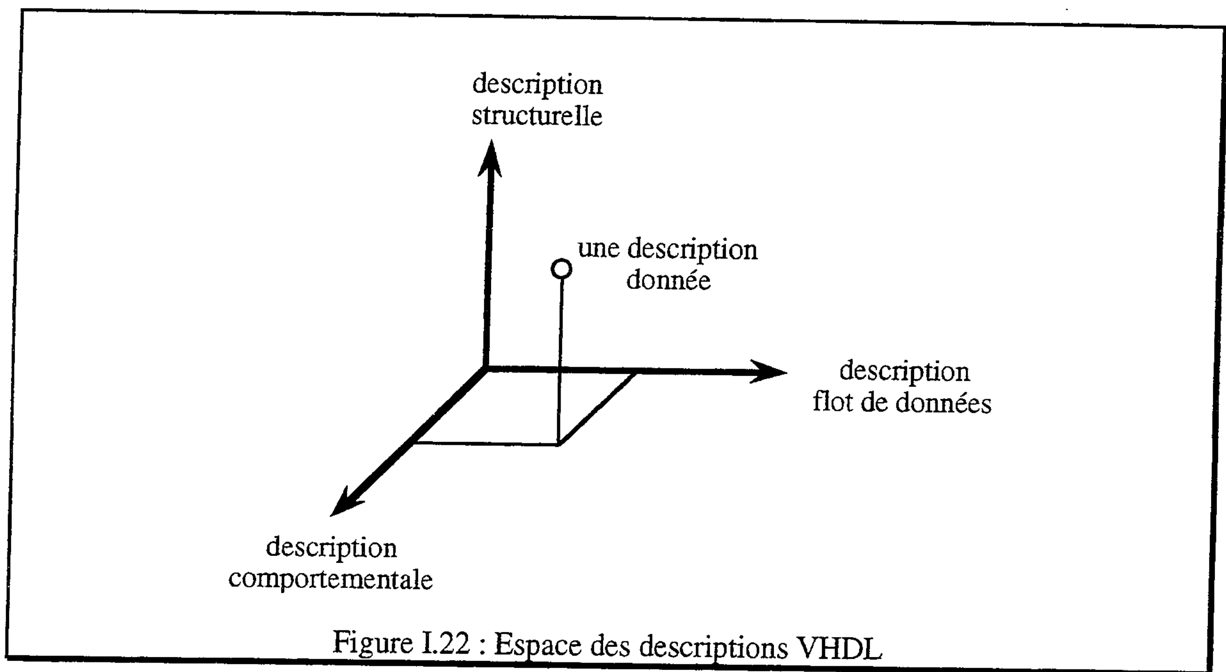

Figure I.22 : Espace des descriptions VHDL

Afin de résumer ce qu'il vient d'être vu et à titre d'exemple, voici trois descriptions d'un additionneur un bit, équivalentes au point de vue comportemental. La figure I.23 donne la description complète de cet additionneur (entité plus architecture); il s'agit d'une description structurelle. L'additionneur est décrit par une composition structurelle de trois éléments : deux demi-additionneurs et une porte logique OU.

```
entity ADD is
port (        X,Y,CIN :  in BIT
              S,COUT :   out BIT      );
end ADD;

architecture A1 of ADD is
    component DEMI_ADD
    port (  A1,B1 :     in BIT;
            S1,COUT :   out BIT      );
    end component;
    component OU
    port (  A1,A2 :     in BIT;
            S, :        out BIT      );
    end component;
    signal A,B,C : BIT;
begin
    U1 : DEMI_ADD port map (X,Y,A,B);
    U2 : DEMI_ADD port map (B,CIN,C,S);
    U3 : OU port map (A,B,COUT);
end A1;
```

Figure I.23 : Description structurelle d'un additionneur un bit

La figure I.24 donne une autre architecture possible pour cet additionneur, mais le style de description utilisé ici est le style flot de données. Les sorties sont affectées par de simples équations.

```
architecture A2 of ADD is
    signal SI : BIT;
begin
    SI <= X xor Y;
    S <= SI xor CIN;
    COUT <= (X and Y) or (S and CIN);
end A2;
```

Figure I.24 : Architecture flot de données d'un additionneur un bit

Enfin, la figure I.25 illustre le troisième style de description : le style comportemental. Cet exemple illustre également très bien l'utilisation des variables. Ici la variable N est temporaire et sert à faire un calcul intermédiaire. Son affectation est immédiate. Dans cette description, la variable N ne peut pas être remplacée par un signal car alors le comportement obtenu ne serait plus celui d'un additionneur.

```
architecture A3 of ADD is
begin
    process
        variable N : INTEGER;
        constant SUM_VECTOR : BIT_VECTOR (0 to 3) := "0101";
        constant CARRY_VECTOR : BIT_VECTOR (0 to 3) := "0011";
    begin
        N := 0;
        if X = '1'      then N := N+1;  end if;
        if Y = '1'      then N := N+1;  end if;
        if CIN = '1'    then N := N+1;  end if;
        S <= SUM_VECTOR(N);
        COUT <= CARRY_VECTOR(N);
        wait on X,Y,CIN;
    end process;
end A3;
```

Figure I.25 : Architecture comportementale d'un additionneur un bit

3. VHDL'92 : les différences par rapport à la norme de 87

En tant que standard IEEE, le langage VHDL est revoté tout les cinq ans. Ainsi en 1991 a été mis en place un processus de re-standardisation et en septembre 1993 la nouvelle norme a été approuvée. Cette nouvelle norme a donné lieu à un nouveau manuel de référence du langage VHDL [IEEE 93] décrivant sa syntaxe et sa sémantique. Il est bien évident que ce vote a rendu le standard précédent désuet. Cependant, les outils (de compilation, de simulation, de synthèse, etc.) qui acceptaient la norme de 87 ne prennent pas forcément tous en compte aujourd'hui, la nouvelle norme de 92. Un impératif imposé par le nombre important d'outils utilisant VHDL, était d'assurer la compatibilité entre VHDL'92 et VHDL'87. Nous verrons quelles ont été les grandes lignes qui ont guidé le processus de re-standardisation au cours de la première section. Les parties suivantes donneront les nouveautés apportées par VHDL'92 par rapport à VHDL'87.

3.1. Les grandes lignes qui ont guidé le processus de re-standardisation

Les grandes lignes qui ont guidé le processus de re-standardisation et qui sont exposées dans [Berg 93a] et [Berg 93b] sont rappelées brièvement dans ce qui suit :

• Assurer la compatibilité avec VHDL'87. Ceci constitue un impératif industriel. Cette compatibilité est assurée à environ 90% [Krol 94a]. Il n'existe que quelques exceptions d'incompatibilité qui sont les mots clés qui ont été rajoutés dans VHDL'92 et ceux qui ont été supprimés.

• Préserver le typage fort. Cette caractéristique de VHDL permet de faire de nombreuses vérifications au cours de la compilation.

• Séparer déclarations et fonctionnalités. Ce point constitue la philosophie réelle de VHDL'87 (sous-programmes, entité-architecture, etc.). C'est pour cette raison que par exemple, la demande d'associer une sémantique particulière à un identificateur nommé CLOCK ou RESET à été rejetée.

• Préserver la notion unique de temps. L'instruction *"S <= A after 10 ns;"* signifie que le signal S prendra la valeur de A exactement après 10 ns par rapport au temps de simulation et non pas au moins 10 ns après comme pourraient l'interpréter certains outils de synthèse.

• Préserver le déterminisme. En VHDL'87, l'utilisation de fonctions de résolution et des opérations de lecture et d'écriture de fichiers peuvent amener dans certains cas à des résultats

non-déterministes. Avec VHDL'92, l'utilisation de variables partagées constitue une troisième source possible d'indéterminisme.

• Préserver la généralité. VHDL est un langage très général ce qui explique son utilisation dans des domaines variés. L'idée développée ici consiste à donner des solutions générales aux problèmes spécifiques qui ont été abordés. Ceci évidement ne simplifie pas l'utilisation du langage dans un domaine précis, mais c'est le prix à payer pour garder un langage général.

• Préserver la faculté de bien décrire du niveau portes logiques au niveau système.

• Préserver la faculté de pouvoir utiliser les différents styles (structurel, flot de données et comportemental) dans une même description.

• Préserver le parallélisme dans le langage VHDL.

• Préserver et améliorer la cohérence du langage.

• Préserver et éventuellement améliorer la portabilité. L'annexe C du LRM [IEEE 93] énumère les constructions non portables.

• Ne pas rajouter de paquetage spécifique à une application dans le LRM. Des paquetages spécifiques à la synthèse pourront être standardisés mais ne feront en aucun cas partie du langage lui-même.

• Minimiser l'impact des modifications sur l'implantation. En effet, beaucoup d'outils se sont développés autour de VHDL et la priorité a été donnée aux solutions qui minimisent l'impact sur implantation.

• Optimiser l'efficacité d'implantation. Un bon exemple pour illustrer ce dernier point est l'introduction des variables globales. A un très haut niveau de description, les concepteurs de systèmes préféreront utiliser des variables plutôt que des signaux. Le coût des signaux (au niveau simulation) est très important (fonctions de résolution, pilotes) et n'est pas nécessaire dans ce cas. La description est si éloignée du résultat physique que seules des variables globales comme celles disponibles dans les langages de programmation sont suffisantes.

Dans les parties suivantes sont énumérées et groupées les nouveautés introduites par VHDL'92 par rapport à VHDL'87. Il ne s'agit pas là de voir et de développer en détails chaque point, mais plutôt de donner un aperçu de VHDL'92. Pour plus de précisions, il est possible de se reporter à [Berg 93b], [Guyl 92], [Börg 94] ou encore au LRM [IEEE 93].

3.2. Les nouveaux mécanismes de simulation

3.2.1. Activation lors du dernier delta délai

Le concept de delta délai (pas de simulation vu comme un délai infinitésimal) est l'artifice VHDL qui impose la causalité au cours de la simulation. Étant donné un temps t0 au cours de la simulation, si un événement intervient au temps t0, le simulateur va calculer les nouvelles valeurs des sorties et pour ce faire va généralement avoir besoin de plusieurs delta délais. Cependant, il se peut que le concepteur veuille ignorer ce qui se passe durant ces delta délais et ne soit intéressé que par l'état stable qui a lieu lors du dernier delta délai. La figure I.26 donne un exemple d'instruction d'assertion qui génère une fausse alarme.

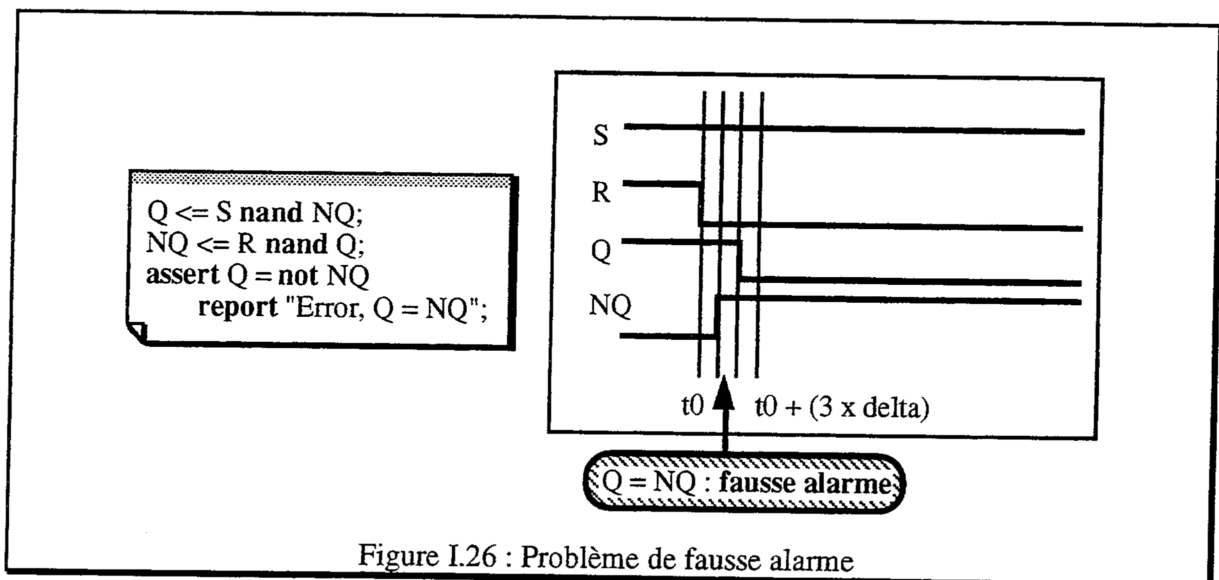

Figure I.26 : Problème de fausse alarme

La solution à ce problème a été apporté par VHDL'92 avec l'introduction de la notion de processus différé (*"postponed"*). Un processus différé n'est activé que pendant le dernier delta délai de simulation. Le problème rencontré dans l'exemple précédent peut facilement être résolu par l'utilisation d'une instruction d'assertion différée, qui correspond également à un processus comme nous l'avons vu auparavant. Une telle instruction ne sera exécutée qu'une seule fois au cours du dernier delta délai de simulation. La figure I.27 donne la solution au problème posé par la figure I.26.

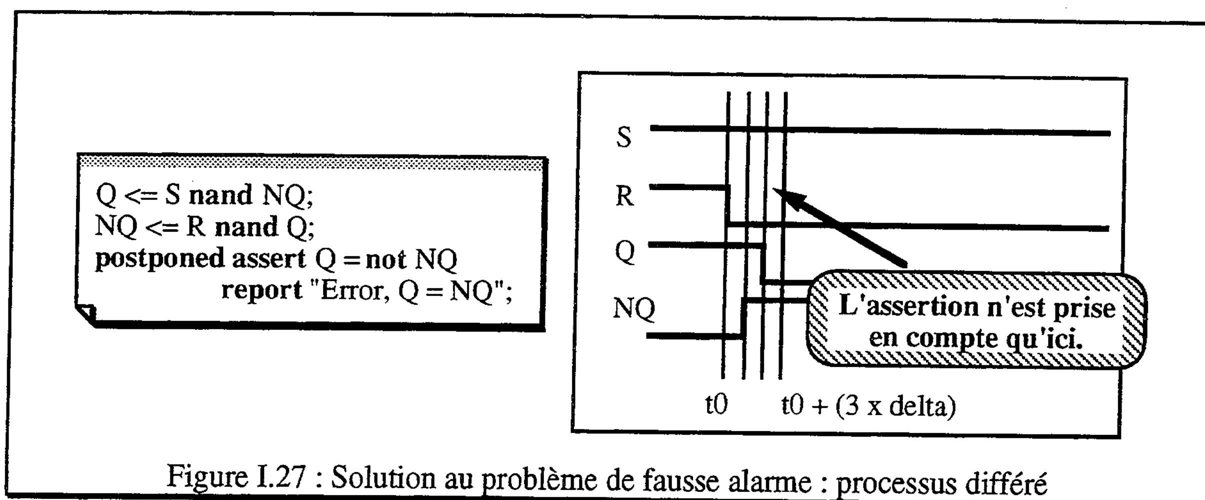

Figure I.27 : Solution au problème de fausse alarme : processus différé

Le mot-clé *"postponed "* s'applique aux processus, aux instructions d'assertion, aux appels concurrents de procédure et aux affectations concurrentes de signaux.

3.2.2. Les variables partagées (ou variables globales)

Une variable partagée est une variable qui peut être accédée (en lecture ou en écriture) par plus d'un processus. Les variables partagées n'ont pas d'équivalent physique. Elles ont été introduites par VHDL'92 pour un besoin particulier : l'utilisation efficace d'objets globaux. Bien que les signaux puissent être utilisés globalement depuis VHDL'87, leur coût en temps de simulation est très cher et leur utilisation en tant qu'objet global n'est pas forcément adaptée à ce qu'il pourrait être permis de faire (passage obligatoire par des fonctions de résolution).

Les variables partagées sont déclarées en utilisant une déclaration de variable classique précédée du mot-clé *"shared "* , dans n'importe quelle région déclarative, excepté dans les sous-programmes et les processus (c'est-à-dire là où peuvent être déclarées les variables classiques). Elles peuvent être affectées à l'intérieur des sous-programmes et des processus comme les variables classiques.

Cependant, ces variables sont à utiliser avec précaution car, dans le cas où elles sont accédées sans restriction, elles peuvent aboutir à des situations non-déterministes. En effet, si une variable partagée est accédée en écriture par plusieurs processus concurrents, sa mise à jour dépend de l'ordre d'exécution de ces processus. Cet ordre peut ne pas être imposé ou ne pas être garanti identique pour les différentes exécutions du même modèle. Un exemple de ce type de problème est illustré à la figure I.28. Ainsi, dans cet exemple la variable V peut avoir la valeur -1, 0 ou 1, mais aucune de ces valeurs ne peut être prédite. Cette valeur dépend du dernier processus exécuté. Malgré tout, ceci permet, par exemple, de modéliser des fonctions renvoyant un résultat aléatoire.

```
architecture A of E is
    shared variable V : INTEGER := 0;
begin
    P1: process
    begin
        V := V - 1;
        wait;
    end process;
    P2: process
    begin
        V := V + 1;
        wait;
    end process;
end A;
```

Figure I.28 : Exemple d'indéterminisme

Un groupe de travail s'est formé (Shared Variable Working Group) afin d'explorer les mécanismes possibles de protection pour éviter ces problèmes d'indéterminisme.

Il est possible cependant, d'éviter ces problèmes, à condition d'utiliser les variables partagées pour des domaines bien particuliers, tels que la programmation orientée objet (afin de connaître l'état interne d'un objet) ou encore la description de systèmes. Par exemple, elles peuvent servir à modéliser une pile ou encore la température de fonctionnement d'un circuit. Un exemple de modélisation d'une pile est donné dans [Berg 93b, pp.35]. Cette pile est modélisée à l'aide d'un paquetage où sont définies deux variables globales, représentant le contenu de la pile et son index, ainsi que des sous-programmes d'empilage et de dépilage utilisant ces variables.

3.3. Les nouveaux mécanismes de structuration

3.3.1. Instanciation directe

En VHDL'87, au niveau structurel, il est nécessaire de configurer chaque instance de composant, c'est-à-dire d'associer à chaque instance un couple entité-architecture. Ce mécanisme de configuration est d'ailleurs très puissant (cf. section 2.5.3.). Cependant il dénote quelques points critiques : il est lourd dans le cas de descriptions simples, il est syntaxiquement et conceptuellement complexe car il n'est pas naturel et d'autres langages concurrents à VHDL possèdent des mécanismes plus simples. Notons néanmoins qu'il existe une configuration par défaut normalisée consistant à prendre en compte la dernière architecture compilée lorsque les profils du composant et de l'entité sont identiques. VHDL'92 offre en plus le mécanisme d'instanciation directe de composants. Dans ce cas la déclaration du composant et la configuration ne sont plus nécessaires : le composant est directement configuré lors de son

instanciation. Un exemple reprenant celui qui avait été donné à la figure I.8. est fourni à la figure I.29.

```
architecture A of E is
    signal RE,RS : BIT;
    signal E1,E2,S : BIT_VECTOR (10 downto 1);
begin
    ADD_10 :    entity work.ADD_N(ARCH)
                generic map (10) port map (RE,E1,E2,S,RS);
end A;
```

Figure I.29 : Exemple d'instanciation directe de composant

Il est évident que ce nouveau mécanisme ne remplace pas l'ancien dans tous les cas : ce dernier reste encore très pratique pour adapter les ports et les paramètres des modèles existants et permet également d'être indépendant de la technologie car il est très simple de changer de bibliothèque. Cependant l'instanciation directe offre un mécanisme simple pour les débutants (pas de concepts de composant et de configuration), allège et rend plus lisible les descriptions où les modèles ne sont instanciés qu'une seule fois (partie opérative et partie contrôle d'un circuit), facilite l'écriture des fichiers de tests et permet une traduction directe des autres langages tels que Verilog, M, etc.

3.3.2. Association incrémentale

En VHDL'87, pour toutes les instances de composants, l'association des paramètres génériques et l'association des ports ne peuvent être spécifiées qu'une seule fois, soit par le biais d'une spécification de configuration au niveau d'une architecture, soit par une configuration de composant à l'intérieur d'une déclaration de configuration. Par contre, VHDL'92 offre la possibilité de séparer ces informations d'associations entre la spécification et la déclaration de configuration. Les indications d'associations données au niveau d'une spécification de configuration sont appelées indications d'associations primaires, alors que celles données au niveau d'une déclaration de configuration sont appelées indications d'associations incrémentales. Ceci signifie que les paramètres génériques qui ont déjà été associés à des valeurs effectives au niveau d'une spécification de configuration peuvent être associés à d'autres valeurs (même différentes) au niveau d'une déclaration de configuration. Ces dernières valeurs écraseront les précédentes. De plus, les ports n'ayant pas été associés à des ports effectifs au niveau d'une spécification de composant, peuvent être connectés plus tard, au niveau de la déclaration de configuration.

L'utilisation principale de l'association incrémentale concerne la spécification des paramètres temporels. Tout d'abord, dans un couple entité-architecture, les paramètres génériques temporels ont par défaut des valeurs unitaires; elles ne dépendent pas d'une instance donnée. Ensuite, dans l'architecture où se trouve instancié le couple entité-architecture, pour certaines instances particulières, des valeurs temporelles spécifiques sont associées aux paramètres génériques. Enfin, au niveau de la déclaration de configuration, de nouvelles valeurs temporelles peuvent être associées, plus précises, dépendantes d'une technologie donnée et résultat d'une rétro annotation.

3.3.3. Notion de groupe

La notion de groupe introduite par VHDL'92 est liée à la notion d'attribut. Les attributs définis par l'utilisateur sont largement utilisés pour donner des informations aux outils. Par exemple, par le biais des attributs, l'utilisateur peut guider la synthèse en donnant des contraintes. Cependant la notion d'attribut ne permet pas d'associer une information à un groupe d'objets. La notion de groupe comble ce manque : elle permet de lier entre eux plusieurs objets (signaux, fonctions, blocs, étiquettes, etc.). De plus, un attribut peut être spécifié pour un groupe.

Comme pour les attributs définis par l'utilisateur, la notion de groupe est utilisée pour passer des informations aux outils. Elle s'utilise plus particulièrement pour définir des chemins dans un circuit et y associer des informations temporelles, ou encore pour guider la synthèse au niveau de l'allocation des ressources, mais aussi pour donner des indications aux outils de placement et de routage. Un exemple d'utilisation de groupe est donné à la figure I.30. La première ligne défini le groupe CHEMIN, la seconde ligne spécifie un chemin : E_VERS_S, la troisième ligne défini l'attribut DELAI et la quatrième ligne spécifie la valeur de l'attribut DELAI associé au chemin E_VERS_S.

```
group CHEMIN is (signal, signal);
group E_VERS_S : CHEMIN (E, S);

attribute DELAI is TIME;
attribute DELAI of E_VERS_S : group is 5 ns;
```

Figure I.30 : Groupe utilisé pour spécifier un chemin et y associer un délai

3.4. Les nouveaux mécanismes d'interface

3.4.1. Architectures et sous-programmes étrangers

Une description de circuit est généralement décomposée en plusieurs parties, mais ces parties peuvent également être décrites en utilisant des langages différents : VHDL, Verilog, C, Ada, etc. Ce mélange est devenu aisé en VHDL'92 par l'introduction d'un nouveau attribut préalablement défini : l'attribut *"'FOREIGN"* défini dans le paquetage *"STANDARD"* . Cet attribut peut s'appliquer soit à des entités, soit à des sous-programmes, et indique respectivement que, soit l'architecture associé à l'entité, soit le corps du sous-programme, est étranger. Ce mécanisme permet évidemment d'utiliser des modèles et d'appeler des sous-programmes décrits par un langage autre que VHDL, mais il fournit également la possibilité de protéger des modèles décrits en VHDL comme par exemple des descriptions de bibliothèques. Cependant, l'utilisation de cet attribut limite la portabilité du modèle. De plus, dans le cas où cet attribut pourrait faire référence à des modèles décrits dans d'autres langages, aucun mécanisme de synchronisation entre ces langages n'a été défini.

3.4.2. Lecture et écriture de fichiers

VHDL'92 offre quatre sortes d'objets : les constantes, les signaux, les variables et les fichiers. Les fichiers sont particulièrement pratiques pour décrire le contenu d'une mémoire morte par exemple. Cependant, les capacités de VHDL'87 dans ce domaine sont pauvres et un certain nombre d'erreurs et d'ambiguïtés coexistent au niveau du LRM [IEEE 87]. VHDL'92 a permis de corriger ces erreurs et d'introduire deux nouvelles fonctionnalités : la procédure *"FILE_OPEN"* permettant de tester l'existence d'un fichier et la procédure *"FILE_CLOSE"* permettant de fermer un fichier dans le but de pouvoir l'ouvrir à nouveau. Ces procédures ont été rajoutées dans le paquetage *"TEXTIO"* .

3.4.3. Fonctions impures

En VHDL'87 il n'existe que des fonctions dites pures, c'est-à-dire renvoyant un résultat ne dépendant que des valeurs des paramètres d'entrée. En VHDL'92 sont introduites les fonctions impures. Ces fonctions peuvent avoir des effets de bord (fonctions sans paramètre, accès à des objets globaux externes, fonctions étrangères, etc.). Les déclarations de ces fonctions sont précédées du mot-clé *"impure "* .

3.5. Environnement préalablement défini : les nouveaux opérateurs, fonctions et attributs

3.5.1. Opérateurs de décalage et de rotation et opérateur *"xnor"*

Afin de compléter l'environnement préalablement défini, de nouveaux opérateurs sont disponibles : il s'agit des opérateurs de décalage et de rotation et de l'opérateur logique *"xnor"*. Il existe quatre opérateurs de décalage : *"sll"* (Shift Left Logical), *"srl"* (Shift Right Logical), *"sla"* (Shift Left Arithmetic) et *"sra"* (Shift Right Arithmetic) et deux opérateurs de rotation : *"rol"* (ROtate Left logical) et *"ror"* (ROtate Right logical). Ceux sont des opérateurs binaires dont l'opérande de gauche est un vecteur de bits ou de booléens et l'opérande de droite est un entier. Si l'entier est nul, aucune opération n'a lieu, si il est positif, l'opération de décalage ou de rotation est répétée ce nombre entier de fois, et s'il est négatif, dans ce cas l'opération de décalage ou de rotation a lieu dans le sens inverse et est également répétée ce nombre entier de fois. Ces opérateurs, comme tous les opérateurs VHDL, peuvent aussi être surchargés pour d'autres types ou pour changer la valeur entrante dans le cas des décalages. La valeur entrante par défaut est '0' ou fausse suivant le type de l'opérande de gauche (vecteur de bits ou de booléens).

3.5.2. Attributs préalablement définis *"'DRIVING_VALUE"* et *"'DRIVING"*

En VHDL'87 il n'est pas permis de lire un signal de sortie et ceci est parfois très limitatif. Pour palier à ceci, un nouveau attribut a été défini : *"'DRIVING_VALUE"*. Cet attribut renvoie la valeur du signal auquel il est appliqué et permet ainsi de récupérer la valeur d'un signal de sortie. L'attribut *"'DRIVING"* renvoie la valeur booléenne faux si le signal auquel il est appliqué a été déconnecté.

3.5.3. Attribut préalablement défini *"'ASCENDING"*

L'attribut *"'ASCENDING"* renvoie la valeur booléenne vraie si le rang du tableau auquel il est appliqué est ascendant et la valeur faux s'il est descendant. Dans le cas général, en VHDL'87, la même fonctionnalité peut être obtenue en comparant les valeurs retournées par les attributs *"'RIGHT"* et *"'LEFT"*. Cependant, ce moyen créé des problèmes dans les cas limites des tableaux nuls ou ne comprenant qu'un seul élément.

3.5.4. Attributs préalablement définis *"BEHAVIOR"* et *"STRUCTURE"*

Ces deux attributs qui sont préalablement définis en VHDL'87 ont été supprimés en VHDL'92 car ils n'étaient pas utilisés et ne correspondaient pas à un besoin réel des concepteurs.

3.5.5. Attributs préalablement définis *"IMAGE"* et *"VALUE"*

En VHDL'92 sont préalablement définis deux nouveaux attributs : *"IMAGE"* et *"VALUE"* permettant de convertir n'importe quel type scalaire ou sous-type (entiers, types énumérés, etc.) en chaîne de caractères et vice versa. Ces mécanismes facilitent par exemple la lecture et l'écriture de fichiers.

3.5.6. Attributs préalablement définis *"SIMPLE_NAME"* , *"PATH_NAME"* et *"INSTANCE_NAME"*

Les attributs *"SIMPLE_NAME"* , *"PATH_NAME"* et *"INSTANCE_NAME"* retournent tous une chaîne de caractères. Celle renvoyée par *"SIMPLE_NAME"* correspond au nom d'un objet. Celle retournée par *"PATH_NAME"* correspond à la liste entière des étiquettes du chemin et donne ainsi une information sur la hiérarchie. Celle renvoyée par *"INSTANCE_NAME"* inclue l'information donné par l'attribut *"PATH_NAME"* plus une information supplémentaire sur la configuration. L'utilisation de ces attributs permet de faciliter la mise au point d'un modèle car il devient ainsi aisé de repérer les endroits qui posent problèmes.

3.6. Améliorations mineures

Les principales améliorations apportées par VHDL'92 viennent d'être développées au cours des sections précédentes. D'autres améliorations de moindre importance ont également été introduites.

Elles concernent notamment l'instruction d'affectation de signaux. En VHDL'87, deux notions sont mélangées. L'instruction suivante *"A <= B after 10 ns;"* donne deux informations : la première indique que le signal A correspond au signal B retardé de 10 ns et la seconde signifie implicitement que tout pic dont la largeur est inférieure à 10 ns sera filtré. En VHDL'92, il est possible de rendre explicite cette seconde information en utilisant le mot-clé *"inertial"* . Ainsi l'instruction précédente devient : *"A <= inertial B after 10 ns;"* . Une nouvelle clause

optionnelle introduite par le mot-clé *"reject"* permet de fixer une limite différente de filtrage des pics. La figure I.31 illustre ces différents cas.

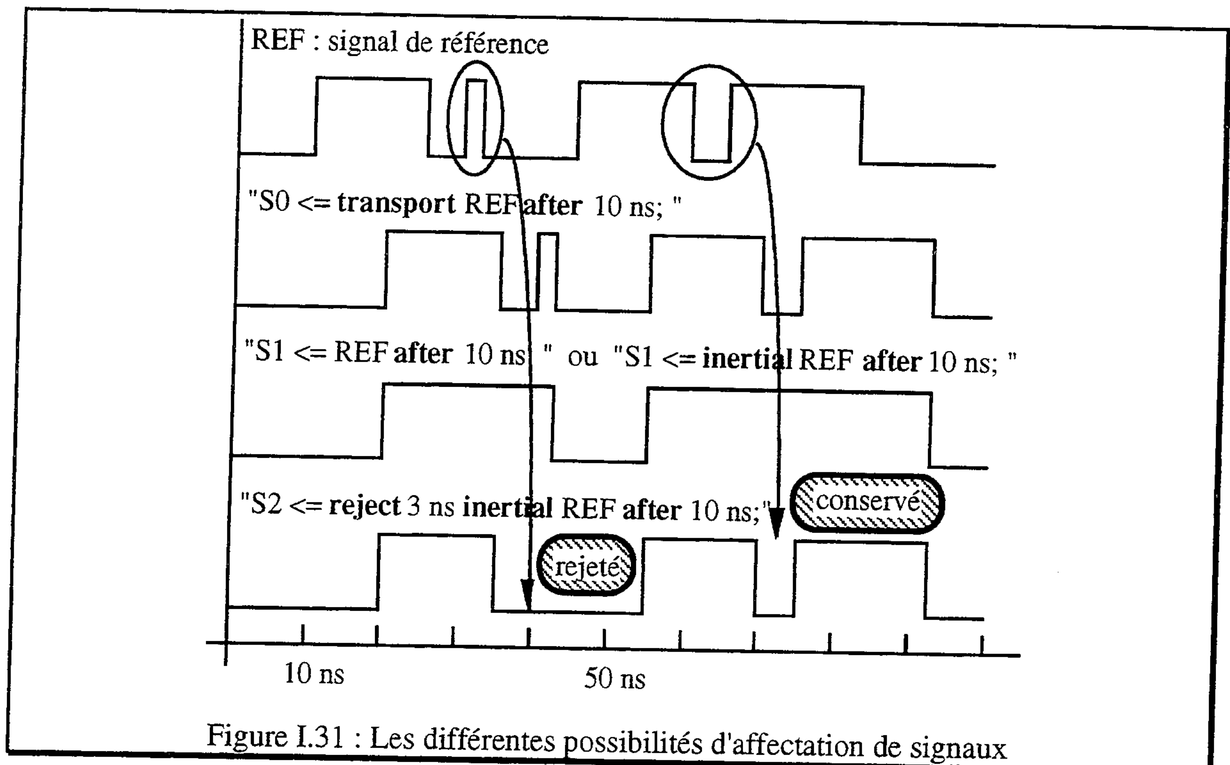

Figure I.31 : Les différentes possibilités d'affectation de signaux

Les affectations concurrentes conditionnelles de signaux ont été étendues et maintenant il est possible d'indiquer qu'un signal n'est pas affecté dans tous les cas. A titre d'exemples, les deux instructions suivantes sont équivalentes : *"S <= A when COND else unaffected;"* , *"S <= A when COND;"* et ces deux écritures sont correctes en VHDL'92.

Concernant l'instanciation de composants, il est maintenant possible d'associer directement des expressions à des ports, au niveau de la clause *"port map"* , comme le montre l'exemple suivant : *"C1 : NOR port map (A,'0',B);"* , alors qu'en VHDL'87 un signal intermédiaire devait être utilisé pour réaliser cela.

L'introduction de l'instruction séquentielle *"report"* facilite l'écriture de messages à l'intérieur du code VHDL afin d'obtenir des traces lors de l'exécution par le simulateur. Il s'agit en fait d'une simplification de l'instruction d'assertion qui requiert une condition et qui par défaut stoppe le simulateur.

VHDL'92 introduit également une généralisation des étiquettes aux instructions séquentielles, une extension de l'ensemble des caractères, une généralisation des identificateurs, une extension du concept d'alias, plus d'autres améliorations mineures concernant notamment la cohérence au niveau de la syntaxe du langage.

Pour plus de précisions sur toutes ces améliorations apportés par VHDL'92 par rapport à VHDL'87, il est possible de se reporter au LRM [IEEE 93] et à [Berg 93b].

4. VHDL par rapport aux autres langages

Bien que VHDL soit maintenant largement accepté et adopté, il n'est pas le seul langage de description de matériel. Pendant ces dernières trente années, comme il est dit dans [HDLs 92a] et [HDLs 92b], beaucoup d'autres langages ont été développés, ont évolués et sont encore utilisés aujourd'hui par les concepteurs de circuits intégrés. Créé pour être un standard, VHDL doit son succès à la fois à ces prédécesseurs et à la maturité de ces principes de base. Il est le résultat de nombreuses études sur les principes de la simulation logique et des langages associés [Cras 85].

La dernière partie de ce chapitre fournit des éléments de comparaison entre VHDL et deux autres langages de descriptions de matériels parfois aussi utilisés que VHDL : Verilog et M. Cette étude, largement développée dans [Berg 92] et [Magi 92], tient surtout à mettre en valeur les caractéristiques les plus fondamentales de VHDL : sa généralité et par là son manque de constructions préalablement définies fournies par d'autres langages plus spécifiques, mais aussi la mise à disposition de constructions puissantes permettant de nouvelles méthodologies de modélisation.

4.1. Historique

En ce qui concerne VHDL, un historique détaillé est donné en début de ce chapitre. Rappelons simplement que VHDL a été développé indépendamment de tout outil : c'est un standard. Par contre, M et Verilog ont été développés par des compagnies privées pour leurs propres besoins (langage de spécification pour leurs outils de simulation).

Verilog a tout d'abord été décrit par la société Gateway Design Automation qui a ensuite fusionné avec la compagnie Cadence Design Systems. Pour que Verilog puisse faire face à VHDL, Cadence a décidé de le rendre public en 1990. Bien que le nom "Verilog" soit une marque déposée par Cadence, le langage lui-même peut être accepté par n'importe quel outil de conception. L'utilisation de Verilog est promue par le groupe Open Verilog International (OVI) qui a publié en Octobre 1991 la première version du manuel de référence du langage Verilog [OVI 91].

Contrairement à Verilog, M [Ment 89] est toujours un langage privé. Il est la propriété de la compagnie Mentor Graphics et aucun signe aujourd'hui ne laisse à supposer que ce langage

sera transféré dans le domaine public un jour. Bien qu'il ne s'agisse pas de standard comme VHDL, Verilog et M sont cependant promus par deux des plus importantes compagnies dans le domaine de la conception assistée par ordinateur.

4.2. Caractéristiques générales

Du point de vue de leur syntaxe, VHDL s'est largement inspiré du langage ADA, Verilog ressemble au langage C ou Pascal et M est totalement fondé sur le langage C. VHDL est certainement le plus difficile à utiliser car il reste très général. De plus, il demande à l'utilisateur d'avoir des habitudes de programmeur (compilation séparée, langage fortement typé, notion de surcharge, etc.). Comparé à Verilog et M qui restent proches de la réalité physique, VHDL est sans aucun doute plus complexe.

VHDL est plus général que Verilog et M, et est donc mieux adapté à la description comportementale de circuits intégrés. Par contre Verilog et M s'utilisent plus facilement pour décrire des circuits au niveau logique voire même inférieur (transistor).

4.3. Différences au niveau de la modélisation

4.3.1. Niveaux de descriptions

Contrairement à VHDL, Verilog et M s'appliquent facilement aux descriptions de bas niveaux. M permet même de décrire des modèles analogiques. Ils fournissent tous les deux un ensemble préalablement définis de portes. Les constructions implantées dans ces langages sont très spécifiques à la modélisation de bas niveau des circuits intégrés.

Par contre, pour décrire des modèles à des niveaux supérieurs, VHDL est plus puissant. Il inclut des facilités de programmation qui n'existent pas dans Verilog et M et qui font de lui un langage adapté pour les modélisations de haut niveau. Il s'agit notamment des caractéristiques suivantes :
- langage fortement typé (vérification de la cohérence),
- compilation séparée,
- paquetages et bibliothèques (réutilisation),
- mécanisme de surcharge,
- types définis par l'utilisateur,
- fonctions de résolutions définies par l'utilisateur,
- séparation entre entité et architecture(s),

- architectures multiples pour une même entité,
- mécanismes de configuration (entités et composants).

4.3.2. Unités de conception

Comme nous l'avons vu, VHDL définit cinq unités de conception : l'entité, l'architecture, la déclaration de paquetage, le corps de paquetage et la configuration. Chacune de ces cinq unités peut être analysée séparément et mise dans une bibliothèque. Verilog et M ne fournissent que le module qui est équivalent à la paire entité-architecture : il n'y a pas de séparation entre la vue externe et interne du modèle, comme cela est possible en VHDL, et l'architecture interne du module est unique. De plus, les paquetages ne sont implantés ni dans Verilog et ni dans M.

4.3.3. Domaines concurrent et séquentiel

Généralement, les langages de descriptions de matériel définissent un domaine concurrent et un domaine séquentiel. Pour VHDL et Verilog cette définition est similaire. Les modules Verilog et les architectures VHDL incluent des instructions concurrentes. Dans les deux langages, les instructions concurrentes peuvent être utilisées pour les descriptions de types structurel, flot de données et comportemental. Les instructions Verilog *"always"* et VHDL *"process"* englobent toutes les deux des algorithmes séquentiels qui peuvent également être encastrés à l'intérieur des sous-programmes de ces langages.

Concernant le domaine concurrent de M, il est réduit à l'utilisation de sortes de processus UNIX : il n'y a pas de descriptions flot de données dans M. Les modules de ce langage renferment essentiellement un code séquentiel très proche du code C.

4.3.4. Objets et types

La généralité de VHDL aussi bien que la spécificité des autres langages de description de matériel sont particulièrement mises en relief par les différences au niveau des objets et des types qu'ils définissent. Quatre classes d'objets sont définies en VHDL : les signaux, les variables, les constantes et les fichiers ; chaque objet a un type donné qui peut être préalablement défini ou bien défini par l'utilisateur.

Par contre, Verilog ne fournit qu'un ensemble très restrictif de types. En particulier, les signaux sont implicitement définis avec le type logique classique et sont associés à l'une des fonctions

de résolution préalablement définies. En Verilog, il est donc impossible de définir des signaux de types abstraits (entier, etc.) ce qui est très pratique pour la modélisation de systèmes.

Le langage M n'offrait au début que des signaux ayant un type logique implicite. Par la suite, les définitions de signaux de types abstraits et de fonctions de résolutions ont été rendues possible. Les variables peuvent être déclarées à l'intérieur des modules avec n'importe quel type C.

4.3.5. Environnement préalablement défini

VHDL a un ensemble préalablement défini de types et de sous-programmes très pauvre comparé à celui de Verilog ou M. En réalité, VHDL peut être vu comme un équipement permettant de définir son propre langage de descriptions de matériel. La philosophie générale de ce langage est de laisser l'utilisateur construire tous les éléments spécifiques dont il a besoin : types particuliers, sous-programmes et opérateurs, entités et composants génériques et (ou) dépendant d'une technologie [Bert 91], etc.

Donc, une méthode efficace et nécessaire pour utiliser VHDL dans une société est de déterminer toutes les constructions qui sont spécifiques à l'application ou au domaine cible et de développer (ou faire développer) les bibliothèques VHDL correspondantes. Une fois que cet environnement est défini, alors il ne reste plus aux concepteurs qu'à réutiliser ces constructions. Étant donné les capacités de VHDL, un tel environnement peut même être plus efficace que l'environnement implicite fourni par les autres langages de descriptions de matériel.

4.3.6. Descriptions structurelles

Au niveau des descriptions structurelles, VHDL dispose d'un mécanisme de configuration très puissant que n'offre pas Verilog et M. En effet, ce mécanisme permet aux modèles d'être plus flexibles car ils peuvent être écrits indépendamment des entités utilisés, facilitant ainsi le passage d'une bibliothèque à une autre et plus généralement d'une technologie à une autre. Comme il a déjà été dit, des adaptations sont également possibles entre l'entité et le composant.

4.4. Conclusion

Quoiqu'il en soit, VHDL, Verilog et M permettent tous les trois de modéliser des circuits intégrés. Bien évidemment, ces trois langages n'ont pas exactement les même caractéristiques mais il existe toujours un moyen (même détourné) pour parvenir à ses fins. En ce qui concerne

sa puissance de modélisation, VHDL n'est pas vraiment adapté pour les descriptions de très bas niveau (analogique, transistors et portes) ; par contre sa supériorité s'affirme dans les descriptions de plus haut niveau (comportementale, fonctionnelle).

Les principales différences entre VHDL et les autres langages de descriptions de matériel sont essentiellement des différences de méthodologie. VHDL est un langage qui reste très général alors que Verilog et M sont davantage dédiés à la modélisation de circuits intégrés. L'environnement préalablement défini de VHDL pourra sembler très pauvre comparé à celui implicite des autres langages. Il y a donc nécessité de développer un environnement VHDL spécifique au domaine d'application avant de commencer à réellement utiliser VHDL dans un projet de conception. Une fois cet environnement défini, alors seulement, les capacités de VHDL pourront être comparées aux capacités offertes par les autres langages de descriptions de matériel plus spécialisés.

Enfin, le degré d'utilisation d'un langage dépend également du nombre et de l'efficacité des outils qui y sont associés. En particulier, dans le domaine de la simulation, l'efficacité des simulateurs VHDL, Verilog et M peut constituer un facteur déterminant lors du choix de l'utilisation de l'un de ces langages.

5. Conclusion

Après avoir exposé la nécessité d'un langage tel que VHDL et donné ses principales caractéristiques, après avoir étudié les améliorations apportées par VHDL'92 et après avoir souligné les points forts de VHDL par rapports aux autres langages de descriptions de matériel, dans le chapitre suivant, nous allons nous intéresser plus particulièrement à un de ses domaines d'application : la synthèse.

Chapitre II

-- ◊ - ◊◊◊ - ◊ --

Modélisations VHDL pour la synthèse

1. Généralités

1.1. Définition

La synthèse est définie comme une succession d'opérations permettant à partir d'une description de circuit dans un domaine fonctionnel (description comportementale) d'obtenir une description équivalente dans le domaine physique (description structurelle). Le processus de synthèse peut être défini comme une boîte noire ayant en entrée une description abstraite en termes de langages de description matériel, et comme sortie une description structurée en termes de dispositifs interconnectés, la traduction en termes de composants physiques (réalisables) étant manifeste.

1.2. La synthèse automatique de circuits : dans quel but ?

Le premier intérêt de la synthèse est de permettre une description la plus abstraite possible d'un circuit physique. Le concepteur a de moins en moins de détails à donner. Par exemple, pour décrire un compteur, la description détaillée des signaux de contrôle explicitement utilisés n'est pas indispensable : seule la fonctionnalité de comptage et les contraintes de synthèse doivent

être indiquées. Le but de l'abstraction est de réduire et de condenser les descriptions au départ et, par conséquent, de faciliter leur correction en cas d'erreurs.

Plus le niveau d'abstraction est élevé et plus la description est fonctionnelle. La synthèse, par respect avec ceci, aide à raccourcir le chemin partant de l'idée du concepteur pour aboutir au résultat physique. Néanmoins, il y a encore un long chemin à parcourir entre la description de la fonctionnalité et le choix de l'architecture qui permet de satisfaire les contraintes de façon optimale. C'est là où l'expérience du concepteur entre en jeu.

Générer automatiquement un circuit physique signifie aussi fournir un résultat de bonne qualité. Puisque les systèmes deviennent de plus en plus complexes, il ne deviendra bientôt plus possible d'envisager leur conception à la main. Avec la synthèse, le nombre d'informations devant être fournies par le concepteur diminue. Ces informations consistent essentiellement en la description comportementale du circuit et des contraintes correspondantes. La synthèse amène sans aucun doute à des circuits plus sûrs, plus robustes et devrait être considérée comme une marque de qualité dans le cycle de conception.

Puisque la synthèse permet de réduire la taille des descriptions, elle permet également de faciliter les remises à jour, de rendre les corrections plus rapides, et de pouvoir explorer un ensemble plus vaste de solutions architecturales. Dans ce contexte, le meilleur compromis entre coût et performance peut plus facilement être atteint par le concepteur.

Le grand nombre de descriptions déjà existantes couplé avec la possibilité de les paramétrer amènent à la création de ressources de bibliothèques réutilisables. Ceci permet d'améliorer encore plus la productivité des circuits électroniques.

Généralement, l'implantation physique du circuit obtenu après synthèse automatique est moins bonne que celle obtenue à la main (bien que les temps de conception ne soient pas comparables). Néanmoins, l'évolution des procédés technologiques amène à un accroissement de la complexité, dans une technologie donnée, tous les deux ou trois ans, et donc offrent de réelles possibilités pour la synthèse.

La complexité croissante des systèmes à concevoir, les exigences élevées de la qualité, la réduction du temps moyen de lancement d'un produit sur le marché et l'intégration automatique de matériel pour le test des circuits sont d'autres raisons qui impliquent l'utilisation des techniques de synthèse.

1.2. Les différents niveaux de synthèse

La figure II.1 schématise les différents niveaux de synthèse et les relations pouvant exister entre ces niveaux.

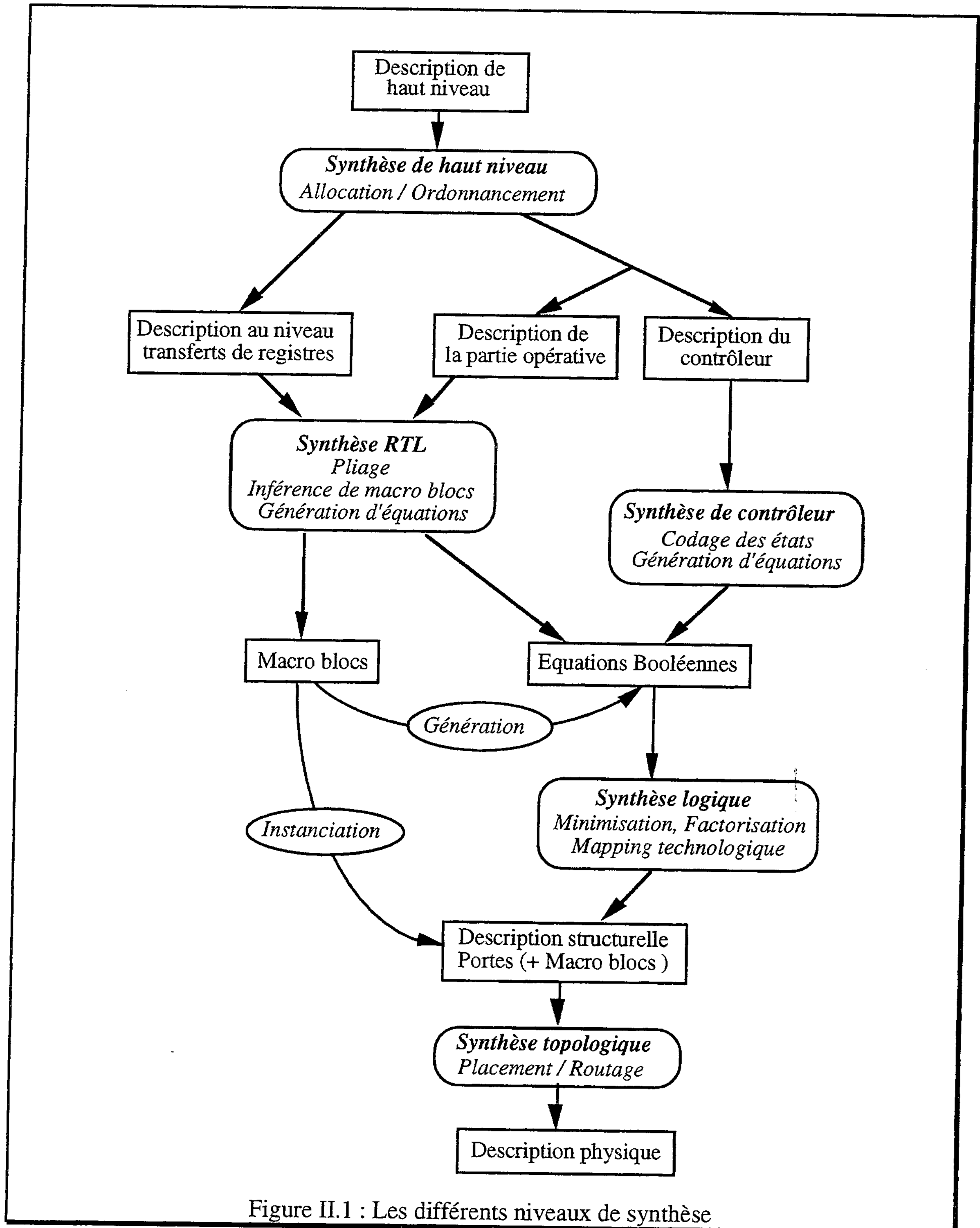

Figure II.1 : Les différents niveaux de synthèse

Ces différents niveaux de synthèse vont être succinctement décrits dans ce qui suit, par ordre décroissant d'abstraction, qui correspond en fait, à l'ordre inverse chronologique d'apparition dans le domaine de la conception assistée par ordinateurs.

1.2.1. Synthèse de haut niveau

Le point d'entrée d'un outil de synthèse de haut niveau est une description comportementale de circuit de type algorithmique. Généralement, cette description contient des objets qui n'ont aucune correspondance matérielle immédiate (par exemple, les variables et les opérations ne peuvent pas être directement associées à des registres et à des opérateurs). De plus, elles contiennent le plus souvent des relations temporelles entre ces opérations ; ces relations peuvent être exprimées en utilisant des constructions algorithmiques classiques telles que *"while"*, *"if"*, *"case"*, etc.

La synthèse de haut niveau comporte deux étapes dépendantes l'une de l'autre : l'ordonnancement des opérations et l'allocation des ressources. La phase d'ordonnancement consiste à affecter les opérations à réaliser à des coups d'horloge. La phase d'allocation des ressources comprend plusieurs tâches : la sélection du type de ressources dans une bibliothèque, la détermination du nombre de ressources de chaque type requis par la spécification initiale et l'assignation des ressources (opérateurs physiques, registres et connexions) aux opérations et variables. Ce type de synthèse a suscité un vif intérêt chez de nombreux chercheurs et a permis de mettre en place un grand nombre d'algorithmes permettant de résoudre toutes les étapes de la synthèse de haut niveau [Marw 86], [Farl 88], [Paul 88], [Camp 89], [Mign 92].

Selon les outils de synthèse de haut niveau considérés, le résultat de synthèse est généralement une spécification au niveau transferts de registres où le découpage entre une partie opérative et une partie contrôle n'est pas forcément explicite. Dans ce cas, le choix de l'architecture est laissé à l'outil de synthèse RTL prenant le relais. Mais ce résultat peut également être constitué de la description d'une partie opérative réalisant les opérations et de la description du contrôleur associé ; dans ce cas, l'architecture est imposée par l'outil de synthèse de haut niveau.

1.2.2. Synthèse au niveau transferts de registres (ou synthèse RTL)

Comme son nom l'indique, la description du circuit spécifie des transferts de registres à registres à travers d'éventuels opérateurs et des barrières temporelles entre ces transferts. Les différentes approches qui peuvent être utilisées lors de la synthèse RTL ont été abordées dans

plusieurs travaux de recherche [Bert 92a], [Mign 93], [Rizz 93], [Safi 95], mais nous ne nous étendrons pas davantage sur ce point.

D'une manière générale, le résultat obtenu après synthèse est constitué d'un ensemble d'équations Booléennes et de macro blocs (additionneurs, registres, compteurs, accumulateurs, etc.).

Rappelons que le contexte général de cette thèse se situe bien dans ce domaine de synthèse, mais les points essentiels que nous allons aborder par la suite, concernent la modélisation et l'utilisation de macro blocs au niveau de la synthèse RTL. Les termes de pliage et d'inférence seront définis ultérieurement dans ce chapitre.

1.2.3. Synthèse de contrôleurs

Les outils de synthèse de contrôleurs partent d'un graphe d'états et calculent le codage optimisé des états avec plusieurs choix pour l'architecture finale : PLA, cellules standards, compteurs ou architectures microprogrammées. Pour ce type de synthèse, les objets de base sont les états et les transitions entre états, les sorties étant associées soit aux états, soit aux transitions [Duff 91], [Gerb 94].

1.2.4. Synthèse d'équations Booléennes

Les fonctions devant être réalisées par le circuit sont décrites sous formes d'équations Booléennes. Le résultat obtenu est une description structurelle du circuit en termes de portes de base de la technologie ciblée. Cette description peut également contenir des macro blocs issus de la synthèse RTL.

Il existe dans le commerce des outils puissants (ASYL+, Compass, Synopsys, etc.) qui permettent de réaliser des listes d'interconnexions de cellules standards (description des éléments contenus dans un circuit et des connexions entre ces éléments) ou bien des réseaux programmables PLD (Programmable Logic Devices) ou FPGA (Fiel Programmable Gate Array). Ces outils possèdent tous les algorithmes nécessaires pour les étapes les plus basses du procédé de conception [Sica 88] : minimisation, factorisation [Abou 92], projection structurelle technologique (ou "mapping" technologique) et optimisation temporelle de réseaux Booléens [Sako 93].

1.2.5. Synthèse topologique

Cette dernière étape dans le flot général du processus de synthèse consiste, à partir d'une description structurelle du circuit, à en obtenir une représentation physique (dessins des masques, etc.). C'est à ce niveau qu'interviennent les techniques de placement et de routage. Le résultat obtenu peut alors être directement utilisé pour la fabrication du circuit dans le cadre des ASICs (Application Specific Integrated Circuits), ou la programmation du boîtier dans le cadre des circuits programmables.

1.3. Les limitations du langage VHDL pour la synthèse

Le langage VHDL ayant été défini en premier lieu pour la simulation, son utilisation en tant que langage d'entrée d'outils de synthèse présente certaines limitations [Liss 88].

La première remarque qui peut être faite est la suivante : le langage VHDL et le domaine de la synthèse sont en apparente contradiction. En effet, VHDL est un langage fondamentalement asynchrone ; la simulation est conduite par les événements et le cycle de simulation (très souvent noté delta : ∂) est infinitésimal. Les actions ont donc lieu à des instants discrets dans le temps et peuvent impliquer d'autres actions à des instants discrets dans le temps. Par contre, la synthèse automatique de circuits ne s'applique aujourd'hui qu'aux circuits synchrones par rapport à un ou plusieurs signaux d'horloge, pouvant être des machines d'états finis ou des extensions de ces dernières. Dans le domaine de la synthèse automatique, les sections asynchrones pouvant être prises en compte sont très limitées (parties concernant la mise à zéro ou la mise à un d'un circuit). La synthèse, elle, travaille au niveau du cycle d'horloge.

VHDL étant un langage orienté vers la simulation, les modèles VHDL peuvent contenir des informations qui n'ont de sens que pour la simulation (comme des informations temporelles détaillées telles que *"A <= B after 12 ns"*). Concernant ce point, la seule solution est de filtrer ces informations. Cependant, il n'est pas possible de décider, uniquement en se référant à la description, si ces informations de temps font partie des spécifications pour la simulation, ou bien s'il s'agit d'une contrainte pour la synthèse, et dans ce cas, s'il s'agit d'une valeur minimale, maximale ou typique. Une première conclusion peut-être faite à ce stade : l'utilisation de VHDL en tant que langage de synthèse conduit manifestement, tout d'abord, à la définition d'un sous-ensemble du langage VHDL [Bert 92b].

Un autre problème peut être soulevé : en utilisant VHDL, il est possible d'écrire plusieurs modèles pour le même circuit. Ces modèles sont équivalents en termes de résultats de simulation, mais ils amènent à des résultats différents lors de la synthèse. Ainsi, par exemple,

pour décrire un contrôleur et pour en faire la synthèse avec un outil particulier, il faudra décrire ce contrôleur en suivant le modèle reconnu par cet outil. Si le contrôleur est décrit différemment, tout en respectant le sous-ensemble défini par l'outil de synthèse utilisé, le contrôleur sera synthétisé, mais le résultat obtenu ne sera pas optimisé car l'outil n'aura pas pu reconnaître un contrôleur. Donc, pour remédier à ce problème, l'utilisation de VHDL en tant que langage d'entrée pour la synthèse implique la définition de modèles pouvant être reconnus par les outils de synthèse [Belh 93].

Une autre limitation provient des objets, des types ou des instructions définis dans le langage VHDL n'ayant aucune relation avec le matériel. Il s'agit par exemple des objets tels que les fichiers (*"file"*) ou des types tels que les pointeurs (*"access"*), ou encore des instructions comme celles d'assertion (*"assert"*), etc. De plus, le langage VHDL permet de décrire des algorithmes tout comme un langage de programmation. Les notions de conditions, d'itération, d'indexage, de récurrence et d'indirection connues en programmation peuvent être décrites en VHDL par les instructions *"if"* , *"case"* , *"loop"* , etc. Lorsque ces instructions sont utilisées, il est souvent difficile de leurs trouver une correspondance directe avec le matériel, mais ce problème peut être résolu par l'utilisation d'algorithmes d'allocation et d'ordonnancement mis en place dans les systèmes de synthèse de haut niveau [Farl 88], [Land 94].

1.4. VHDL, un langage pour la synthèse : oui ou non ?

L'objectif premier du langage VHDL n'était pas la synthèse. Comme il vient d'être dit, toutes les constructions définies dans ce langage ne sont pas synthétisables. Cependant, il est tout à fait possible de définir un langage dans lequel toutes les constructions sont synthétisables. Donc, est-ce que VHDL peut réellement être utilisé pour la synthèse ?

Le langage VHDL est capable de supporter le processus complet de synthèse. Mais pour utiliser VHDL en tant que langage de description de circuits pour la synthèse, il est nécessaire auparavant, de définir un sous-ensemble du langage rejetant tout ce qui n'a pas de sens pour la synthèse. Il est impératif de définir aussi un ensemble de modèles ou de règles de description permettant de garantir un comportement synchrone. Ces modèles pouvant être facilement reconnus par l'outil de synthèse, le circuit obtenu n'en sera que davantage optimisé. Cependant, pour pouvoir accepter le langage VHDL en tant que langage d'entrée, les outils de synthèse sont contraints à respecter la sémantique VHDL afin de permettre une comparaison au niveau des résultats de simulation entre la spécification initiale du circuit à synthétiser et le résultat de la synthèse. Il est bien évident qu'un outil de synthèse ne respectant pas cette contrainte n'a aucune chance de voir le jour sur le marché. Attention tout de même, par respect de la

sémantique VHDL, il est attendu que les résultats de simulation entre la spécification initiale et le circuit synthétisé soient équivalents au niveau du cycle d'horloge.

Ces conditions étant posées, et VHDL étant un standard depuis 1987, de nombreux outils de synthèse se sont développés et figurent actuellement sur le marché de la conception assistée par ordinateurs : ASYL+, Synopsys, Compass, Racal-Redac, Examplar, etc. Chacun de ces outils possède son propre sous-ensemble VHDL ([ASYL], [COMP], [SYNO]) mais bien souvent ces sous-ensembles sont différents et incompatibles ce qui ne permet pas aisément la portabilité des descriptions VHDL.

De plus, pour décrire un même circuit, les modèles acceptés par les outils de synthèse diffèrent très souvent selon l'outil de synthèse utilisé. Actuellement, ces modèles ne sont pas normalisés. Lors de la spécification du nouveau standard VHDL en 1992, une des principales demandes qui a été faite par les utilisateurs de VHDL pour la synthèse, était d'ajouter une instruction spécifique pour décrire les machines d'états finis. Cependant, cette demande n'a pas été prise en compte dans VHDL'92. En effet, beaucoup de méthodes existent déjà pour décrire des machines d'états finis et en ajouter une nouvelle aurait été redondant.

Il faut savoir aussi que la description VHDL d'un système pour en faire la synthèse comprend aussi la description VHDL d'un test permettant de simuler ce système. Ce test fait partie intégrante de la description ; il est indispensable. Il décrit l'environnement du système à synthétiser et permet ainsi de le tester. La séparation entre le système à synthétiser et le test est importante. Une fois le système synthétisé, le même test peut être réutilisé pour simuler le résultat du processus de synthèse et en vérifier sa cohérence et sa robustesse. Ceci suppose évidemment que l'outil de synthèse considéré puisse délivrer un résultat sous la forme d'une description VHDL pouvant être une liste d'interconnexions de portes, également appelée "netlist", décrite structurellement. De même, les descriptions VHDL des éléments constituant cette netlist doivent également être fournies de sorte à pouvoir simuler la netlist en question. La figure II.2 illustre cette méthodologie.

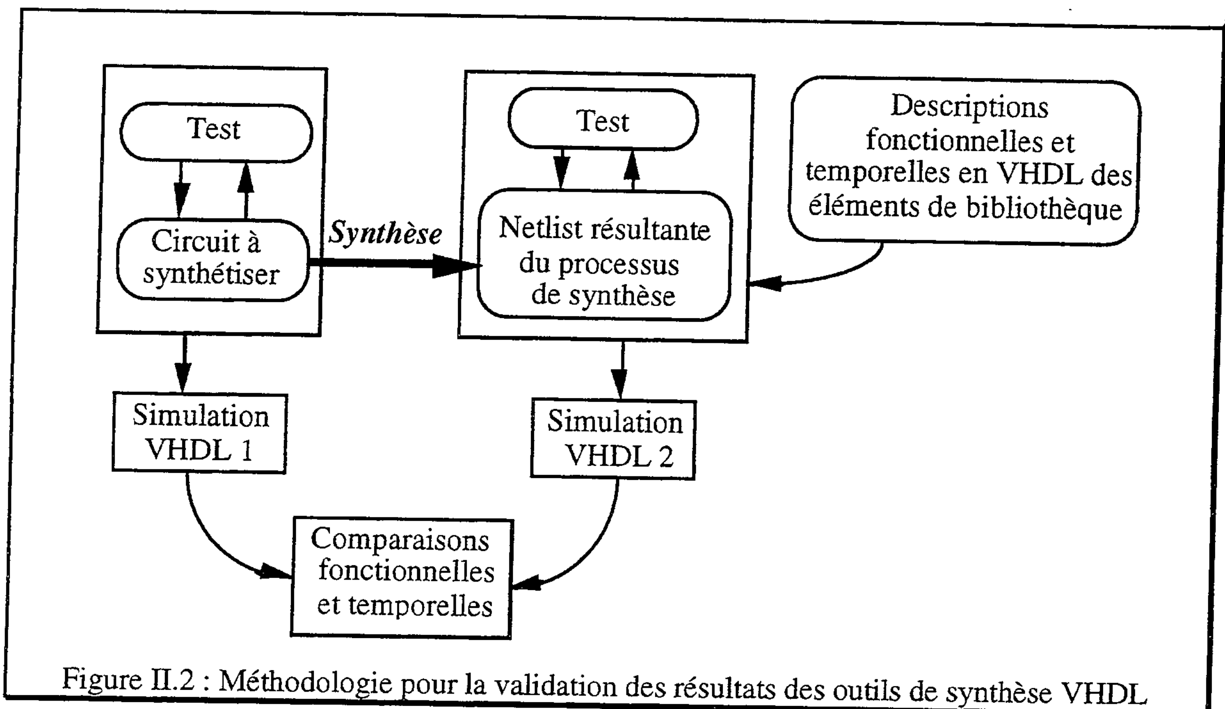

Figure II.2 : Méthodologie pour la validation des résultats des outils de synthèse VHDL

La description d'un test est souvent vue comme une énumération de stimuli, mais bien souvent elle peut être beaucoup plus complexe et contenir la description de systèmes très sophistiqués. Cependant, toute la puissance du langage VHDL peut être utilisée pour décrire les tests puisqu'ils ne seront pas synthétisés.

En dernier point, il est à noter que les techniques de synthèse vont encore beaucoup évoluer durant les années à venir. Ceci est plus particulièrement vrai pour la synthèse comportementale (RTL et de haut niveau) puisque les outils traitant ce type de synthèse apparaissent depuis peu de temps sur le marché. Là encore, il est évident que le langage VHDL, choisi en tant que langage d'entrée d'outils de synthèse, de par sa généralité et sa puissance, ne bloquera pas les progrès qui restent à faire dans le domaine de la synthèse à n'importe quel niveau d'abstraction.

1.5. Méthodologie de conception d'un circuit à partir d'une description VHDL

Maintenant, la question est de savoir comment intégrer le processus de synthèse à partir d'une spécification de circuit en VHDL, à l'intérieur d'une méthodologie de conception déjà existante [Airi 94]. Les questions clés peuvent se résumer ainsi :

Quelles sont les étapes d'une telle méthodologie ?

Que peut-on réellement attendre des outils de synthèse ?

Quelles sont leurs caractéristiques principales ?

1.5.1. Les étapes du cycle de conception

Elles peuvent être divisées en étapes de conception proprement dites et en étapes de validation. Les étapes de validation sont essentielles afin d'assurer que le circuit physique produit possède bien les propriétés désirées, quelle que soit la qualité des outils de synthèse. Cependant, des différences peuvent apparaître au niveau des comportements avant et après synthèse.

La figure II.3 illustre le cycle de conception. Un tel cycle peut être utilisé aussi bien pour les ASICs (Application Specific Integrated Circuits) que pour les circuits programmables de type FPGA (Field Programmable Gate Arrays) ou CPLD (Complex Programmable Logic Devices).

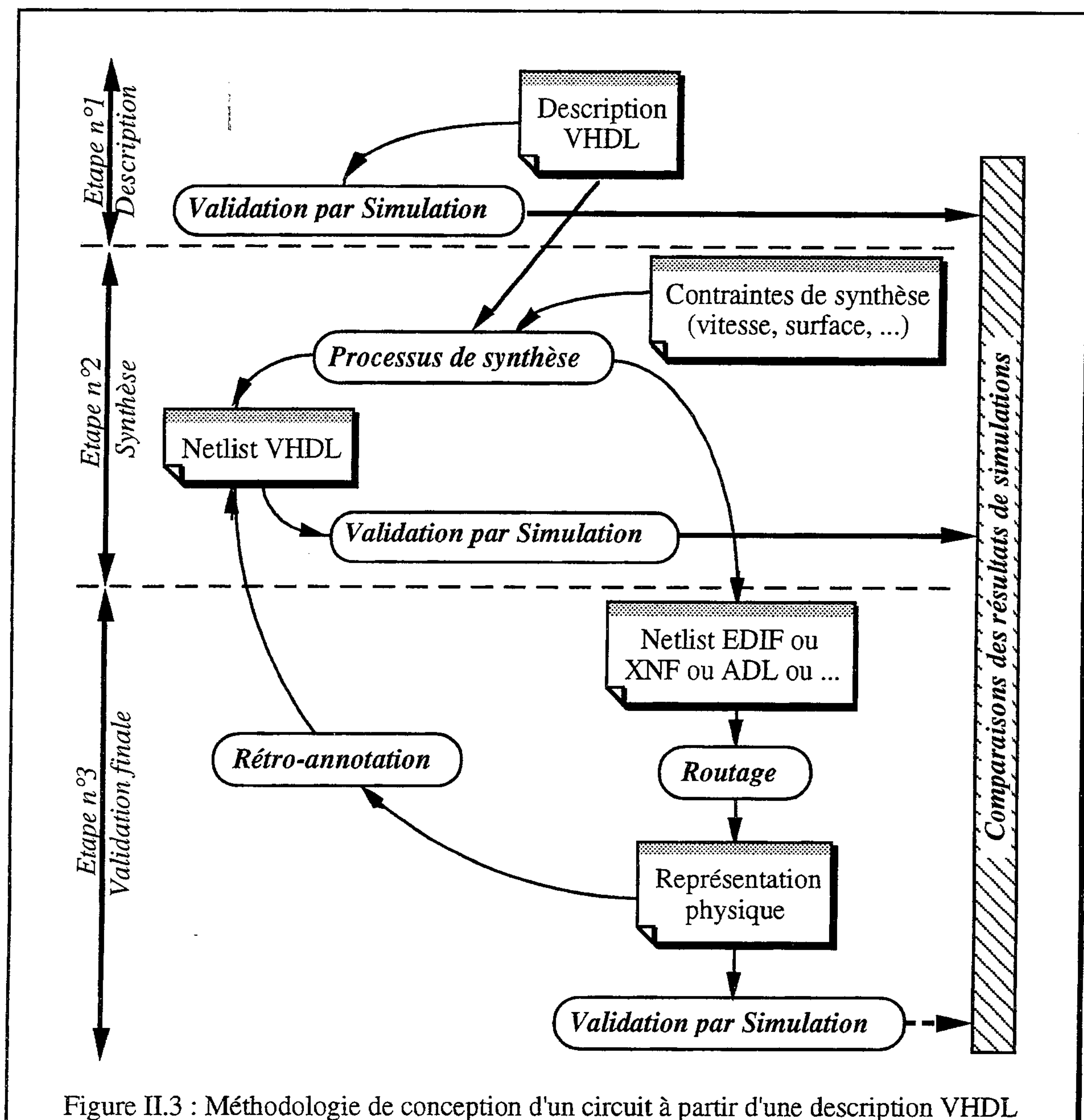

Figure II.3 : Méthodologie de conception d'un circuit à partir d'une description VHDL

1.5.2. La description du circuit pour la synthèse

La description du circuit est la première étape. A ce niveau, les spécifications doivent être claires et sont codées en VHDL. Le style de description utilisé est important et cette tâche de transcription en VHDL consiste essentiellement à trouver un bon compromis entre le niveau d'abstraction et la précision de la description. Le choix des types de données (entiers, types énumérés, etc.) et le choix des constructions VHDL (sous-programmes, instructions de boucle, etc.) sont essentiels.

Plus la description est abstraite et plus sa validation, sa maintenance et sa compréhension par quelqu'un d'autre sont aisées. Une description plus précise aboutira à plus de détails. Cependant, généralement ces détails ont un effet négatif sur la lisibilité et la maintenance de la description, mais peuvent accroître de façon significative le contrôle de l'étape de synthèse en vue d'obtenir le résultat escompté. Très souvent, plusieurs passages sont nécessaires, entre les phases de description et d'analyse des résultats de simulation après synthèse, pour affiner (et donc rendre moins abstraite) une partie de la description initiale, en vue d'obtenir en contrepartie un résultat optimal après synthèse.

La description finalement obtenue est validée par simulation. Pour cette tâche, toute la puissance du langage VHDL pour décrire l'environnement de simulation spécifique à l'application, c'est-à-dire le test, est à la disposition de l'utilisateur. VHDL a le gros avantage de fournir de telles possibilités : dans certains domaines, la description du test (génération des stimuli, analyse de la trace) peut être complexe et aboutir à plus de lignes de code que le modèle lui-même.

1.5.3. Le processus de synthèse

La seconde phase du cycle de conception est le processus de synthèse. Le code VHDL, validé pendant la phase précédente, est traduit en une netlist générique. Les éléments de mémorisation ainsi que les opérateurs apparaissent explicitement. Le comportement tout entier est maintenant décrit par des appels à des éléments d'une bibliothèque générique. La richesse de cette bibliothèque est directement reliée au savoir-faire de l'outil.

Ce concept de netlist générique permet d'autres transformations liées aux contraintes données par l'utilisateur. Par exemple, une addition est implantée en utilisant un additionneur à retenue bondissante (Ripple Carry Adder) si les contraintes de vitesse sont assez faciles à respecter. Un additionneur avec calcul anticipé de la retenue (Carry Look Ahead Adder) est utilisé dans le cas contraire.

Des optimisations spécifiques sont réalisées sur les équations Booléennes. Leurs buts sont multiples. Quand le but est de réduire la surface du circuit, l'ensemble des équations Booléennes est minimisé en groupant les termes communs et en simplifiant. Quand l'idée est de parvenir à un meilleur contrôle des performances de temps du circuit, le nombre de niveaux des portes logiques est réduit même si cette transformation entraîne la duplication de certaines expressions (et donc accroît la surface du circuit).

Finalement, la description résultante doit être projetée structurellement sur la bibliothèque physique ciblée. Cette étape est appelée "mapping technologique". Le résultat du processus de synthèse doit respecter les contraintes de vitesse et de surface. Généralement, les algorithmes du mapping technologique considèrent les contraintes de temps en priorité par rapport aux contraintes de surface : les contraintes de temps doivent strictement être respectées tout en occupant la plus petite surface possible.

A ce stade, les outils de synthèse sont capables de proposer une première estimation du coût (en termes de surface) des opérations de placement et de routage. Cette estimation permet d'avoir des résultats plus réalistes.

Le résultat du processus de synthèse est une netlist qui sert de point d'entrée aux outils de placement et de routage. Le format EDIF [EDIF], un format standard créé initialement pour des échanges de descriptions de schémas, est généralement utilisé pour coder cette netlist.

Cependant, cette netlist peut aussi être utilisée pour valider le résultat de synthèse. Cette validation se fait habituellement par simulation. Deux approches différentes sont possibles pour cette simulation : une simulation dans le contexte spécifique des outils d'aide à la conception automatique de circuits, ou une simulation réutilisant le test VHDL déjà décrit pour la validation du circuit avant synthèse. Cette seconde approche est à choisir en priorité puisqu'elle évite de réécrire les stimuli. Cette approche implique que l'outil de synthèse soit capable de générer une netlist VHDL et fournisse les descriptions VHDL des éléments physiques de la bibliothèque. Ces descriptions VHDL doivent donner le comportement en fonction du temps de ces éléments (caractéristiques de temps des éléments des bibliothèques physiques).

La simulation après synthèse permet de vérifier que la fonctionnalité désirée est obtenue mais fournit aussi une première estimation du comportement en fonction du temps. Quelles sont les différences possibles entre les résultats de simulation avant synthèse et ceux après synthèse ?

Si le type BIT est utilisé, très peu de différences sont attendues. Si les comportements avant et après synthèse ne sont pas en accord, la fréquence de l'horloge probablement trop élevée, doit

être vérifiée. Réduire considérablement la fréquence de l'horloge pour vérifier la fonctionnalité est un moyen efficace d'exclure (ou non) un problème lié à l'horloge. Malheureusement, si les désaccords persistent, il n'existe aucune bonne stratégie pour isoler systématiquement la source du problème. Statistiquement, un grand nombre d'erreurs restantes à cette étape de la conception est lié à une mauvaise initialisation des éléments de mémorisation. Pour être sûr de la cohérence des descriptions avant et après synthèse, il est néanmoins essentiel de forcer la valeur initiale des éléments de mémorisation en question. Cette opération doit être effectuée dans la description avant synthèse. Elle peut paraître fastidieuse mais c'est le prix à payer pour assurer la qualité du circuit synthétisé.

1.5.4. La validation finale

Cette validation a lieu après les opérations de placement et de routage. Son but est de vérifier que la fonctionnalité initiale a été respectée et que les caractéristiques de temps du circuit généré satisfont les contraintes de temps. Deux méthodes pour parvenir à ce but sont possibles :

- réaliser la simulation dans l'environnement propre. Dans ce cas, le simulateur utilise les valeurs de chargement des noeuds calculées après les opérations de placement et de routage.
- rétro annoter la netlist VHDL résultant du processus de synthèse avec les caractéristiques déduites des opérations de placement et de routage. Le gros avantage de cette méthode est de pouvoir rester dans le domaine de la simulation VHDL.

Il est intéressant de noter que les informations fournies par les outils de placement et de routage peuvent être très utiles pour les outils de synthèse. Par exemple, des informations plus précises sur le chargement d'un port donné peut permettre une optimisation supplémentaire. Cependant, les résultats obtenus après répétition de la boucle synthèse puis routage peuvent ne pas converger. Pour éviter ce problème, une nouvelle approche est maintenant proposée par les derniers outils de synthèse. Leur stratégie consiste essentiellement à fortement coupler les outils de synthèse avec ceux de placement et de routage. Les évaluations des outils de placement et de routage sont accessibles au travers du processus de synthèse et sont prises en compte pendant la synthèse au niveau des optimisations. Finalement, l'outil de synthèse ne génère pas seulement une netlist, mais aussi un ensemble de contraintes de placement et de routage dans le but d'optimiser encore et encore le chemin critique.

1.6. Contrôle du processus de synthèse

1.6.1. Contrôle du processus de synthèse : de quoi s'agit-il ?

Le modèle VHDL est l'entrée essentielle des outils de synthèse : il donne une description du comportement désiré. Puisque le circuit physique doit aussi respecter d'autres contraintes, un moyen pour exprimer ces dernières est nécessaire. Cependant, aucun langage standardisé pour faire cela n'est disponible et chaque outil de synthèse offre sa propre solution. Quelle que soit cette solution, deux types de contraintes sont généralement rencontrées :

- des contraintes sur les caractéristiques physiques du circuit (contraintes de temps, de surface, de chargement),
- des contraintes (options) sur le flot de conception lui-même (formats des interfaces, contrôle de la hiérarchie).

1.6.2. Contraintes sur les caractéristiques physiques du circuit

Lorsque aucune contrainte n'est fournie au processus de synthèse, les outils de synthèse minimisent le coût du circuit généré. Il existe différentes façons de calculer (ou d'évaluer) le coût du circuit généré. Lorsque le circuit est un ASIC, ce coût est simplement lié à la surface et est exprimé soit en micron carré soit en nombre de portes de base ou portes équivalentes (inverseurs de taille minimale ou porte "et" à deux entrées par exemple). Si le circuit doit être implanté sur un composant programmable (de type FPGA ou CPLD), l'évaluation du coût est plus complexe puisque sa valeur n'est pas continue : la cible est un ensemble de circuits standards, et l'un d'entre eux doit être choisi. Le nombre d'éléments de base ne constitue pas l'unique critère pour ce choix : une évaluation du placement et du routage doit aussi être prise en compte.

Il existe beaucoup de contraintes physiques. Certaines d'entre elles permettent de décrire l'environnement dans lequel fonctionnera le circuit, les autres indiquent les performances qui devront être obtenues.

La première catégorie permet de fournir les informations suivantes :

- la valeur des capacités de chargement sur les ports d'entrée et de sortie,
- la valeur des résistances des ports d'entrée et de sortie,
- l'intervalle de température dans lequel doit fonctionner le circuit,
- la tension d'alimentation.

Dans la seconde catégorie, les informations suivantes peuvent être données :

• le temps minimum ou maximum entre des entrées et sorties spécifiques,

• le délai de propagation minimum ou maximum sur un chemin donné,

• la fréquence d'horloge,

• la différence de phase acceptable entre tous les noeuds propageant le même signal d'horloge à l'intérieur du circuit tout entier (ce phénomène de déphasage est plus communément appelé "skew" d'horloge). Cette contrainte doit être prise en compte par la stratégie du placement et du routage et au moment du choix de la taille des amplificateurs.

1.6.3. Formats d' interfaces

Différents formats et langages peuvent être rencontrés au moment de la description des entrées du processus de synthèse.

Un modèle comportemental du circuit à synthétiser est généralement écrit en VHDL ou Verilog. Plus d'informations sur les différences entre ces deux langages sont données au chapitre premier.

Si le but du processus de synthèse est de traduire un circuit déjà existant dans un autre format, le format d'entrée peut être le format EDIF ou une description VHDL structurelle. Il est à noter que dans le cas particulier où les technologies de départ et d'arrivée sont différentes, cette opération de transcription est appelée "reciblage" technologique (ou "migration").

Parmi ces formats d'entrée figure celui de la description de la bibliothèque physique (contenant les éléments caractérisés). Malheureusement, il n'existe pas réellement encore de standard pour décrire ces bibliothèques physiques, mais des travaux dans cette direction sont en cours [VITAL]. Donc, des formats propres à chaque outil sont utilisés. Bien souvent ces formats sont déposés et confidentiels et ne peuvent donc pas être réutilisés.

Il n'existe pas non plus de standard au niveau de l'expression des contraintes fournies à l'outil de synthèse. Elles apparaîtront sous la forme d'attributs VHDL définis par les outils de synthèse, de commandes spécifiques ou de commentaires syntaxiques.

Les formats d'interfaces suivant sont utilisés pour décrire les sorties du processus de synthèse :

• La netlist, résultat du processus de synthèse, est généralement fournie en EDIF pour les outils de placement et de routage, et en VHDL (ou Verilog) dans le cadre d'une simulation après synthèse.

• Les contraintes de placement et de routage n'ont pas de format standard. Des formats

propres à chaque outils sont généralement proposés.

• Les estimations approximatives des caractéristiques du placement et du routage telles que les temps de traversés des noeuds et des cellules sont stockées en utilisant le format standard SDF (Standard Delay Format [SDF]).

1.7. Environnement VHDL pour la synthèse

Comme nous l'avons vu dans le premier chapitre, l'environnement préalablement défini de VHDL est très pauvre comparé à celui des autres langages de descriptions de matériel. Donc, pour pouvoir utiliser VHDL dans le domaine de la synthèse, un certain nombre de paquetages standards ou en cours de standardisation sont décrits en VHDL. Notons que ces paquetages ne font pas partie du langage et donc ne sont pas définis dans le LRM.

1.7.1. Paquetage *"STD_LOGIC_1164"*

Un paquetage définissant un type énuméré composé de neuf valeurs, ainsi que les fonctions de résolution et les opérateurs logiques associés, a été voté en 1992 et a donné lieu au paquetage standard *"STD_LOGIC_1164"* . Ce paquetage défini le type *"STD_ULOGIC"* ainsi que le sous type résolu correspondant *"STD_LOGIC"* tel que le montre la figure II.4.

```
package STD_LOGIC_1164 is
    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     );
    subtype STD_LOGIC is RESOLVED STD_ULOGIC;
    ...
end STD_LOGIC_1164;
```

Figure II.4 : Définition du type "STD_ULOGIC" et du sous-type "STD_LOGIC".

Les valeurs 'X', '0' et '1' prédominent sur les valeurs 'W', 'L', et 'H' qui elles-mêmes prédominent sur la valeur 'Z'. Les valeurs '1' et '0' représentent des tensions proches respectivement de la tension d'alimentation et de la masse. Les valeurs 'U', 'X', 'W' et '-' sont dites valeurs métalogiques ; elles remplissent une fonction au niveau de la simulation mais elles n'ont pas de sens physique. La valeur 'U' est la valeur d'initialisation par défaut. Les valeurs 'X' et 'W' représentent les états qui ne peuvent pas être déterminés par le simulateur. Pour

l'instant, 'L' et 'H' n'ont pas de sémantique standard pour la synthèse puisque les outils de synthèse ne sont pas encore capable de faire la différence entre les forces des valeurs ('0' est plus fort que 'L', etc.). La valeur 'Z' peut être utilisée en simulation comme le résultat d'une fonction de résolution lorsqu'aucun pilote n'est activé. Pour la synthèse, l'utilisation de la valeur 'Z' implique la création de porte(s) trois-états. Enfin, la valeur '-' est généralement interprétée par les outils de synthèse comme une valeur sans importance pour le concepteur (plus communément appelées "don't care") et elle est remplacée soit par '0', soit par '1' afin d'optimiser le circuit. La sémantique exacte de toutes ces valeurs pour la synthèse est définie par le groupe IEEE SSIG.

1.7.2. Groupe IEEE travaillant sur la synthèse

Le groupe IEEE SSIG (Synthesis Special Interest Group) travaille sur la mise en place d'un environnement VHDL pour la synthèse. Il est chargé d'établir un document définissant la sémantique pour la synthèse des types déclarés dans le paquetage *"STD_LOGIC_1164"* (notamment pour les valeurs 'L' et 'H' mais également pour la valeur '-' dans le cas de comparaisons, etc.) et de faire en sorte que la sémantique pour la simulation et la synthèse soit la même. Ce groupe doit également définir des paquetages arithmétiques travaillant sur des types numériques, ainsi que des attributs spécifiques permettant de donner des informations aux outils de synthèse. Les résultats de ces travaux sont actuellement en cours de standardisation. De plus amples renseignements sont donnés dans [Airi 94].

Après avoir défini et donné les objectifs de la synthèse automatique de circuits, énuméré les différents niveaux de synthèse, exposé les limitations du langage VHDL pour la synthèse et expliqué la méthodologie de conception de circuits à partir d'une description VHDL, nous allons maintenant aborder plus en détails la modélisation VHDL de circuits pour la synthèse, en considérant tout d'abord une bibliothèque cible constituée uniquement d'éléments de base, puis en considérant également l'appel et l'utilisation d'éléments plus complexes appelés macro blocs.

2. Modélisations VHDL d'éléments de bas niveau

Précisons que nous nous intéressons ici à la spécification initiale d'un circuit en VHDL et non pas à la description des éléments d'une bibliothèque en VHDL. Ceci est plutôt l'objet de VITAL (VHDL Initiative Towards ASIC Libraries) sur lequel nous reviendrons au cours du chapitre suivant. Un circuit peut se décomposer en sous-blocs jusqu'à obtention d'éléments très simples pouvant correspondre à des éléments de bibliothèque. C'est pourquoi les modèles VHDL qui sont donnés ici sont généraux et ne s'appliquent pas à un élément donné d'une bibliothèque particulière. Notamment, les informations de temps ont été volontairement omises ; seul le comportement est décrit. Ceci est suffisant pour faire une première synthèse du circuit. Suivant les résultats obtenus, des contraintes temporelles ou autres pourront être spécifiées par les différents moyens que nous avons cités au cours du chapitre précédent. Les modèles qui vont être donnés par la suite constituent une base pour un concepteur débutant en VHDL.

2.1. Définition

Les ***éléments de bas niveau*** également appelés éléments de base englobent les portes logiques simples (AND, OR, NOT, XOR, NAND, NOR, XNOR), les portes logiques complexes (AO,AOI, OA,OAI), les portes d'entrée-sortie, les portes trois-états, ainsi que les éléments séquentiels simples de type bascule ou verrou (bascules D, RS, T, JK, etc.). Il s'agit des éléments utilisés par la synthèse de bas niveau. Cet ensemble d'éléments est suffisant pour construire un circuit complet.

2.2. Modélisations VHDL de blocs combinatoires

2.2.1. Définition

Un ***modèle combinatoire*** est un modèle qui n'implique aucune mémorisation : ses sorties ne dépendent que de ses entrées et non pas d'un état interne. Si le temps de propagation du modèle n'est pas spécifié, un changement d'état sur les entrées implique donc immédiatement un changement d'état sur les sorties.

2.2.2. Modélisation de portes logiques de base

Pour modéliser les portes logiques de base, le plus simple est d'utiliser les opérateurs logiques VHDL : *"and"* , *"or"* , *"xor"* , *"not"* , *"nand"* , *"nor"*, *"xnor"* , au niveau d'une affectation concurrente de signal. Ces opérateurs sont préalablement définis pour les types *"BOOLEAN"* ,

"BIT" et *"BIT_VECTOR"* pour des vecteurs de même taille. Les mêmes opérateurs surchargés pour les types définis dans le paquetage standard *"STD_LOGIG_1164"* sont également disponibles dans ce paquetage. La figure II.5 donne un exemple de modélisation d'une porte NAND à deux entrées utilisant le type *"BIT"*.

```
entity NAND2 is
    port (  A, B :   in BIT;
            S :      out BIT      );
end NAND2;

architecture ARCH of NAND2 is
begin
    S <= A nand B;
end ARCH;
```

Figure II.5 : Modélisation d'une porte NAND à deux entrées.

Il est bien évident que d'autres descriptions peuvent être utilisées, mais celle-ci est la plus simple et la plus directe.

2.2.3. Modélisations de portes logiques complexes par des équations Booléennes

La modélisation de portes logiques complexes par des équations Booléennes utilise également les opérateurs logiques VHDL cités précédemment au niveau d'affectations concurrentes de signaux. Un parenthésage est nécessaire pour indiquer la priorité des opérations. Un exemple est donné à la figure II.6.

```
entity EQ_BOOL is
    port (  A, B, C :   in BIT;
            S :         out BIT      );
end NAND2;

architecture ARCH of EQ_BOOL is
begin
    S <= (A and B) or (not C);
end ARCH;
```

Figure II.6 : Modélisation d'une équation Booléenne à trois entrées.

Remarque :

Ces équations sont ensuite traitées par la synthèse de bas niveau et peuvent donc être optimisées par les étapes de minimisation, de factorisation et de projection structurelle sur la bibliothèque. Le résultat physique obtenu après synthèse de l'exemple donné à la figure II.6 n'est donc pas

forcément celui de la figure II.7. Ce résultat dépend des éléments contenus dans la bibliothèque et des contraintes données par le concepteur à l'outil de synthèse ainsi que du savoir-faire de cet outil de synthèse.

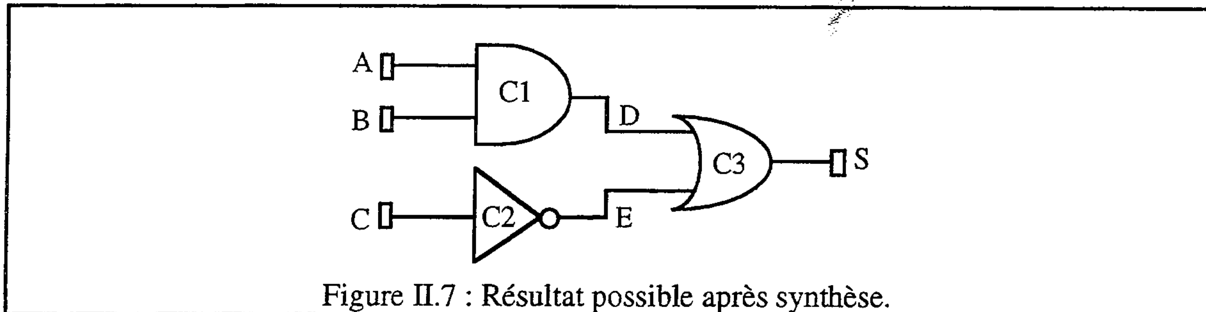

Figure II.7 : Résultat possible après synthèse.

Pour obtenir un tel résultat, en supposant que la bibliothèque contienne au moins les portes logiques AND, OR et NOT, il est nécessaire de décrire ces équations structurellement comme le montre la figure II.8 et de forcer l'outil de synthèse à conserver cette hiérarchie. Bien souvent, une des options par défaut dans les outils de synthèse consiste à mettre à plat la description dans le cas de technologies FPGA, alors que cette hiérarchie est généralement conservée pour les technologies ASIC. Ceci peut aboutir à de bons résultats pour de petits modèles, mais est pénalisant pour des plus gros, aussi bien au niveau de la surface que du temps de propagation du circuit obtenu.

```
architecture ARCH2 of EQ_BOOL is
    signal D, E : BIT;
begin
    C1 : entity WORK.AND2(ARCH)      port map (A,B,D);
    C2 : entity WORK.NOT1(ARCH)      port map (C,E);
    C3 : entity WORK.OR2(ARCH)       port map (D,E,S);
end ARCH2;
```

Figure II.8 : Modélisation structurelle d'une équation Booléenne à trois entrées.

Il est à noter que le modèle donné à la figure II.8 suppose que les descriptions des portes AND2, NOT1 et OR2 qui sont référencées soient déjà écrites et compilées dans la bibliothèque VHDL de travail. De plus, ce modèle n'est autorisé qu'avec le nouveau standard VHDL'92 car il fait intervenir la notion d'instanciation directe déjà vue au cours du premier chapitre.

2.2.4. Modélisations de portes complexes par des tables de vérité

Une autre manière de décrire une porte combinatoire complexe, éventuellement multi-sorties, est de décrire la table de vérité correspondante. Considérons par exemple la table de vérité donnée à la table 1.

Table 1 : Table de vérité

A	B	C	S
0	0	0	-
0	0	1	-
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	-
1	1	1	-

La modélisation des tables de vérité se fait par l'affectation conditionnelle de signaux en utilisant les opérateurs logiques VHDL. Notons que dans la table de vérité donnée à la table 1, les deux premières et les deux dernières valeurs de la sortie S n'ont pas d'importance. Pour modéliser ces valeurs, l'utilisation des types définis dans le paquetage standard *"STD_LOGIC_1164"* est donc nécessaire. Ce paquetage est compilé dans la bibliothèque VHDL *"IEEE"* et il est possible d'y faire référence dans une description VHDL par une clause *"use"*. Le modèle décrivant la table de vérité est donné à la figure II.9.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity TABLE is
port (  A,B,C :      in STD_LOGIC;
        S :          out STD_LOGIC     );
end TABLE;

architecture ARCH of TABLE is
begin
    S <=    '0' when (  ((A = '0') and (B = '1') and (C = '0')) or
                        ((A = '1') and (B = '0') and (C = '0'))  ) else
            '1' when (  ((A = '0') and (B = '1') and (C = '1')) or
                        ((A = '1') and (B = '0') and (C = '1'))  ) else
            '-';
end ARCH;
```

Figure II.9 : Modélisation d'une table de vérité.

L'instruction concurrente d'affectation conditionnelle de signaux *"with ... select"* peut également être utilisée.

Tous les modèles que nous venons de voir peuvent également être décrits en utilisant un processus VHDL combinatoire équivalent.

En synthèse, un processus VHDL est appelé ***processus combinatoire*** lorsqu'il ne fait appel à aucun élément de mémorisation. Pour ce faire, ce processus, s'il ne contient pas de variable, doit remplir deux conditions. La première condition consiste à énumérer au niveau de la liste de sensibilité du processus, tous les signaux lus dans ce même processus, c'est-à-dire tous les signaux figurant en parties droites d'affectations et figurant au niveau des conditions. La seconde condition consiste à affecter les signaux dans tous les chemins possibles, c'est-à-dire dans tous les cas. Une troisième condition vient s'ajouter dans le cas où des variables sont utilisées : ces variables doivent être lues après avoir été affectées. Un exemple de processus combinatoire décrivant la table de vérité précédente est donné à la figure II.10.

```
architecture ARCH2 of TABLE is
begin
    process (A,B,C)  -- tous les signaux lus sont dans la liste de sensibilité
    begin
        if (    ((A = '0') and (B = '1') and (C = '0')) or
                ((A = '1') and (B = '0') and (C = '0'))  )  then
            S <= '0';
        elsif  ( ((A = '0') and (B = '1') and (C = '1')) or
                 ((A = '1') and (B = '0') and (C = '1'))  )  then
            S <= '1';
        else
            S <= '-';       -- le signal S est affecté dans tous les chemins
        end if;
    end process;
end ARCH2;
```

Figure II.10 : Modélisation d'une table de vérité par un processus combinatoire.

2.2.5. Modélisation de portes trois-états

Les portes trois-états sont utilisées lorsque plusieurs blocs accèdent en même temps au même bus. En VHDL, les fonctions de résolution ont été définies pour décrire un tel comportement et résoudre ainsi les conflits pouvant apparaître lors des accès aux bus. Un moyen très simple pour modéliser une porte trois-états est de faire intervenir la valeur 'Z', donc il est nécessaire d'utiliser les types définis dans le paquetage *"STD_LOGIC_1164"* . Un signal affecté à la valeur 'Z' dans certaines conditions, correspond à une sortie de porte trois-états commandée par ces conditions. La figure II.11. donne une description de porte trois-états et la figure II.12 en donne la représentation physique correspondante.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity TROIS_ETATS is
port (  A,COND :   in STD_LOGIC;
        S :        out STD_LOGIC     );
end TROIS_ETATS;

architecture ARCH of TROIS_ETATS is
begin
    S <=    A when (COND = '1') else
            'Z';
end ARCH;
```

Figure II.11 : Modélisation d'une porte trois-états.

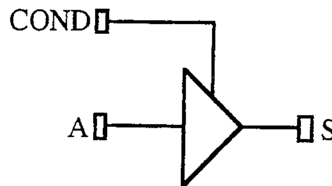

Figure II.12 : Porte trois-états.

2.3. Modélisations VHDL de blocs séquentiels

2.3.1. Définitions

Un ***modèle séquentiel*** est un modèle qui implique de la mémorisation. Les valeurs des sorties de ce modèle ne dépendent donc pas uniquement des valeurs sur les entrées, mais également des valeurs des états internes.

Un ***modèle synchrone*** est un modèle dans lequel figure un signal spécifique de contrôle généralement appelé horloge. Si tous les points de synchronisation se réfèrent à cette horloge, alors le modèle est purement synchrone. Puisque la synchronisation implique la mémorisation, un modèle synchrone, purement ou non, est séquentiel.

Un ***modèle asynchrone*** est donc un modèle pour lequel il n'existe pas de signal spécifique de contrôle. Un modèle combinatoire en est un exemple. Un modèle qui n'est pas purement synchrone est donc en partie asynchrone.

2.3.2. Modélisations de verrous

Le verrou est le point de mémorisation le plus simple. Le comportement du verrou peut se résumer ainsi : la sortie Q du verrou recopie l'entrée D pendant le niveau actif (haut ou bas) de l'horloge. Lorsque l'horloge est inactive, la dernière valeur de l'entrée D est mémorisée sur la sortie Q. Le moyen le plus simple de modéliser un verrou est d'utiliser un processus comme le montre la figure II.13.

```
entity VERROU is
port (  D,CLK :     in BIT;
        Q:          out BIT      );
end VERROU;

architecture ARCH of VERROU is
begin
    process (D,CLK)
    begin
        if CLK = '1' then
            Q<=D;
        end if;
    end process;
end ARCH;
```

Figure II.13 : Modélisation d'un verrou actif sur le niveau haut de l'horloge.

Une autre écriture plus lourde utilisant des affectations concurrentes de signaux et nécessitant l'intervention d'un signal interne est donnée à la figure II.14.

```
architecture ARCH2 of VERROU is
    signal Q_INT : BIT;
begin
    Q_INT <=  D when CLK = '1' else
              Q_INT;
    Q <= Q_INT;
end ARCH2;
```

Figure II.14 : Modélisation d'un verrou actif sur le niveau haut de l'horloge.

VHDL'92 permet de simplifier cette dernière écriture et en propose trois nouvelles. Chaque affectation concurrente donnée à la figure II.15 peut remplacer les deux affections décrites dans la figure II.14 et évite la déclaration supplémentaire du signal interne Q_INT.

```
Q <=   D when CLK = '1';
Q <=   D when CLK = '1' else unaffected;
Q <=   D when CLK = '1' else Q'DRIVING_VALUE;
```

Figure II.15 : Modélisations en VHDL'92 d'un verrou actif sur le niveau haut de l'horloge.

La caractéristique principale de toutes ces descriptions est que l'entrée D figure dans la liste de sensibilité du processus. Afin de modéliser une mise à un ou à zéro asynchrone d'un verrou , il faut également faire intervenir le signal de contrôle correspondant dans la liste de sensibilité. La modélisation d'un verrou avec mise à zéro asynchrone et prioritaire par rapport au signal d'horloge est donnée à la figure II.16.

```
entity VERROU_ASYNC is
port (  D,CLK,RST :   in BIT;
        Q:            out BIT      );
end VERROU_ASYNC;

architecture ARCH of VERROU_ASYNC is
begin
    process (D,CLK,RST)
    begin
        if RST = '1' then
            Q <= '0';
        elsif CLK = '1' then
            Q<=D;
        end if;
    end process;
end ARCH;
```

Figure II.16 : Modélisation d'un verrou avec mise à zéro asynchrone.

Pour modéliser le même verrou mais cette fois avec une remise à zéro synchrone par rapport au signal d'horloge, il faut supprimer le signal de contrôle correspondant de la liste de sensibilité. Le modèle d'un tel verrou est donné à la figure II.17.

```
architecture ARCH of VERROU_SYNC is
begin
    process (D,CLK)
    begin
        if CLK = '1' then
            if RST = '1' then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;
end ARCH;
```

Figure II.17 : Modélisation d'un verrou avec mise à zéro synchrone par rapport à l'horloge.

2.3.3. Modélisations de bascules

Le comportement d'une bascule est très proche de celui d'un verrou : l'entrée D est mémorisée sur la sortie Q de la bascule lorsqu'un front (montant ou descendant) intervient sur le signal d'horloge, au lieu d'un niveau dans le cas d'un verrou. Les modèles VHDL sont donc également très proches. La seule différence consiste à supprimer l'entrée D de la liste de sensibilité du processus. La description d'une bascule active sur le front montant de l'horloge est donnée à la figure II.18.

```
entity BASCULE is
port (  D,CLK :    in BIT;
        Q:         out BIT      );
end BASCULE;

architecture ARCH of BASCULE is
begin
    process (CLK)
    begin
        if CLK = '1' then
            Q <= D;
        end if;
    end process;
end ARCH;
```

Figure II.18 : Modélisation d'une bascule active sur le front montant de l'horloge.

Notons que pour la modélisation de bascules, beaucoup d'outils de synthèse imposent l'écriture redondante : *"if CLK'EVENT and CLK = '1' then ..."* à la place de *"if CLK = '1' then ..."*. D'autres écritures sont également possibles. Deux exemples sont donnés à la figure II.19.

```
architecture ARCH2 of BASCULE is
begin
    process
    begin
        wait until CLK = '1';
        Q <= D;
    end process;
end ARCH2;

architecture ARCH3 of BASCULE is
    signal Q_INT : BIT;
begin
    Q_INT <=  D when (CLK'EVENT and CLK = '1') else
              Q_INT;
    Q <= Q_INT;
end ARCH3;
```

Figure II.19 : Modélisations d'une bascule active sur le niveau haut de l'horloge.

Notons que dans le dernier modèle le test sur *"CLK'EVENT"* est indispensable sinon le comportement obtenu n'est plus celui d'une bascule mais bien celui d'un verrou.

Pour décrire une remise à un ou à zéro synchrone ou asynchrone, les mêmes règles que celles vues pour la modélisation des verrous restent valables. La modélisation d'une bascule comprenant un signal d'horloge, plus un signal de mise à zéro et un signal de validation synchrones par rapport à cette horloge, est donnée à la figure II.20. Notons que dans ce modèle, le signal de mise à zéro est prioritaire par rapport à celui de validation. Comme il a été vu précédemment pour les verrous, l'ordre de priorité des signaux est donné par l'ordre des instructions séquentielles dans le processus.

```
entity BASCULE_SYNC is
port (  D,CLK,RST,VAL :  in BIT;
        Q:               out BIT      );
end BASCULE_SYNC;

architecture ARCH of BASCULE_SYNC is
begin
    process (CLK)
    begin
        if CLK = '1' then
            if RST = '1' then
                Q <= '0';
            elsif VAL = '1' then
                Q <= D;
            end if;
        end if;
    end process;
end ARCH;
```

Figure II.20 : Modélisation d'une bascule avec mise à zéro et validation synchrones par rapport à l'horloge.

3. Modélisations VHDL de macro blocs

3.1. Définitions

Les ***macro blocs*** sont des blocs de bibliothèque plus complexes que les éléments de base. Il s'agit plus précisément de multiplexeurs, de décodeurs, de comparateurs, d'opérateurs arithmétiques (additionneurs, soustracteurs, multiplieurs, incrémenteurs, etc.), de décaleurs, d'unités arithmétique et logique, de registres, de compteurs, etc. Ces éléments ne sont pas utilisés par la synthèse de bas niveau. Le nombre de bits de ces blocs est généralement supérieur à un. Ces blocs sont généralement utilisés dans les parties opératives et les chemins de données des circuits intégrés.

Les ***générateurs*** sont des macro blocs paramétrés possédant au moins un paramètre. Pour ne citer qu'un exemple, le paramètre le plus souvent utilisé est le nombre de bits du bloc. Le résultat obtenu lorsque tous les paramètres du générateur sont fixés est une instance de ce générateur. Le même générateur permet donc d'obtenir autant d'instances que de combinaisons possibles des valeurs de ses paramètres. Des exemples de générateurs sont fournis dans les bibliothèques XBLOX de Xilinx [XBLOX], ACTgen d'Actel [ACTgen] et LPM (Library of Parameterized Modules) du standard [LPM].

3.2. Modélisations VHDL de multiplexeurs

La fonctionnalité du multiplexeur consiste à sélectionner une entrée parmi plusieurs et de la propager vers une simple sortie. Les multiplexeurs sont généralement utilisés en entrée ou en sortie d'opérateurs et de registres de sorte à minimiser leur nombre lorsque ceux-ci sont utilisés de manière mutuellement exclusive. Il est possible de décrire un multiplexeur en écrivant tout simplement son équation logique en suivant les règles qui ont déjà été données au paragraphe 2.2.3. Une autre méthode consiste à utiliser les instructions conditionnelles concurrentes ou séquentielles VHDL. Deux exemples sont donnés aux figures II.21 et II.22.

```
entity MUX2x1 is
port (  A,B,SEL_A :     in BIT;
        S :             out BIT     );
end MUX2x1;

architecture ARCH of MUX2x1 is
begin
    S <=    A when SEL_A = '1' else
            B;
end ARCH;
```

Figure II.21 : Modélisation d'un multiplexeur deux entrées vers une sortie.

Le modèle de la figure II.22 donne le processus équivalent de l'instruction utilisée à la figure II.21. Rappelons que pour modéliser un multiplexeur à l'aide d'un processus, ce processus doit être combinatoire, ce qui implique que la sortie du multiplexeur doit être affectée dans toutes les branches du processus et que tous les signaux d'entrée doivent figurer dans la liste de sensibilité du processus.

```
architecture ARCH2 of MUX2x1 is
begin
    process (A,B,SEL_A)
    begin
        if SEL_A = '1' then
            S <= A;
        else
            S <= B;
        end if;
    end process;
end ARCH2;
```

Figure II.22 : Modélisation d'un multiplexeur deux entrées vers une sortie.

Notons que les entrées et la sortie du multiplexeur doivent être de même type mais pas forcément du type *"BIT"* (il peut s'agir de vecteurs, d'entiers, etc.). Notons également que la

condition de sélection peut être une expression faisant intervenir plusieurs signaux. Des modèles de multiplexeurs plus complexes sont donnés aux figures II.23 et II.24. Ces modèles décrivent un multiplexeur quatre entrées : A,B,C et D vers une sortie : S. A,B,C,D et S sont des signaux de même type :*"STD_LOGIC_VECTOR"* dans ce cas. Les signaux SEL1, SEL2, SEL3 et SEL4 intervenant au niveau des expressions de sélection sont de type *"STD_LOGIC"*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity MUX4x1 is
port (  A,B,C,D :        in STD_LOGIC_VECTOR (7 downto 0);
        SEL1,SEL2 :      in STD_LOGIC;
        SEL3,SEL4 :      in STD_LOGIC;
        S :              out STD_LOGIC_VECTOR (7 downto 0)      );
end MUX4x1;

architecture ARCH of MUX4x1 is
begin
    S <=    A when (SEL1 = '0') else
            B when ((SEL2 = '1') and (SEL3 = '1')) else
            C when (SEL4 = '1') else
            D;
end ARCH;
```

Figure II.23 : Modélisation d'un multiplexeur quatre entrées vers une sortie.

Pour les signaux de sélection, le type *"STD_LOGIC"* a été choisi de sorte à pouvoir faire intervenir la valeur '-' dans le modèle donné à la figure II.24.

```
architecture ARCH2 of MUX4x1 is
    signal COM : STD_LOGIC_VECTOR (3 downto 0);
begin
    COM <= SEL1 & SEL2 & SEL3 & SEL4;
    with COM select
    S <=    A when "0---",
            B when "111-",
            C when "10-1" | "1-01",
            D when others;
end ARCH2;
```

Figure II.24 : Modélisation d'un multiplexeur quatre entrées vers une sortie.

Ces deux exemples illustrent un point très important : les formes conditionnelle (figure II.23) et sélective (figure II.24) ne sont pas réellement équivalentes. Par concept, l'instruction conditionnelle implique une priorité dans l'exécution alors que dans la forme sélective, les branches sont accédées avec la même priorité.

Remarque :

Dans l'exemple donné à la figure II.24 la valeur '-' est utilisée plusieurs fois dans les conditions des branches sélectives, or cette valeur doit être utilisée avec précaution notamment lors de comparaisons implicites ou explicites. En effet, lors de la simulation de cet exemple avant synthèse, si la valeur du signal COM est "0000", elle est différente au niveau de la simulation de la valeur "0---" et donc la sortie S ne prendra pas la valeur de A, comme nous pourrions nous y attendre, mais celle de D. Cependant, lors de la simulation après synthèse, pour cette même valeur "0000" du signal COM, la sortie S prendra la valeur de A. Donc, la simulation avant et après synthèse de cet exemple donne des résultats différents, ce qui n'est pas acceptable. Pour remédier à ce problème d'incohérence, le groupe IEEE SSIG, chargé de spécifier la sémantique des valeurs du type *"STD_ULOGIC"* pour la synthèse, a défini la fonction *"STD_MATCH"* figurant dans le paquetage *"NUMERIC_STD"*. Cette fonction permet de comparer des bits ou des vecteurs contenant la valeur '-' en considérant la valeur '-' comme une valeur sans importance (un "dont'care"). L'utilisation de *"STD_MATCH"* permet donc de filtrer la comparaison de sorte à rendre cohérents les résultats de simulation avant et après synthèse. Le modèle donné à la figure II.25 permet de remédier, par l'utilisation de la fonction *"STD_MATCH"*, au problème d'incohérence introduit au modèle de la figure II.24. Dans l'exemple de la figure II.25, lors de la simulation avant et après synthèse, le signal S prend bien la valeur de A lorsque COM vaut "0000".

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

architecture ARCH3 of MUX4x1 is
    signal COM : STD_LOGIC_VECTOR (3 downto 0);
begin
    COM <= SEL1 & SEL2 & SEL3 & SEL4;
    S <=    A when STD_MATCH(COM,"0---") else
            B when STD_MATCH(COM,"111-") else
            C when STD_MATCH(COM,"10-1") or
                   STD_MATCH(COM,"1-01") else
        D;
end ARCH3;
```

Figure II.25 : Modélisation d'un multiplexeur quatre entrées vers une sortie.

3.3. Modélisations VHDL de comparateurs

Les comparateurs sont modélisés à l'aide des opérateurs relationnels VHDL : =, /=, <, <=, >, et >=. Ces opérateurs sont généralement utilisés au niveau des conditions des instructions concurrentes et séquentielles conditionnelles et au niveau des expressions situées en partie droite d'affectations. Un exemple de modélisation de comparateur huit bits est donné à la figure

II.26. La sortie EG est vraie si les valeurs des deux vecteurs d'entrées A et B sont égales sinon elle est fausse, la sortie PG est vraie si la valeur de A est plus grande que celle de B sinon elle est fausse et la sortie PP est vraie si la valeur de A est plus petite que celle de B sinon elle est fausse.

```
entity COMP is
port (  A,B :           in BIT_VECTOR (7 downto 0);
        EG, PG, PP :    out BOOLEAN );
end COMP;

architecture ARCH of COMP is
begin
    EG <= A = B;
    PG <= A > B;
    PP <= A < B;
end ARCH;
```

Figure II.26 : Modélisation d'un comparateur huit bits.

Notons que les opérateurs relationnels tels qu'ils sont préalablement définis permettent de comparer deux entiers ou deux vecteurs de bits de même taille et renvoient une valeur Booléenne. L'utilisation de l'opérateur d'égalité pour comparer deux vecteurs de taille différente amènera toujours à un résultat faux même si ces vecteurs ont la même valeur ("0011" est différent de "011"). Ces opérateurs peuvent également être surchargés de sorte à pouvoir être utilisés avec d'autres types.

3.4. Modélisations VHDL d'opérateurs arithmétiques

Les opérateurs arithmétiques sont modélisés à l'aide des opérateurs VHDL : + (addition), - (soustraction), * (multiplication), / (division), ** (puissance), mod (modulo), rem (reste). Ces opérateurs sont préalablement définis pour le type entier. Pour pouvoir les utiliser avec des vecteurs, il faut les surcharger. Il est bien souvent préférable d'utiliser des vecteurs afin d'accéder directement aux bits pour tester par exemple un dépassement ou bien pour tronquer ou étendre facilement les vecteurs. Dans les exemples qui suivent, nous supposerons que ces opérateurs surchargés ont été définis auparavant dans un paquetage VHDL appelé *"PKG_ARITH"* . Nous voyons là tout l'intérêt d'un tel paquetage en tant qu'outil pour la modélisation. La description d'un additionneur huit bits est donnée à la figure II.27. Il est bien évident que ce n'est pas la seule manière de modéliser des additionneurs, mais c'est la plus simple et la plus rapide ; et la qualité du résultat obtenu, qui dépend du savoir faire de l'outil de synthèse, peut être tout à fait satisfaisante. D'autres exemples beaucoup plus complexes de modélisations VHDL d'additionneurs sont également disponibles dans [Bert 91].

```
library WORK;
use WORK.PKG_ARITH.all;

entity ADD is
port (  A,B :   in BIT_VECTOR (7 downto 0);
        S :     out BIT_VECTOR (7 downto 0) );
end ADD;

architecture ARCH of ADD is
begin
   S <= A + B;
end ARCH;
```

Figure II.27 : Modélisation d'un additionneur huit bits.

Pour obtenir le modèle des autres opérateurs, il suffit de remplacer l'opérateur + dans la description précédente par l'opérateur arithmétique VHDL correspondant.

Notons toutefois que si des entiers sont utilisés à la place des vecteurs et si ces entiers ne sont pas contraints à un certain rang, alors ils seront généralement codés par l'outil de synthèse sur trente deux bits. Donc, bien souvent, les entiers utilisés devront être contraints. Si il s'agit d'entiers contraints tels qu'ils soient compris entre les valeurs entières N et M, alors, si N est positif, une représentation non signée est choisie sur une largeur L de bits, L correspondant au nombre le plus petit tel que (2 ** L) - 1 soit plus grand ou égal à M. Par contre, si N est négatif, une représentation signée (généralement en complément à deux) est nécessaire et dans ce cas L est le nombre le plus petit tel que (2 ** (L-1)) -1 soit supérieur ou égal à M et tel que -2**(L-1) soit inférieur ou égal à N.

Remarquons également que l'opérateur de division n'est pas accepté par les outils de synthèse excepté lorsque le diviseur est une puissance de deux ce qui revient à un décalage. Ceci reste très réaliste puisque il n'existe pas à ce jour de diviseur dans les bibliothèques. De même, les opérateurs **, mod et rem ne sont acceptés que si l'opérande de droite est une puissance de deux.

3.5. Modélisations VHDL de décaleurs

Ces opérateurs sont généralement utilisés pour des vecteurs. Deux sortes d'opérateurs de décalage et de rotation peuvent être distinguées : les opérateurs arithmétiques et les opérateurs logiques.

Comme nous venons de voir, il est possible de modéliser un décaleur arithmétique en utilisant un opérateur de multiplication (décalage à gauche) ou de division (décalage à droite) dont

l'opérande de droite est une puissance de deux. Le même comportement peut également être obtenu en utilisant les opérateurs de décalage arithmétique *"sla"* et *"sra"*, préalablement définis par VHDL'92 pour les vecteurs de *"BIT"* et de *"BOOLEAN"*. Notons que ces deux opérateurs permettent de conserver le bit de signe du vecteur à décaler. La sémantique de ces deux opérateurs est représentée à la figure II.28.

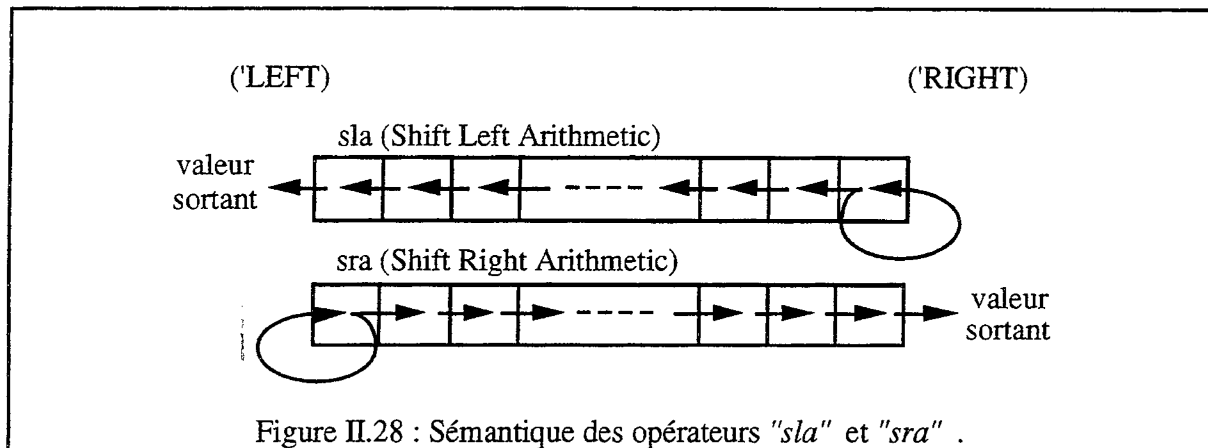

Figure II.28 : Sémantique des opérateurs *"sla"* et *"sra"* .

Les opérateurs logiques de décalage concernent les opérateurs de décalage et de rotation, vers la gauche ou vers la droite. Pour modéliser ces opérateurs logiques, il est possible d'utiliser les opérateurs : *"sll"*, *"srl"*, *"rol"* et *"ror"* , préalablement définis par VHDL'92 pour les vecteurs de *"BIT"* et de *"BOOLEAN"* . La sémantique de ces opérateurs est représentée à la figure II.29. Notons que la valeur d'entrée utilisée lors des décalages correspond à la valeur la plus à droite du type énuméré de base utilisé, soit '0' pour des vecteurs de *"BIT"* et *"FALSE"* pour les vecteurs de *"BOOLEAN"* .

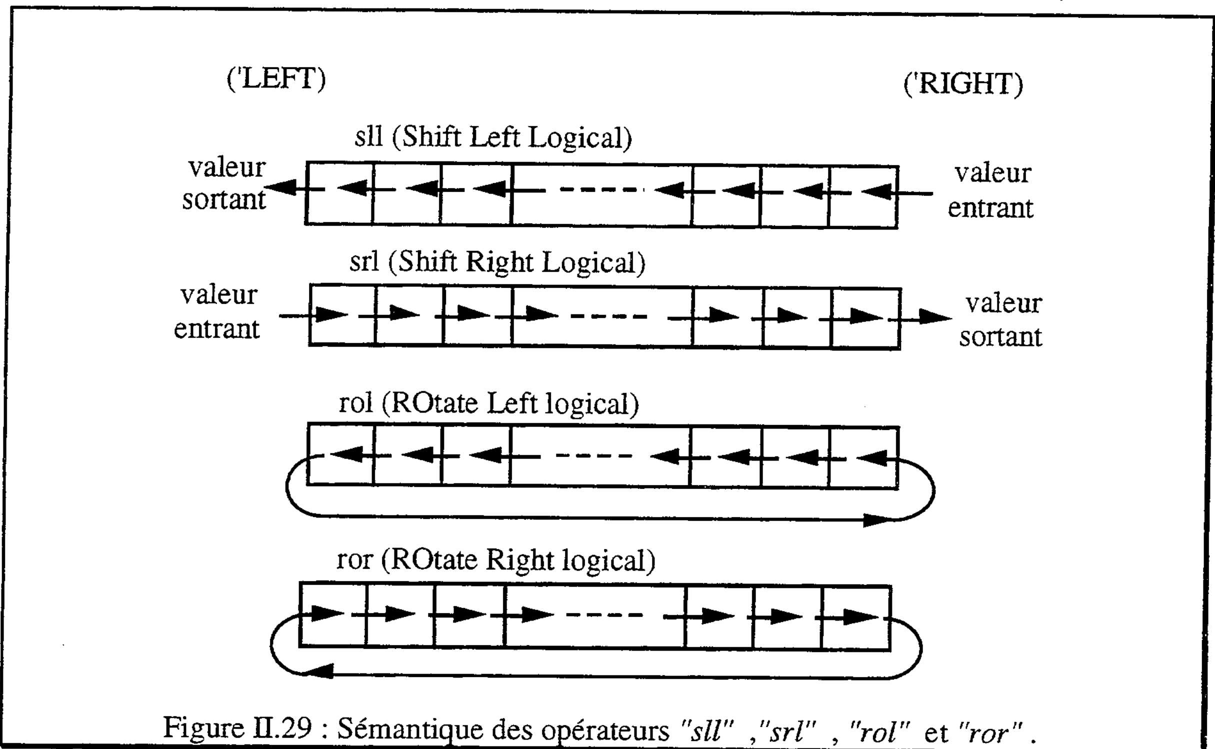

Figure II.29 : Sémantique des opérateurs *"sll"* ,*"srl"* , *"rol"* et *"ror"* .

Il est également possible de décrire tous ces opérateurs en utilisant des affectations de signaux par tranches. Deux modèles équivalents au niveau du comportement sont donnés à la figure II.30. Il s'agit de la description d'un décaleur logique huit bits décalant un bit vers la gauche.

```
entity DECAL is
port (  A:      in BIT_VECTOR (7 downto 0);
        S:      out BIT_VECTOR (7 downto 0) );
end DECAL;

architecture ARCH1 of DECAL is
begin
    S <= sll (A,1);
end ARCH1;

architecture ARCH2 of DECAL is
begin
    S(0) <= '0';
    S(7 downto 1) <= A(6 downto 0);
end ARCH2;
```

Figure II.30 : Modélisations d'un décaleur logique huit bits.

3.6. Modélisations VHDL de registres

Tout ce que nous avons déjà dit au paragraphe 2.3.3 sur les modélisations de bascules reste vrai pour les modélisations de registres. Il suffit simplement d'utiliser des vecteurs à la place de

simple bits pour l'entrée et la sortie des données. La figure II.31 donne un exemple de modélisation d'un registre huit bits actif sur le front montant de l'horloge, ayant une remise à zéro asynchrone et une validation des données synchrone par rapport à l'horloge.

```
entity REGISTRE is
port (  D:                 in BIT_VECTOR (7 downto 0);
        CLK,RST,VAL :      in BIT;
        Q:                 out BIT_VECTOR (7 downto 0) );
end REGISTRE;

architecture ARCH of REGISTRE is
begin
    process (CLK,RST)
    begin
        if RST = '1' then
            Q <= "00000000";
        elsif CLK = '1' then
            if VAL = '1' then
                Q<=D;
            end if;
        end if;
    end process;
end ARCH;
```

Figure II.31 : Modélisation d'un registre huit bits avec mise à zéro asynchrone et validation de données synchrone par rapport à l'horloge.

3.7. Modélisations VHDL de compteurs

Pour modéliser un compteur, il est nécessaire de décrire à la fois une partie séquentielle permettant de mémoriser les sorties du compteur et une partie combinatoire permettant d'incrémenter ou de décrémenter ces sorties. Il n'est pas indispensable de séparer ces deux parties au niveau de la description et il est tout à fait possible et très simple d'utiliser un seul processus pour décrire un compteur. La figure II.32 donne le modèle d'un compteur huit bits possédant une remise à zéro asynchrone, ainsi qu'un signal de chargement synchrone par rapport à l'horloge, permettant de charger la valeur à partir de laquelle doit s'effectuer le comptage. D'autres modèles de compteurs sont également disponibles dans [Bert 91].

```
library WORK;
use WORK.PKG_ARITH.all;

entity COMPTEUR is
port (  D:                      in BIT_VECTOR (7 downto 0);
        CLK,RST,CHARG :         in BIT;
        Q:                      out BIT_VECTOR (7 downto 0) );
end COMPTEUR;

architecture ARCH of COMPTEUR is
signal COUNT : BIT_VECTOR (7 downto 0);
begin
    process (CLK,RST)
    begin
        if RST = '1' then
            COUNT <= "00000000";
        elsif CLK = '1' then
            if CHARG = '1' then
                COUNT <= D;
            else
                COUNT <= COUNT + "00000001";
            end if;
        end if;
    end process;
    Q <= COUNT;
end ARCH;
```

Figure II.32 : Modélisation d'un compteur huit bits avec mise à zéro asynchrone et chargement de données synchrone.

Si la bibliothèque utilisée pour faire la synthèse ne contient pas de compteur, pour réaliser ce modèle, un registre huit bits avec remise à zéro asynchrone devra être utilisé pour mémoriser la sortie Q du compteur, puis un additionneur ou un incrémenteur sera utilisé pour compter. Un multiplexeur deux entrées vers une sortie peut alors être mis en entrée du registre, permettant ainsi de le charger, soit avec l'entrée D, soit avec le résultat du comptage, suivant la valeur de l'entrée CHARG. Nous venons donc de voir qu'un même processus peut engendrer à la fois des parties combinatoires et des parties séquentielles.

Remarque :

La plus part des modèles qui ont été donnés jusqu'ici utilisent des vecteurs de bits et non pas des entiers. Des entiers peuvent également être utilisés. Cependant les entiers s'utilisent de préférence au niveau des descriptions de systèmes. De plus les modèles utilisant les vecteurs sont plus naturels pour les concepteurs de circuits et permettent d'accéder à chaque bit ce qui est parfois indispensable selon ce qui est modélisé.

3.8. Modélisations VHDL de mémoires RAM

Une mémoire RAM (Read Access Memory) peut être considérée comme un tableau de M mots de N bits, chaque mot pouvant être lu ou écrit de façon synchrone par rapport à une horloge. Le contrôle se fait généralement par l'intermédiaire d'un ou plusieurs bus d'adresse indiquant l'adresse du mot à lire ou à écrire, ainsi que par un port de lecture/écriture permettant de lire ou d'écrire un mot de la mémoire.

Un exemple simple de modélisation d'une RAM MxN est donnée à la figure II.33. La modélisation d'une RAM passe nécessairement par l'utilisation d'un tableau. De plus, une seconde condition est indispensable : il doit y avoir au moins une référence et une affectation du tableau par cycle d'horloge. Ici, une référence au tableau signifie qu'un élément du tableau se trouve en partie droite d'une affectation, et une affectation du tableau signifie qu'un élément du tableau figure en partie gauche d'une affectation.

```
library WORK;
use WORK.PKG_ARITH.all;

entity RAM_MxN is
generic (  M :     POSITIVE := 4;  -- 2**4 = 16 mots
           N :     POSITIVE := 8  );
port (     CLK :            in BIT;
           LEC_ECR :        in BIT;
           AD_LEC :         in BIT_VECTOR (M_1 downto 0);
           AD_ECR :         in BIT_VECTOR (M_1 downto 0);
           ENTREE :         in BIT_VECTOR (N-1 downto 0);
           SORTIE :         out BIT_VECTOR (N-1 downto 0)   );
end RAM_MxN;

architecture ARCH of RAM_MxN is
    type MEMOIRE is array ((2**M)-1 downto 0) of
            BIT_VECTOR (N-1 downto 0);
    signal TABLEAU : MEMOIRE;
begin
    process (CLK)
    begin
        if CLK = '1' then
            if LEC_ECR = '1' then
                TABLEAU(TO_NATURAL(AD_ECR)) <= ENTREE;
            end if;
        end if;
        SORTIE <= TABLEAU(TO_NATURAL(AD_LEC));
    end process;
end ARCH;
```

Figure II.33 : Modélisation d'une RAM MxN.

Notons également que dans cet exemple, la fonction de conversion TO_NATURAL permet de convertir un vecteur de bits en naturel. Cette fonction est supposée être définie dans le paquetage PKG_ARITH.

A l'heure actuelle, tous les outils de synthèse automatique n'acceptent pas forcément les tableaux et les index variables, et parmi ceux qui l'acceptent, très peu sont capables d'utiliser des mémoires RAM. Cependant, un outil développé à Thomson reconnaissant et utilisant automatiquement des RAMs est développé dans [Berth 92]. Récemment, certains outils de synthèse de haut niveau s'interêssent également à l'utilisation des mémoires RAMs. C'est le cas de l'outil TODOS capable d'utiliser plusieurs mémoires RAM pouvant avoir différentes configurations [Marw 95]. Mais lorsqu'elles sont utilisées par la synthèse de haut niveau, les mémoires RAM ne sont pas décrites explicitement comme nous l'avons fait à la figure II.33.

4. Synthèse utilisant des macro blocs

4.1. Introduction

Nous allons traiter ici les différentes étapes d'un outil de synthèse capable de générer ou d'utiliser des macro blocs à partir d'une description VHDL. Ces étapes ont été implantées dans l'outil de synthèse ASYL+ [Bert 93b], [Bert 94a] développé au laboratoire CSI (Conception de Systèmes Intégrés).

Historiquement, les premiers travaux de synthèse automatique de circuits ne concernaient que la synthèse logique. A ce niveau, le circuit est décrit par des équations logiques qui sont ensuite factorisées, minimisées et projetées sur une bibliothèque de portes de base afin d'obtenir le circuit optimisé correspondant. L'objet de ce paragraphe n'est pas de développer les différentes méthodes de synthèse logique. Néanmoins, pour plus de détails sur ces méthodes et sur les diverses étapes de la synthèse logique, il est possible de se reporter à [Bray 89], [Abou 92], [Sako 93] et [Sica 88] pour ne citer que quelques références. Beaucoup de travaux ont été publiés et sont encore publiés aujourd'hui sur ce thème.

Mais le monde de la micro-électronique a évolué et des langages très puissants de description de circuits sont apparus (VHDL, Verilog). Ces langages permettent non seulement de décrire des équations logiques, mais également des équations utilisant des opérateurs arithmétiques (+, -, *) ou relationnels (=, /=, <, <=, >, >=). Pour traiter correctement ces opérateurs, les outils de synthèse ont donc du s'adapter. En effet, la génération des équations d'un additionneur à partir d'une description contenant un simple "+", sur huit bits par exemple, peut prendre beaucoup de temps et le résultat obtenu n'est pas forcément très satisfaisant au niveau du chemin critique et

de la surface. A partir d'un nombre supérieur de bits le système de synthèse peut même exploser en mémoire. Pour plus de détails sur ces résultats, il est possible de se reporter à [Bert 91]. Les outils de synthèse ont donc été amenés à évoluer de sorte à améliorer ces résultats.

Dans un premier temps, pour palier à ce manque, le savoir faire des concepteurs en matière de conception de macro blocs (additionneurs, soustracteurs, etc.) a du être intégré dans les systèmes de synthèse, de sorte à générer automatiquement les équations optimisées de ces macro blocs, en fonction des critères de vitesse et de surface donnés par l'utilisateur. Les avantages d'une telle méthode sont notamment développés dans [Thee 95].

Dans un second temps, il a fallu tenir compte du fait que beaucoup de bibliothèques de macro blocs ont été développées, et donc, il a fallu essayer d'utiliser au maximum les éléments de ces bibliothèques. Ces bibliothèques sont des bibliothèques d'éléments fixes ou génériques ([ACTgen], [XBLOX]). L'utilisation de ces éléments permet encore d'améliorer les performances au niveau du chemin critique et de la surface par rapport à la génération automatique des équations. En effet, les concepteurs qui ont réalisé ces bibliothèques ont déployé tout leur savoir faire, mais ils ont également pris en compte la technologie qu'ils ont utilisée. Des résultats prouvant l'éfficacité des générateurs de la bibliothèque ACTgen sont donnés dans [Bidd 95].

4.2. Flot de synthèse

Le flot de synthèse, implanté dans ASYL+, à partir d'une description VHDL est donné à la figure II.34. Tout d'abord, la syntaxe et la sémantique de la description VHDL sont analysées et si la description est correcte, une structure interne sous forme d'un arbre abstrait est remplie, reprenant toutes les informations contenues dans la description VHDL.

L'arbre abstrait ne sert en fait que d'interface entre l'analyse et la synthèse. A partir de l'arbre abstrait est remplie une seconde structure : la structure RTL qui est utilisée par la synthèse. La structure RTL est une structure qui reste encore très proche de celle de l'arbre abstrait. A ce niveau, le circuit est décrit comme un ensemble d'affectations reliant les sorties et les signaux internes, aux signaux internes et aux entrées. En partie droite de ces affectations peuvent apparaître de simple opérateurs Booléens ou des opérateurs plus complexes tels que des opérateurs arithmétiques ou relationnels. Ces affectations peuvent être conditionnelles et la condition peut être une simple expression, ou quelque chose de plus particulier, comme l'attente d'un événement (front montant par exemple) sur un signal, qui sera alors reconnu comme signal d'horloge. La structure RTL doit son nom au fait qu'elle permette de décrire des

de type RTL, c'est-à-dire des opérations Booléennes, arithmétiques et relationnelles entre des fils ou des registres. Notons que cette structure est indépendante du langage de description utilisé et qu'elle peut tout aussi bien être remplie à partir d'une description dans un langage VHDL que dans tout autre langage de description de haut niveau tel que Verilog par exemple.

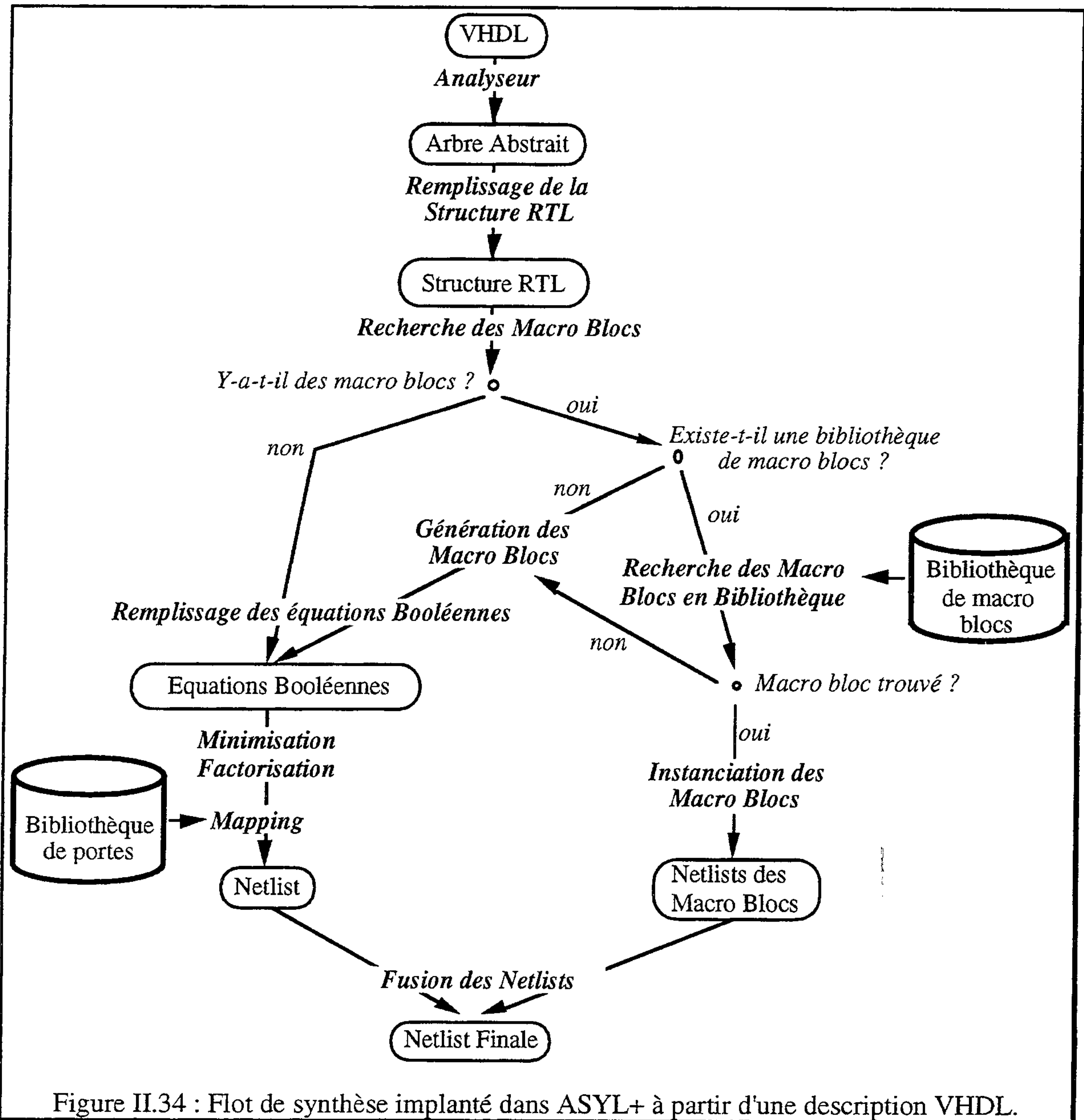

Figure II.34 : Flot de synthèse implanté dans ASYL+ à partir d'une description VHDL.

C'est sur cette structure RTL que se fait la recherche des macro blocs. Chaque affectation est considérée l'une après l'autre et suivant les expressions apparaissant en partie droite et les conditions pour les affections conditionnelles, un certain nombre de macro blocs peuvent être décelés. Il s'agit plus particulièrement des additionneurs, des soustracteurs, des multiplieurs,

des incrémenteurs, des décrémenteurs, des comparateurs, des multiplexeurs, des décodeurs, des registres, des compteurs et des RAMs.

Suivant le résultat de la recherche de macro blocs, le flot de synthèse se divise en deux branches. Si le résultat est négatif, cela signifie qu'aucun macro bloc n'a pu être décelé et donc le circuit est constitué d'un ensemble d'équations faisant éventuellement intervenir un certain nombre de portes de base du style bascule, verrou ou porte trois-états. Dans ce cas, la structure RTL est traduite en une liste d'équations Booléennes. C'est sur ces dernières que vont travailler les algorithmes de synthèse de bas niveau réalisant les étapes de minimisation, de factorisation et de mapping. Lors du mapping, la bibliothèque de portes de bas niveau de la technologie cible est prise en compte de sorte à obtenir une netlist, c'est-à-dire une description structurelle du circuit contenant les éléments physiques de la technologie considérée.

Si le résultat est positif, cela signifie qu'au moins un macro bloc a été trouvé et éventuellement plusieurs. Dans ce cas, le circuit est donc composé d'un ou plusieurs macro blocs, plus probablement, un ensemble d'équations et de portes de base (de type bascule, porte trois-états, ect.). La synthèse des équations et des portes de base suit le flot décrit précédemment. Pour l'ensemble des macro blocs, un traitement spécifique intervient. Deux cas et donc deux flots peuvent être distingués selon si il existe ou non une bibliothèque de macro blocs.

Dans le premier cas, si il n'existe pas de bibliothèque de macro blocs, alors chaque macro bloc va être généré. Pour chaque macro bloc, cette génération se fait suivant les critères de vitesse et de surface donnés par l'utilisateur, par un choix au niveau des équations pré stockées en interne dans le système de synthèse. Plus de détails sur la génération de ces macro blocs seront donnés dans le paragraphe qui suit. Ensuite, après avoir choisi les équations appropriées, le flot de synthèse logique habituel s'opère. Après les étapes de minimisation, factorisation et mapping, une netlist est générée, incluant également des macro blocs, mais dans ce cas, ces macro blocs sont réalisés à partir des portes de base de la technologie.

Dans le second cas, l'utilisateur spécifie une bibliothèque de macro blocs. En effet, pour certaines technologies, il existe déjà des bibliothèques de macro blocs conçus de telle sorte que leurs performances en surface et en chemin critique sont bien meilleures que celles obtenues dans le cas d'une génération telle que nous l'avons décrite plus haut [Bidd 95]. (Sinon, pourquoi se casser la tête à faire des bibliothèques, me direz-vous ?) Donc, dans ce cas, chaque macro bloc est considéré séparément et une requête est faite au gestionnaire de bibliothèque de sorte à rechercher dans la bibliothèque l'élément correspondant. Plusieurs types de requête ont été mis en place et nous y reviendrons plus tard. Ces requêtes permettent notamment de prendre en compte les critères de synthèse spécifiés par l'utilisateur. Ensuite, deux cas peuvent se

présenter suivant si la recherche du macro bloc en bibliothèque échoue ou non. Si la recherche échoue, alors dans ce cas l'issue de secours consiste à retourner sur le flot précédent permettant de générer le macro bloc. Si la génération n'est pas possible (par exemple, s'il n'existe pas encore de traitement pour le macro bloc considéré) alors il ne reste plus rien à faire si ce n'est de générer une boîte vide contenant le bon nombre de ports avec les bons noms. A l'utilisateur ensuite à remplir lui-même la boîte vide qui figurera au niveau de la netlist finale du circuit. Par contre, si la recherche aboutit, alors dans ce cas le macro bloc est instancié et la netlist correspondante est générée. Ensuite il ne reste plus qu'à fusionner les netlists des macro blocs avec la netlist correspondant au reste du circuit pour obtenir la netlist finale. Suivant la technologie considérée, cette fusion peut se faire soit à l'intérieur du système de synthèse (comme c'est le cas pour la bibliothèque ACTgen d'Actel), soit à l'extérieur (comme c'est le cas pour la bibliothèque XBLOX de Xilinx). Nous reviendrons sur ces diverses interfaces en fin de ce chapitre.

4.3. Génération de macro blocs

Lorsque des macro blocs figurent au niveau de la description VHDL sous forme d'opérateurs arithmétiques (+, -, *) ou relationnels (=, =/, <, <=, >, >=), qu'ils ont été détectés au niveau de la structure RTL et qu'aucune bibliothèque de macro blocs n'a été spécifiée par l'utilisateur, ou que les macro blocs correspondant ne figurent pas au niveau cette bibliothèque, alors ils sont générés automatiquement par le système de synthèse. Ces outils de génération de macro blocs et leur intégration dans un système de synthèse architecturale ont notamment été abordés dans [Chen 90] et [Thee 95].

Cette génération prend en compte les critères de synthèse fournis par l'utilisateur ainsi que la technologie utilisée. Actuellement, diverses équations ont été implantées dans le système de synthèse ASYL+. Peuvent être générés automatiquement les macro blocs suivant : additionneurs, soustracteurs, incrémenteurs, décrémenteurs, comparateurs et multiplieurs.

4.3.1. Additionneurs

Suivant le critère de synthèse (surface minimale ou chemin critique minimal) et la technologie cible, plusieurs équations sont proposées pour les additionneurs.

Si le critère de synthèse est la surface, les équations bien connues des additionneurs à retenue bondissante sont utilisées. Ces équations sont ensuite mappées sur une bibliothèque de portes de base. Si cette bibliothèque contient des cellules élémentaires d'additionneur deux bits, alors ces cellules sont automatiquement utilisées.

Si le critère de synthèse est la vitesse, dans ce cas le calcul de la retenue (qui conditionne le chemin critique de l'additionneur) doit être anticipé. La retenue est alors générée par une structure d'arbres binaires dont les noeuds sont les termes de propagation et de génération utilisés pour le calcul de la retenue. En fait, selon la technologie cible, deux sortes d'additionneurs basés sur cette structure sont utilisés. Si la technologie est construite à base de multiplexeurs comme c'est le cas pour Actel ou Quicklogic, alors l'additionneur généré est basé sur l'additionneur de Brent-Kung. Pour les autres technologies comprenant un ensemble de portes de base, l'additionneur généré est basé sur l'additionneur de Von Neuman.

Pour d'autres technologies plus particulières, l'additionneur généré utilise un mécanisme offert par la technologie, spécialement conçu pour permettre un calcul rapide de la retenue, et ceci quelque soit le critère de synthèse. Il s'agit plus précisément des technologies XC4000 de Xilinx et FLEX8000 d'Altera.

Pour plus de détails sur toutes ces structures d'additionneurs et pour avoir des résultats comparatifs entre ces différentes structures, il est possible de se reporter à [Guyo 94], [Soue 89], [Mull 89] ou encore à [ASYL].

4.3.2. Soustracteurs

Pour obtenir un soustracteur à partir d'un additionneur, il suffit d'inverser une opérande plus éventuellement la retenue entrante. En fait, l'équation A - B est dérivée de A + (-B) où (-B) est le complément à deux de B. Rappelons que pour réaliser le complément à deux de B il suffit d'inverser B et de rajouter un. Donc, les équations des soustracteurs sont les même que celles utilisées pour les additionneurs moyennant l'adaptation qui vient d'être décrite.

4.3.3. Incrémenteurs et décrémenteurs

Les équations utilisées pour les incrémenteurs et les décrémenteurs sont les même que celles utilisées pour les additionneurs et les soustracteurs. Notons simplement que l'opérande constante est propagée à l'intérieur des équations de sorte à optimiser ces équations. D'ailleurs ceci est fait d'une façon plus générale chaque fois qu'un opérateur est utilisé et que l'une de ces deux opérandes est constante.

4.3.4. Comparateurs

Comparateurs d'égalité et d'inégalité :

Indépendamment du critère de synthèse, un seul type d'équation est utilisé pour les comparateurs d'égalité et d'inégalité. Pour un comparateur d'égalité sur N bits, l'équation est :

S = not [[A(0) xor B(0)] or [A(1) xor B(1)] or ... or [A(N-1) xor B(N-1)]]

Pour un comparateur d'inégalité sur N bits, l'équation est :

S = [A(0) xor B(0)] or [A(1) xor B(1)] or ... or [A(N-1) xor B(N-1)]

Ces équations peuvent être implantées selon deux critères de synthèse qui sont la surface et le chemin critique.

Comparateurs de supériorité, de supériorité ou égalité, d'infériorité, et d'infériorité et inégalité :

Pour ces quatre types d'opérateurs, les équations utilisées sont les même que celles utilisées pour les soustracteurs.

4.3.5. Multiplieurs

Pour les multiplieurs, pour l'instant un seul type d'équations basées sur le multiplieur de Braun est implanté [Kore 93], [Touz 93]. Ces équations peuvent être implantées selon deux critères de synthèse qui sont la surface et le chemin critique. Un autre type de générateur de multiplieurs ainsi qu'un générateur de diviseurs sont décrits dans [Thee 95].

4.4. Appel à des macro blocs de bibliothèques

A partir d'une description VHDL, il existe trois manières d'appeler des macro blocs de bibliothèques. La plus simple de ces méthodes du point de vue de l'écriture de la description VHDL est l'appel par inférence, ensuite vient l'appel par fonctions ou procédures, et enfin, la plus lourde mais la plus précise est l'appel structurel par instanciation de composants. Notons que ces trois formes d'appel peuvent être mélangées au sein d'une même description. Ces trois types d'appel ont été implantés dans ASYL+ [Bert 93b], [Bert 94b].

4.4.1. Appel par inférence

Définition: ***Inférence*** est le terme employé pour désigner l'action de faire appel à un macro bloc soit par un opérateur arithmétique ou relationnel, soit par un modèle de description dans un langage évolué de type VHDL, Verilog, etc. Il est donc possible de distinguer deux types d'inférence : les ***inférences simples*** ne faisant intervenir qu'un opérateur pour faire appel à

un macro bloc (inférences d'additionneurs, de comparateurs, etc.), ou les ***inférences complexes*** faisant intervenir tout un modèle de description (inférences de registres, compteurs, multiplexeurs, décodeurs, etc.).

La figure II.35 donne un exemple de description VHDL permettant d'inférer un additionneur huit bits. Il ne s'agit là que d'une description incomplète faisant intervenir une inférence simple. Notons que la fonction "+" qui est utilisée dans cet exemple a été surchargée pour le type *"BIT_VECTOR"* et que sa définition figure dans le paquetage VHDL *"PKG_ARITH"* compilé dans la bibliothèque *"ASYL"*.

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity CIRCUIT is
port (      CLK :         in BIT;
            ...
            SORTIE :      out BIT_VECTOR (7 downto 0) );
end CIRCUIT;

architecture ARCH of CIRCUIT is
    signal A, B : BIT_VECTOR (7 downto 0);
    ...
begin
    ...
    SORTIE <= A + B;
    ...
end ARCH;
```

Figure II.35 : Description VHDL permettant d'inférer un additionneur huit bits.

La figure II.36 donne le squelette qui doit être utilisé dans ASYL+ pour inférer un compteur. Il s'agit donc là d'une inférence complexe. Si le compteur est décrit différemment, alors il ne sera pas reconnu en temps que tel par le système de synthèse. L'inférence de compteurs dans ASYL+ a été largement abordée dans [Chou 95].

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity COMPTEUR is
port (      CLK, RST:           in BIT;
            VALID, CHARG:       in BIT;
            INCR_DECR:          in BIT
            ENTREE:             in BIT_VECTOR (N-1 downto 0);
            SORTIE :            out BIT_VECTOR (N-1 downto 0)    );
end COMPTEUR;

architecture ARCH of COMPTEUR is
begin
    process (CLK, RST)
    begin
        if (RST = <'0' | '1'>) then
            SORTIE <= "00..00";
        elsif (CLK'EVENT and CLK = <'0' | '1'>) then
            if (VALID = <'0' | '1'>) then
                if (CHARG = <'0' | '1'>) then
                    SORTIE <= ENTREE;
                elsif (INCR_DECR = <'0' | '1'>) then
                    SORTIE <= SORTIE + '1';
                else
                    SORTIE <= SORTIE - '1';
                end if;
            end if;
        end if;
    end process;
end ARCH;
```

Figure II.36 : Squelette VHDL permettant d'inférer un compteur dans l'outil de synthèse ASYL+.

Lorsqu'un appel à un macro bloc par inférence est utilisé, cela implique au niveau du gestionnaire de bibliothèque qu'il existe une requête permettant de faire une recherche par type d'opération. En effet, si nous considérons par exemple la description donnée à la figure II.35, le gestionnaire de bibliothèque va tout d'abord rechercher les macro blocs réalisant une opération d'addition. Ensuite, parmi cet ensemble de macro blocs, il va rechercher les macro blocs permettant de faire une addition entre deux ports d'entrée sur huit bits et un port de sortie sur huit bits. L'additionneur en question peut posséder une retenue entrante, mais dans le cas où elle n'est pas utilisée, une valeur par défaut doit être indiquée au niveau de la description de la bibliothèque. Enfin, parmi les éléments répondant à tous ces critères, celui qui est choisi est celui qui répond le mieux aux souhaits de l'utilisateur en ce qui concerne la surface et le chemin critique. Nous reparlerons plus en détail au cours du chapitre III des formats de bibliothèques et des données à modéliser en vue de l'utilisation de ces bibliothèques par la synthèse automatique de circuit.

4.4.3. Appel structurel par instanciation de composants

Définition: ***Appel structurel par instanciation de composants*** désigne l'action d'appeler un macro bloc par l'intermédiaire d'une instanciation de boîte au niveau d'une description structurelle de circuit dans un langage de haut niveau tel que VHDL, Verilog, etc.

La figure II.38 donne un exemple VHDL d'appel à un additionneur huit bits par une instanciation de composant. Le composant instancié est le composant "FADD8". Il a été défini au préalable dans le paquetage VHDL *"AC3_COMP"* compilé dans la bibliothèque *"ASYL"*.

```
library ASYL;
use ASYL.AC3_COMP.all;

entity CIRCUIT is
port (       CLK :        in BIT;
             ...
             SORTIE :     out BIT_VECTOR (7 downto 0) );
end CIRCUIT;

architecture ARCH of CIRCUIT is
    signal A, B : BIT_VECTOR (7 downto 0);
    signal UN : BIT;
    ...
begin
    ...
    UN <= '1';
    ADD: FADD8 port map (A,B,UN,SORTIE,open);
    ...
end ARCH;
```

Figure II.38 : Description VHDL faisant appel à un additionneur huit bits par l'intermédiaire d'une instanciation de composant.

FADD8 est un macro bloc de la bibliothèque Actel3. Il s'agit d'un additionneur sur huit bits possédant deux ports d'entrée sur huit bits, une retenue entrante sur un bit dont la valeur par défaut est un lorsqu'elle n'est pas utilisée, un port de sortie sur huit bits donnant le résultat de l'addition et enfin une retenue sortante sur un bit. Les retenues entrante et sortante n'étant pas utilisées dans cet exemple, la première est affectée à la valeur un et la seconde n'est pas connectée.

Pour faire appel à un macro bloc par une instanciation de composant, cela implique donc qu'il existe un paquetage VHDL où sont déclarés les composants correspondant aux macro blocs de la bibliothèque considérée. Les couples entité-architecture correspondant à ces composants peuvent également être décrits mais ne seront en aucun cas synthétisés. Ils peuvent être

[ACTgen], [CSAM], [LPM], il en ressort qu'un certain nombre de paramètres reste commun à l'ensemble de ces bibliothèques, le plus général d'entre eux étant bien évidemment le nombre de bits.

Les valeurs de ces paramètres sont spécifiées différemment suivant l'appel utilisé : appel par inférence, appel par fonctions ou procédures et appel par instanciation de composants. La méthode la plus naturelle de passage de ces valeurs est celle qui est utilisée dans le cas d'un appel de générateurs par instanciation de composants. Dans ce cas nous avons vu qu'un paquetage VHDL définissant les composants de la bibliothèque considérée devait être décrit. Comme ces composants correspondent à des générateurs, non seulement leurs ports sont décrits, mais également leurs paramètres, par l'intermédiaire de la clause générique prévue à cet effet. Un exemple de déclaration de composant correspondant au générateur ADD_SUB de la bibliothèque XBLOX de Xilinx est donné à la figure II.39. Le générateur ADD_SUB est un générateur d'additionneurs, de soustracteurs et d'additionneur-soustracteurs.

```
component ADD_SUB
generic (  N: integer range 1 to MAX_SIZE_XBLOX_MODULE;
           STYLE :           STYLE_ARITH := UNSPECIFIED;
           USE_RLOC :        BOOLEAN := TRUE;
           RLOC_ORIGIN :     STRING := "";
           RLOC_RANGE :      STRING := "";
           TNM :             STRING := ""        );
port (  A :          in BIT_VECTOR (N - 1 downto 0);
        B :          in BIT_VECTOR (N - 1 downto 0);
        C_IN :       in BIT := '0';
        ADD_SUB :    in BIT := '1';
        FUNC :       out BIT_VECTOR (N - 1 downto 0);
        C_OUT :      out BIT;
        OVFL :       out BIT     );
end component;
```

Figure II.39 : Déclaration VHDL du composant correspondant au générateur ADD_SUB de la bibliothèque XBLOX

Notons que dans cet exemple, des valeurs par défaut sont spécifiées pour certains paramètres ainsi que pour certains ports. Ces valeurs sont celles données dans la documentation décrivant la bibliothèque. Au moment de l'instanciation d'un tel composant, la valeur de chaque paramètre peut être fournie au niveau de la clause VHDL *"generic map"* . Les paramètres n'ayant aucune valeur par défaut doivent obligatoirement être spécifiés au niveau de cette clause. La figure II.40 donne un exemple d'instanciation d'un tel composant. Dans cet exemple, nous supposons que le composant ADD_SUB décrit à la figure II.39 est déclaré dans un paquetage VHDL nommé *"XBLOX_BV"* et que ce paquetage est compilé dans la bibliothèque VHDL *"ASYL"* . Dans l'exemple de la figure II.40, seuls les paramètres N

(correspondant au nombre de bits) et STYLE (correspondant à l'architecture implantée) sont positionnés respectivement aux valeurs 8 et RIPPLE. Au niveau des ports, les ports d'entrée C_IN et ADD_SUB ne sont pas connectés et ont pour valeur les valeurs par défaut spécifiées au niveau de la déclaration VHDL du composant.

```
library ASYL;
use ASYL.XBLOX_BV.all;

entity CIRCUIT is
port (      CLK :       in BIT;
            ...
            SORTIE :    out BIT_VECTOR (7 downto 0) );
end CIRCUIT;

architecture ARCH of CIRCUIT is
    signal I1, I2 : BIT_VECTOR (7 downto 0);
    ...
begin
    ...
    ADD: ADD_SUB
        generic map ( N => 8, STYLE => RIPPLE);
        port map (  A => I1, B => I2, FUNC => SORTIE,
                    C_OUT => open, OVFL => open     );
    ...
end ARCH;
```

Figure II.40 : Description VHDL faisant appel au générateur ADD_SUB de la bibliothèque XBLOX par l'intermédiaire d'une instanciation de composant.

Dans le cas d'un appel à un générateur par inférence ou par une fonction ou une procédure, les valeurs des paramètres doivent être déduites par l'outil de synthèse. Nous pouvons distinguer à ce niveau trois catégories de paramètres. La première catégorie concerne les paramètres dont valeurs peuvent être spécifiées implicitement au niveau de la description VHDL. Par exemple, la taille d'un additionneur est implicitement donnée par la taille des opérandes sur lesquelles s'effectue l'addition. Cependant, il n'est pas possible de déduire la valeur de tous les paramètres associés à un générateur à partir de la description VHDL. C'est là qu'interviennent les deux autres catégories de paramètres. Tout d'abord, il y a ceux dont les valeurs peuvent être déduites par l'outil de synthèse en fonction des performances du circuit requises par l'utilisateur. Par exemple, pour le paramètre STYLE du générateur ADD_SUB, correspondant à l'architecture implantée, sa valeur peut être positionnée par l'outil de synthèse en fonction des critères de synthèse spécifiés par l'utilisateur. En effet, au niveau de la documentation, plusieurs valeurs sont proposées et par exemple, la valeur RIPPLE est conseillée afin d'obtenir une surface minimale. Enfin, la troisième catégorie de paramètres englobe tous les paramètres restant, c'est-à-dire tous ceux pour lesquels aucune valeur ne peut être déduite par l'outil de

synthèse à partir de la description VHDL ou des critères de synthèse spécifiés par l'utilisateur. Parmi ces paramètres, nous trouvons par exemple les paramètres USE_RLOC, RLOC_ORIGIN et RLOC_RANGE du générateur ADD_SUB de la bibliothèque XBLOX ; ces trois paramètres permettent de donner des informations au niveau du placement des blocs qui seront prisent en compte par l'outil de placement et de routage de Xilinx. Ces valeurs ne pouvant pas être déduites doivent donc forcément être spécifiées explicitement par l'utilisateur par l'intermédiaire d'un fichier de directives fourni au moment de la synthèse ou par des méta commentaires figurant au niveau de la description VHDL. Dans ce dernier cas, le compilateur VHDL utilisé doit aussi tenir compte des commentaires.

De sorte à générer correctement le bloc fixe demandé, le générateur doit donc avoir à sa disposition le nom du bloc à générer ainsi que la valeur des paramètres s'y rapportant. C'est le système de synthèse qui doit fournir ces informations. L'interface entre l'outil de synthèse et le générateur peut se faire soit par l'intermédiaire d'une netlist où figurent toutes ces informations, soit par une commande permettant d'appeler directement le générateur dans le cadre d'un appel système. Ces deux méthodes seront illustrées dans les deux points suivant, respectivement pour les technologies Xilinx et Actel.

4.5.2. Exemple d'interface avec la bibliothèque de générateurs XBLOX de Xilinx

Xilinx propose une bibliothèque de générateurs : XBLOX. Le système de synthèse ASYL+ permet de faire appel à ces générateurs par les différentes méthodes que nous venons de voir : appel par inférence, appel par fonction ou procédure et appel par instanciation de composant [Bert 93a]. L'interface entre l'outil de synthèse ASYL+ et les outils de Xilinx se fait par l'intermédiaire de netlists. Cette interface ainsi que le flot de synthèse mis en place dans ASYL+ pour utiliser la bibliothèque XBLOX sont illustrés à la figure II.41. Pour être lues par les outils de Xilinx, ces netlists doivent être écrites dans un format propre à Xilinx : le format XNF [XNF].

Pour chaque macro bloc instancié, une netlist XNF donnant le nom et les ports du générateur XBLOX, ainsi que les valeurs des paramètres associés est générée par ASYL+. La netlist générale décrivant l'interconnexion de ces macro blocs avec le reste du circuit est également fournie par ASYL+. Ensuite, ces netlists sont traitées par les outils de Xilinx. Celles correspondant aux macro blocs sont lues par l'outil XBLOX permettant d'en générer le contenu et les netlists résultantes sont fusionnées avec la netlist générale du circuit afin d'obtenir la netlist globale finale du circuit. C'est cette dernière qui est utilisée par les étapes ultimes de placement, routage et programmation du boîtier Xilinx.

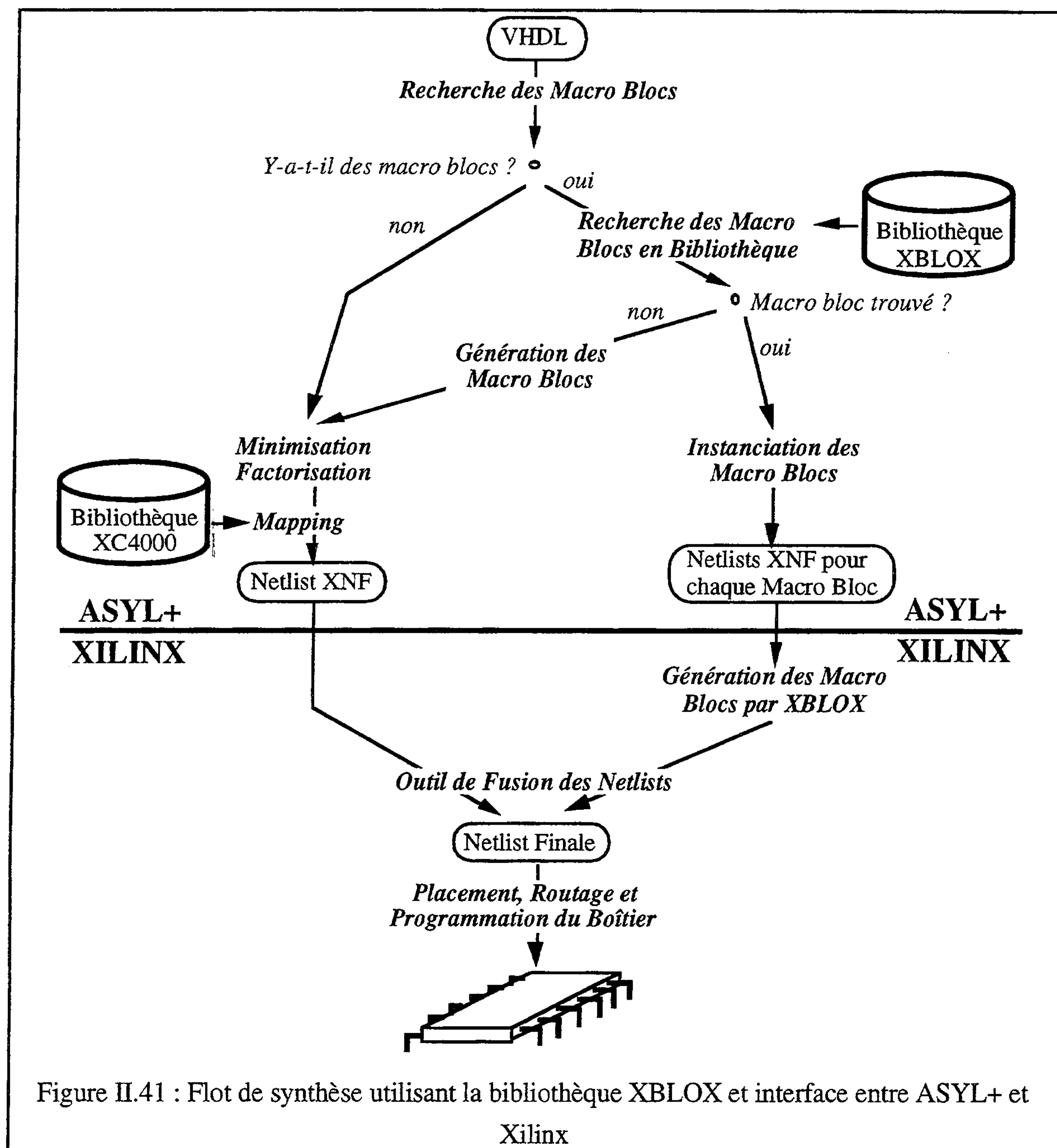

Figure II.41 : Flot de synthèse utilisant la bibliothèque XBLOX et interface entre ASYL+ et Xilinx

4.5.3. Exemple d'interface avec la bibliothèque de générateurs ACTgen d'Actel

Actel propose également une bibliothèque de générateurs : ACTgen. Ces générateurs peuvent être appelés dans le système de synthèse ASYL+. Le flot de synthèse ainsi que l'interface entre ASYL+ et Actel dans le cadre de l'utilisation des générateurs ACTgen sont illustrés à la figure II.42. Lors de l'instanciation d'un macro bloc, ASYL+ appelle directement le générateur ACTgen d'Actel par une commande système où sont indiqués le nom et les valeurs des paramètres nécessaires pour générer le macro bloc requis. Ensuite Actel retourne une netlist

correspondant au macro bloc généré. Cet appel système est totalement transparent pour l'utilisateur. Après avoir récupéré les netlists de chaque macro bloc et après avoir synthétisé la netlist globale du circuit, ASYL+ réalise alors la fusion de ces netlists afin d'obtenir la netlist finale du circuit. Ces netlists sont écrites dans le format standard EDIF [EDIF] ou le format ADL spécifique à Actel. La netlist finale est alors lue par les outils d'Actel qui réalisent le placement, le routage puis la programmation du boîtier.

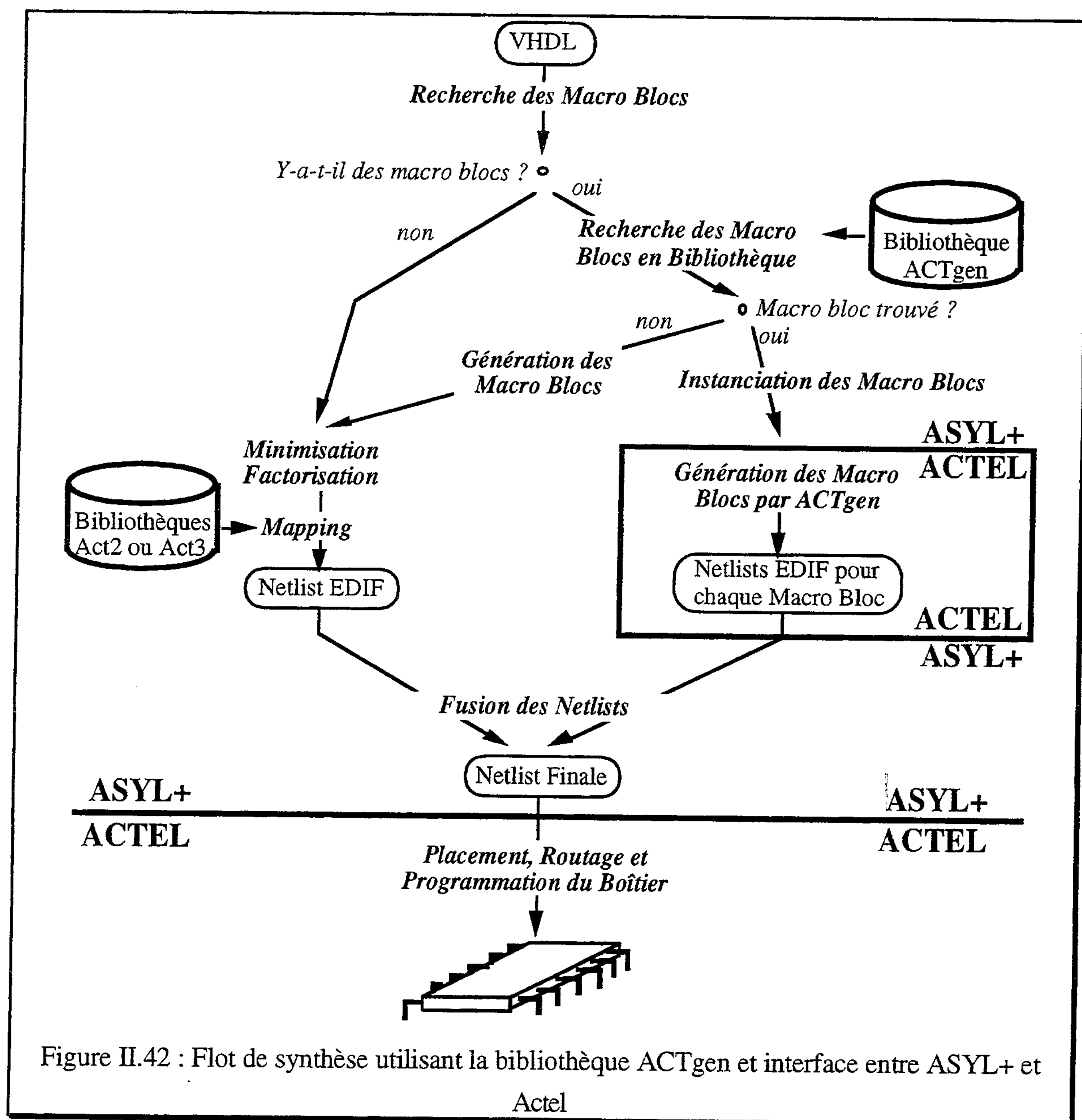

Figure II.42 : Flot de synthèse utilisant la bibliothèque ACTgen et interface entre ASYL+ et Actel

4.6. Optimisations

Concernant les macro blocs et leur utilisation, un certain nombre d'optimisations très simples ont été mises en place dans le système de synthèse ASYL+ et permettent de minimiser la surface ou (et) le temps de réponse du circuit.

4.6.1. Optimisations apportant un gain en surface

Afin de diminuer la surface du circuit, dans certains cas, il est possible, dans une première étape, de partager des opérateurs, et il est également possible, dans une seconde étape, de minimiser le nombre de multiplexeurs se trouvant en entrée de ces opérateurs.

Le ***partage d'opérateurs*** , appelé également ***pliage d'opérateurs*** , consiste à n'utiliser qu'un seul opérateur pour réaliser plusieurs opérations spécifiées. Pour ce faire, si deux opérations doivent être exécutées au même instant, il faut que les conditions portant sur ces opérations soient mutuellement exclusives et il faut également que ces deux opérations puissent être réalisées par le même opérateur. Ceci constitue donc une première étape au niveau de l'optimisation. Un algorithme efficace d'identification d'opérateurs mutuellement exclusifs dans le cadre d'une description algorithmique de haut niveau est présenté dans [Juan 94].

Lorsqu'il y a partage d'opérateurs, l'opérateur résultant devant réaliser plusieurs opérations, un certain nombre de multiplexeurs vont donc figurer en entrée de cet opérateur, permettant, selon une condition, de réaliser l'opération requise. Lorsque les opérations réalisées par ce même opérateur ont des entrées communes, il est également possible de minimiser le nombre de ces multiplexeurs en jouant sur la commutativité des opérations. Pour plus de détails sur les algorithmes pouvant être mis en jeu pour minimiser les multiplexeurs, il est possible de se reporter à [Kuku 90], [When 91], [Mign 92] ou encore [Jian 94]. Ceci constitue la seconde étape au niveau de l'optimisation.

La figure II.43 donne un exemple de description VHDL où sont spécifiées deux opérations d'addition. Les conditions sur ces deux additions sont exclusives donc un seul additionneur est nécessaire. La figure II.44(b) donne le résultat obtenu après cette première étape d'optimisation. De plus, l'opération d'addition étant commutative, il est possible de permuter les opérandes d'une addition, de sorte à ce que l'entrée A figure sur une même entrée de l'additionneur, afin de n'utiliser qu'un seul multiplexeur au lieu de deux. La figure II.44(c) donne le résultat obtenu après cette seconde et dernière étape d'optimisation.

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXEMPLE is
port (  A,B,C :     in BIT_VECTOR (7 downto 0);
        E :         in BIT;
        S :         out BIT_VECTOR (7 downto 0)   );
end EXEMPLE;

architecture ARCHI of EXEMPLE is
begin
    with E select
        S <=    A + B when '1',
                C + A when others;
end ARCHI;
```

Figure II.43 : Description VHDL faisant intervenir le partage des opérateurs et la minimisation des multiplexeurs.

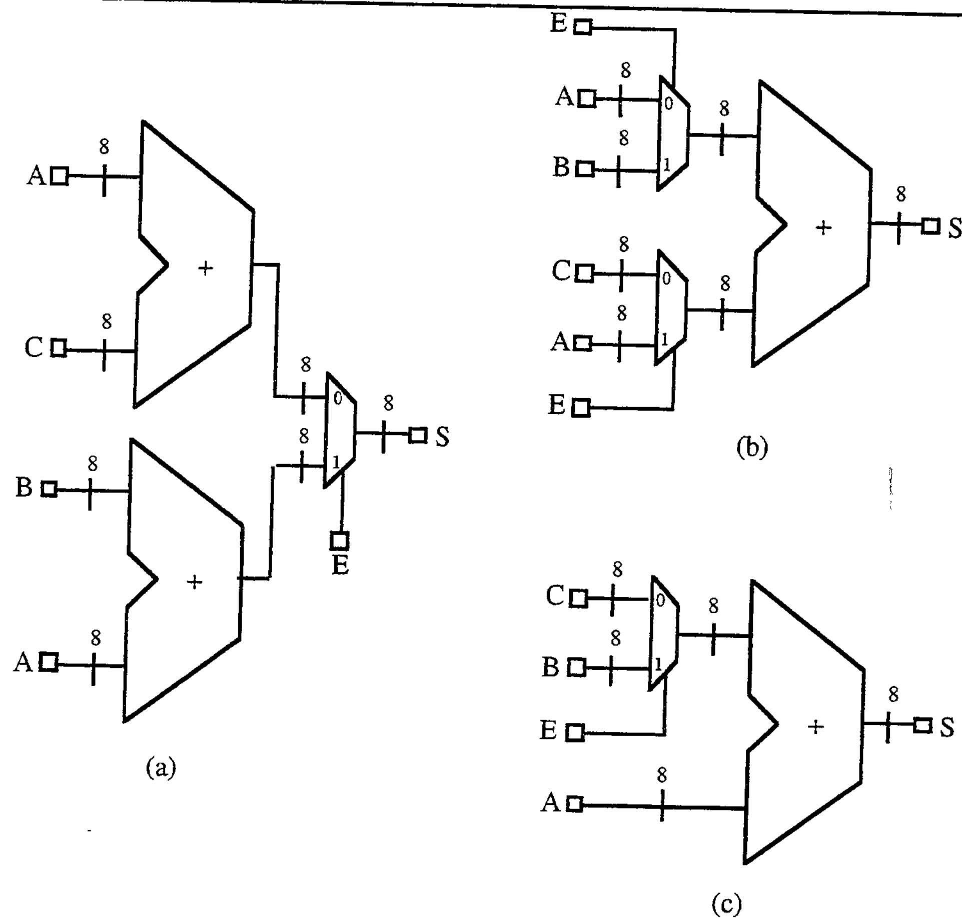

Figure II.44 : Circuit synthétisé correspondant à la description donnée à la figure II.43.
(a) Circuit initial (b) Circuit après le partage d'opérateurs
(c) Circuit final après minimisation du nombre de multiplexeurs en entrées d'opérateurs.

Le circuit synthétisé correspondant à la description VHDL de la figure II.43, après optimisations, est donné à la figure II.44(c). Le circuit optimisé n'est constitué que d'un additionneur et un multiplexeur sur une entrée au lieu de deux additionneurs et un multiplexeur en sortie pour le même circuit non optimisé (figure II.44(a)).

De manière encore plus générale, le système de synthèse ASYL+ est également capable de reconnaître des sous-expressions communes et de les partager. Ceci revient en fait à une factorisation.

Par exemple, dans la description donnée à la figure II.45, la sortie S est affectée soit par (A * B) + (C * D) quand F vaut '1', soit par (D * E) + (A * B) quand F vaut '0'. Les conditions portant sur l'affectation de S sont exclusives. Nous allons donc chercher à minimiser le circuit en essayant de partager des opérateurs, voire même des sous-expressions au niveau des deux expressions figurant en partie droite de l'affectation de S. La sous-expression (A * B) est reconnue comme étant commune aux deux expressions et est donc factorisée. Pour réaliser les deux opérations d'addition, un seul additionneur est nécessaire puisque les conditions sur F sont exclusives. De même pour réaliser les opérations (C * D) et (D * E) intervenant respectivement dans la première et la seconde expression, un seul multiplieur est nécessaire. En jouant sur la commutativité des opérations d'addition et de multiplication, il est possible de permuter les entrées de la seconde addition de sorte à ce que l'expression (A * B) figure toujours en opérande gauche de l'addition, et ainsi supprimer un multiplexeur au niveau du résultat de synthèse. De même, au niveau des sous-expressions (C * D) et (D * E), la multiplication étant également commutative, les entrées de la seconde multiplication peuvent être permutées de sorte à supprimer un multiplexeur.

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXEMPLE is
port (  A,B,C,D,E :     in BIT_VECTOR (7 downto 0);
        F :             in BIT;
        S :             out BIT_VECTOR (15 downto 0)   );
end EXEMPLE;

architecture ARCHI of EXEMPLE is
begin
    with F select
        S <=    (A * B) + (C * D) when '1',
                (D * E) + (A * B) when others;
end ARCHI;
```

Figure II.45 : Description VHDL faisant intervenir la reconnaissance d'expressions communes.

Le circuit optimisé synthétisé à partir de la description donnée à la figure II.45 est représenté à la figure II.46. Ce circuit ne contient qu'un additionneur, deux multiplieurs et un multiplexeur au lieu de deux additionneurs, quatre multiplieurs et un multiplexeur si aucune optimisation n'est réalisée.

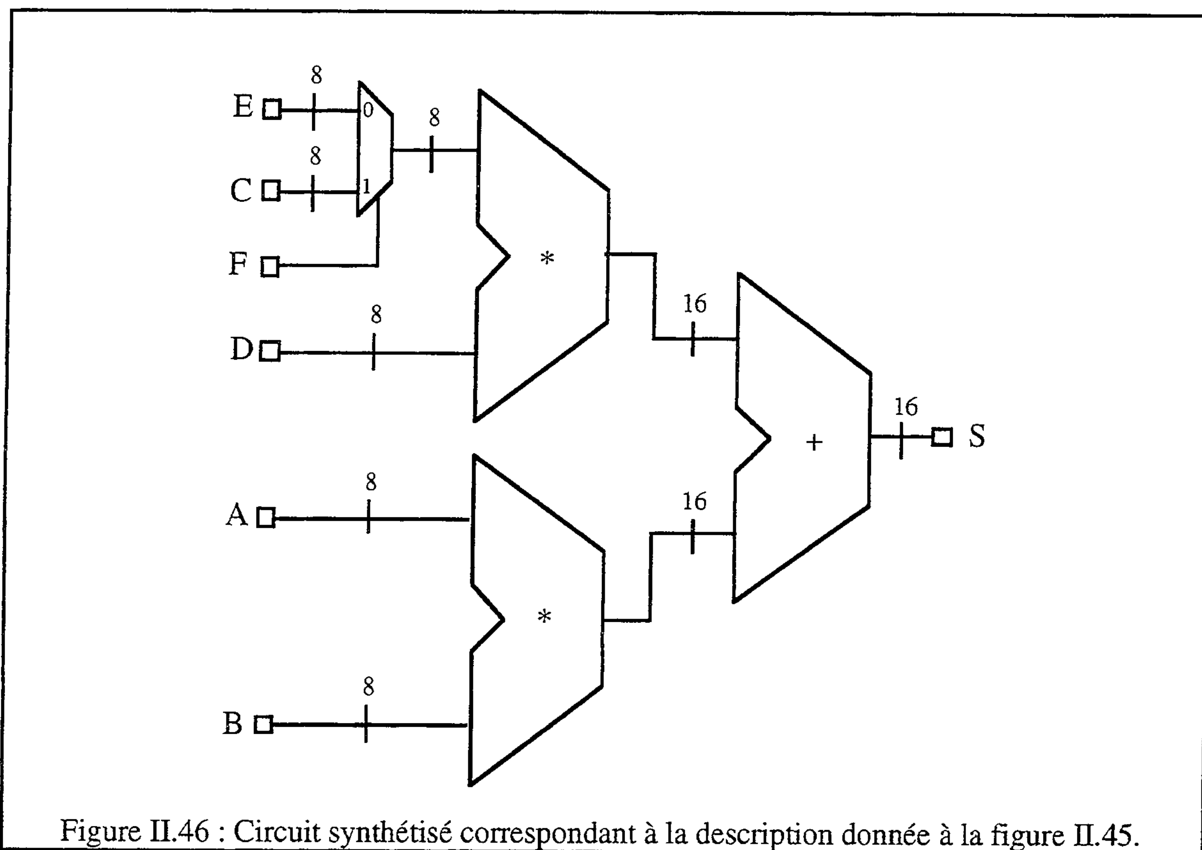

Figure II.46 : Circuit synthétisé correspondant à la description donnée à la figure II.45.

4.6.2. Optimisation apportant un gain en temps

Lorsque la même opération est utilisée plusieurs fois dans une expression, un moyen efficace pour minimiser le temps de traverser du circuit réalisant cette expression est de construire des arbres bien équilibrés. Si l'opération considérée est une opération binaire, la profondeur minimale de l'arbre construit est une fonction en logarithme de deux du nombre d'opérations.

Dans l'exemple qui est donné à la figure II.47, la sous-expression A + B + D + E peut être partagée. Pour réaliser cette sous-expression, un arbre bien équilibré de trois additionneurs est construit. Cet arbre a une profondeur égale à deux. Un multiplexeur permettant en fonction de la valeur de G de sélectionner l'entrée C ou F à additionner à cette sous-expression figure en entrée du quatrième additionneur.

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXEMPLE is
port (  A,B,C,D,E,F :  in BIT_VECTOR (7 downto 0);
        G :             in BIT;
        S :             out BIT_VECTOR (7 downto 0)   );
end EXEMPLE;

architecture ARCHI of EXEMPLE is
begin
   with G select
       S <=   A + B + C + D + E when '0',
              E + B + D + A + F when others;
end ARCHI;
```

Figure II.47 : Description VHDL faisant intervenir la reconnaissance d'expressions.

La figure II.48 donne le schéma du circuit obtenu après optimisation. Le chemin critique de ce circuit passe par trois additionneurs au lieu de quatre dans le cas où les additionneurs seraient mis en cascade les uns derrière les autres.

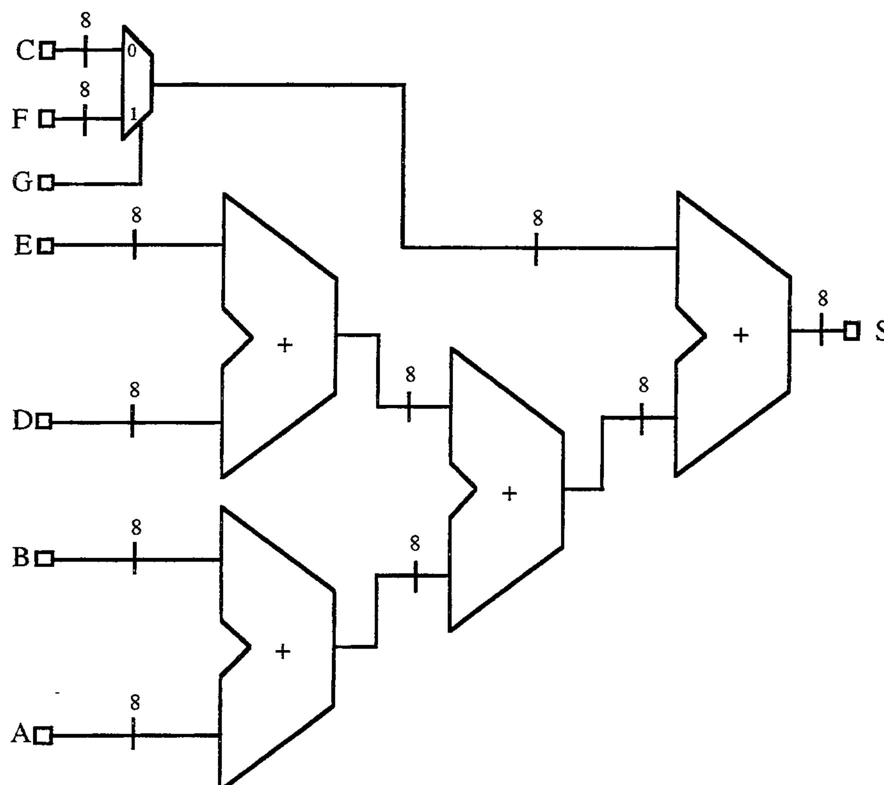

Figure II.48 : Circuit synthétisé correspondant à la description donnée à la figure II.47.

4.6.3. Optimisation apportant un gain en surface et en temps

Une autre optimisation très simple intégrée dans ASYL+ consiste à simplifier les expressions. En effet, dans l'exemple donné à la figure II.49, les expressions A - (2 * A) + A et A - A peuvent être simplifiées car, quelque soit la valeur de A, ces expressions valent toujours zéro. Ainsi, le circuit correspondant à cette description ne contient aucun opérateur et la sortie S est directement connectée à zéro.

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXEMPLE is
port (  A :      in BIT_VECTOR (7 downto 0);
        B :      in BIT;
        S :      out BIT_VECTOR (7 downto 0)   );
end EXEMPLE;

architecture ARCHI of EXEMPLE is
begin
    with B select
        S <=    A - (2 * A) + A when '0',
                A - A when others;
end ARCHI;
```

Figure II.49 : Description VHDL faisant intervenir la simplification d'expressions.

4.6.4. Autres optimisations possibles non intégrées dans ASYL+

D'autres optimisations peuvent également être employées, mais par manque de temps elles n'ont pas encore été toutes intégrées dans ASYL+.

Un type d'optimisations pouvant être appliqué à l'utilisation des macro blocs en synthèse est basé sur la dérivation. Deux classes de dérivation peuvent être distinguées selon si la fonctionnalité ou la taille du macro bloc est considérée.

La dérivation basée sur la fonctionnalité du macro bloc consiste à décomposer cette fonctionnalité en sous-fonctionnalités plus simples. Par exemple, la fonctionnalité de comptage peut être décomposée en deux sous-fonctionnalités : celle d'incrémentation-décrémentation et celle de mémorisation. Un certain nombre de règles de dérivation peuvent ainsi être définies afin de décomposer un macro bloc en sous-macro blocs plus simples. Ainsi, si il n'existe pas de compteur dans la bibliothèque technologique considérée, mais si celle-ci contient un incrémenteur-décrémenteur et un registre, alors ils pourront être utilisés pour réaliser un

compteur. Le résultat obtenu sera alors meilleur que celui fourni après génération des équations d'un compteur.

Le second type de dérivation est basée sur la taille du macro bloc. Deux cas peuvent également être distingués selon si la taille du ou des macro blocs dérivés est supérieure ou inférieure à la taille du macro bloc initial ; mais dans tous les cas, la fonctionnalité du ou des macro blocs dérivés reste identique à celle du macro bloc initial.

Dans le premier cas, le macro bloc initial est dérivé en un autre macro bloc de taille supérieure. Pour ne citer qu'un exemple, prenons le cas d'un additionneur sur quinze bits. Si la bibliothèque technologique ne contient que des additionneurs sur seize bits, il est préférable d'utiliser cet additionneur sur seize bits en mettant les bits de poids forts des opérandes à zéro, plutôt que de générer les équations d'un additionneur quinze bits.

Dans le second cas, le macro bloc initial est dérivé en sous-macro blocs de taille inférieure. Considérons par exemple un additionneur sur seize bits, si il n'existe que des additionneurs huit bits en bibliothèque, il est préférable d'utiliser deux de ces additionneurs en connectant la retenue sortante du premier à la retenue entrante du second, plutôt que de générer les équations d'un additionneur seize bits.

Pour ce second type d'optimisation à partir de dérivation basée sur la taille du macro bloc, le plus difficile est de fixer les limites permettant de savoir à partir de quel moment il est préférable de générer les équations plutôt que d'utiliser les éléments de la bibliothèque. Ceci dépend non seulement de la taille du macro bloc à dériver mais également de la taille des macro blocs disponibles dans la technologie utilisée. Ces optimisations ne pouvant pas être réalisées automatiquement à ce jour, l'utilisateur garde cependant le moyen de forcer par le biais de la description VHDL qu'il donne, l'utilisation des macro blocs de bibliothèque ou des équations générées en internes. En effet, il peut décrire un additionneur seize bits à partir d'additionneur huit bits existant en bibliothèque, comme il peut décrire un additionneur quinze bits en utilisant un additionneur seize bits.

Ce type d'optimisation par dérivation nous a été inspiré par les optimisations proposées dans [Marw 90] et [Land 93], basées essentiellement sur la transformation d'expressions arithmétiques par application de règles algébriques, tout en prenant en compte certaines propriétés associées aux opérations, telles que la commutativité ou l'élément neutre.

5. Conclusion

Dans ce chapitre, nous avons tout d'abord donné des généralités sur les différents niveaux de synthèse ; puis nous avons introduit les limitations que présente le langage VHDL en vue de son utilisation pour la synthèse. Nous avons ensuite présenté la méthodologie de conception d'un circuit à partir d'une description VHDL. Cette méthodologie se décompose en trois étapes : la description, la synthèse et enfin la validation finale. Nous avons également expliqué comment l'utilisateur peut contrôler le processus de synthèse en spécifiant des contraintes. Enfin, nous avons clos ces généralités par quelques mots sur la mise en place d'un environnement VHDL pour la synthèse et l'implication du groupe IEEE dans ce travail.

Dans la seconde partie de ce chapitre, nous avons donné différents modèles VHDL pouvant être utilisés pour la synthèse. Nous avons différencié les modèles de blocs combinatoires (portes logiques de base, équations Booléennes, tables de vérité, portes trois-états), des modèles de blocs séquentiels (verrous, bascules) et nous avons défini la notion de processus VHDL combinatoire. Ce premier ensemble de modèles ne fait intervenir au niveau de la synthèse que des éléments de base des bibliothèques technologiques.

Dans la troisième partie de ce chapitre, nous avons voulu donner des modèles VHDL plus complexes pouvant faire appel à des macro blocs de bibliothèques. Les modèles qui sont donnés sont des modèles de multiplexeurs, de comparateurs, d'opérateurs arithmétiques, de décaleurs, de registres, de compteurs et de mémoires RAM.

Enfin, dans la dernière partie, nous nous sommes davantage intéressés à la synthèse utilisant ces macro blocs. Le flot de synthèse utilisant ces éléments à partir d'une description VHDL a été exposé et nous avons pu distinguer deux cas selon l'existence ou la non existence d'une bibliothèque de macro blocs. Dans le cas où il n'existe pas de bibliothèque de macro blocs, nous avons vu comment les macro blocs peuvent être générés automatiquement. Par contre dans le cas où une telle bibliothèque est spécifiée, nous avons vu les trois méthodes permettant d'appeler ces macro blocs : appel par inférence, appel par fonctions ou procédures et enfin appel structurel par instanciation de composants. Nous avons également exposé le cas particulier de l'utilisation des bibliothèques de générateurs telles que les bibliothèques XBLOX de Xilinx et ACTgen d'Actel. En dernier lieu, nous avons présenté les optimisations mises en place au niveau de l'outil de synthèse ASYL+ utilisant de tels macro blocs.

Au cours de cette dernière partie, nous avons vu que le système de synthèse utilisant des macro blocs de bibliothèques doit faire appel à un gestionnaire de bibliothèque, afin de rechercher l'élément répondant au mieux à la requête formulée. Il en est de même pour la synthèse de bas

niveau utilisant des bibliothèques de portes de base. Pour ce faire, un certain nombre d'informations doit figurer au niveau des descriptions des éléments en bibliothèque. Dans le chapitre suivant, nous allons voir comment sont modélisés les éléments de bibliothèque, qu'il s'agisse de portes de base ou bien d'éléments plus complexes tels que des macro blocs.

Chapitre III

-- ◊ - ◊◊◊ - ◊ --

Modélisation d'éléments de bibliothèques

1. Introduction

Nous venons de voir au cours du chapitre précédent, qu'un système de synthèse automatique de circuits délivre à partir d'une description de ce circuit la netlist correspondante. Cette netlist est constituée d'éléments physiques de la technologie cible considérée et des interconnexions entre ces éléments. Pour ce faire, le système de synthèse doit donc avoir accès à ces éléments qui sont fournis par des fabriquants tels que Actel, Xilinx, VLSI, etc. Les informations les concernant sont généralement décrites sous forme textuelle dans un fichier, chaque fichier correspondant à la liste des éléments d'une même bibliothèque. A l'heure actuelle, il n'existe pas réellement de format universel de représentation de ces informations. C'est ainsi que nous avons vu se développer une multitude de formats de description de ces éléments. Chaque outil de synthèse utilise son propre format et il s'agit par conséquent de formats déposés et protégés, la plupart du temps, par les sociétes qui les développent. Comme tout autre outil de synthèse, ASYL+ possède aussi son propre format. Ce format, les limitations qui en découlent et la définition d'un nouveau format palliant à ses déficiences actuelles vont être exposés dans ce chapitre.

2. Modélisation des éléments de bibliothèques dans ASYL+

Chaque format de bibliothèque étant propre à l'outil de synthèse qui l'utilise et chaque format étant protégé, nous allons ici nous borner à décrire le format de bibliothèque mis en place dans ASYL+. Tout d'abord nous allons spécifier le format défini pour décrire les cellules de faible complexité utilisées par la synthèse de bas niveau. Ensuite, nous verrons comment a été défini le format permettant de décrire les macro blocs utilisés par la synthèse RTL. Enfin, nous verrons quelles sont les limitations de ces deux formats et nous proposerons un nouveau format.

2.1. Modélisation des cellules de bas niveau : format de bibliothèque de bas niveau

Historiquement parlant, ASYL+ a tout d'abord été un outil de synthèse logique (Aide à la SYnthèse Logique). Dans un premier temps, il a fallu définir un format pour décrire les éléments de bibliothèques utilisés par la synthèse de bas niveau. Rappelons que les informations contenues dans ces bibliothèques sont utilisées lors du mapping, c'est-à-dire lors de la projection structurelle des équations Booléennes sur les éléments de la technologie cible. Ces informations sont également nécessaires lors du calcul du chemin critique dans le cadre d'une optimisation ou d'une analyse temporelle du circuit.

Les éléments décrits dans une bibliothèque de bas niveau ASYL+ peuvent se classer en trois catégories : les portes combinatoires, les portes préalablement définies et les portes inutilisables par la synthèse.

2.1.1. Les portes combinatoires

Les portes combinatoires englobent les portes combinatoires simples (AND, OR, NOT, etc.), les portes combinatoires complexes (AOX, OAIX, etc.), les multiplexeurs, etc. Au niveau de la grammaire du format de bibliothèque, ces éléments sont introduits par le mot-clé "GATE". Les informations permettant de décrire ces éléments afin de les utiliser lors de la synthèse sont les suivantes :

- le nom de l'élément dans la bibliothèque ("Gate Name"). Ce nom devra figurer au niveau de la netlist du circuit.

- une information indiquant que l'élément ne doit pas être modifié pendant la synthèse ("Don't Touch").

- une information indiquant que l'élément ne doit pas être utilisé pendant la synthèse ("Don't Use").

- la surface de l'élément ("GateArea").

- la sortance maximale de l'élément ("MaxFanout").

- une équation définissant la fonctionnalité de l'élément. Cette équation relie la sortie de l'élément à ses entrées. Il s'agit d'une équation Booléenne utilisant les opérateurs logiques AND ("*"), OR ("+") et NOT ("!"). Cette expression peut être factorisée. Pour cela des parenthèses peuvent être utilisées.

- la liste des ports d'entrée de l'élément.

Pour chaque port d'entrée, introduit par le mot-clé "PIN", il est indiqué :

- le nom du port figurant dans la bibliothèque ("PinName"). Ce même nom doit figurer au niveau de l'expression donnant la fonctionnalité de l'élément.

- une information indiquant la phase du port ("PinPhase") qui peut être de type inversif, non inversif ou indéterminé.

- une liste d'informations permettant de calculer le chemin critique du circuit. Il s'agit plus précisément de la capacité d'entrée ("InputCapacitance"), de l'entrance ("InputLoad"), du temps de montée ("RiseDelay"), du temps de montée intrinsèque ("RiseFanoutDelay"), du temps de descente ("FallDelay") et du temps de descente intrinsèque ("FallFanoutDelay").

Un exemple décrivant une porte AND à deux entrées est donné à la figure III.1.

```
# Porte AND à deux entrées
GATE     0   0   "an02d"    1.33   3.13   z=(a1*a2);
PIN  a1  NONINV  0.06       0.06   0.62   3.19   0.78   2.69
PIN  a2  NONINV  0.06       0.06   0.62   3.19   0.88   2.69
```

Figure III.1 : Description en bibliothèque d'une porte AND à deux entrées.

2.1.2. Les portes préalablement définies

Les portes préalablement définies englobent les portes trois-états, les portes séquentielles : bascules et verrous, et les cellules d'entrée-sortie. Au niveau de la grammaire du format de bibliothèque, ces éléments sont introduits par le mot-clé "PGATE". Les informations décrivant ces éléments sont les suivantes :

- une information indiquant que l'élément ne doit pas être modifié pendant la synthèse ("Don't Touch").

- une information indiquant que l'élément ne doit pas être utilisé pendant la synthèse ("Don't Use").

- une information indiquant le type de la porte ("ASYLGate Name"). Pour ce faire une liste de noms indépendants de la bibliothèque et correspondants à chaque type d'élément a été

définie. Cette liste est la suivante : - DG pour les bascules D,

- DMUXG pour les bascules D avec entrées multiplexées,
- TG pour les bascules T,
- JKG pour les bascules JK,
- RSG pour les bascules RS,
- LG pour les verrous,
- LMUXG pour les verrous avec entrées multiplexées,
- THG pour les portes trois-états non inversives,
- ITHG pour les portes trois-états inversives,
- BUF pour les portes d'entrée-sortie.

- la définition de l'élément ("GateDefinition"). Cette définition donne les informations suivantes dépendantes de la bibliothèque :

- le nom de l'élément dans la bibliothèque ("LibGate Name"). Ce nom devra figurer au niveau de la netlist du circuit,

- la surface de l'élément ("Area"),

- la liste des ports de l'élément en question. L'ordre dans lequel doivent être donnés ces ports dépend du type de l'élément. C'est d'après cet ordre qu'est déduit la fonctionnalité de chaque port. Par exemple, pour une bascule D, les informations concernant les ports doivent être données dans cet ordre :

- nom du port d'horloge ("Clock"),
- type de synchronisation (front montant ou descendant, ou niveau haut ou bas) ("CType"),
- nom du port de mise à zéro ("Reset"),
- nom du port de mise à un ("Set"),
- nom du port de validation d'horloge ("ClockEnable"),
- nom du port de contrôle de la sortie trois-états ("3StateCtrl"),
- nom du port d'entrée ("Input"),
- nom du port de sortie ("Output"),
- nom du port de sortie complémentée si elle existe ("CompOutput"),
- nom du port de sortie haute impédance si elle existe ("HiImpOutput").

Notons que dans cette grammaire, tous les signaux de contrôle sont actif à un, à moins qu'ils ne soient précédés du signe "!". De même, le signe "*" est utilisé lorsqu'un port n'est pas disponible.

- une liste d'informations temporelles associée à chaque sortie. Pour chaque sortie, cette liste est introduite par le mot-clé "RELATED_OUTPUT_PIN" suivi du nom du port de sortie considéré ("OutputName"), de la sortance maximale ("MaxFanout"), d'une liste donnant les informations temporelles associées à chaque entrée. Pour chaque entrée considérée, cette liste

est introduite par le mot-clé "PIN" suivi des informations suivantes :

- le nom du port d'entrée figurant dans la bibliothèque ("PinName").
- une information indiquant la phase du port ("PinPhase") qui peut être de type inversif, non inversif ou indéterminé.
- une liste d'informations permettant de calculer le chemin critique du circuit. Il s'agit plus précisément de la capacité d'entrée ("InputCapacitance"), de l'entrance ("InputLoad"), du temps de montée ("RiseDelay"), du temps de montée intrinsèque ("RiseFanoutDelay"), du temps de descente ("FallDelay"), du temps de descente intrinsèque ("FallFanoutDelay"), du temps d'arrivée (SetupTime") et du temps de maintien (HoldTime) dans le cas d'un élément séquentiel.

Un exemple décrivant une bascule D active sur front montant d'horloge est donné à la figure III.2.

```
# Bascule D active sur front montant d'horloge
PGATE   0   0   DG     "DF1"   1.00    CLK RECK * * * * D Q * *
   RELATED_OUTPUT_PIN    Q   16
       PIN     CLK    UNKNOWN     1.60  1.00  7.30  0.50  7.30  0.50  0.00  0.00
       PIN     D      UNKNOWN     1.60  1.00  8.10  0.00  8.10  0.00  0.00  0.00
```

Figure III.2 : Description en bibliothèque d'une bascule D active sur front montant d'horloge.

2.1.2. Les portes inutilisables par la synthèse

Certains éléments ne sont pas utilisables directement par la synthèse de bas niveau. Il peut s'agir notamment de portes combinatoires réalisant plusieurs fonctions Booléennes complexes comme par exemple des additionneurs un ou deux bits. Ces éléments doivent tout de même être décrits au niveau de la bibliothèque car lors du calcul du chemin critique dans le cadre d'une optimisation de circuit, les temps de traversée de ces éléments devront être pris en compte. Au niveau de la grammaire du format de bibliothèque, ces éléments sont introduits par le mot-clé "UGATE". Ces éléments sont décrits par les informations suivantes :

- une information indiquant que l'élément ne doit pas être modifié pendant la synthèse ("Don't Touch").
- une information indiquant que l'élément ne doit pas être utilisé pendant la synthèse ("Don't Use").
- le nom de l'élément dans la bibliothèque ("LibGate Name").
- la surface de l'élément ("Area").
- le nombre de ports d'entrée de l'élément ("NbIn").
- le nom de chaque port d'entrée ("InPinName").

- le nombre de ports de sortie de l'élément ("NbOut").

- le nom de chaque port de sortie ("OutPinName").

- une liste d'informations associée à chaque sortie. Pour chaque sortie, cette liste est introduite par le mot-clé "RELATED_OUTPUT_PIN" suivi du nom du port de sortie considéré ("OutputName"), de la sortance maximale ("MaxFanout"), d'une liste donnant les informations temporelles associée à chaque entrée. Pour chaque entrée considérée, cette liste est introduite par le mot-clé "PIN" suivi des informations suivantes :

- le nom du port d'entrée figurant dans la bibliothèque ("PinName").

- une information indiquant la phase du port ("PinPhase") qui peut être de type inversif, non inversif ou indéterminé.

- une liste d'informations permettant de calculer le chemin critique du circuit. Il s'agit plus précisément de la capacité d'entrée ("InputCapacitance"), de l'entrance ("InputLoad"), du temps de montée ("RiseDelay"), du temps de montée intrinsèque ("RiseFanoutDelay"), du temps de descente ("FallDelay"), du temps de descente intrinsèque ("FallFanoutDelay"), du temps d'arrivée (SetupTime") et du temps de maintien (HoldTime) dans le cas d'un élément séquentiel.

- un mot-clé supplémentaire permettant d'identifier les cellules d'additionneur un et deux bits.

Un exemple décrivant un additionneur un bit à trois entrées : A, B et CI et deux sorties : S et CO, est donné à la figure III.3.

```
# Additionneur un bit
UGATE    0   0   FA1A   2.00
   3   A   B   CI
   2   S   CO
   RELATED_OUTPUT_PIN   S    16.00
       PIN    A   UNKNOWN    3.43  3.00  6.40  0.35  6.40  0.35  0.00  0.00
       PIN    B   UNKNOWN    3.43  3.00  6.40  0.35  6.40  0.35  0.00  0.00
       PIN    CI  UNKNOWN    3.43  3.00  6.40  0.35  6.40  0.35  0.00  0.00
   RELATED_OUTPUT_PIN   CO   16.00
       PIN    A   UNKNOWN    3.43  3.00  6.40  0.35  6.40  0.35  0.00  0.00
       PIN    B   UNKNOWN    3.43  3.00  6.40  0.35  6.40  0.35  0.00  0.00
       PIN    CI  UNKNOWN    3.43  3.00  6.40  0.35  6.40  0.35  0.00  0.00
UGATE_IS     adder1  A  B  CI  S  CO   END_UGATE_IS
```

Figure III.3 : Description d'un additionneur un bit.

La grammaire complète de ce format de bibliothèque est donnée en annexe 1. Tous les noms donnés entre guillemets font référence à cette grammaire.

2.2. Modélisation de macro blocs : format de bibliothèque de haut niveau

Dans le cadre de l'outil de synthèse ASYL+, après s'être intéressés à la synthèse logique, nous nous sommes également intéressés à la synthèse RTL. A ce niveau, les informations stockées dans les bibliothèques de cellules de bas niveau ne sont pas adaptées à ce nouveau type de synthèse. C'est ainsi qu'un second format de bibliothèque entièrement conçu pour ce type de synthèse a été défini. Les informations ainsi que les éléments contenus dans ce type de bibliothèque diffèrent de ceux contenus dans les bibliothèques de bas niveau. En effet, il s'agit généralement d'éléments plus complexes et leur fonctionnalité ne peut pas être décrite par une simple équation.

Au niveau de la synthèse RTL, l'élément de base utilisé est une ressource effectuant un ensemble d'opérations. Les informations contenues par une ressource ont été définies après analyse d'un ensemble complet de macro blocs.

Au niveau de la grammaire, chaque élément est introduit par le mot-clé "Resource" dans le cas d'un élément fixe ou par "Generic" dans le cas d'un élément paramétré. Chaque élément est décrit par les informations suivantes :

- le nom de l'élément dans la bibliothèque.
- le type de l'élément. Une liste de mots clés correspondant à chaque type d'élément a été définie. Il peut s'agir d'un opérateur, d'un registre, d'un multiplexeur, etc.
- la surface, la hauteur et la largeur de l'élément.
- la liste des paramètres dans le cas d'un élément paramétré. Chaque paramètre est décrit par son nom, son type (il peut s'agir soit d'un entier, soit d'une chaîne de caractères), l'ensemble des valeurs qu'il est possible de donner à ce paramètre et une valeur par défaut.
- la liste des ports de l'élément. Chaque port est défini par son nom, son type (port des données d'entrée, port des données de sortie, port de commande, etc.) et sa largeur en nombre de bits. Pour les éléments paramétrés, cette largeur peut s'exprimer en fonction d'un des paramètres.
- la liste des opérations que peut effectuer l'élément.

Chaque opération est définie par :

- son nom,
- son type ; une liste de mots clés correspondants à chaque type d'opération (addition, soustraction, décalage, mise à zéro synchrone, etc.) a été mise en place,
- le temps maximum nécessaire pour réaliser cette opération ; ce temps peut être spécifié de trois façons différentes : soit par une constante, soit par une expression, soit par un appel à une fonction.

- une information indiquant si il s'agit d'une opération pipelinée, et dans ce cas, une information supplémentaire permettant d'en indiquer la latence,

- la liste des ports de commande ainsi que la valeur à affecter à chacun de ces ports pour réaliser l'opération en question (notons que la valeur '-' peut être spécifiée afin de permettre d'éventuelles optimisations).

- la liste des ports de données utilisés par cette opération ; pour chaque port d'entrée il est possible d'indiquer si il s'agit d'un port commutatif et il est également possible d'indiquer une valeur par défaut à affecter sur le port d'entrée considéré dans le cas où il ne serait pas utilisé.

Un exemple d'additionneur-soustracteur générique décrit en bibliothèque est donné à la figure III.4. Il s'agit plus particulièrement de l'élément "ADD_SUB" de la bibliothèque XBLOX [XBLOX].

```
# Additionneur générique sur N bits #
(Generic ADD_SUB
   (ResType OPE)
   (Area        FunctionCall "/asyl/lib/info/add_sub.area"      )
   (Height      FunctionCall "/asyl/lib/info/add_sub.height"    )
   (Width       FunctionCall "/asyl/lib/info/add_sub.width"     )
(ParamList
   (ParamDesc N (ParamType INT) (ParamValues 1 Inter 64) (ParamDefaultValue 16)))
(PortList
   (PortDesc A          (PortType INPUT)      (PortWidth (ParamUsed N)))
   (PortDesc B          (PortType INPUT)      (PortWidth (ParamUsed N)))
   (PortDesc C_IN       (PortType INPUT)      (PortWidth 1))
   (PortDesc ADD_SUB    (PortType COMMAND)    (PortWidth 1))
   (PortDesc FUNC       (PortType OUTPUT)     (PortWidth (ParamUsed N)))
   (PortDesc C_OUT      (PortType OUTPUT)     (PortWidth 1))
   (PortDesc OVFL       (PortType OUTPUT)     (PortWidth 1)))
(OpList
   (OpDesc ADD
      (OpType ADD)
         (Delay   FunctionCall "/asyl/lib/info/add_sub.delay" )
         (Pipeline 0)  (Latency 0)
      (OpPortCommandList
         (PortCommandDesc ADD_SUB    (PortDefaultValue 1)))
      (OpPortDataList
         (PortDataDesc A       (PortCommutList B))
         (PortDataDesc B       (PortCommutList A))
         (PortDataDesc C_IN  (PortDefaultValue 0))
         (PortDataDesc FUNC)
         (PortDataDesc C_OUT)
         (PortDataDesc OVFL)))
   (OpDesc SUB
      (OpType SUB)
         (Delay   ( * 0.12 (ParamUsed N)))
         (Pipeline 0)  (Latency 0)
      (OpPortCommandList
         (PortCommandDesc ADD_SUB    (PortDefaultValue 0)))
      (OpPortDataList
         (PortDataDesc A)
         (PortDataDesc B)
         (PortDataDesc C_IN (PortDefaultValue 1))
         (PortDataDesc FUNC)
         (PortDataDesc C_OUT)
         (PortDataDesc OVFL)))
   )
)
```

Figure III.4 : Description d'un additionneur générique sur N bits.

La grammaire complète de ce format de bibliothèque est donnée en annexe 2.

2.3. Limitations de ces formats

2.3.1. Limitations liées à l'existence de deux formats séparés

Une première limitation liée à cette approche est qu'il existe deux formats tout à fait différents. Deux formats impliquent deux analyseurs, deux structures et deux gestionnaires de bibliothèques tout à fait différents. En résumé, cela équivaut à deux fois plus de code à développer.

Un second point négatif est la duplication d'informations. En effet, les éléments de haut niveau peuvent et doivent être décrits dans la bibliothèque de bas niveau pour être pris en compte lors du calcul du chemin critique d'un circuit. Par contre, certains éléments de bas niveau comme les multiplexeurs doivent également être décrits dans la bibliothèque de haut niveau car ils peuvent être reconnus et utilisés directement par la synthèse de haut niveau. La limite entre ces deux bibliothèques est donc très floue et le contenu de chacune d'elles dépend plus des capacités des algorithmes de synthèse qui les utilisent que des éléments à décrire.

2.3.2. Limitations liées au format de bibliothèque de bas niveau

D'autres limitations sont liées aux formats eux-mêmes. Concernant le format de bibliothèque de bas niveau, une première limitation très importante provient du fait qu'il est impossible à l'heure actuelle de décrire des éléments combinatoires ayant plus d'une sortie. Par exemple, un additionneur un bit possédant deux sorties (la somme et la retenue de l'addition) ne peut pas être décrit correctement par une seule équation. Pour un tel élément, seules les informations temporelles s'y rapportant pourront être données.

Une autre limitation provient de la description des informations temporelles. Pour des éléments assez complexes c'est-à-dire ayant plusieurs entrées et plusieurs sorties, il devient rapidement très lourd de spécifier les informations temporelles. En effet, ces informations sont données pour chaque sortie par rapport à chaque entrée, donc pour N sorties et M entrées, la complexité est de N fois M. Actuellement, il est impossible de regrouper des entrées ou des sorties pour simplifier la description de ces informations et leur traitement.

Toujours à propos de la spécification des informations temporelles, les valeurs qui sont données sont des constantes et ne peuvent pas être des expressions pouvant faire intervenir des paramètres spécifiques à l'élément comme le nombre de bits, ou des paramètres généraux à la bibliothèque tel que le processus technologique.

Une dernière limitation également très gênante est liée au fait qu'il est impossible de décrire des éléments paramétrés dans ce format. Il existe dans certaines bibliothèques des éléments de bas niveau sur N bits ; c'est le cas notamment de la bibliothèque XBLOX qui offre des opérateurs Booléens sur N bits (AND, OR, NAND, etc.). Actuellement, ces éléments ne peuvent pas être décrits dans la bibliothèque de bas niveau.

2.3.3. Limitations liées au format de bibliothèque de haut niveau

Une importante défaillance du format de bibliothèque de haut niveau est qu'il n'est pas possible d'indiquer la fonctionnalité des ports. Par exemple, dans le cas d'un additionneur, il n'est pas possible de distinguer le port de retenue sortante du port de dépassement de capacité.

De plus, toujours concernant les informations spécifiques aux ports, il n'est pas possible d'indiquer une valeur par défaut pour un port d'entrée qui ne serait pas utilisé. Pour un même bloc et pour un même port de ce bloc, cette valeur peut varier selon l'opération effectuée. Par exemple, pour un additionneur-soustracteur, la valeur par défaut du port de retenue entrante est généralement '0' pour une addition, et l'inverse, soit '1', pour une soustraction. Ces valeurs dépendent également de la cible technologique considérée et sont indiquées dans les catalogues fournis par les constructeurs.

Actuellement, seul le format de bibliothèque de haut niveau permet de décrire des générateurs paramétrés. Cependant, de nombreuses limitations nuisent encore à la description de tels éléments. En effet, les paramètres considérés ne peuvent être que des entiers ou des chaînes de caractères. De plus, ces paramètres ne peuvent pas être utilisés dans des expressions. Par exemple, s'il est possible de décrire des ports de taille N, N étant un paramètre entier défini précédemment, il n'est pas possible de décrire un port de taille 2xN (taille du port de sortie d'un multiplieur N bits) ou $\log_2(N)$ (taille du port de commande d'un multiplexeur N bits).

Enfin, une dernière limitation provient de la spécification des informations temporelles. Tout d'abord ces informations sont très insuffisantes car il n'est possible d'indiquer qu'une seule valeur de délai par opération réalisée par l'élément. Cette valeur de délai est soit une constante, soit une expression, soit le résultat renvoyé par un appel à une fonction. De plus, les expressions décrites ne peuvent utiliser qu'un nombre restreint d'opérateurs (+, -, *, /) et l'interface avec les fonctions appelées n'est pas suffisamment bien décrite. De plus il existe des éléments de bibliothèque, notamment ceux de la bibliothèque VDP300 [VLSI], pour lesquels les informations temporelles sont données par des expressions différentes selon l'intervalle

dans lequel se situe le nombre de bits de l'élément. Il est bien évident que ceci ne peut pas être décrit par le format de bibliothèque de haut niveau actuel.

2.3.4. Insuffisances générales

Il n'est pas possible de décrire dans chacun de ces deux formats et donc de tenir compte des conditions de fonctionnement du circuit tels que la température, la tension d'alimentation, le processus technologique utilisé lors de la fabrication, etc. Or, tous ces paramètres influent sur les temps de traversée des cellules.

Une seconde limitation provient de la spécification des informations temporelles. Quelle que soit le format de bibliothèque considéré, les méthodes proposées pour les décrire sont inadaptées aux éléments de bibliothèque qui existent aujourd'hui.

Un troisième point moins important concerne les unités employées. Aucune unité n'est précisée dans ces formats de bibliothèque. Les valeurs constantes données sont sans unité. Il peut s'agir de ns comme de pF, or ces unités devraient être utilisées par l'estimateur employé en fin de synthèse pour donner la surface et le chemin critique approximatifs du circuit synthétisé.

2.4. Objectifs du nouveau format

Afin de pallier à toutes ces limitations et de remédier à tous les problèmes rencontrés lors de l'utilisation de ces deux formats de bibliothèque, nous avons décidé de définir un troisième format en remplacement des deux précédents que nous venons de décrire. Les objectifs de ce nouveau format consistent bien évidemment à combler toutes les déficiences des deux précédents que nous venons d'énumérer.

De plus, ce nouveau format doit permettre de faciliter la migration technologique également appelée transfert technologique. Cette opération consiste tout simplement à traduire un circuit implanté sur une cible technologique particulière en un circuit équivalent sur une autre cible technologique. Pour ce faire, un circuit décrit au niveau portes est alors traduit en équations qui sont ensuite traduites à nouveau en portes d'une cible technologique différente. Si cette opération reste encore assez simple pour un circuit n'utilisant que des éléments de bibliothèque de bas niveau, elle se complique lorsque le circuit se compose d'éléments plus complexes tels que des additionneurs, etc. Lors de la définition du nouveau format de bibliothèque, ce problème devra être pris en compte de sorte à faciliter la migration technologique dans le cas de circuits contenant des macro blocs.

Enfin, ce nouveau format doit permettre de faciliter l'estimation du chemin critique en comprimant les données à exploiter. Par exemple, il sera possible de spécifier ces données pour un port de plusieurs bits et non pas obligatoirement pour chaque bit du port. Ce nouveau format permettra également d'améliorer les résultats de cette estimation par la prise en compte de nouveaux paramètres tels que le processus technologique employé lors de la fabrication de ces éléments.

2.5. Proposition de nouveau format

2.5.1. Remarques générales

La description d'une bibliothèque dans le nouveau format se décompose en deux fichiers : un fichier indiquant uniquement les informations temporelles de chaque élément et un fichier donnant toutes les autres informations sur les éléments de la bibliothèque, c'est-à-dire essentiellement les informations structurelles et fonctionnelles. Cette description sous forme de deux fichiers a été choisie pour deux raisons. La première consiste à alléger la description d'une bibliothèque en séparant les informations temporelles, qui sont généralement lourdes, du reste de la description, de sorte à rendre cette dernière plus lisible. La seconde raison provient de la mise en place d'un format standard appelé SDF pour Standard Delay Format [SDF], permettant de décrire les informations temporelles associées à une liste d'éléments interconnectés. Dans le cas où le logiciel de synthèse ASYL+ serait amené à lire ou à générer un fichier dans un tel format, la décomposition en deux fichiers de la description des éléments de bibliothèque facilitera ces opérations.

Ce nouveau format devra permettre la description d'éléments très simples ainsi que d'éléments très complexes. Pour décrire ces derniers de nombreuses informations devront être spécifiées. De sorte à alléger la description des éléments simples, beaucoup d'informations seront optionnelles dans ce nouveau format.

2.5.2. Format du fichier décrivant les informations structurelles et fonctionnelles des éléments de bibliothèque

2.5.2.1. Les différentes méthodes de description d'un élément de bibliothèque

Dans ce nouveau format, un élément de bibliothèque peut être décrit de trois façons différentes. Une seule des ces trois manières sera choisie selon l'élément à décrire. Ces trois méthodes sont les suivantes :

- description par une liste d'équations Booléennes. Cette méthode très simple est utilisée pour décrire les éléments de faible complexité utilisés par la synthèse de bas niveau et la migration technologique. Pour chaque élément, une liste d'équations Booléennes donne sa fonctionnalité. Ces équations permettent d'exprimer la valeur des ports de sortie en fonction de celles des ports d'entrée.

- description par une liste d'opérations. Cette méthode dérive de celle utilisée par l'ancien format de bibliothèque de haut niveau. Ce type de description est plus complexe mais reste indispensable pour décrire et reconnaître efficacement un élément de bibliothèque complexe tel qu'un additionneur 16 bits.

- description structurelle hiérarchique. Cette dernière méthode consiste à décrire la structure de l'élément. Dans ce cas il s'agit d'éléments complexes pouvant se décomposer en plusieurs sous-éléments. Par exemple, un compteur peut se décomposer en un additionneur plus un registre. Cette méthode de description pourra être utilisée lors de la migration technologique d'éléments complexes. Par exemple, pour transférer un compteur d'une technologie à une autre, si aucun compteur n'existe dans la technologie cible, ce compteur sera décomposé en sous-éléments jusqu'à obtention d'une décomposition en sous-éléments ayant des équivalents dans la technologie cible.

2.5.2.2. Informations générales décrites par le nouveau format de bibliothèque

D'une manière générale, un fichier correspond à la description de l'ensemble des éléments d'une bibliothèque. Des informations sont donc données sur la bibliothèque puis sur chaque élément ou cellule de cette bibliothèque. Des paramètres généraux à toute la bibliothèque peuvent être décrits. Ces paramètres pourront ensuite être repris lors de la description des cellules de la bibliothèque.

La spécification de chaque élément se divise en trois parties : la description de ses paramètres spécifiques, celles de ses ports et celle de sa fonctionnalité. Cette fonctionnalité peut être décrite selon les trois méthodes énoncées précédemment : description par une liste d'équations, par une liste d'opérations ou bien par la structure hiérarchique de l'élément. La figure suivante représente les informations générales décrites par le nouveau format de bibliothèque ainsi que les liens hiérarchiques, schématisés par des flèches, existant entre ces informations. Plus de détails sur ces informations sont donnés par la suite.

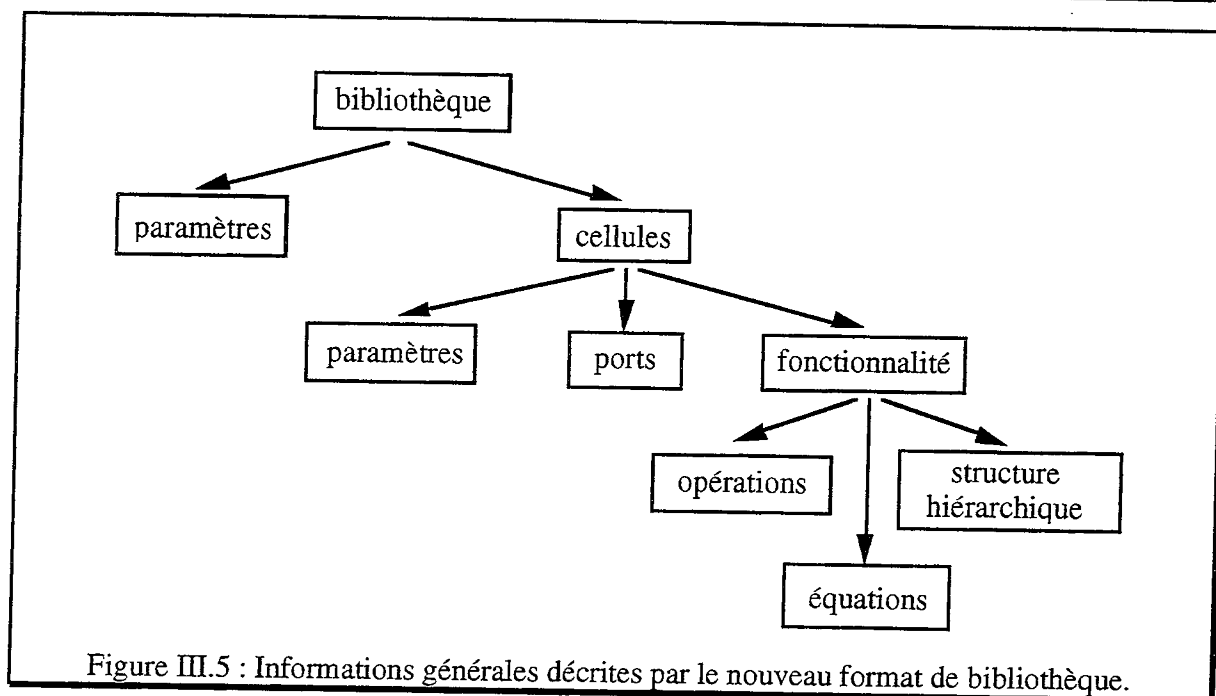

Figure III.5 : Informations générales décrites par le nouveau format de bibliothèque.

2.5.2.3. Informations générales à toutes les cellules de la bibliothèque

Les informations générales à toutes les cellules de la bibliothèque sont :
- le nom de la bibliothèque,
- le style d'indexage des bits d'un vecteur ; suivant les fondeurs l'indexage des bits d'un vecteur se fait de différente manière. Par exemple, le quatrième bit d'un vecteur A peut être noté "A4", "A:4", "A.4", "A[4]", "A_4", etc. selon la technologie utilisée. Mais quelle que soit cette notation, elle reste vraie pour toutes les cellules d'une même bibliothèque.
- les unités de longueur et de surface ; les valeurs indiquant la largeur, la hauteur et la surface des cellules seront toutes données dans ces unités.
- des paramètres généraux à toutes les cellules de la bibliothèque peuvent également être définis à ce moment là.

Après ces informations d'ordre général est donnée la liste des cellules contenue dans la bibliothèque. Un exemple décrivant ces informations est donné à la figure III.6.

```
Library BASHITO
BusStyle "[" "]"
Unit_Length "um"   Unit_Area "um sq"
Param ( Int N, M  1 Inter 32  Default 8 )
Name ...
```

Figure III.6 : Description des informations générales à toutes les cellules d'une bibliothèque.

2.5.2.4. Informations liées aux cellules de la bibliothèque

Une cellule de bibliothèque est décrite par :
- son nom,
- son type : une vingtaine de types ont été définis. Il existe un type pour les compteurs, les multiplexeurs, les portes combinatoires, etc. La liste de ces types est donnée en annexe 3. Il n'existe pas de type "additionneur" car un additionneur est un élément combinatoire (de type "OPE") réalisant une opération d'addition.
- sa généricité ; un élément générique est un générateur. Par exemple, il peut s'agir d'un additionneur N bits.
- sa surface, sa hauteur et sa largeur ; ces valeurs seront utilisées pour estimer la surface du circuit synthétisé.
- ses possibilités d'utilisation en synthèse. Toutes les cellules d'une bibliothèque ne sont pas forcément utilisables par les outils de synthèse. Il peut s'agir par exemple d'un diviseur d'horloge. Cependant, même si de tels éléments ne peuvent pas être utilisés par la synthèse, ils doivent tout de même être pris en compte lors de l'estimation du chemin critique du circuit et doivent également être reconnus lors de la lecture de la description du circuit utilisant de tels éléments.
- la liste de ces paramètres dans le cas d'un générateur. Il peut s'agir par exemple de la taille du bloc, du style d'implantation du bloc, des valeurs de mise à zéro ou à un synchrone ou asynchrone, etc.
- la liste de ses ports. Les informations définissant les ports sont données ci-dessous.
- sa fonctionnalité. Les informations définissant la ou les fonctionnalités d'une ressource sont données ci-dessous.

2.5.2.5. Informations liées aux ports des cellules de la bibliothèque

Le port d'une cellule de bibliothèque est défini par :
- son nom,
- son mode : il peut s'agir d'un port d'entrée, de sortie ou d'entrée-sortie.
- sa fonctionalité : cette information décrit la fonctionalité du port : il peut s'agir d'un port de retenue entrante, d'un port d'horloge, d'un port de donnée, etc. La liste de ces fonctionnalités se trouve en annexe 3.
- son aptitude à sélectionner une opération. Certains ports sont appelés ports de commande car ils permettent de sélectionner l'exécution d'une opération parmi l'ensemble des opérations réalisables par la cellule. Par exemple, un additionneur-soustracteur a généralement un port de commande indiquant à la cellule de faire une addition si la valeur du port est à zéro ou une

soustraction si cette valeur est à un. Cette information est surtout utilisée par les systèmes de synthèse dominée par le contrôle, où le circuit synthétisé se compose d'une partie opérative et d'une partie contrôle.
- son opposition de phase. Cette information indique l'opposition de phase entre l'entrée et la sortie considérées. Par exemple, pour un inverseur, la sortie est en opposition de phase par rapport à l'entrée. Cette information est utilisée dans le cadre de l'optimisation en synthèse de bas niveau.
- son masque d'inversion. Certains ports d'entrée ou de sortie peuvent être inversés. Pour des ports sur plusieurs bits, ce masque est précisé pour chaque bit. Cette information permet également de préciser si un port d'entrée est actif à un ou à zéro (dans le cas d'un port de remise à zéro par exemple).
- sa synchronisation. Cette information précise si le port considéré permet de réaliser une opération synchrone ou asynchrone ainsi que sa priorité par rapport aux autres opérations. Par exemple une mise à zéro asynchrone peut être plus prioritaire qu'une mise à un asynchrone.
- son codage. Pour les ports sur plusieurs bits, il est possible d'en indiquer son codage : notation binaire standard, notation en complément à deux, notation signée. L'implantation de la cellule pourra varier en fonction de ce codage.
- son aptitude à être optionnel. Dans le cadre des générateurs, certains ports peuvent être optionnels. Si le port en question n'est pas utilisé, la logique connectée à ce port n'est pas implantée et la cellule obtenue est donc optimisée.
- son incompatibilité avec d'autres ports (exclusion). Certains ports optionnels peuvent être incompatibles entre eux. Dans ce cas, il faut utiliser soit l'un soit l'autre.
- sa nécessité par rapport à d'autre ports (inclusion). Parmi les ports optionnels, des groupes indissociables peuvent être définis. Autrement dit, la présence d'un des ports du groupe implique la présence de tous les autres ports de ce même groupe.
- sa taille dans le cas d'un port sur plusieurs bits. Un port peut avoir une taille fixe ou variable. Dans ce dernier cas il s'agit d'un port générique. La taille du port est précisée par un intervalle croissant.
- son bit de poids fort dans le cas d'un port sur plusieurs bits.

Un exemple décrivant les informations générales relatives à une cellule de bibliothèque ainsi que la description de ces ports est donné à la figure III.7.

```
# description d'un additionneur : informations générales et description des ports
Name ADDER        TG_Ope      Generic
Area N*4   Length N*2  Width N*2
Touch True        Use True
Port (  A,B       In   Pt_Std   Not_Cmd    Unknow      Port_Not_Inv
                  As   Unsigned     Necessary    Width N,
# description complète des ports A et B.
            S     Out        Width N       )
# description abrégée du port S.
```

Figure III.7 : Description des informations générales et des ports d'un additionneur.

2.5.2.6. Informations liées à la fonctionnalité des cellules de la bibliothèque

Comme il a déjà été précisé, la fonctionnalité d'un élément peut se décrire de trois manières différentes. Ces trois méthodes sont expliquées plus en détails dans ce paragraphe.

• Description par une liste d'équations Booléennes :

Ces équations Booléennes permettent de donner la fonctionnalité de chaque sortie par rapport aux entrées. Elles utilisent les opérateurs suivants : le ET logique (*), le OU logique (+) et l'opérateur d'inversion (!). Un exemple d'équation reliant les entrées a, b et c à la sortie z est donnée ci-après : "z = a * b + ! c".

• Description par une liste d'opérations.

Cette description est plus complexe. Elle s'applique davantage aux macro blocs car il est difficile de décrire leur fonctionnalité par une liste d'équations Booléennes. Une opération est définie par :

- son nom,
- son type : addition, rotation à gauche, etc. La liste de ces types est donnée en annexe 3.
- la liste facultative des équations correspondant à cette opération, qui pourront être utilisées dans le cadre d'une resynthèse au cours d'une migration.
- la liste des ports commutatifs pour l'opération considérée. Par exemple, considérons un additionneur-soustracteur, pour une opération d'addition les ports d'entrée peuvent être commutés alors que pour une opération de soustraction ils ne le peuvent pas. Cette information est utilisée lors de la minimisation des cellules d'interconnexions (multiplexeurs, etc.).
- les valeurs qui doivent être positionnées sur les ports de commande pour réaliser l'opération considérée.
- la liste des ports intervenant lors de cette opération. Tous les ports de la cellule ne sont pas

forcément impliqués par une opération particulière réalisée par cette cellule.
- la liste des conséquences liées à cette opération.
- une information indiquant si il s'agit une opération pipelinée ou non. Dans ce cas l'opération nécessite plusieurs cycles d'horloge pour être effectuée.
- une information indiquant la latence dans le cas d'une opération pipelinée. La latence est le nombre de cycles nécessaires pour effectuer l'opération pipelinée.

Un exemple décrivant l'opération d'addition réalisée par l'additionneur décrit à la figure précédente est donné à la figure III.8.

```
Op ADD Op_Add
Comm (A,B)
Data (A,B,S)
```

Figure III.8 : Description d'une opération d'addition.

• Description structurelle hiérarchique.

Dans ce dernier cas, la fonctionnalité de l'élément est décrit par un assemblage structurel d'autres éléments plus simple contenus dans cette même bibliothèque. Cet assemblage peut également être couplé avec des équations Booléennes.

Ce type de description se décompose en deux parties : une première partie où sont déclarés les signaux internes ainsi que leur taille respective, et une seconde partie reliant ces signaux internes aux ports d'entrée et de sortie par l'intermédiaire d'autres éléments de la bibliothèque ou d'équations Booléennes. Rappelons que ce type de description a été introduit pour faciliter la migration technologique d'éléments complexes.

Un exemple d'accumulateur 8 bits pouvant se décomposer en trois éléments de plus faible complexité : un additionneur, un multiplexeur et un registre, est donné à la figure III.9. La modélisation correspondante de cet élément selon ce type de description est donnée à la figure III.10. Dans cet exemple, l'additioneur, le multiplexeur et le registre sont supposés être des éléments de la bibliothèque et avoir été décrits auparavant.

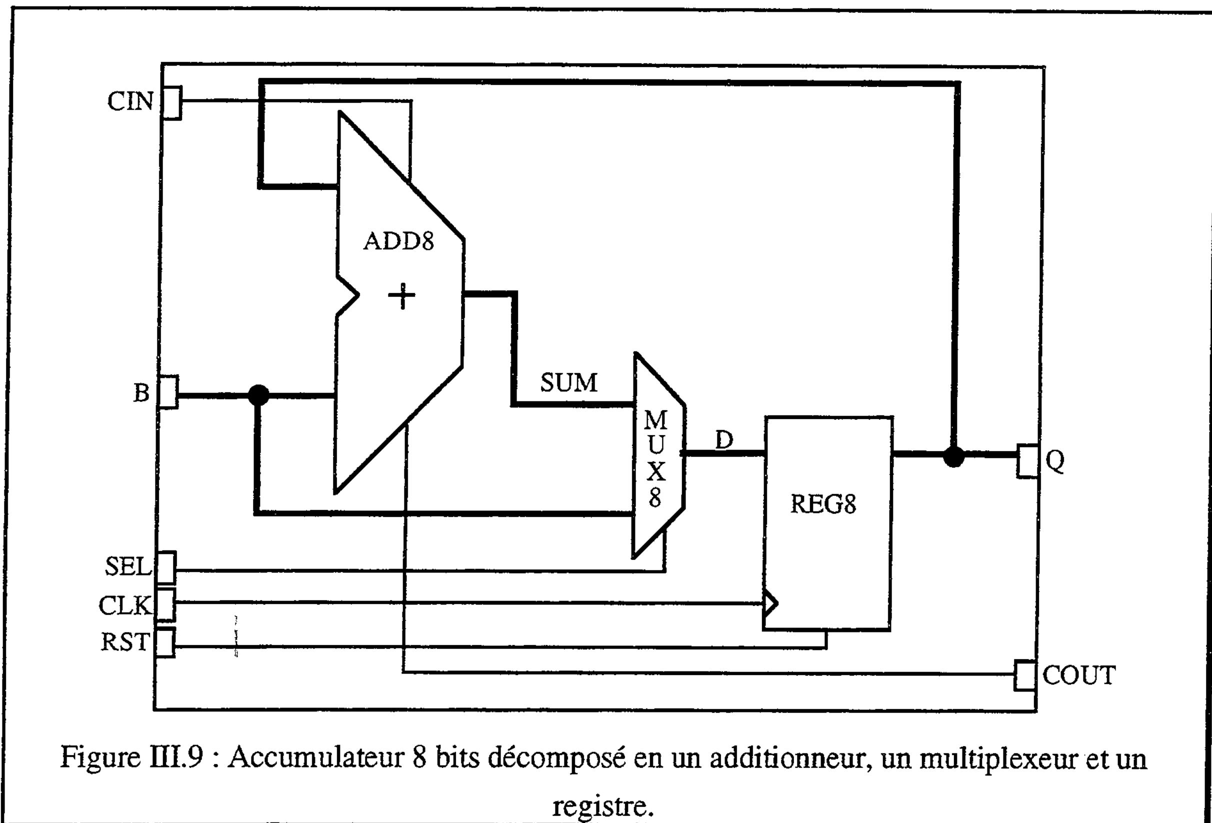

Figure III.9 : Accumulateur 8 bits décomposé en un additionneur, un multiplexeur et un registre.

```
Name "ACCUM"    TG_Accum
Port (  B         In        Width 8,
        CIN, SEL, CLK, RST   In        Width 1,
        Q         Out       Width 8,
        COUT      Out       Width 1     )
# déclaration des signaux internes
Declaration      (SUM 8, D 8)
# description structurelle hiérarchique
Construction
   (    ADD8 (Q, B, CIN, SUM, COUT),
        MUX8 (SUM, B, SEL, D),
        REG8 (D, Q, CLK, RST)
   )
```

Figure III.10 : Description structurelle d'un accumulateur 8 bits.

2.5.2.7. Exemples réels de description avec le nouveau format de bibliothèque

Deux exemples réels sont donnés dans cette partie. Le premier exemple donné à la figure III.11 est la description d'un additionneur de la bibliothèque LPM [LPM] sur laquelle nous reviendrons plus en détail par la suite. Il s'agit là d'une cellule générique pouvant être utilisée par la synthèse RTL ou la synthèse de haut niveau.

```
Library LPM
Name LPM_ADD_SUB
TGate TG_Ope
Generic
Param (
   Int "WIDTH"
   Str REPRESENTATION "UNSIGNED", "SIGNED" Default "SIGNED"
   Str LPM_POLARITY PORT_INV  )
Port (
   DATAA In  Port_Invmask LPM_POLARITY Width "WIDTH" LSB ,
   DATAB In  Port_Invmask LPM_POLARITY Width "WIDTH" LSB,
   SUM Out Port_Invmask LPM_POLARITY Width "WIDTH" LSB ,
   CIN In PT_Cin Port_Invmask LPM_POLARITY Opt,
   COUT Out PT_Cout Port_Invmask LPM_POLARITY Opt Exclude( OVERFLOW),
   OVERFLOW Out PT_Ovfl Port_Invmask LPM_POLARITY Opt Exclude (COUT),
   ADD_SUB In PT_addSub Cmd Port_Invmask LPM_POLARITY Opt)
Op ADDITION OP_ADD
   Comm ( DATAA, DATAB )
   Cond ( ADD_SUB 1 )
   Data ( DATAA, DATAB, SUM, CIN 0, COUT )
   Put (REPRESENTATION "UNSIGNED")
Op ADDITIONC2 OP_ADDC2
   Comm ( DATAA, DATAB )
   Cond ( ADD_SUB 1 )
   Data ( DATAA, DATAB, SUM, CIN 0, OVERFLOW )
   Put (REPRESENTATION "SIGNED")
Op SUBTRACTION OP_SUB
   Cond ( ADD_SUB 0 )
   Data ( DATAA, DATAB, SUM, CIN  1, COUT )
   Put (REPRESENTATION "UNSIGNED")
Op SUBTRACTIONC2 OP_SUBC2
   Cond ( ADD_SUB 0 )
   Data ( DATAA, DATAB, SUM, CIN 1, OVERFLOW )
   Put (REPRESENTATION "SIGNED")
```

Figure III.11 : Description d'un additionneur-soustracteur de la bibliothèque LPM.

Le second exemple donné à la figure III.12 est la description d'une bascule multiplexée à deux entrées. Cette bascule est un élément de la bibliothèque VSC370 de cellules standard [VLSI]. Il s'agit là d'un élément de base relativement complexe pouvant être utilisé par la synthèse de bas niveau.

```
Library VSC370
Unitwidth um
# Description d'une bascule multiplexée avec remise à zéro et à un asynchrone
Name MFBTNB
TGate TG_Dmux
Width 115.2
Touch Tue  Use True
Port (  SA      In  PT_sel             Cmd   Unknown          Width 1,
        DA,DB   In                           Unknown          Width 1,
        CP      In  PT_clk             Cmd   Unknown          Width 1,
        CDN     In  PT_rst  Port_Inv   Cmd   Unknown   As 0   Width 1,
        SDN     In  PT_set  Port_Inv   Cmd   Unknown   As 1   Width 1,
        Q       Out                          Unknown          Width 1,
        QN      Out                          Unknown          Width 1)
Eq(     Q = SA * DA + !SA * DB,
        QN = !Q     )
```

Figure III.12 : Description d'une bascule multiplexée de la bibliothèque VSC370.

2.5.3. Format du fichier décrivant les informations temporelles des éléments de bibliothèques

2.5.3.1. Informations générales décrites par ce fichier

Les informations contenues dans ce second fichier seront utilisées par l'outil d'estimation du chemin critique. Ces données viennent en complément des données décrites précédemment, contenues dans le premier fichier. Il est bien évident que les données fournies dans ces deux fichiers sont étroitement liées. La figure III.13 illustre ces liens. Dans cette figure, les rectangles blancs représentent les informations contenues dans le premier fichier que nous venons de décrire, alors que les rectangles gris représentent les informations temporelles contenues dans le second fichier que nous allons décrire.

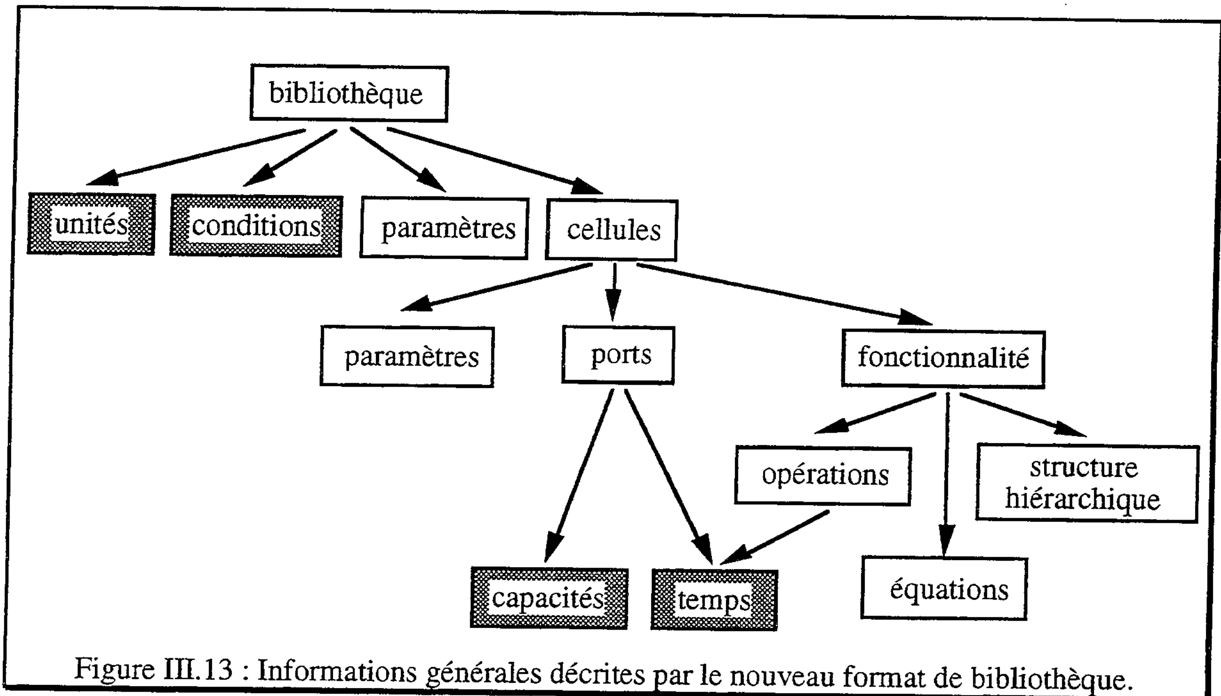

Figure III.13 : Informations générales décrites par le nouveau format de bibliothèque.

2.5.3.2. Informations générales sur la bibliothèque

Dans ce second fichier, un certain nombre d'informations générales à toute la bibliothèque sont décrites. Ces informations sont les suivantes :

- le nom de la bibliothèque. Ce nom sert de lien entre les deux fichiers permettant de décrire la bibliothèque.
- les unités de base employées pour représenter les données de température, de tension, de temps, de capacité de routage et de capacité des ports.
- les conditions opératoires qui sont les conditions dans lesquelles fonctionnera le circuit. Ces conditions sont : la température, la tension, le processus, les coefficients de dérivation, la capacité de routage. Plusieurs valeurs peuvent être données pour chacune de ces informations, définissant ainsi plusieurs conditions opératoires dont les conditions limites de fonctionnement.
- les conditions opératoires par défaut.

La figure III.14 donne un exemple de définition de ces informations pour une bibliothèque.

```
Library BASHITO

Units
   Temp    "°C"
   Volt    "V"
   Time    "ns"
   RCap    "nF/um"
   Capa    "nF"

OpCond     OC1
   Temp    25.0
   Volt    5.0
   Proc    1.0
   Kt      1.0
   Kv      1.0
   Kp      1.0
   RCap    0.6

DefOpCond      OC1
```

Figure III.14 : Exemple de description des informations générales sur une bibliothèque.

2.5.3.3. Informations temporelles sur les éléments de la bibliothèque

Un élément est décrit par :
- son nom. Ce nom qui est unique permet de faire le lien entre les informations temporelles associées à cet élément et les informations structurelles et fonctionnelles qui s'y rapportent, décrites dans le premier fichier.
- les capacités de ses ports d'entrée et de sortie,
- les temps de traversée de cet élément qui peuvent être associés à ses ports d'entrée-sortie ou aux opérations qu'il réalise.

2.5.3.4. Informations sur les capacités des ports d'entrée et de sortie

Chaque port d'entrée est défini par :
- son nom,
- sa capacité d'entrée,
- son entrance.

Chaque port de sortie est défini par :
- son nom,
- sa capacité de sortie,
- sa sortance.

2.5.3.5. Informations sur les temps de traversée

Un temps de traversée est toujours associé à un couple de ports d'entrée et de sortie. Ce temps peut être associé à un port de sortie et un port d'entrée, à un port de sortie et un ensemble de ports d'entrée ou à un ensemble de ports de sortie et un ensemble de ports d'entrée. Un port est défini par son nom et un ensemble de ports est défini par une liste de noms ou bien par le caractère spécial "*" spécifiant l'ensemble de tous les ports de sortie ou d'entrée. Pour les ports sur plusieurs bits, un index, un segment ou le port tout entier peut être spécifié.

Ces informations temporelles associées à des couples d'entrée-sortie, peuvent soit être directement reliées à un élément, soit être reliées aux opérations réalisables par cet élément.

Pour un couple de ports d'entrée et de sortie, les informations de temps pouvant être spécifiées sont les suivantes :
- le temps de montée,
- le temps de montée intrinsèque,
- le temps de descente,
- le temps de descente intrinsèque,
- le temps de précharge (pour les éléments séquentiels),
- le temps de maintien (pour les éléments séquentiels).

Dans le cas d'éléments pouvant réaliser plusieurs opérations, toutes ces informations peuvent être répétées pour chaque opération réalisable par l'élément. Ainsi, les informations temporelles ne sont plus directement associées à l'élément de la bibliothèque, mais aux opérations réalisables par cet élément.

La figure III.15 donne un exemple de description des informations temporelles associées à un élément simple : un inverseur.

```
# Informations temporelles associées à un inverseur
Name INV
   # Informations sur la capacité du port de sortie Z
   POutC  Z   10  *
   # Informations sur la capacité du port d'entrée I
   PInC   I   *   14
   # Informations sur le temps de traversée de cet inverseur
   PTim   Z  I  6  6  6  6
# le caractère "*" est utilisé ici pour indiquer une valeur inconnue
```

Figure III.15 : Exemple de description des informations temporelles associées à un élément simple.

La figure III.16 donne un exemple de description des informations temporelles associées à un élément complexe : un décaleur. Ce décaleur permet de réaliser deux opérations : un décalage à droite ou bien un décalage à gauche. Les temps de traversée de cet élément sont donnés pour ces deux opérations.

```
# Informations temporelles associées à un décaleur
Name CLSHIFT
   # Informations sur la capacité des ports de sortie
   POutC  UNDERFLOW, OVERFLOW, RESULT   8   8
   # Informations sur la capacité des ports d'entrée
   PInC   DATA, DISTANCE[0]   9   9
   # Informations sur les temps de traversée associés à l'opération SHLEFT
   PTim   Op SHLEFT      RESULT   *   6   6   6   6
   # Informations sur les temps de traversée associés à l'opération SHRIGTH
   PTim   Op SHRIGTH              *   *   *   *   *   *
```

Figure III.16 : Exemple de description des informations temporelles associées à un élément complexe.

Il est à noter que toutes les valeurs pouvant être données dans ce format de bibliothèque peuvent être des entiers, des réels ou des expressions pouvant éventuellement faire intervenir des paramètres.

La grammaire complète des deux fichiers de ce nouveau format de bibliothèque est donnée en annexe 3.

2.6. Conclusion

Le nouveau format de bibliothèque qui vient d'être exposé n'a pas encore été implanté dans le système de synthèse ASYL+ suite à des contraintes de temps. En effet, l'implantation de ce nouveau format implique la mise en place de nouveaux gestionnaires de bibliothèque tant pour la synthèse de bas niveau que pour la synthèse de haut niveau. Afin d'utiliser efficacement ce nouveau format, les procédures définies pour estimer le chemin critique du circuit synthétisé doivent également être revues. Enfin, l'outil de migration doit entièrement être repris et complété.

De plus, l'utilisation de ce nouveau format implique la réécriture de toutes les bibliothèques déjà utilisées par ASYL+ dans ce nouveau format. Cependant, afin de faciliter cette tâche, un utilitaire a été développé permettant d'écrire automatiquement dans le nouveau format les bibliothèques décrites dans les anciens formats de bas niveau et de haut niveau. Ceci ce fait par le passage des informations des anciennes structures intermédiaires de bibliothèque vers les

nouvelles. Il est bien évident que plus d'informations peuvent être décrites par ce nouveau format, mais ces informations n'auront qu'à être rajoutées.

3. Les nouveaux standards : VITAL et LPM

3.1. Problèmes liés à la description et à l'utilisation des bibliothèques

3.1.1. Problèmes liés à la description des bibliothèques

Le problème de la définition d'un format de bibliothèque est un problème général qui se pose à toutes les personnes développant un outil de CAO. Actuellement, chaque outil possède son propre format de description de bibliothèque. Ces formats sont protégés et appartiennent aux sociétés développant ces outils. Les descriptions des bibliothèques ne sont pas accessibles et sont généralement codées à l'intérieur de chaque outil. Pour un concepteur utilisant plusieurs outils de CAO (simulation, synthèse, placement et routage, etc.) pour produire son circuit, cela implique qu'il doit avoir à sa disposition plusieurs descriptions de la même bibliothèque technologique utilisée, chaque description correspondant à chaque outil utilisé. De plus, il n'existe pas d'outil permettant de traduire une bibliothèque décrite dans un format donné vers un autre format puisque ces formats sont confidentiels. Tout ceci pose donc des problèmes au concepteur lors du passage d'un outil à un autre.

Le problème qui se pose aux sociétés développant ces outils de CAO est non seulement celui de la définition d'un format de description de bibliothèques, mais aussi et surtout celui de la validité de ces descriptions. En effet pour être valables, ces descriptions doivent être certifiées conformes par les fondeurs créant ces bibliothèques. Posséder des descriptions de bibliothèques certifiées est une garantie supplémentaires de la qualité de l'outil de CAO utilisé et aujourd'hui beaucoup de sociétés ont des difficultés pour faire certifier leurs descriptions de bibliothèques.

Pour pallier à ces deux graves problèmes, un format standard de description de bibliothèque est en train d'émerger. Ce format standard s'appelle VITAL pour "VHDL Initiative Towards ASIC Libraries". De nombreuses sociétés s'intéressent à ce nouveau format. Nous y reviendrons plus en détails dans ce qui suit.

3.1.2. Problèmes liés à l'utilisation des bibliothèques

Pour un outil de synthèse, il est important de savoir utiliser efficacement le plus grand nombre de cibles technologiques existantes sur le marché. Or, toutes ces cibles évoluent et les bibliothèques comportent de plus en plus d'éléments complexes ou spécifiques à la technologie. Ainsi, il devient de plus en plus difficile pour les outils d'utiliser tous ces éléments nouveaux et complexes. C'est une des raisons pour laquelle un nouveau standard : LPM, pour "Library of Parameterized Modules", a vu le jour. LPM est en fait une bibliothèque virtuelle d'éléments paramétrés mise au point afin de faciliter et d'améliorer la connexion entre les outils de synthèse et les outils spécifiques à chaque cible technologique. Plus d'informations sur ce nouveau standard vont être apportées à la fin de ce chapitre.

3.2. VITAL

3.2.1. Motivations

Si nous nous référons à la figure III.17 tirée de [Patt 95], nous nous apercevons que l'utilisation de VHDL croît de façon linéaire au cours des années depuis 1987 en Europe et depuis 1991 aux Etats-Unis d'Amérique, alors que l'utilisation de Verilog atteint quasiment aujourd'hui un palier dans ces deux zones géographiques. En Europe, contrairement aux Etats-Unis, l'utilisation de VHDL a toujours prédominé sur celle de Verilog. Cependant, même aux Etats Unis la tendance semble s'inverser et selon les prédictions, d'ici 1997, les utilisateurs de VHDL seront aussi nombreux que ceux de Verilog.

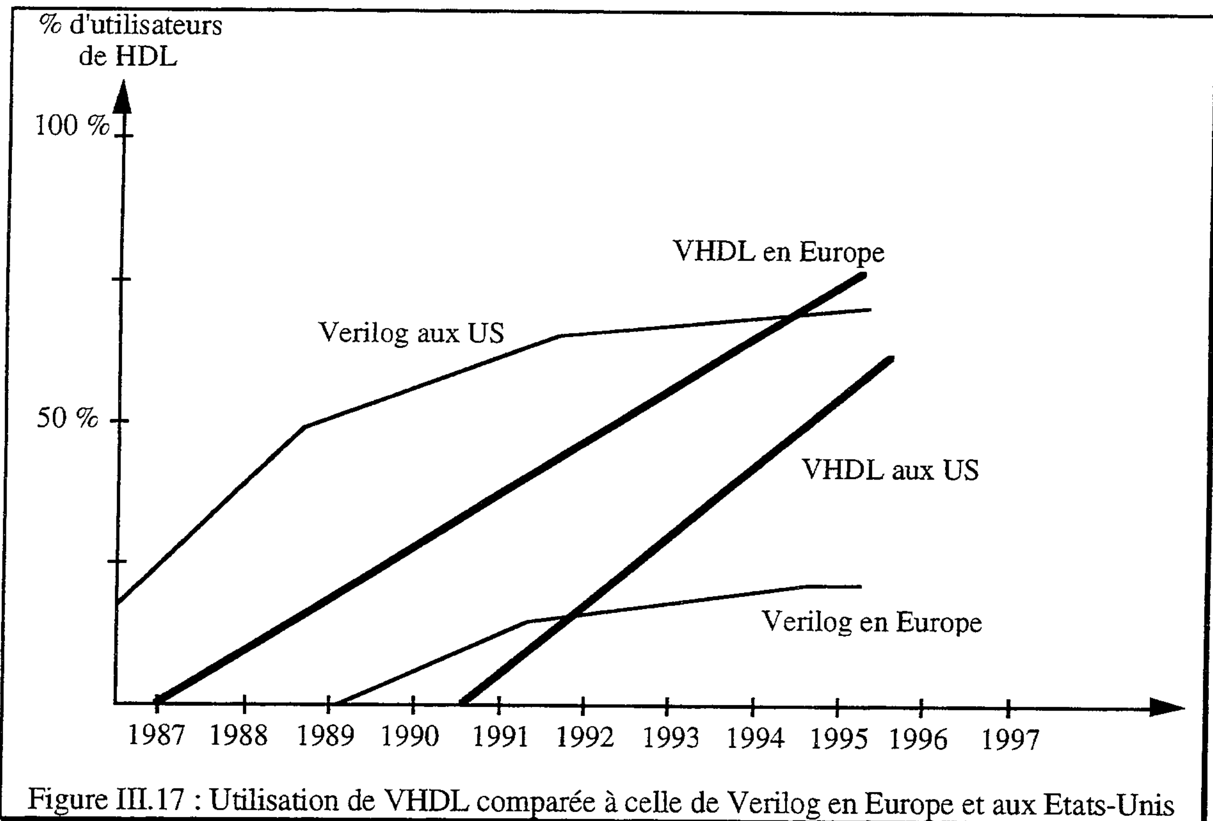

Figure III.17 : Utilisation de VHDL comparée à celle de Verilog en Europe et aux Etats-Unis

Néanmoins, le combat entre ces deux langages reste toujours aussi difficile et pour que ces prédictions soient réalisées et que VHDL soit plus largement adopté, il faut mettre à la disposition des utilisateurs de VHDL des bibliothèques décrites en VHDL comme cela est déjà fait pour Verilog depuis plusieurs années. Ceci explique donc la naissance de VITAL.

Le but essentiel de VITAL est d'accélérer le développement de bibliothèques d'ASICs décrites en VHDL pour la simulation [VITAL]. Les modèles comportent des informations temporelles précises permettant d'obtenir des résultats de simulation très proches de la réalité.

La figure III.18 tirée de [Berm 94] donne l'estimation de l'évolution du nombre de bibliothèques ASIC certifiées décrites en VITAL par rapport à Verilog.

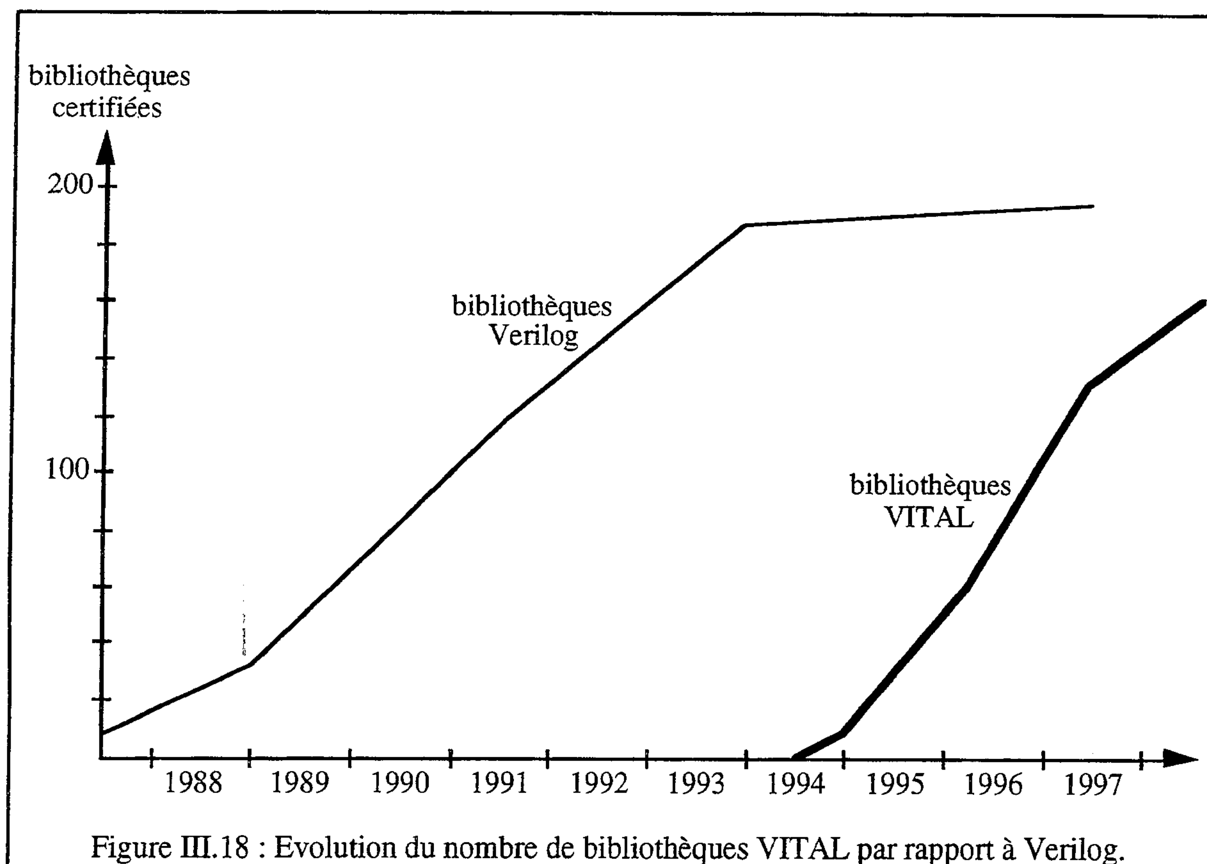

Figure III.18 : Evolution du nombre de bibliothèques VITAL par rapport à Verilog.

3.2.2. Objectifs

VITAL ("VHDL Initiative Towards ASIC Libraries") est le fruit de la coopération d'une quarantaine de compagnies internationales de fondeurs et de développeurs d'outils d'aide à la conception automatisée. Parmi ces compagnies nous retrouvons les plus grandes telles que Synopsys, Compass, Cadence, Hewlett-Packard, Mentor Graphics, etc. Cette collaboration a abouti à un effort de standardisation de modèles VHDL pour les bibliothèques d'ASICs.

L'initiative de VITAL a vu le jour en 1992 lors de la conférence "VHDL Users Forum" et a abouti à une première spécification en Février 1993 [VITAL 93]. D'autres versions ont suivies et actuellement la plus récente est la version 3.0 de Juillet 1995 [VITAL 95] qui vient d'être proposée au comité de standardisation IEEE. A ce jour, le vote en vue d'obtenir le titre de standard IEEE 1076.4 vient d'avoir lieu et le résultat du vote a été positif.

Les constatations qui ont débouché sur la mise en place de VITAL sont essentiellement les suivantes :

- le manque de bibliothèques VHDL ASIC certifiées et portables,
- le manque d'un flot de conception entièrement basé sur VHDL,
- la lenteur des simulateurs VHDL part rapport aux simulateurs logiques.

VITAL a pour objectifs :
- de permettre le développement accru de bibliothèques certifiées et portables en VHDL,
- de fournir une solution permettant de transcrire des données temporelles en VHDL afin d'établir des modèles précis dans le domaine temporel,
- de fournir une méthode de modélisation concrète pour une simulation efficace,
- de fournir un mécanisme de rétro-annotation uniforme.

VITAL définit deux niveaux de conformité :
- le niveau 0 de conformité est atteint lorsque le modèle utilise les conventions de nommage et de type pour les informations temporelles définies par VITAL,
- le niveau 1 de conformité est atteint lorsque le modèle est conforme au niveau 0, et lorsque le modèle suit les règles de description spécifiées par VITAL en utilisant que les fonctions définies par VITAL pour décrire la fonctionnalité, la propagation des délais et pour détecter les instabilités et les violations temporelles à l'intérieur du modèle.

Nous reviendrons plus en détails sur ces deux niveaux de conformités, mais avant tout, nous allons décrire les informations temporelles prises en compte au niveau d'un modèle VITAL.

3.2.3. Informations temporelles prise en compte par VITAL

Les informations temporelles considérées sont les suivantes :
- le temps de propagation : temps nécessaire à un changement d'état survenu sur un signal d'entrée pour apparaître sur le signal de sortie correspondant,
- le temps d'arrivée d'un signal synchrone ("setup time") : temps antérieur à un front actif du signal d'horloge durant lequel la valeur spécifiée du signal d'entrée synchrone ne doit pas changer,
- le temps de maintien d'un signal synchrone ("hold time") : temps postérieur à un front actif du signal d'horloge durant lequel la valeur spécifiée du signal d'entrée synchrone ne doit pas changer,
- le temps d'arrivée d'un signal asynchrone ("release time") : temps antérieur à un front actif du signal d'horloge pendant lequel la valeur spécifiée sur un signal asynchrone ne doit pas changer,
- le temps de maintien d'un signal asynchrone ("removal time") : temps postérieur à un front actif du signal d'horloge pendant lequel la valeur spécifiée sur un signal asynchrone ne doit pas changer,
- la période : temps écoulé entre deux fronts actifs successifs du signal d'horloge.

La figure III.19 illustre ces informations temporelles.

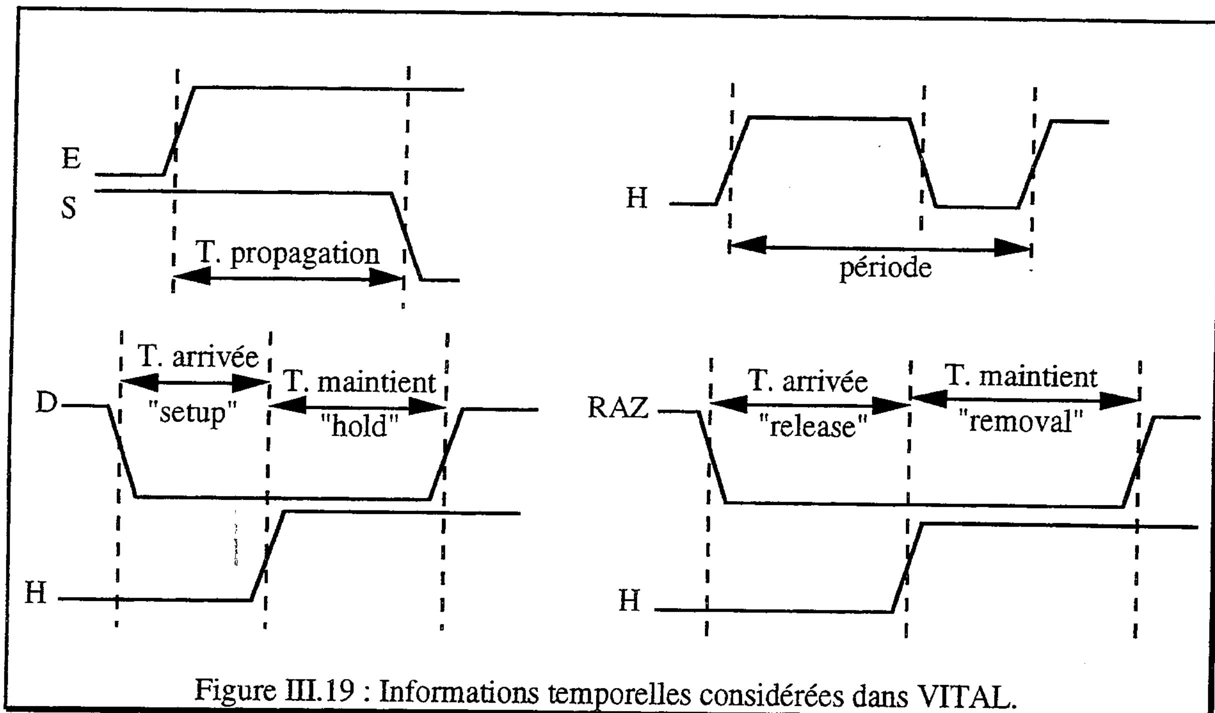

Figure III.19 : Informations temporelles considérées dans VITAL.

3.2.4. Le niveau 0 de conformité

Un modèle VITAL est en fait un modèle VHDL respectant un certain nombre de règles définies par la norme VITAL. VITAL ajoute donc au standard VHDL un ensemble de règles permettant de décrire des éléments de bibliothèques en VHDL, qui pourront ensuite être utilisés efficacement en simulation et même en synthèse. Ces règles se divisent en deux sous-ensembles. Si le premier sous-ensemble de règles est respecté par le modèle VHDL, alors le modèle est dit compatible à VITAL au niveau 0. Ce premier sous-ensemble de règles est présenté dans cette section. Lorsque le modèle VHDL respecte également les règles énoncées dans le second sous-ensemble, alors le modèle est dit compatible à VITAL au niveau 1. Ce second sous-ensemble de règles est présenté dans la section suivante.

Un modèle VITAL est tout d'abord défini par son interface. L'interface permet d'identifier les ports et les paramètres génériques. Les informations temporelles associées à un modèle sont uniquement définies par des paramètres génériques. VITAL propose une standardisation des noms de ces paramètres de sorte à ce qu'une rétro-annotation via SDF puisse être accomplie. Les informations temporelles dépendantes de l'environnement doivent être calculées à l'extérieur du modèle VITAL ou bien importées directement via un fichier SDF, et ensuite elles doivent être fournies au modèle VITAL seulement en tant que valeurs effectives des paramètres génériques du modèle. Ainsi, le modèle VITAL reste indépendant de la méthodologie employée

pour calculer ces valeurs temporelles, ce qui simplifie le modèle et diminue son temps d'exécution et son occupation mémoire.

Les ports et les signaux définis dans un modèle VITAL utilisent les types et sous-types définis dans le paquetage standard *"STD_LOGIC_1164"*. Des types et sous-types spécifiques aux paramètres génériques modélisant les informations temporelles ont été définis dans le paquetage *"VITAL_Timing"*.

Le nommage des paramètres génériques représentant les informations temporelles est normalisé. Le nom du paramètre commence obligatoirement par un préfixe indiquant la nature de l'information, suivi éventuellement par d'autres informations telles par exemple qu'un nom de port pour préciser, le cas échéant, le port auquel est rattachée l'information. Pour ne citer que deux exemples, "tpd" et "tipd" sont les préfixes utilisés pour représenter respectivement le temps de propagation intrinsèque et le temps de propagation lié aux interconnexions.

Le squelette d'un modèle général est donné à la figure III.20. Ce modèle est compatible au niveau 0. Seule l'entité est décrite. Les lignes en italique sont les lignes spécifiques à un modèle VITAL.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
library VITAL;
use VITAL.VITAL_Timing.all;
use VITAL.VITAL_Primitives.all;

entity Squelette is
    generic (
            tipd_a1 :     DelayType01 := (0.00 ns, 0.00 ns);
            -- tipd_* : délai d'interconnexion lié à l'entrée *
            ...
            tpd_a1_s1 :  DelayType01 := (0.48 ns, 0.55 ns);
            -- tpd_*_# : temps de propagation entre l'entrée * et la sortie #
            ...

            TimingCheckOn : Boolean := True;
            -- Permet d'activer ou d'inhiber les vérifications temporelles
            ... );
    port (
            a1 :     in   Std_Logic    := 'U';
            ...
            s1 :     out  Std_Logic;
            ... );
end Squelette;
```

Figure III.20 : Squelette d'une entité VITAL compatible au niveau 0

3.2.5. Le niveau 1 de conformité

Les modèles compatibles au niveau 0 à VITAL, c'est-à-dire respectant les règles qui viennent d'être décrites et s'appuyant sur le squelette présenté à la figure III.20, peuvent être utilisés pour la rétro-annotation via SDF. Cependant, étant donné qu'aucune règle précise n'a été donnée pour la description de l'architecture, les modèles compatibles uniquement au niveau 0 sont très longs à simuler. Ceci est d'autant plus pénalisant lors de la simulation VHDL de grands circuits ASICs comprenant plusieurs milliers de portes. Par exemple, la simulation d'un simple vecteur sur un ASIC de 200 000 portes prend 50 minutes avec un traditionnel simulateur au niveau portes, alors que 15 heures sont nécessaires avec un simulateur VHDL typique [Crow 94]. Néanmoins, ces résultats peuvent être améliorés en utilisant un simulateur VHDL accéléré. Pour le cas précédent, le temps de simulation est ramené à seulement 30 minutes pour un simulateur VHDL accéléré [Crow 94].

C'est de cette constatation qu'est apparue la nécessité de définir un niveau de compatibilité supérieur au niveau 0 : le niveau 1. Un modèle compatible à VITAL au niveau 1 doit tout d'abord être compatible au niveau 0 et doit également respecter un certain nombre de règles supplémentaires spécifiques au niveau 1. Ces règles régissent la description de l'architecture du modèle afin de la rendre suffisamment rigide pour permettre une accélération de la simulation VHDL, mais également suffisamment générale pour couvrir un grand nombre d'éléments de bibliothèques.

Pour ce faire, des outils de base ont été définis. Il s'agit des fonctions et des procédures définies dans les paquetages VHDL "VITAL_Timing" et "VITAL_Primitives", permettant de prendre en compte et de propager les informations temporelles associées à chaque modèle, ainsi que de décrire la fonctionnalité. Quelques unes de ces fonctions et procédures sont présentes dans le squelette donné à la figure III.22. La liste complète figure dans [VITAL]. Ce sont ces fonctions et ces procédures qui sont accélérées au cours d'une simulation VHDL suite, par exemple, à leur implantation directe dans le coeur du simulateur, comme il a été fait pour le simulateur Vulcan de la société VEDA [Crow 94]. D'autres sociétés telles que Synopsys et Mentor Graphics proposent également d'accélérer les paquetages VITAL dans leur simulateur VHDL [Levi 94], [Casl 94].

La structure générale d'un modèle VHDL compatible au niveau 1 à VITAL est présentée à la figure III.21. L'architecture d'un tel modèle se décompose en quatre parties.

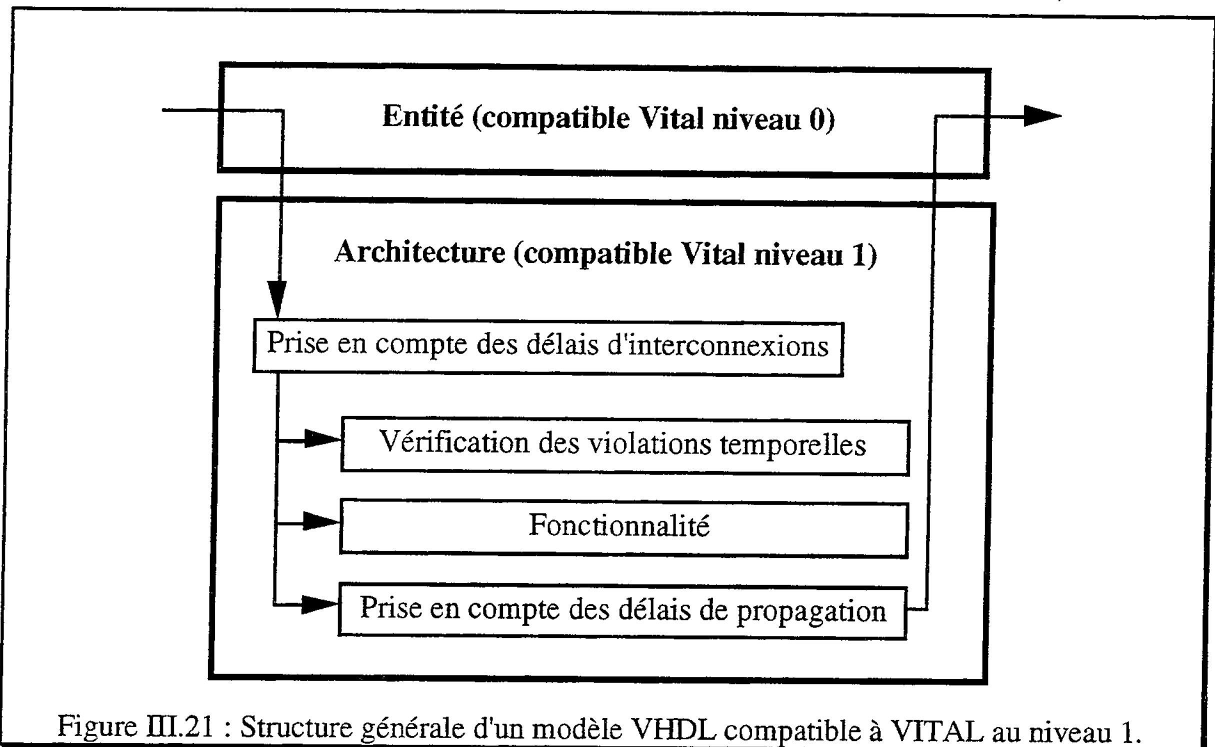

Figure III.21 : Structure générale d'un modèle VHDL compatible à VITAL au niveau 1.

La première partie consiste à propager les délais d'interconnexion spécifiés en tant que paramètres génériques au niveau de l'entité. Pour ce faire, pour chaque port d'entrée est déclaré un signal interne correspondant. Seuls ces signaux seront utilisés ailleurs dans l'architecture. Ensuite, à chacun de ces signaux est affecté le port d'entrée respectif retardé du délai d'interconnexion correspondant. Ceci se fait par l'intermédiaire de la procédure "VitalPropagateWireDelay" définie dans le paquetage "VITAL_Timing".

La seconde partie consiste à vérifier les contraintes temporelles imposées notamment au niveau des éléments séquentiels par les temps d'arrivée ou les temps de maintien. Ces temps sont fournis par les paramètres génériques décrits au niveau de l'entité.

La troisième partie consiste à décrire la fonctionnalité de l'élément sans se préoccuper des informations temporelles. Plusieurs possibilités sont offertes : l'utilisation de fonctions ou de procédures définies dans le paquetage "VITAL_Primitives" et l'utilisation de tables d'états notamment pour représenter des éléments séquentiels.

La quatrième et dernière partie consiste à prendre en compte les délais de propagation de l'élément pour chaque sortie par rapport à chaque entrée. Ces délais proviennent des paramètres génériques fournis au niveau de l'entité. Ceci se fait par l'intermédiaire de la procédure "VitalPropagatePathDelay", affectant le port de sortie par la variable interne correspondante employée dans la partie précédente, accompagné du délai de propagation approprié. Ce délai est

calculé en fonction des délais de toutes les entrées par rapport à cette sortie et en fonction des instants des derniers changements sur toutes les entrées.

Ces quatre parties sont présentées à la figure III.22 donnant le squelette VHDL d'une architecture VITAL compatible au niveau 1.

```
architecture Arch of Squelette is
   attribute VITAL_Level_1 of Arch : architecture is True;
   signal a1_ipd :    Std_Logic    := 'X';
   -- *_ipd : signaux d'entrée prenant en compte les délais d'interconnexion
begin
   -- Bloc modélisant les délais d'interconnexion
   Wire_Delay : block
   begin
        VitalPropagateWireDelay (...);
        -- est fonction d'une entrée (a1), du signal interne correspondant retardé (a1_ipd),
        -- et du délai d'interconnexion associé à cette entrée (tipd_a1);
   end block;
   -- Partie décrivant le comportement
   VITALBehavior : process (a1_ipd, ...)
        variable s_zd :    Std_Logic    := 'X';
        -- *_zd : variables de sortie ne prenant pas en compte les délais de propagation
   begin
        -- Partie vérifiant les contraintes temporelles
        TimingCheck : if (TimingChecksOn) then
            ....
        end if;
        -- Partie décrivant la fonctionnalité
            s_zd := VitalAnd (...);
            -- est fonction des signaux d'entrée internes retardés (*_ipd).
        -- Prise en compte du temps de propagation
            VitalPropagatePathDelay (...);
            -- est fonction d'une sortie (s1), de la sortie interne correspondante non
            -- retardée (s1_zd), des temps de propagation intrinsèques liés à cette sortie
            -- par rapport à chaque entrée (tpd_*_s1) et des instants de dernier
            -- changement de chaque entrée (*_ipd'last_event).
   end process;
end Arch;
```

Figure III.22 : Squelette d'une architecture VITAL compatible au niveau 1

3.2.6. Flot de conception VHDL utilisant VITAL

Le flot de conception VHDL utilisant VITAL est décrit à la figure III.23. Il a été présenté dans [Bouc 94].

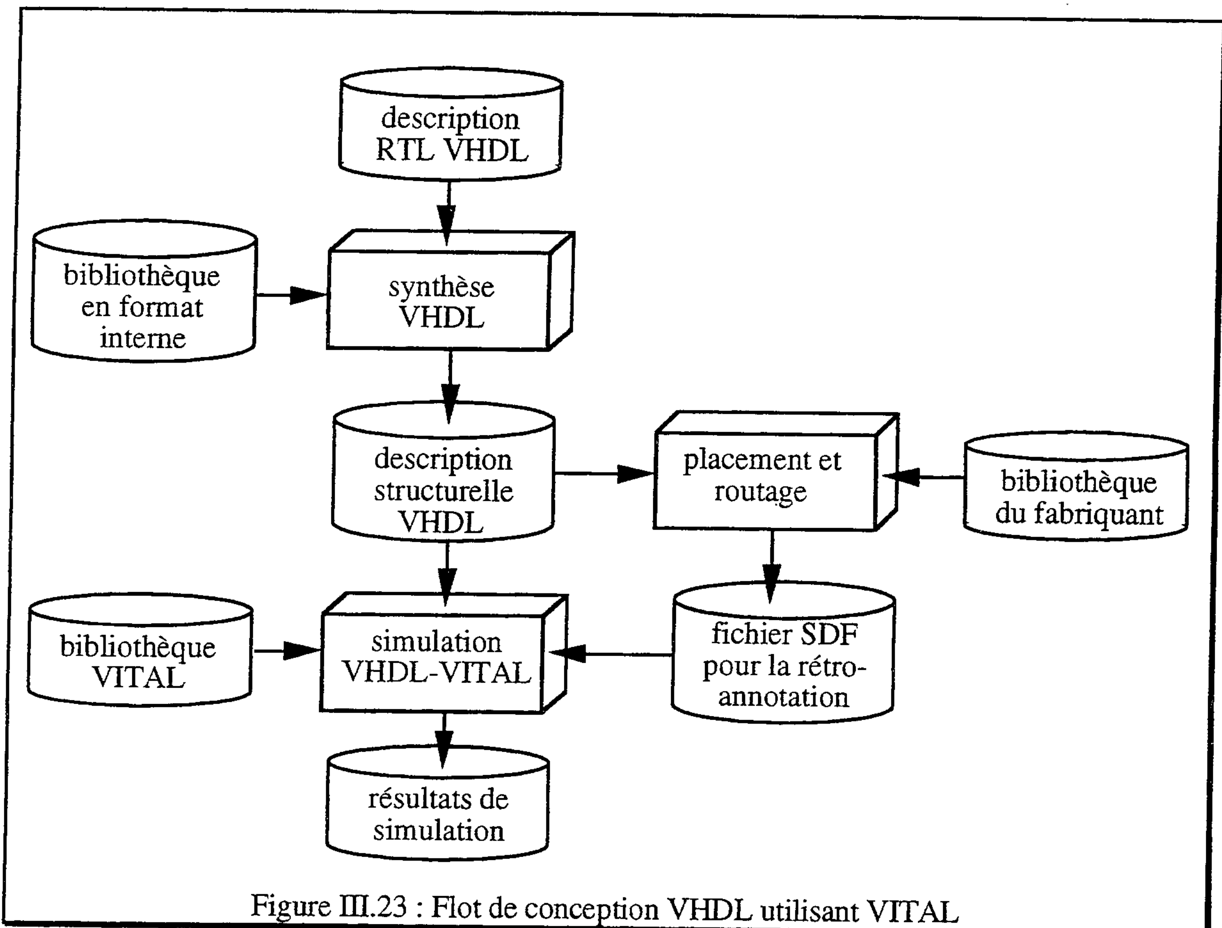

Figure III.23 : Flot de conception VHDL utilisant VITAL

Ce flot de conception utilise une description comportementale de type RTL décrite en VHDL. Cette description est alors synthétisée par un outil de synthèse automatique de circuit tel que ASYL+, Synopsys, etc. utilisant pour ce faire la description de la bibliothèque technologique ciblée dans leur propre format interne. Par exemple, pour ASYL+, il s'agit du format qui vient d'être présenté en début de ce chapitre. L'outil de synthèse génère alors en sortie une description structurelle du circuit, exprimée sous la forme d'un ensemble de portes de base et de macro blocs interconnectés. Cette description peut alors être utilisée par un simulateur VHDL logique utilisant la même bibliothèque technologique, mais décrite cette fois dans le format VITAL. Les résultats de simulation obtenus permettent alors de vérifier la fonctionnalité du circuit synthétisé mais ne tiennent pas compte des délais liés aux interconnexions des éléments du circuit. Pour ce faire, la description structurelle obtenue après synthèse doit être placée et routée. L'outil de placement et de routage utilise alors une troisième description de la même bibliothèque technologique provenant généralement directement du fabriquant. Après le placement et le routage, les valeurs des capacités et des résistances liées aux interconnexions peuvent alors être extraites. Ces valeurs sont utilisées par un estimateur calculant, selon un modèle approprié aux dimensions technologiques, les délais de propagation des interconnexions entre les éléments. Ces délais sont alors répertoriés dans un fichier au format

SDF [SDF] et peuvent alors être utilisés par la rétro-annotation. Une seconde simulation VHDL utilisant les informations décrites dans le fichier SDF et la bibliothèque au format VITAL permet alors de vérifier non seulement la fonctionnalité mais également le comportement du circuit en fonction du temps.

Par rapport à un flot normal, ce flot de conception à l'avantage d'être plus universel car les fichiers intermédiaires entre les divers outils utilisent des formats standards, tels que VHDL pour la description structurelle obtenue après synthèse, SDF pour les informations temporelles liées aux interconnexions après placement et routage, et VITAL pour la description de la bibliothèque utilisée par le simulateur VHDL. Ainsi, les descriptions sont portables d'un outil à un autre et par exemple, pour la simulation VHDL, seule une description VITAL de la bibliothèque est nécessaire, quelque soit le simulateur VHDL utilisé. De plus, les performances de chaque outil peuvent être comparées, ce qui engendre une certaine compétition entre les outils

3.2.7. Mécanisme de rétro-annotation via SDF

Le mécanisme de translation des données temporelles contenues dans un fichier SDF vers les paramètres génériques d'un modèle VITAL est également spécifié dans le document définissant le format VITAL. Cette spécification s'appuie sur la version 2.1 du format SDF [SDF]. Les simulateurs VHDL compatibles à VITAL doivent suivre les règles définissant cette translation, de sorte à rétro-annoter automatiquement le circuit VHDL avec les valeurs des informations temporelles contenues dans le fichier SDF.

Deux méthodes peuvent être utilisées pour annoter la description structurelle d'un circuit VHDL utilisant des modèles VITAL. La première consiste à utiliser la configuration VHDL. Tous les éléments du circuit peuvent alors être configurés en utilisant les valeurs temporelles propres à chacun d'eux. Cependant cette méthode requiert un temps de compilation assez long pour des circuits comprenant plusieurs centaines de milliers de lignes de code VHDL. La seconde méthode qui paraît plus efficace pour de gros circuits, consiste à importer directement les données temporelles provenant du fichier SDF à l'intérieur du simulateur. C'est cette méthode qui a été retenue pour les simulateurs Vulcan de VEDA [Crow 94] et QuickVHDL de Mentor Graphics [Casl 94]. La structure d'un tel simulateur dans un environnement VITAL est présentée à la figure III.24.

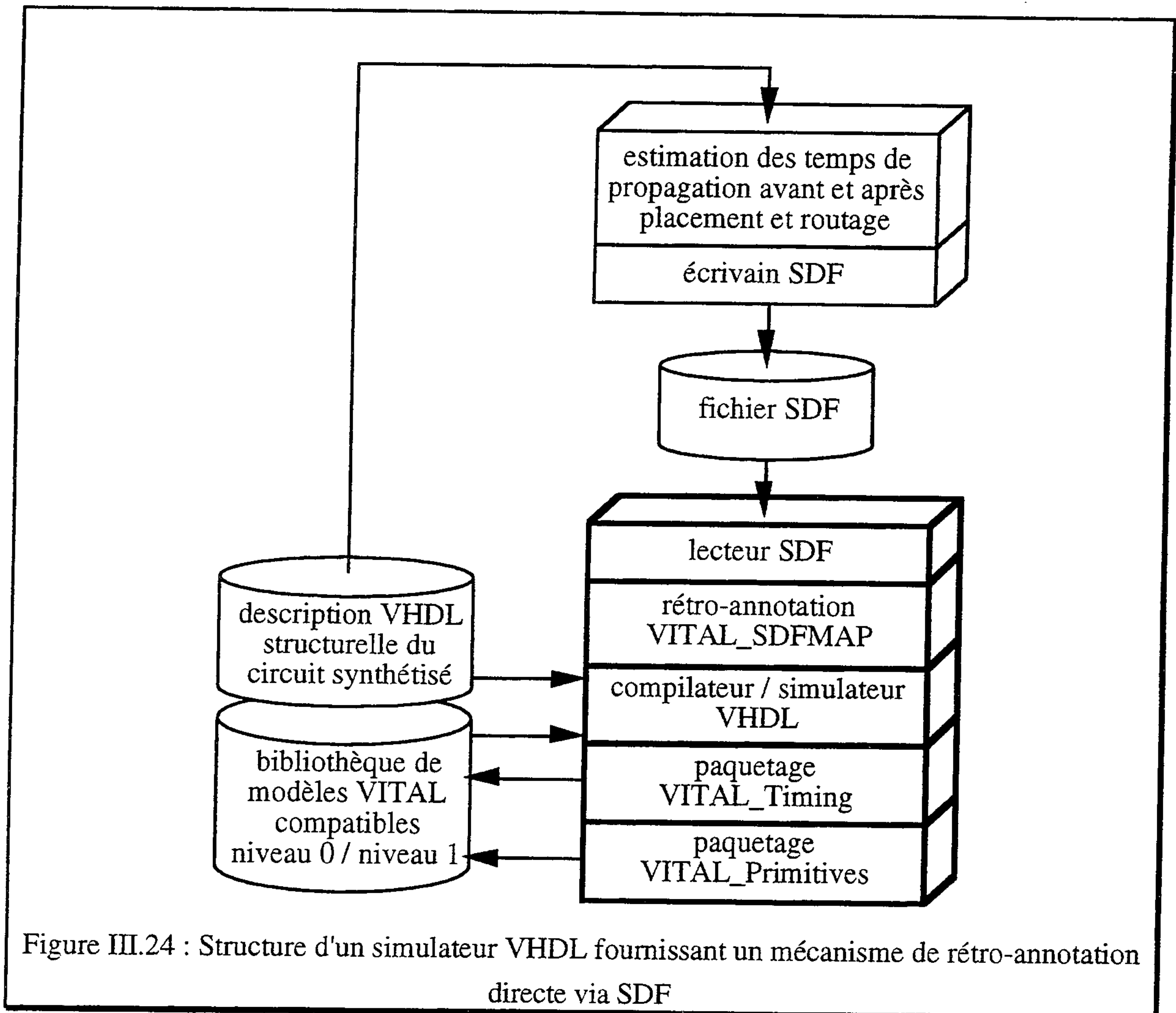

Figure III.24 : Structure d'un simulateur VHDL fournissant un mécanisme de rétro-annotation directe via SDF

3.2.8. VITAL et SDF : liens avec la synthèse

Pour être compatible avec VITAL, il est évident qu'un outil de synthèse doit savoir générer la description structurelle d'un circuit dans le langage VHDL. Les éléments interconnectés au niveau de cette description correspondent aux composants VHDL utilisant les modèles VITAL de la bibliothèque technologique considérée.

Afin de permettre une rétro-annotation, la norme VITAL s'est appuyée sur la norme SDF. La spécification de SDF [SDF] (pour Standard Delay Format) a été développé par OVI afin de permettre un transfert efficace et non ambigu des données temporelles entre les outils d'aide à la conception automatique de circuits nécessitant ces informations.

Concernant plus particulièrement l'utilisation de fichiers SDF lors de la conception automatique d'un circuit, plusieurs points peuvent être considérés. Rappelons tout d'abord qu'un fichier SDF peut contenir des informations de diverse nature. Il peut s'agir tout d'abord des conditions

de fonctionnement du circuit telles que la température, la tension d'alimentation, le processus technologique employé, etc. Ensuite, au niveau des informations temporelles, deux catégories sont à distinguer : les contraintes temporelles spécifiées par l'utilisateur et les données temporelles dont les valeurs peuvent être calculées avant ou après le placement et le routage du circuit. Ces données temporelles permettent de décrire les délais de propagation à l'intérieur d'un élément ainsi que les délais de traversée des interconnexions. Les informations contenues dans un fichier SDF pouvant être de plusieurs natures, plusieurs fichiers SDF peuvent donc être utilisés au cours des diverses étapes de la conception d'un circuit.

Avant la synthèse, un premier fichier SDF fourni par l'utilisateur peut être utilisé pour spécifier les contraintes temporelles que doit respecter le circuit. Dans ce cas, l'outil de synthèse doit être capable de lire les informations indiquées dans ce fichier et doit en tenir compte au cours de la synthèse.

Après la synthèse, si l'outil est capable d'estimer les temps de traversée des éléments du circuit en fonction du nombre des interconnexions de chaque noeuds (une même sortie pouvant piloter plusieurs entrées), dans ce cas un fichier SDF peut être généré. Ce fichier contient alors les temps de propagation des éléments utilisés dans la description structurelle associée générée après synthèse. Comme nous venons de voir, ce fichier SDF peut servir de base à une rétro-annotation au cours d'une simulation VHDL de la description structurelle.

Afin, comme nous l'avons vu, un troisième fichier SDF peut être généré par les outils de placement et de routage. Dans ce cas, le fichier SDF contient les valeurs des temps de propagation des éléments et des interconnexions entre ces éléments, calculées en fonction des caractéristiques physiques extraites de la description topologique du circuit.

Les trois utilisations d'un fichier SDF qui viennent d'être présentées sont illustrées à la figure III.25.

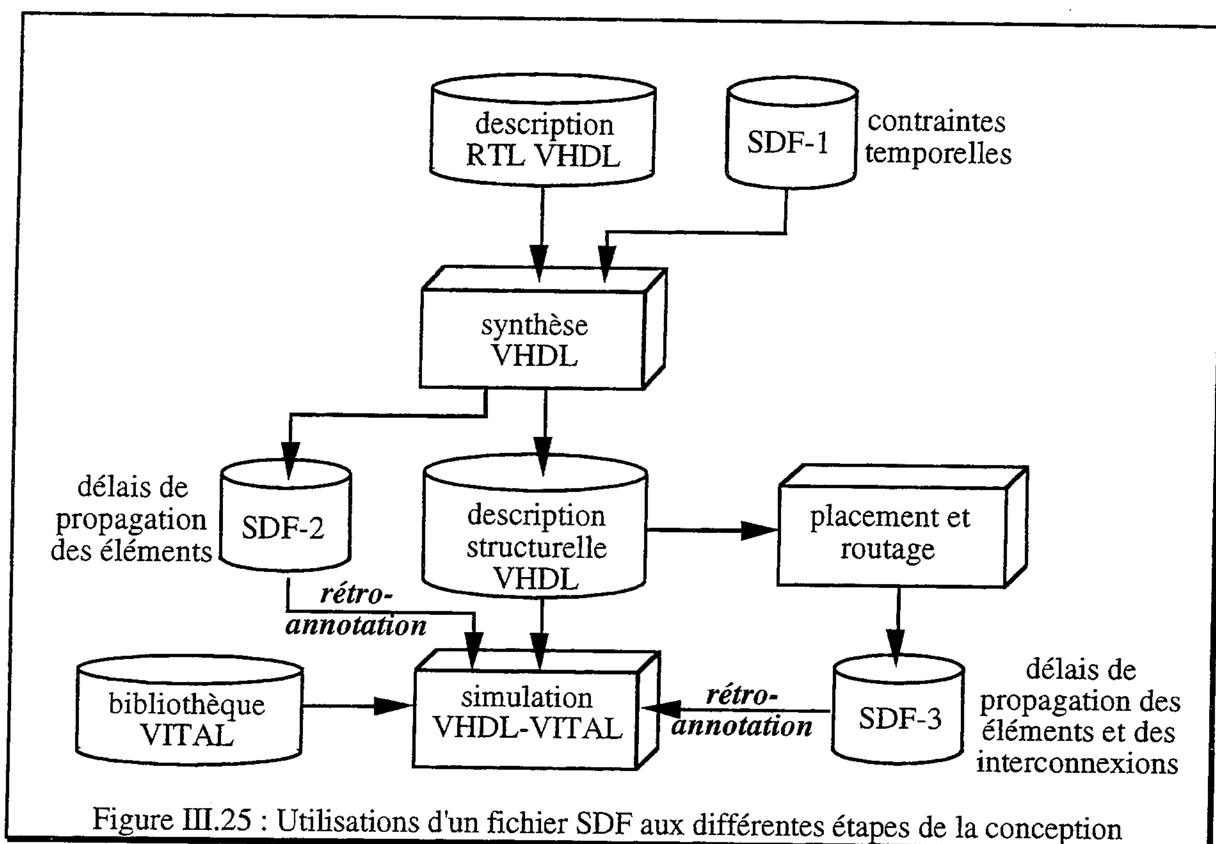

Figure III.25 : Utilisations d'un fichier SDF aux différentes étapes de la conception

Maintenant nous allons nous intéresser au cas où le circuit obtenu à la fin du cycle de conception ne vérifie pas les contraintes temporelles de départ. Ceci peut être dû, par exemple, aux mauvaises estimations des valeurs des délais de propagation, notamment lors de l'utilisation des technologies en dessous du micron. Rappelons que ces contraintes temporelles peuvent être vérifiées par une simple simulation VHDL utilisant la bibliothèque technologique au format VITAL et la description VHDL structurelle générée après synthèse, rétro-annotée par les informations temporelles provenant du fichier SDF fourni après le placement et le routage. Le résultat obtenu ne correspondant pas aux attentes du concepteur, il faut donc trouver une solution pour l'améliorer. Cette solution consiste à optimiser ou ré-optimiser la description structurelle obtenue après synthèse, tout en tenant compte des informations temporelles fournies après le placement et le routage. La convergence de cette méthode dépend de la capacité de l'outil d'optimisation à générer un fichier de contraintes pour le placement et le routage, et de la capacité de l'outil de placement et de routage à prendre en compte ce fichier de contraintes. La méthode qui vient d'être exposée est illustrée à la figure III.26.

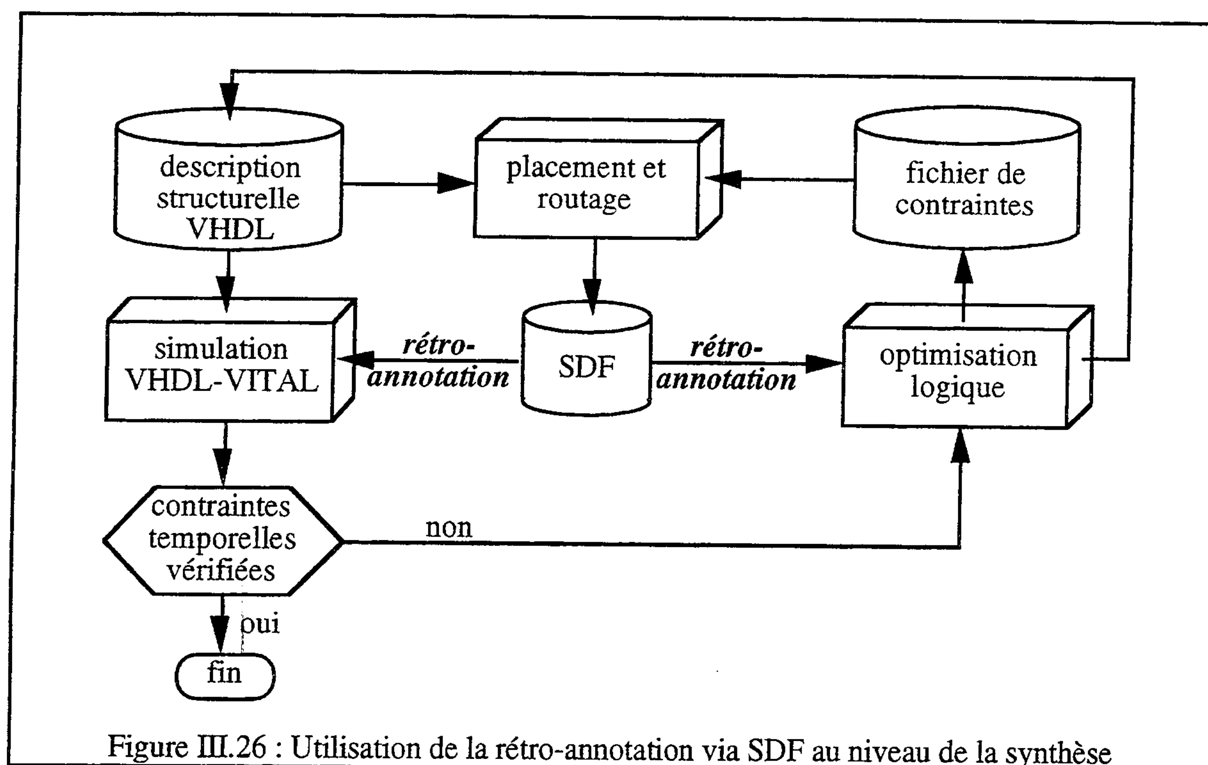

Figure III.26 : Utilisation de la rétro-annotation via SDF au niveau de la synthèse

3.2.9. Limitations

Une première limitation provient de la difficulté à obtenir une bibliothèque VITAL certifiée. Pour créer une telle bibliothèque, il faut posséder toutes les informations physiques et temporelles liées à chaque élément de la bibliothèque, les transcrire dans les modèles VITAL et vérifier enfin que toutes ces données sont en accord. Manuellement, cette tâche est estimée à six hommes mois pour une bibliothèque ASIC [Crow 94]. Ceci entraîne donc un retard important entre la mise en place d'un nouveau processus technologique et son utilisation en conception. Cependant, ce délai peut être réduit par la création automatique de bibliothèque VITAL. Plusieurs sociétés se sont intéressées à cet aspect et des outils capables de générer de telles bibliothèques sont en train de voir le jour. De tels outils sont à l'étude ou existent déjà chez Synopsys, chez Meta-Software [Crow 94], chez Compass [Krol 94b], chez Mentor Graphics [Hama 94] et chez Cadence [Naya 94].

Un second point venant compliquer l'utilisation de VITAL provient du fait que les modèles doivent respecter un certain style de description afin de permettre une accélération au niveau de la simulation VHDL. Pour permettre cette accélération, les modèles doivent être compatibles au niveau 1 et utiliser les paquetages VHDL "VITAL_Timing" et "VITAL_Primitives". De plus, posséder une bibliothèque VITAL "X" ne signifie pas forcément pouvoir l'utiliser efficacement avec un simulateur "Y", ceci dépend de la façon dont sont décrits les modèles [Levi 94]. Les règles de description spécifiées dans VITAL restent encore assez souples, cependant afin

d'utiliser au mieux certains simulateurs, ces règles sont rendues plus rigides. Par exemple, pour utiliser efficacement le simulateur Vulcan, la société VEDA recommande d'utiliser des équations Booléennes pour modéliser des parties combinatoires et des tables pour modéliser des parties séquentielles.

D'autres limitations sont liées à la spécification même du format VITAL. Par exemple, dans l'état actuel du format VITAL version 2.2b, il n'est pas possible de décrire de mémoires RAM ou ROM. De plus, VITAL s'appuie sur la norme VHDL IEEE de 1987 et non pas celle de 1992, alors que la norme de 1992 est moins restrictive, notamment en ce qui concerne les identificateurs, ce qui permet de remédier aux problèmes pouvant survenir lors du nommage des éléments ou de leurs ports. La liste des imprécisions de la spécification et des problèmes répertoriés et non résolus par cette version de VITAL figure en annexe D du document [VITAL].

Remarque :
Les informations données dans cette section s'appuie sur la version 2.2b de VITAL [VITAL] datant du 25 Mars 1994. Cependant une nouvelle version : la 3.0, vient d'être acceptée comme standard IEEE en Novembre 1995. N'ayant pas pu prendre connaissance de cette nouvelle version, il ne m'est pas permis aujourd'hui de fournir de plus amples informations. Souhaitons seulement que cette nouvelle spécification remédie aux problèmes qui ont été évoqués.

3.3. LPM

LPM (pour "Library of Parameterized Modules") est un ensemble d'éléments permettant de décrire les opérations logiques d'un circuit digital.

3.3.1. Motivations

La motivation essentielle de LPM provient du désir de conserver, dans le cadre d'une description structurelle d'un circuit digital, les fonctions de haut niveau, plutôt que de les éclater en fonctions logiques de base, tout en restant indépendant de la technologie.

Pour chaque technologie, qu'il s'agissent des CPLDs, des FPGAs ou des ASICs, il existe généralement une méthode unique et efficace pour implanter les fonctions de haut niveau telles que des additionneurs, des compteurs, etc. Si ces fonctions de haut niveau sont éclatées en équations logiques, il devient alors difficile, pour ne pas dire impossible, de retrouver la fonctionnalité de départ à partir de ces équations, de sorte à implanter efficacement le circuit sur la technologie ciblée.

Avec l'introduction des FPGAs, capables d'implanter des fonctions complexes possédant chacune une architecture unique pour chaque type de FPGA, le besoin de LPM s'est fait encore davantage ressentir. En effet, deux difficultés interviennent lors de la création d'un outil de synthèse capable d'utiliser efficacement les FPGAs. Tout d'abord, les personnes chargées de développer l'outil de synthèse n'obtiennent généralement pas assez suffisamment et pas assez rapidement les informations provenant des fabriquants de FPGAs. Par contre, les fabriquants de FPGAs qui ont ces informations manquent de connaissance en synthèse. LPM est là pour combler le fossé entre ces deux mondes : celui de la synthèse et celui des FPGAs.

A l'initiative de Cecil Kaplinsky de Plus Logic, un groupe de cinq compagnies comprenant Data I/O, Mentor Graphics, Viewlogic et Xilinx s'est réuni en Octobre 1990 avec pour objectifs la définition du nouveau standard LPM. Une version préliminaire a été écrite et diffusée en Mars 1991. Trois autres sociétés ont rejoint le groupe de travail initial : AT&T, Exemplar Logic et Minc. Le flot de conception utilisant LPM a alors été clarifié donnant naissance à une seconde version préliminaire en Juin 1991. Au début de 1992, avec l'aide apportée par NeoCAD et Actel, le travail fait jusqu'à lors a été repris, ce qui a eu pour aboutissement la mise en place d'une troisième version : la version 2.0 [LPM]. LPM a été accepté en tant que standard EIA (Electronic Industries Association) en Avril 1993 comme complément au standard EDIF 200 sous la référence EIA/IS-103.

3.3.2. Objectifs

Le premier objectif de LPM est de permettre une utilisation rapide et efficace des technologies CPLDs, FPGAs et ASICs par les concepteurs, sans qu'ils aient besoin par cela de connaître les détails architecturaux de ces technologies. L'utilisation des éléments LPM se fait soit par l'intermédiaire d'un outil de synthèse, soit par celle d'un éditeur schématique.

Le second objectif de LPM est de fournir un ensemble d'éléments génériques et indépendants de la technologie, utilisables pour concevoir un circuit, afin d'obtenir de meilleures performances à partir d'un grand nombre de technologies.

Dans le but d'atteindre ces objectifs, LPM a été conçu suivant ces règles :
- spécifier entièrement un circuit. L'ensemble des éléments LPM permet de décrire entièrement la logique de la plupart des circuits. Seulement quelques fonctions exotiques non implantées dans les technologies FPGAs ne sont pas représentées dans LPM.
- permettre une correspondance efficace avec les technologies. Quelque soit le circuit conçu avec n'importe quel outil de synthèse utilisant LPM, la description résultante, interconnectant

des blocs LPM, peut être implantée sur n'importe quelle technologie possédant un traducteur des éléments LPM vers ceux de la technologie.
- permettre de décrire le circuit indépendamment de la technologie. Le circuit ainsi décrit peut être traduit vers n'importe quelle technologie, voire même sur plusieurs technologies à la fois, différentes parties du même circuit pouvant être implantées sur différentes technologies.
- séparer la description logique de la description physique. Les éléments LPM ne contiennent pas d'informations physiques car elles sont trop liées à la technologie. LPM reste donc indépendant de la technologie et décrit seulement la fonctionnalité du circuit.
- permettre la description d'informations temporelles en vue de guider la synthèse. LPM, qui se veut indépendant de la technologie, permet de décrire la fonctionnalité du circuit ainsi que certaines contraintes temporelles pouvant être exploitées par des outils de synthèse, de placement-routage, etc.

3.3.3. Flot général d'utilisation de LPM

Le flot général de conception utilisant LPM est présenté à la figure III.27.

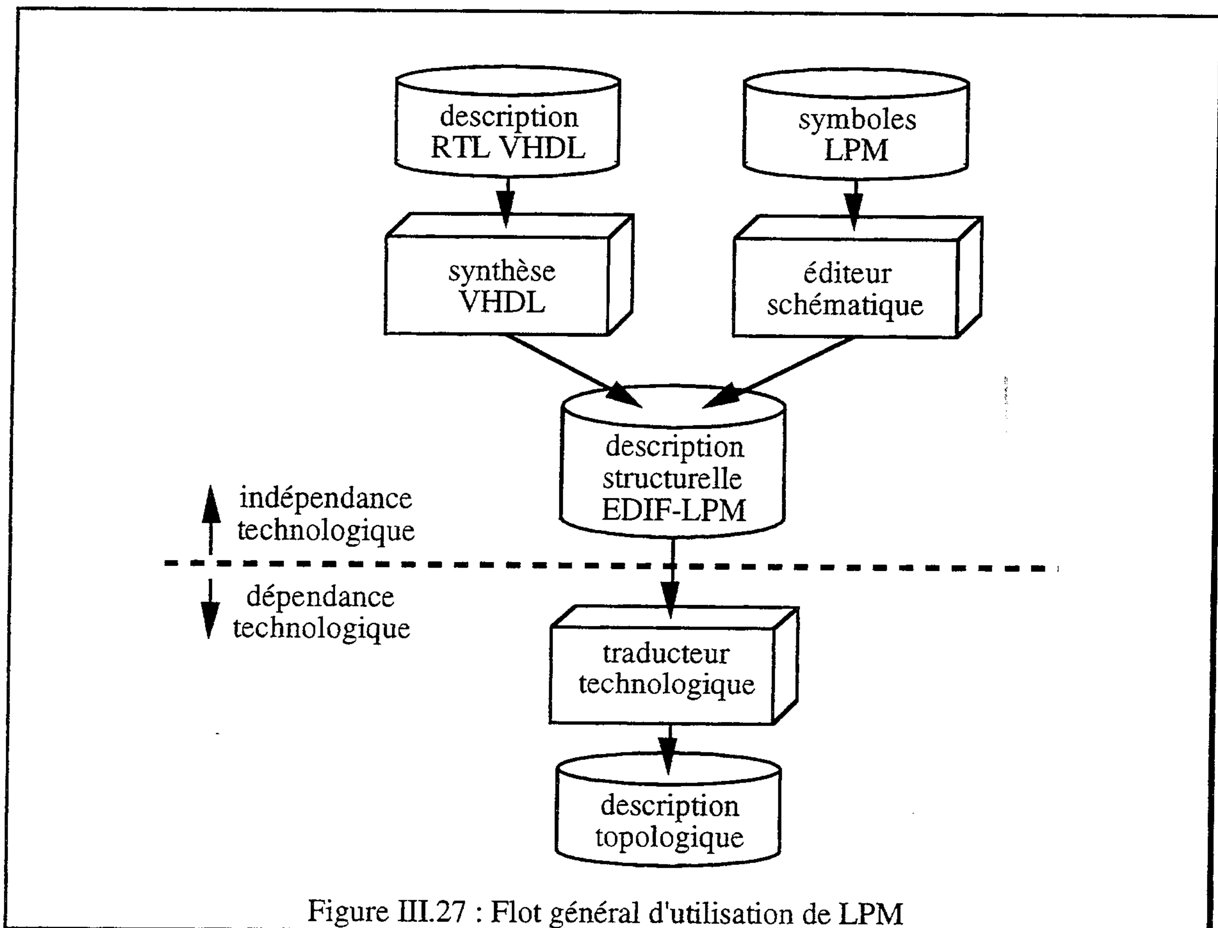

Figure III.27 : Flot général d'utilisation de LPM

La description structurelle interconnectant les éléments LPM peut être obtenue soit par un outil de synthèse à partir d'une description comportementale du circuit, soit à partir d'un éditeur schématique ayant à sa disposition les symboles LPM dans sa base de données. La description structurelle représente le circuit sous la forme d'une liste d'éléments LPM interconnectés. Cette description n'utilise que des éléments LPM. Le format de représentation utilisé est le format EDIF 200 [EDIF]. D'autres fichiers peuvent accompagner cette description donnant, par exemple, le contenue d'une ROM, celui d'une machine d'états finis, ou encore celui d'une table de vérité contenue dans le circuit, tous ces éléments étant des éléments LPM. Le traducteur technologique au sens de [LPM] implante la description structurelle logique, indépendante de la technologie, en une description topologique, spécifique à la technologie ciblée. Ce traducteur inclue les étapes de placement et de routage.

3.3.4. Passage des éléments LPM à la technologie ciblée

Lors du passage des éléments LPM aux cellules de la technologie ciblée, les contraintes liées à cette technologie doivent être prises en compte. Ces contraintes limitent à la fois la surface, liée à la contenance du boîtier dans le cas des CPLDs et des FPGAs, et la vitesse du circuit à synthétiser. Dans le but de pouvoir optimiser en surface et en temps la description structurelle EDIF-LPM, l'outil de synthèse a besoin de connaître plusieurs informations :
- la liste des ressources existantes dans la technologie considérée,
- le nombre de ressources utilisées et la vitesse estimée lors de l'implantation d'un élément LPM sur la technologie.

Le flot complet d'utilisation des éléments LPM dans le cycle de conception et la description du passage des éléments LPM à la technologie ciblée est schématisé à la figure III.28.

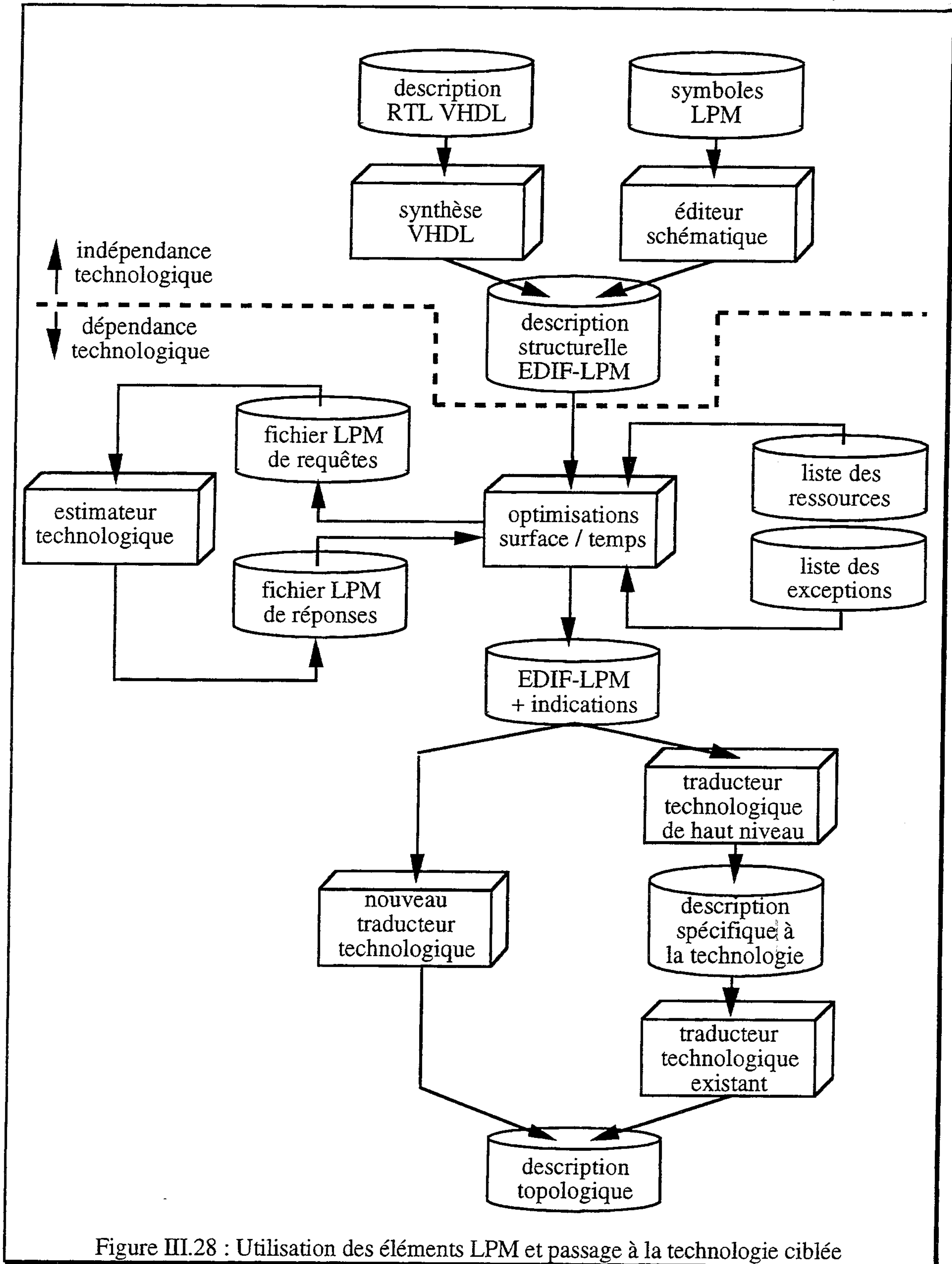

Figure III.28 : Utilisation des éléments LPM et passage à la technologie ciblée

Génération par l'outil d'optimisation du fichier de requêtes :

Un élément LPM peut être implanté de différentes façons sur la même technologie. Lors de l'optimisation, un fichier LPM de requêtes est passé à un estimateur spécifique à la technologie considérée. Ce fichier de requêtes peut être constitué de toute la description structurelle EDIF-LPM du circuit ou bien seulement d'une partie de cette description. Les requêtes sont associées aux instances des éléments LPM interconnectées dans cette description. Il peut s'agir, soit de valeurs temporelles limites, comme le temps de propagation maximum d'une entrée vers une sortie d'une certaine instance LPM, soit du nombre maximum de ressources à employer accompagné parfois également du type de ressources à utiliser, comme par exemple, utiliser quatre CLBs pour implanter une instance LPM sur une technologie Xilinx.

Génération par l'estimateur du fichier de réponses :

L'estimateur technologique tente d'implanter chaque instance LPM en prenant en compte les requêtes qui y sont associées et renvoie un fichier de réponses à l'outil d'optimisation. Les requêtes sont considérées comme des contraintes par l'estimateur. Pour chaque instance LPM accompagnée de requêtes, trois réponses peuvent être envisagées. Dans le premier cas, quelles que soient les requêtes temporelles, si les contraintes concernant les ressources n'ont pas pu être atteintes lors de la tentative d'implantation de l'instance LPM sur la technologie ciblée, alors l'estimateur indique qu'il a échoué. Dans le second cas, les contraintes concernant les ressources ont pu être respectées, mais pas les contraintes temporelles. L'estimateur renvoie dans ce cas les informations de temps et de surface de l'implantation qu'il juge la meilleure. Dans le dernier cas, toutes les contraintes ont pu être respectées. L'estimateur renvoie alors les informations de temps et de surface de toutes les implantations vérifiant ces contraintes.

Génération par l'outil d'optimisation de la description EDIF-LPM accompagnée d'indications :

A partir du fichier LPM de réponses fourni par l'estimateur spécifique à la technologie, l'optimisation sélectionne la meilleure implantation pour chaque instance LPM. Ce choix se fait à partir des informations de temps et de surface déduites par l'estimateur et en fonction des critères de temps et de surface spécifiés par l'utilisateur. Dans la description structurelle EDIF-LPM, ce choix se traduit sous la forme d'indications spécifiques à la technologie accompagnant les instances LPM. Le résultat de l'optimisation est donc une description structurelle en EDIF interconnectant des éléments LPM, auxquels sont associées des informations permettant de guider le traducteur lors de l'implantation physique. Ces informations peuvent être des informations de surface et de temps et seront alors considérées comme des contraintes par le traducteur. Il peut également s'agir directement des implantations effectuées par l'estimateur et choisies par l'optimisation. Ces informations ne sont pas clairement définies dans le standard LPM et dépendent en fait de la spécificité de chaque traducteur technologique.

A partir de cette description, deux flots peuvent être envisagés selon le type de traducteur.

Utilisation d'un traducteur technologique prenant en compte les éléments LPM :
Le premier flot (branche de gauche de la figure III.28) considère l'utilisation d'un traducteur sachant prendre en compte les éléments LPM afin de les implanter sur la technologie considérée. Il réalise également le placement et le routage du circuit. Cependant, ce type de traducteur commence à peine à être développé par les fabriquants.

Utilisation des traducteurs technologiques existants via un traducteur de haut niveau :
Le second flot (branche de droite) tient compte des traducteurs déjà existants. Ces traducteurs ne savent pas utiliser les éléments LPM ; ils ne réalisent que le placement et le routage du circuit à partir d'une description dont le format et les cellules sont spécifiques à la technologie ciblée. Afin de fournir cette description, un traducteur de haut niveau spécifique à la technologie est donc employé. Ce traducteur doit faire correspondre à chaque élément LPM un ou plusieurs composants de la technologie en tenant compte des indications éventuelles associées à chaque élément LPM.

Utilisation du fichier d'exceptions :
Notons également que certaines technologies ne sont pas en état d'accepter tous les éléments LPM dans toutes leurs configurations possibles. Par exemple, certaines technologies ne possèdent pas de bascules D avec mise à un et mise à zéro asynchrones. Ce type de configurations n'ayant pas de correspondant dans la technologie considérée, doit être connu de l'outil de synthèse. A cet effet, une liste d'exceptions permet d'indiquer les configurations des éléments LPM supportées par une technologie donnée. Cette liste est prise en compte par l'optimisation lors du choix de l'implantation. De plus, les traducteurs mis à l'étude par certains fabriquants ne planifient le support que de certains éléments LPM [ATT]. Dans ce cas, seuls les éléments LPM supportés doivent figurer dans cette liste, accompagnés éventuellement des configurations acceptées pour la technologie considérée, dans le cas où elles ne sont pas toutes permises. Aucun fichier d'exception n'a besoin d'être défini lorsque tous les éléments LPM, dans toutes leurs configurations, sont acceptés par la technologie ciblée.

3.3.5. Les règles suivies lors de la définition des éléments LPM

Toute description structurelle est constituée de signaux et de composants. EDIF donne la syntaxe utilisée pour écrire une telle description et LPM ajoute une bibliothèque de composants de sorte à ce que cette description puisse être créée de manière standard. Plusieurs règles ont été établies afin de mettre en place la liste des éléments LPM. Ces règles sont les suivantes :

- ensemble complet. LPM fournit un ensemble complet de fonctions logiques. N'importe quelle fonction Booléenne peut être implantée en utilisant les éléments LPM.
- généralisation. Souvent, certaines fonctions peuvent être implantées de nombreuses manières différentes. LPM essaye de réduire une telle duplication. Par exemple, LPM ne compte pas de registre à décalage, mais ce registre peut être construit en utilisant la fonction optionnelle de décalage de l'élément registre LPM_DFF.
- popularité. Dans le but de ne définir qu'un nombre raisonnable d'éléments standards, seules les fonctions les plus répandues ont été définies en tant qu'éléments séparés.
- décomposition difficile. Si une fonction logique est difficile à reconnaître une fois qu'elle a été décomposée au niveau portes, alors elle devient candidate à l'inclusion.
- représentation. La représentation des données utilisées par les éléments LPM est un attribut associé à chaque élément. Un ensemble minimum de représentations couvrant la plupart des besoins en conception a été choisi.

3.3.6. Caractéristiques générales des éléments LPM

LPM fournit une liste de 25 éléments tous configurables. Chaque élément possède un nombre différent de paramètres permettant de fixer la fonctionnalité et la taille de l'élément.

Un paramètre permet également de fixer la représentation des données sur les ports d'entrée et de sortie des éléments. Cette représentation qui peut être non-signée ou signée (en complément à deux) influe sur l'implantation de certains éléments LPM.

Certains éléments possèdent également des ports optionnels. Si ces ports ne sont pas utilisés, alors toute la logique associée à ce port n'est pas implantée ce qui permet d'optimiser considérablement la taille de l'élément en question. Certaines combinaisons fonctionnelles sont mutuellement exclusives ce qui amène à la définition de combinaisons illégales de ports. Par exemple, pour l'élément LPM_DFF, l'utilisation du port ACONST interdit l'utilisation du port ACLR ou ASET.

Tous les éléments de mémorisation possèdent de nombreuses options notamment pour les fonctionnalités synchrones ou asynchrones de mise à un ou à zéro, etc. ainsi que pour la testabilité lors de l'utilisation de l'élément en mode de test dans le cas d'une technologie ASIC. Il est bien évident que les options de test peuvent être omises notamment lorsque la technologie considérée est une technologie CPLD ou FPGA.

Les éléments LPM peuvent être classés en cinq catégories : les portes logiques de base, les éléments arithmétiques, les éléments de mémorisation, les tables (tables de vérité ou machines

d'états finies) et les plots d'entrée-sortie. La liste des éléments LPM classés selon ces cinq catégories figure à la table III.1.

Table III.1 : Liste des éléments LPM classés par catégories

Portes	Arithmétique	Mémorisation	Tables	Plots
LPM_Constant	LPM_Add_Sub	LPM_Latch	LPM_Ttable	LPM_Inpad
LPM_Inv	LPM_Compare	LPM_Dff	LPM_Fsm	LPM_Outpad
LPM_And	LPM_Mult	LPM_Tff		LPM_Bipad
LPM_Or	LPM_Abs	LPM_Ram_Dq		
LPM_Xor	LPM_Counter	LPM_Ram_Io		
LPM_Bustri		LPM_Rom		
LPM_Mux				
LPM_Decode				
LPM_Clshift				

Chaque élément LPM est défini par son nom, ses ports d'entrée et de sortie, ses paramètres et sa fonctionnalité. Le nom permet d'identifier l'élément LPM. Chaque élément possède un certain nombre de ports d'entrée et de sortie. Parmi ces ports, certains sont nécessaires et d'autres peuvent être optionnels. Des combinaisons illégales de ports peuvent être définies dans ce cas. Un paramètre commun à tous les éléments LPM est la taille. D'autres paramètres peuvent être spécifiés suivant les éléments ; il peut s'agir de la représentation des données, de la valeur limite de comptage dans le cas d'un compteur, etc. La définition détaillée de chaque élément LPM figure dans [LPM].

3.3.7. Description structurelle EDIF-LPM

Un élément LPM est donc un élément de bibliothèque standard possédant un certain nombre de paramètres et éventuellement des ports optionnels. Ces éléments sont identifiables par leur nom débutant par "LPM_" ; la liste en est donnée à la table III.1.

Avant d'être utilisé dans une description structurelle EDIF, l'élément LPM doit être configuré, c'est-à-dire que les valeurs de ses paramètres doivent être fixées, ainsi que son interface, dans le cas de la présence de ports optionnels. Dans une description structurelle EDIF, la valeur associée à chaque paramètre est introduite par le mot clé "property". A chaque configuration

différente d'un élément LPM correspond une cellule EDIF identifiable par son nom. La figure III.29 définit la cellule AND2 au niveau d'une description structurelle dans le format EDIF. La cellule AND2 correspond à une configuration particulière de l'élément LPM_AND pour lequel les paramètres WIDTH et SIZE ont été respectivement fixés aux valeurs 1 et 2. Le paramètre LPM_TYPE permet d'identifier l'élément LPM auquel est rattachée cette configuration. Une fois définie, la cellule AND2 va pouvoir être utilisée dans la description structurelle EDIF.

```
(external LPM_LIBRARY
...
(cell AND2
    (cellType GENERIC)
    (view NETVIEW
        (viewType NETLIST)
        (interface
            (port DATA0_0) (port DATA0_1) (port RESULT0)
            (property LMP_TYPE (string "LPM_AND"))
            (property WIDTH (integer 1))
            (property SIZE (integer 2))
        )
    )
)
... )
```

Figure III.29 : Cellule EDIF correspondant à une configuration de l'élément LPM_AND

Une description structurelle est composée uniquement d'instances et de signaux interconnectant ces instances. Une instance correspond à une copie d'une cellule et une cellule est copiée autant de fois qu'elle est utilisée. Autrement dit, le nombre d'instances d'une même cellule dans un circuit correspond au nombre de fois où est utilisée cette cellule. Chaque instance est identifiable par son nom. La figure III.30 décrit deux instances : INST1 et INST2 de la même cellule AND2.

```
...
(instance INST1
    (viewRef NETVIEW
        (cellRef AND2 (libraryRef LPM_LIBRARY)))
)
(instance INST2
    (viewRef NETVIEW
        (cellRef AND2 (libraryRef LPM_LIBRARY)))
)
...
```

Figure III.30 : Deux instances de l'élément AND2 dans une description EDIF

La figure III.31 schématise le passage d'un élément LPM à une ou plusieurs cellules EDIF par le biais de la configuration de l'élément LPM, et le passage d'une cellule EDIF à une ou plusieurs instances suite à l'utilisation de cette cellule dans la description structurelle du circuit.

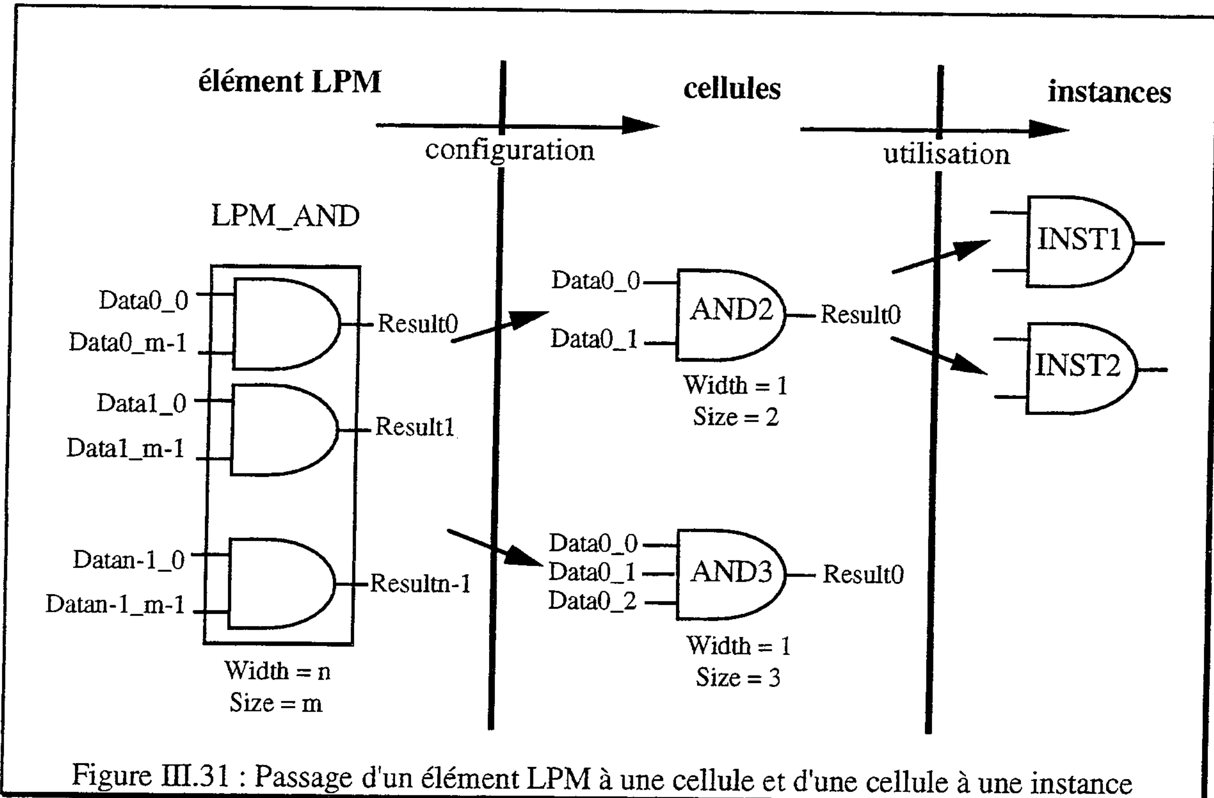

Figure III.31 : Passage d'un élément LPM à une cellule et d'une cellule à une instance

3.3.8. Autres fichiers associés à LPM

Nous avons vu au cours du flot LPM présenté précédemment qu'un fichier de requêtes pouvait être émis par l'outil d'optimisation. Les requêtes temporelles associées à une instance LPM sont de quatre types : le temps maximal de propagation ("tpd"), le temps minimal d'arrivée ("tsetup"), le temps minimal de maintien ("thold) et la période minimale ("tperiod"). Ces temps ont été définis auparavant. Les requêtes concernant les ressources à employer ("resource_limits") pour implanter une instance LPM permettent d'indiquer le nombre maximal de ressources à utiliser et éventuellement le type de ces ressources. Le fichier de requêtes est constitué de la description EDIF-LPM d'une partie du circuit ou du circuit complet, dans laquelle des requêtes sont associées aux instances LPM. Ces requêtes sont donc spécifiées en format EDIF. Un exemple en est donné à la figure III.32. Dans cet exemple, deux requêtes sont associées à l'instance INST1 de la cellule ADD4 définie dans la bibliothèque LPM_LIBRARY. La première requête indique que le temps de propagation entre l'entrée CIN et la sortie COUT de l'instance INST1 ne doit pas dépasser 10 ns. Cinq itérations pourront être effectuées par

l'estimateur pour parvenir à une solution. La seconde requête indique que pas plus de cinq CLBs devront être utilisés pour implanter l'instance INST1 sur la technologie considérée.

```
...
(instance INST1
    (viewRef NETVIEW
        (cellRef ADD4 (libraryRef LPM_LIBRARY)))
    (userData TPD Q1 5 (portRef CIN) (portRef COUT) 10)
    (userData RESOURCE_LIMITS Q1 5
        (userData RESOURCE CLB 5))
)
...
```

Figure III.32 : Exemple d'une partie de fichier de requêtes

En retour, un fichier de réponses est fournit par l'estimateur à l'outil d'optimisation. Ce fichier utilise également le format EDIF. Un exemple en est donné à la figure III.33. Dans cet exemple, deux implantations répondant aux requêtes formulées à la figure III.31 ont été trouvées. La première solution propose un temps de propagation de 7,5 ns pour 4 CLBs utilisés alors que la seconde propose 6,5 ns pour le même nombre de CLBs utilisés.

```
(userData LPM_RESPONSE
...
    (userData QUERY_ID Q1
    (userData ANSWER_ID (string "Implantation_ADD4_Q1_100")
        (userData LPM_STATUS YES)
        (userData ITERATION 3)
        (userData TPD 7.5)
        (userData RESOURCE_SET
            (userData RESOURCE CLB 4))
    (userData ANSWER_ID (string "Implantation_ADD4_Q1_101")
        (userData LPM_STATUS YES)
        (userData ITERATION 5)
        (userData TPD 6.5)
        (userData RESOURCE_SET
            (userData RESOURCE CLB 4))
...
```

Figure III.32 : Exemple d'une partie de fichier de réponses

Parmi toutes les implantations possibles proposées par l'estimateur, l'outil d'optimisation sélectionne la meilleure. Cette solution peut alors être passée au traducteur technologique via la propriété "LPM_HINT" associé à une instance de la description structurelle LPM-EDIF. Dans l'exemple donné à la figure III.3, la seconde implantation a été choisie et est passée au traducteur technologique.

```
...
(instance INST1
    (viewRef NETVIEW
        (cellRef ADD4 (libraryRef LPM_LIBRARY)))
    (property LPM_HINT
        (userData ANSWER_ID (string "Implantation_ADD4_Q1_101")))
)
...
```

Figure III.33 : Exemple de description structurelle EDIF-LPM contenant une indication

D'autres indications peuvent être fournies au traducteur technologique. Ces indications peuvent correspondre aux requêtes formulées auparavant et, dans ce cas, elles sont utilisées comme des contraintes par le traducteur technologique.

Les grammaires précises de tous ces fichiers ainsi que celle du fichier d'exceptions figurent dans le manuel LPM [LPM].

3.3.9. Utilisation au niveau d'ASYL+

Le flot général de synthèse employé dans ASYL+ utilisant des macro blocs a été présenté au cours du chapitre II. Ce flot a été adapté pour chaque nouvelle cible technologique que nous avons voulu prendre en compte, ce qui a abouti à un flot de synthèse spécifique par technologie considérée. Notamment, les flots spécifiques à Actel et Xilinx ont été présentés au cours du chapitre II. L'inconvénient majeur de tous ces flots provient du fait que la technologie ciblée est prise en compte dès le début du flot de synthèse, lors de la recherche de macro blocs. L'introduction de LPM nous a permis d'envisager un nouveau flot de synthèse unifié utilisant les macro blocs, en remplacement aux flots spécifiques existants. Ce nouveau flot est dit unifié car il permet, dans un premier temps, de rechercher les macro blocs indépendamment de la technologie considérée, qui ne sera prise en compte que dans un second temps. La transition entre ces deux étapes se fait au travers des éléments LPM. Le nouveau flot de synthèse basé sur LPM est schématisé à la figure III. 34.

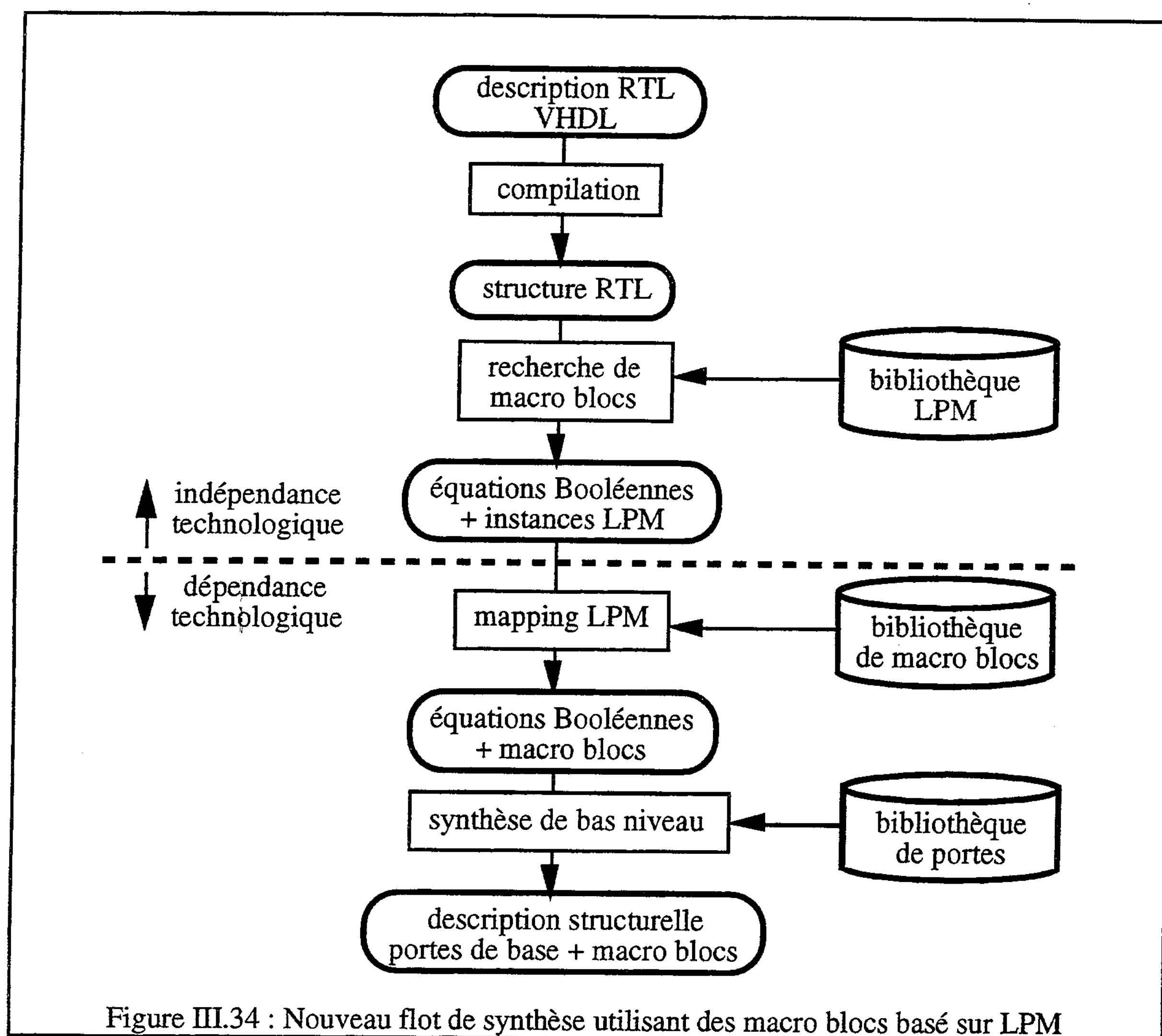

Figure III.34 : Nouveau flot de synthèse utilisant des macro blocs basé sur LPM

Ce nouveau flot de synthèse part toujours d'une description RTL spécifiée en VHDL (ou Verilog). Après compilation, la structure RTL interne est remplie. La recherche des macro blocs se fait sur cette structure. Les macro blocs recherchés correspondent à ceux définis dans la bibliothèque LPM qui est indépendante de la technologie considérée. Contrairement aux flots de synthèse précédents, la recherche de macro blocs reste donc identique quelle que soit la cible technologique.

Actuellement tous les éléments définis dans la norme EDIF LPM ne sont pas encore pris en compte dans ce nouveau flot. Les éléments décrits dans la bibliothèque LPM utilisée aujourd'hui sont : l'additionneur-soustracteur (LPM_ADD_SUB), le comparateur (LPM_COMPARE), le compteur (LPM_COUNTER), le multiplexeur (LPM_MUX) et le décodeur (LPM_DECODE). Deux autres éléments ne figurant pas dans la norme EDIF LPM ont été rajoutés dans cette bibliothèque. Il s'agit d'un incrémenteur-décrémenteur (LPM_INC_DEC) et d'un accumulateur (LPM_ACCUM). Bien que ne figurant pas dans le

standard EDIF LPM, ces deux éléments ont été rajoutés car leurs fonctionnalités, qui restent de haut niveau, diffèrent de celles des éléments déjà utilisés et car il s'agit de fonctionnalités courantes dans les technologies employées.

Le résultat de cette première étape de synthèse est donc une structure composée d'instances LPM correspondant aux macro blocs trouvés, plus des équations Booléennes correspondant au reste du circuit. Jusqu'à ce niveau, la cible technologique n'a pas encore été prise en compte.

Elle est prise en compte dans l'étape suivante appelée "mapping LPM", qui consiste à rechercher pour chaque instance LPM un macro bloc équivalent. Cette étape sera reprise plus en détail dans la section suivante. La structure obtenue après l'étape de mapping LPM est donc constituée de macro blocs spécifiques à la technologie, correspondant aux instances LPM pour lesquelles un équivalent a été déterminé, ainsi que d'équations Booléennes correspondant au reste du circuit. Notons que les instances LPM pour lesquelles aucun équivalent n'a pu être trouvé ont donc du être resynthétisées et figurent à ce niveau sous la forme d'équations Booléennes.

La dernière étape de synthèse de bas niveau consiste à optimiser et à mapper les équations Booléennes sur la bibliothèque de portes de base de la technologie considérée. A la fin de cette dernière étape, le circuit est donc décrit sous la forme d'un ensemble d'éléments interconnectés spécifique à la technologie. Cet ensemble d'éléments peut être constitué d'un mélange de portes de base et de macro blocs.

3.3.10. Mapping LPM et optimisation par dérivation

Le mapping LPM consiste à faire correspondre à chaque instance LPM l'instance d'un élément équivalent dans la technologie considérée. Cet élément peut être soit un macro bloc, soit un assemblage de portes, soit un mélange des deux. La méthode employée pour réaliser le mapping LPM est schématisée à la figure III.35.

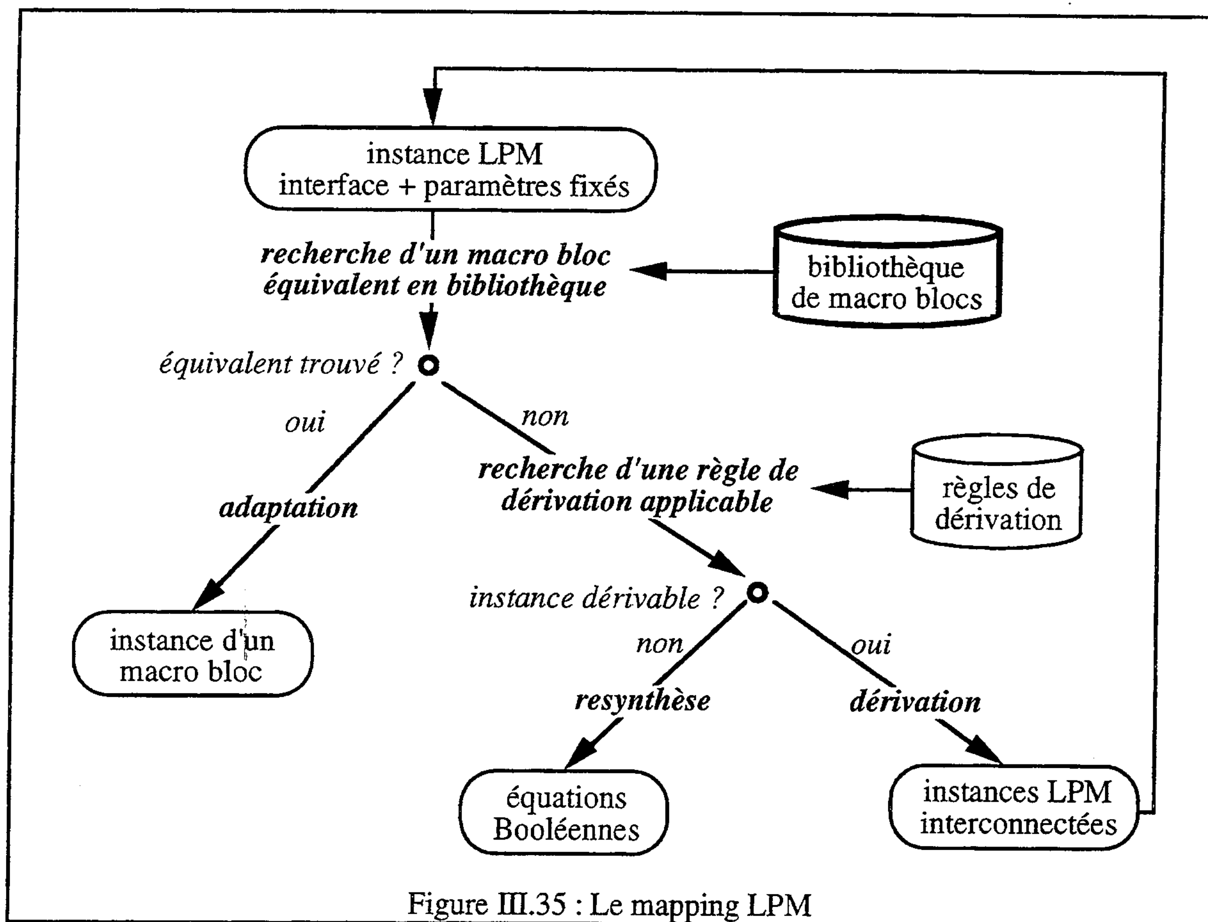

Figure III.35 : Le mapping LPM

Une instance LPM correspond à un élément LPM pour lequel l'interface et les paramètres ont été fixés. Un élément LPM étant indépendant de la technologie, le problème consiste donc à trouver pour chaque instance LPM une implantation physique équivalente dans la cible technologique.

Le principe général de cette méthode est le suivant :

Pour chaque instance LPM une tentative d'implantation directe sur la bibliothèque de macro blocs est effectuée. Si cette tentative réussie, l'instance d'un macro bloc est substituée à l'instance LPM. Une mise à jour des ports et des paramètres entre l'instance LPM et celle du macro bloc est réalisée. Des équations d'adaptation peuvent également être générées afin d'adapter l'interface de l'instance du macro bloc à l'instance LPM, comme par exemple dans le cas où les signaux de commande ne sont pas actifs sur les même niveaux, au quel cas une inversion de ces signaux doit être effectuée.

Si la tentative échoue, deux cas peuvent se présenter. Si l'instance LPM peut être représentée par d'autres instances LPM plus simples, c'est-à-dire si une dérivation est possible, alors l'instance LPM d'origine est remplacée par d'autres instances LPM interconnectées [Reve 95]. Le même traitement est ensuite opéré sur ces sous-instances. Dans le cas contraire,

l'instance LPM est resynthétisée en équations Booléennes équivalentes qui seront ensuite mappées sur la bibliothèque de portes de base de la cible technologique.

Le passage d'une instance LPM à celle d'un macro bloc de bibliothèque se fait par correspondance entre la liste des fonctionnalités de l'instance LPM et la liste des fonctionnalitées du macro bloc. Davantage de détails sont donnés dans [Reve 95] et [Troi 95]. La liste de ces fonctionnalités correspond en fait à la liste des opérations que peut effectuer ce bloc. Cette liste est spécifiée par le format de bibliothèque présentée en début de ce chapitre.

Lorsqu'il n'existe pas de macro bloc équivalent en bibliothèque pouvant remplacer l'instance LPM, une tentative de décomposition ou dérivation de cette instance en instances LPM plus simples interconnectées est effectuée. Le principe de cette décomposition peut être formalisé par un ensemble de règles de dérivation. Les règles de dérivation actuellement implantées dans le mapping LPM permettent de décomposer un accumulateur en un additionneur-soustracteur et un registre, un compteur en un incrémenteur-décrémenteur et un registre et un incrémenteur-décrémenteur en un additionneur-soustracteur. Une autre règle permet également de dériver un additionneur-soustracteur en additionneur. Ces règles ne prennent en compte que la fonctionnalité des instances LPM, mais d'autres règles permettant également de prendre en compte la taille peuvent également être définies.

LPM nous a donc permis d'unifier notre flot de synthèse utilisant les macro blocs de bibliothèques. Ce nouveau flot permet de prendre en compte plus rapidement les macro blocs des nouvelles technologies puisque toute une partie du traitement reste indépendante de la technologie. L'utilisation de LPM nous a également permis de mettre en place de nouvelles optimisations basées sur des règles de dérivation. Ces optimisations peuvent être réalisées quelle que soit la cible technologique et permettent d'utiliser efficacement les macro blocs de ces bibliothèques.

4. Conclusion

Au cours de ce dernier chapitre nous avons tout d'abord étudié les deux formats de bibliothèques utilisés dans ASYL+ pour représenter respectivement les portes de base et les macro blocs d'une cible technologique. Après avoir énoncé les limitations de chacun de ces formats, nous avons alors proposé un troisième format remédiant aux problèmes rencontrés lors de l'utilisation des formats précédents.

Nous nous sommes ensuite intéressé au format de bibliothèque VITAL qui vient d'être accepté en tant que standard IEEE. Après avoir évoqué les motivations et les principales caractéristiques

de ce format basé sur le langage VHDL, nous avons souligné les bénéfices apportés par ce format lors de son utilisation en simulation, puis en synthèse, par le lien d'un fichier SDF permettant de rétro-annoter une description structurelle VHDL utilisant des éléments de bibliothèque VITAL.

Dans la dernière partie de ce chapitre, nous avons présenté un autre standard : LPM. LPM est une extension à la norme EDIF utilisé pour représenter une description structurelle de circuit. LPM ajoute à cette norme une bibliothèque de composants indépendants de toute cible technologique. Cet ensemble de composants ainsi que le flot général de conception utilisant de tels éléments ont été décrits. En fin de cette partie, nous avons exposé le nouveau flot intégré dans ASYL+ permettant d'utiliser plus efficacement les macro blocs des différentes technologies au travers de LPM. L'utilisation de LPM nous a également permis de mettre en place dans ce flot de synthèse des optimisations basées sur des règles de dérivation permettant de mieux s'adapter aux spécificités et au contenu des bibliothèques de chaque cible technologique.

Conclusion Générale

-- ◊ - ◊◊◊ - ◊ --

Cette thèse concerne l'utilisation du langage VHDL dans la synthèse de circuits dédiés et en particulier l'étude de macro blocs.

Nous avons rappelé les principales caractéristiques de ce langage en justifiant le fait de sa standardisation en 1987. Les améliorations qui y ont été apportées lors de la mise à jour du standard en 1992 sont également précisées à ce niveau. Le choix qui s'est porté sur VHDL a ensuite été expliqué à l'aide d'éléments de comparaison par rapport à d'autres langages tels que Verilog par exemple.

L'essentiel du travail a porté sur l'aspect synthèse. Après avoir exposé les limitations introduites par l'utilisation du langage VHDL en synthèse, nous avons présenté la méthodologie de conception de circuits à partir de spécifications en VHDL. Ensuite nous avons donné plusieurs exemples de modèles VHDL décrivant des éléments plus ou moins complexes pouvant aller des portes combinatoires jusqu'aux compteurs ou RAMs en passant par les multiplexeurs ou autres bascules.

Une des contributions de ce travail nous paraît l'étude de l'utilisation automatisée des macro blocs en synthèse. Ce problème a été posé suite à deux constatations : la première provient du fait que les fabriquants développent de plus en plus de bibliothèque de macro blocs [ACTgen], [XBLOX], [LPM] et la seconde, qui découle en fait de la première, est que l'utilisation de ces macro blocs amène évidemment à des résultats meilleurs en temps et en surface [Bidd 95]. Après avoir défini la notion de macro bloc, nous avons précisé les différentes méthodes

permettant d'y faire appel à partir d'une spécification donnée en langage VHDL. La méthodologie employée dans le système de synthèse ASYL+ afin d'utiliser ces macro blocs a alors été exposée. L'utilisation concrète des macro blocs de deux fabriquants : ceux de Xilinx et ceux d'Actel, a permis d'expérimenter cette méthodologie et a aboutie à des résultats très satisfaisants [Bert 94a]. Enfin, plusieurs optimisations implantées dans le système de synthèse ASYL+ et s'appliquant aux macro blocs ont été proposées. Il s'agit essentiellement du pliage d'opérateurs dans le cas d'opérations mutuellement exclusives, de la reconnaissance de sous-expressions communes et de la simplifications d'expressions.

Après nous être intéressés à l'utilisation automatique des macro blocs en synthèse, nous avons abordé le problème de la modélisation de ces éléments en bibliothèque. Un nouveau format de bibliothèque a été proposé, permettant de décrire toutes les informations relatives à un élément, en vue de son utilisation en synthèse. Ces informations ont pu être répertoriées suite à l'étude de diverses bibliothèques et de leur incapacité à être entièrement décrites dans les deux formats anciennement utilisés dans ASYL+. La modélisation des éléments de bibliothèque étant un problème majeur soulevé par toutes les personnes travaillant dans le domaine de la CAO, un effort de standardisation a été fait en vue de mettre au point un format international. Il s'agit plus précisément du format VITAL : "VHDL Initiative Towards ASIC Libraries" dont les caractéristiques principales ont été vues au cours du chapitre III. Un autre standard : LPM ("Library of Parameterized Modules"), définissant une bibliothèque d'éléments virtuels, a également vu le jour. Son but essentiel est de simplifier l'interface entre les outils de synthèse et les outils des fabriquants développant des bibliothèques de macro blocs. Le contenu de cette bibliothèque et les avantages de son utilisation ont été résumés.

Pour conclure, nous pensons que le point original de cette thèse réside dans l'utilisation automatique des macro blocs lors de la synthèse et la mise au point d'un format efficace de description des éléments en bibliothèque. Ce travail s'inscrit en partie dans le cadre du projet européen LINK (projet ESPRIT BRA 6855) destiné à développer les connexions entre la synthèse de haut niveau à celle de bas niveau.

La méthodologie d'utilisation des macro blocs est actuellement implantée dans le système de synthèse ASYL+ et a abouti à de bons résultats. Cependant, ces résultats peuvent sûrement encore être améliorés par une étude plus approfondie des optimisations possibles. Certaines optimisations non encore implantées dans ASYL+ ont été soulignées et peuvent servir d'extension à ce travail.

Notons également que les blocs de bibliothèque deviendront de plus en plus complexes. Or, les outils de synthèse doivent suivre coûte que coûte l'évolution technologique de sorte à toujours

être à la pointe du progrès. C'est la raison pour laquelle ce travail de recherche qui a consisté à utiliser au mieux les macro blocs de bibliothèque devra être poursuivi dans l'avenir.

Bibliographie et références bibliographiques

-- ◊ - ◊◊◊ - ◊ --

[Abou 92] P. Abouzeid,
"Méthodes de factorisation algébrique dédiées aux circuits intégrés complexes",
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue en Décembre 1992.

[Actel] "ACT Family Macro Library Guide",
Actel, 1992.

[ACTgen] "ACTgen Macro Builder, User's guide",
Actel, 1992.

[Airi 90] R. Airiau, J.M. Bergé, V. Olive, J. Rouillard,
"VHDL, du langage à la modélisation",
Presses Polytechniques Universitaires Romandes, 1990.

[Airi 94] R. Airiau, J.M. Bergé, V. Olive,
"Circuit synthesis with VHDL",
Kluwer Academic Publishers, 1994.

[ASYL] "ASYL+/PLS : Users Manuel",
Innovatives Synthesis Technologies, 1995.

[ATT] "AT&T Field-Programmable Gate Arrays Data Book",
AT&T Microelectronics, Avril 1995.

[Belh 93] H. Belhadj, L. Gerbaux, M.C. Bertrand, G. Saucier,
"Specification and Synthesis of Communicating Finite State Machines",
Publié dans le livre intitulé "Synthesis for control dominated circuits" par "Elsevier science publishers" en 1993, pp. 91-102.

[Berg 92] J.M. Bergé, A. Fonkoua, S. Maginot, J. Rouillard,
"VHDL Designer's Reference",
Kluwer Academic Publishers, 1992.

[Berg 93a] J.M. Bergé,
"VHDL'92: the new VHDL standard",
Tutorial présenté à la conférence VLSI 93, à Grenoble, France, le 10 Septembre 1993.

[Berg 93b] J.M. Bergé, A. Fonkoua, S. Maginot, J. Rouillard,
"VHDL'92",
Kluwer Academic Publishers, 1993.

[Berm 94] V. Berman, Cadence,
"VITAL - A key standard for ASIC libraries"
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[Bert 91] M.C. Bertrand,
"Etude et synthèse automatique d'une librairie de blocs fonctionnels en langage VHDL",
Rapport de stage de DEA de Micro-électronique soutenue à Grenoble, France, en Juin 1991.

[Bert 92a] M.C. Bertrand, A. Mignotte, M. Crastes, J. Fron, J. Rampon,
"Interactive Register Transfer Level Synthesis Using Library Blocks",
Proc. EURO ASIC'92, pp. 53-58, Paris, France, 1-5 Juin 1992.

[Bert 92b] M.C. Bertrand, M. Crastes, A. Mignotte,
"VHDL Subset for Control Dominated Synthesis Using Macro Block",
Proc. IFIP Workshop on Control Dominated Synthesis from a Register Transfer Level Description, pp. 162-169, Grenoble, France, 3 et 4 Septembre 1992.

[Bert 93a] M.C. Bertrand, B. Babba, F. Préaud, M. Schnebelen,
"Synthesis from a VHDL description using the Xilinx Blox library",
Proc. EDAC - EUROASIC'93, pp. 117-120, Paris, France, 22-25 Fevrier 1993.

[Bert 93b] M.C. Bertrand, A. Mignotte, L. Chouraki,
"Efficient use of library blocks from a VHDL description",
Proc. IFIP Workshop on Logic and Architecture Synthesis, pp. 371-379, Grenoble, France, 6-8 Décembre 1993.

[Bert 94a] M.C. Bertrand, J. Rampon, H. Belhadj,
"A break through in VHDL synthesis on FPGAs by an efficient use of library macro blocks",
Proc. EDAC - ETC - EUROASIC 1994, pp. 183-187, Paris, France, 28 Février - 3 Mars 1994.

[Bert 94b] M.C. Bertrand, A. Mignotte, O. Khalil, G. Saucier,
"RTL synthesis system using FPGA macro block capabilities"
Proc. FPGA'94 Second International ACM/SIGDA Workshop on Field-Programmable Gate Arrays, Session 7 : Partitioning and Technology Mapping, Berkeley, CA, USA, 13-15 Février, 1994.

[Berth 92] C. Berthet, J. Rampon, L. Sponga
"Synthesis of VHDL Arrays on RAM Cells",
Proc. EURO-DAC'92 - EURO-VHDL'92, pp. 726-731, Hambourg, Allemagne, 7-10 Septembre 1992.

[Bidd 95] A. Biddle,
"High Performance Synthesis Environment Using Efficient Macro Generators",
Proc. User Forum of EDTC, EuroASIC'95, pp. 9-14, Paris, France, 6-9 Mars 1995.

[Börg 94] A. Börger, Uwe Glässer, Wolfgang Müller,
"The semantics of behavioral VHDL'93 descriptions",
Proc. European Design Automation Conference - Euro-VHDL'94, pp. 500-505,
Grenoble, France, 19-23 Septembre 1994.

[Borr 87] D. Borrione, J.L. Paillet,
"An approach to the formal verification of VHDL descriptions",
Rapport interne N°683-I, IMAG/ARTEMIS, Grenoble, Novembre 1987.

[Bouc 94] F. Bouchard, J-P. Caïsso, F. Igier, Matra-MHS,
"Development of a VITAL gate library",
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[Brac 94] T. Brackhahn,
"Gestion des informations temporelles pour les bibliothèques de cellules du logiciel ASYL+",
Rapport de stage 3ième année ENSIMAG, Grenoble, France, Juin 1994.

[Bray 89] R. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli,
"Logic Minimization Algorithms for VLSI Synthesis",
Kluwer Academic Publishers, 1989.

[Camp 89] R. Camposano, W. Rosenstiel,
"Synthesizing Circuits From Behavioral Descriptions",
IEEE Transactions on CAD, Vol. 8, No. 2, pp. 171-180, Février 1989.

[Cast 94] B. Caslis, Mentor Graphics,
"VITAL simulation with QuickVHDL",
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[Chen 90] G-D. Chen, D.D. Gajski,
"An intelligent component database for behavioral synthesis",
Proc. 27th DAC, pp. 150-155, Orlando, FL, 24-28 Juin 1990.

[Chou 95] L. Chouraki, G. Saucier,
"An efficient technique for inferring counters and accumulators from HDL specifications",
Rapport interne, Laboratoire CSI, INPG, Grenoble, France.

[COMP] "ASIC Synthesizer",
Compass, 1992.

[Cras 85] M. Crastes,
"Spécification et simulation fonctionnelles de circuits complexes : le système CADOC",
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 28 Novembre 1985.

[Cras 91] M. Crastes, A. Fonkoua,
"HDLs and logic synthesis",
European CAD Integration Project, ESPRIT, Project 2072, WP2 Deliverable, Janvier 1991.

[Crow 94] A. Crow, Veda,
"The accelerated performance of VITAL compliant simulation with the VULCAN simulator",
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[CSAM] "Bibliothèque TSBC3 Manuel de référence GDT/Genesil", Version 3.1., 1990, "Bibliothèque TSBC3 / GDT Manuel Utilisateur", 1989, Thomson - Composants Militaires et Spatiaux.

[Duff 91] C. Duff, "Codages d'automates et théorie des cubes intersectants", Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 1er Mars 1991.

[Dutt 91] N.D.Dutt, J.R.Kipps, "Bridging High-Level Synthesis to RTL Technology Libraries", Proc. 28th DAC, pp. 526-529, San Francisco, CA, 17-21 Juin 1991.

[EDIF] "EDIF - Electronic Design Interchange Format", Version 2 0 0 , Electronics Industries Association, 1987.

[Farl 88] M.C. McFarland, A.C. Parker, R. Camposano, "Tutorial on High-Level Synthesis", Proc. 25th DAC, pp. 330-336, Anaheim, CA, 12-15 Juin 1988.

[Gerb 94] L. Gerbaux, "Synthèse de contrôleurs complexes avec partitionnement", Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 12 Décembre 1994.

[Guyl 92] A. Guyler, "VHDL 1076-1992 languages changes", Proc. European Design Automation Conference - Euro-VHDL'92, pp. 672-678, Hambourg, Allemagne, 7-10 Septembre 1992.

[Guyo 94] A. Guyot, M. Belrhiti, G. Bosco, "Adders Synthesis", Proc. IFIP Workshop on Logic and Architecture Synthesis, pp. 107-117, Grenoble, France, 19-20 Décembre 1994.

[Hama 94] S. Hamacher, Mentor Graphics, "VITAL library development - Automatic translation from AutoLogic synthesis libraries", Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[HDLs 92a] "Three decades of HDLs - Part1 : CDL through TI-HDL", IEEE Design & Test of Computers, Juin 1992, pp. 69-81.

[HDLs 92b] "Three decades of HDLs - Part2 : Conlan through Verilog", IEEE Design & Test of Computers, Septembre 1992, pp. 54-63.

[IEEE 87] IEEE Std 1076-1987 "IEEE Standard VHDL Langage Reference Manual", IEEE Standards Board, 345 East 47th Street, New York, NY 10017, Mars 1988.

[IEEE 93] IEEE Std 1076-1993 "IEEE Standard VHDL Langage Reference Manual", IEEE Standards Board, 345 East 47th Street, New York, NY 10017, 1993.

[Jian 94] Y-M. Jiang, T-F. Lee, TT Hwang, Y-L. Lin, "Performance-Driven Interconnection Optimization for Microarchitecture Synthesis", IEEE Transactions on CAD, Vol. 13, No. 2, February 1994.

[Juan 94] H.P. Juan, V. Chaiyakul, D. Gajski, "Condition Graphs for High-Quality Behavioral Synthesis", Proc. ICCAD, pp. 170-174, San Jose, CA, 6-10 Novembre 1994.

[Kore 93] I. Koren,
"Computer Arithmetic Algorithms",
Publié par Prentice Hall, Englewood Cliffs, New Jersey, 07632, en 1993.

[Krol 94a] S. Krolikoski, P. Menchini,
"Designing with VHDL'93: an introduction",
Tutorial présenté à la conférence EuroDAC'94, à Grenoble, France, le 23 Septembre 1994.

[Krol 94b] S. Krolikoski, Compass,
"Automatic generation of VITAL libraries",
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[Kucu 90] K. Küçükçakar, A.C. Parker,
"Data Path Tradeoffs using MABAL",
Proc. 27th DAC, pp. 511-516, Orlando, FL, 24-28 Juin 1990.

[Land 93] B. Landwehr, P. Marwedel,
" Intelligent Library Component Selection and Management in an IP-model based High Level Synthesis System",
Proc. IFIP Workshop on Logic and Architectural Synthesis, pp.381-400, Grenoble, France, 6-8 Decembre 1993.

[Land 94] B. Landwehr, P. Marwedel, R. Dömer,
"OSCAR : optimum simultaneous scheduling, allocation and resource binding based on integer programming",
Université de Dortmund, Allemagne, Rapport n° 484, Avril 1994.

[Levi 94] O. Levia, Synopsys,
"How VITAL "guarantees" accurate sign-off simulation results",
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[Lis 88] J.S. Lis, D.D. Gajski,
"Synthesis from VHDL",
Proc. ICCD, pp. 378-381, New-York, 3-5 Octobre 1988.

[LPM] "Library of Parameterized Modules (LPM)", Version 2.0.,
EDIF, A division of Electronics Industries Association, EIA/IS-103, Mai 1993.

[Lunv 94] O.Lunven,
"Gestion des bibliothèques d'ASYL",
Stage de 3ième année ESIAL soutenue à Nancy, France, en Juin 1994.

[Magi 92] S. Maginot,
"Evaluation criteria of HDLs : VHDL compared to Verilog, UDL/I & M",
Proc. European Design Automation Conference - Euro-VHDL'92, pp. 746-751, Hambourg, Allemagne, 7-10 Septembre 1992.

[Marw 86] P. Marwedel,
"A New Synthesis Algorithm for the MIMOLA Software System",
Proc. 23rd DAC, pp. 271-277, Las Vegas, NE, 29 Juin - 2 Juillet 1986.

[Marw 90] P. Marwedel,
"Matching System and Component Behaviour in MIMOLA Synthesis Tools",
Proc. EDAC, pp. 146-156, Glasgow, Scotland, 12-15 Mars 1990.

[Marw 95] P. Marwedel, B. Landwehr,
"RAM-based Architectural Synthesis",
Publié dans le livre intitulé "Logic and Architecture Synthesis, State-of-the-art and novel approaches", édité par G. Saucier et A. Mignotte et publié par Chapman&Hall en 1995, pp. 233-244.

[Ment 89] "M Language Users Guide & Reference", Version 4.0,
Mentor Graphics, Mai 1989.

[Mign 92] A. Mignotte,
"Synthèse architecturale de circuits intégrés",
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 26 Novembre 1992.

[Mign 93] A. Mignotte, M.C. Bertrand, M. Crastes, J. Rampon, G. Saucier,
"ASYL: A Control Driven RTL Synthesis System using Library Blocks",
Publié dans le livre intitulé "Synthesis for control dominated circuits" par "Elsevier science publishers" en 1993, pp. 275-291.

[Mull 89] J.M. Muller, chargé de recherche au CNRS,
"Arithmétique des ordinateurs - Opérateurs et fonctions élémentaires",
Publié par la collection Masson, Paris, en 1989.

[Naya 94] S. Nayak, Cadence,
"Translating existing libraries to VITAL",
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[OVI 91] "Verilog Hardware Description Langage Reference Manual", Version 1.0,
Open Verilog International, October 1991.

[Patt 95] A. Patterson,
"An Outside Perspective on European EDA",
Présenté à la conférence "Electronic Design Automation - the European Dimension", Siemens, Munich, 16 Octobre 1995.

[Paul 88] P.G. Paulin,
"High-Level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms",
Ph.D thesis, faculty of engineering, Carleton University, Canada, January 1988.

[Reve 95] L. Revéret,
"Intégration de macro blocs en synthèse RTL",
Rapport de stage de DEA de Micro-électronique soutenue à Grenoble, France, en Septembre 1995.

[Rizz 93] H. Rizzo, A. Mille,
"Synthèse VHDL",
Stage de 3ième année ENSIMAG soutenue à Grenoble, France, en Juin 1993.

[Safi 95] C. Safinia,
"Optimisations et analyse de performances en synthèse RTL orientée par le contrôle",
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 1er Février 1995.

[Sako 93] K. Sakouti,
"Synthèse et optimisation temporelle de réseaux Booléens",
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 26 Novembre 1993.

[Sale 92] A. Salem,
"Vérification formelle des circuits digitaux décrits en VHDL",
Thèse de doctorat de l'Université Joseph Fourier, Grenoble I, France, soutenue le 2 Octobre 1992.

[SDF] "Standard Delay Format Specification", Version 2.0.,
Open Verilog International, Juin 1993.

[Sica 88] P. Sicard,
"Nouvelles méthodes de synthèse logique",
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 2 Septembre 1988.

[Sina 94] P. Sinander, ESA/ESTEC WD, European Space Agency,
"Some experiences of using VITAL timing check routines for board-level modeling",
Présenté au séminaire "VITAL - The Standard Modeling Methodology for ASIC Libraries in VHDL", à Grenoble, France, le 26 Septembre 1994.

[Soue 89] M. Soueidan,
"Conception d'un microprocesseur reconfigurable",
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, France, soutenue le 14 Avril 1989.

[SYNO] "Design Compiler Reference Manual",
Synopsys, 1992.

[Thee 95] J.F.M. Theeuwen,
"Module Generators and their Integration in an Architectural Synthesis System",
Publié dans le livre intitulé "Logic and Architecture Synthesis, State-of-the-art and novel approaches", édité par G. Saucier et A. Mignotte et publié par Chapman&Hall en 1995, pp. 245-251.

[Touz 93] O.Touzet, J-C. Da Cunha,
"Modélisation et Conception de Systèmes Electroniques Complexes",
Stage de 3ième année ENSIMAG soutenue à Grenoble, France, en Juin 1993.

[Troi 95] P. Troin,
"Inférence de macro blocs complexes",
Rapport de stage de DEA Informatique soutenue à Grenoble, France, en Septembre 1995.

[VITAL] "Model Development Specification", Version V2.2.b,
VHDL Initiative Toward ASIC Libraries, 25 Mars 1994.

[VITAL 93] "VITAL : VHDL Initiative Toward ASIC Libraries Model Development Specification", Version 1.0, 5 Février 1993.

[VITAL 95] "Standard VITAL ASIC Modeling Specification",
Draft, IEEE 1076.4, Juillet 1995.

[VLSI] "VDP300 Library", "VSC370 Library"
VLSI Technology Inc., V8R3, Novembre 1991.

[Wehn 91] N. Wehn, J. Biesenack, M. Pilsl,
"A new approach to multiplexor minimization in the CALLAS synthesis environment",
VLSI 91, Proc. IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration, pp. 203- 213, Edinburgh, Scotland, 20-22 August, 1991.

[XBLOX] "X-BLOX User Guide",
Xilinx XACT, Avril 1994.

[Xilinx] "XACT Libraries Guide",
Xilinx XACT, Avril 1994.

[XNF] "Xilinx XNF Netlist Specification", Version 5.00.,
Xilinx Inc., 9 Juillet 1993.

Annexe I

-- ◊ - ◊◊◊ - ◊ --

Format de description en bibliothèque des éléments utilisés par la synthèse de bas niveau

Annexe I :
Format de description en bibliothèque des éléments utilisés par la synthèse de bas niveau

I.1. Grammaire du format

-------------------------- ASYL / library definition ---------------------
A library is described as a set of gates and must be described using the following format (MIS-like format) :
<StdCell Library> ::=
[<Default Section>]
{<GateDefinition>}+
where
<Default Section> ::= DEFAULT Max_Fanout <Floating point value>
The default section is used to indicate a default value to the
<MaxFanout> parameters of the gates. Default is 999.00
<GateDefinition> ::= <CombinationalGate Definition> |
<PredefinedGate Definition> |
<UnknownGate Definition>

------------------1. Combinational Gate Definition -------------------------
<CombinationalGate Definition> ::=
GATE <Don'tTouch> <Don'tUse> <GateName> <GateArea> [<MaxFanout>]
Z = <Expression>
{PIN <PinName> <PinPhase> <InputCapacitance> <InputLoad>
<RiseDelay> <RiseFanoutDelay> <FallDelay> <FallFanoutDelay>}+
<Don'tTouch> is either 0 or 1 and is used to specify gates that must not be modified during synthesis (if 1).
<Don'tUse> is either 0 or 1 and is used to specify gates that must not be used during synthesis (if 1).
<GateName> is the name of the gate in the library, the resulting netlist will be in terms of these names.
<GateName> may be enclosed with "".
<Expression> is an equation written using classical conventions (* for AND, + for OR, ! for negation and parentheses for grouping). The names used in <Expression> define the outputs of the cells. These outputs must be defined in the PIN section (one PIN line per Input).
<Expression> must be factorized and "minimal". For example, the correct expression for an OAI21 gate is !(A.(B+C)) and not !(A.B + A.C).
Variables used in <Expression> must appear only once, except for the Mux21, Xor2 and XNor2 gates. It means that the gates whose expressions are not trees are not recognized (except Mux21, Xor2 and XNor2).
<PinPhase> is either UNKNOWN, INV or NOTINV indicates the phase of the
input.
<InputCapacitance>, <InputLoad>, <RiseDelay>, <RiseFanoutDelay>, <FallDelay>, <FallFanoutDelay> are floating point numbers, in any unit convenient for
the user. When these data are the same for all the Pins, the * convention
may be used for the <PinName>.
<MaxFanout> is a floating point number, this value is optional.

```
--------------------Examples ---------------------------------------
GATE 0 0          "AOI1D"  1.00   16.0    Y = !(! A*! B+! C);
 PIN      C UNKNOWN  1.60 1.00    7.30 0.50 7.30 0.50
 PIN      B UNKNOWN  1.60 1.00    7.30 0.50 7.30 0.50
 PIN      A UNKNOWN  1.60 1.00    7.30 0.50 7.30 0.50
GATE 0 0          "AOI1C"  1.00   16.0    Y = !(! A*! B+ C);
 PIN      * UNKNOWN  1.60 1.00    7.30 0.50 7.30 0.50
--------------------Remarks ----------------------------------------
```

The following Gates MUST EXIST in your library :

```
 GATE       <Const0 Name>   0    Z=CONST0;       => constant 0
 GATE       <Const1 Name>   0    Z=CONST1;       => constant 1
 GATE       <Inverter Name> ...  Z = !A; ...     => Inverter
```

plus one out of the following gates : NOR2, NAND2, OR2, AND2.
Beware : the system does not check for gate name duplication.

-------------------2. Predefined Gate Definition -----------------------------
This section allows to define non combinational gates such as FlipFlops, latches, Buffers, ...
<PredefinedGate definition> ::=
PGATE <Don'tTouch> <Don'tUse> <ASYLGateName>
<GateDefinition>
{<OptionalTiming>}*
<Don'tTouch> is either 0 or 1 and is used to specify gates that must not be modified during synthesis (if 1).
<Don'tUse> is either 0 or 1 and is used to specify gates that must not be used during synthesis (if 1).
<ASYLGateName> is the absolute Name of the gate in the ASYL system. This
name will be used as reference to migrate designs from one library to another. For example, JKG will indicate a JK gate whatever the library.
The following <ASYLGateName> are defined :
TG : T FlipFlop
DG : D-type FlipFlop
LG : Latch
JKG : JK FlipFlop
RSG : RS FlipFlop
DMUXG : D FlipFlop with muxed Input LMUXG : Latch with muxed Input THG : non-inverting 3-state gate ITHG : inverting 3-state gate
BUF : Buffers
<GateDefinition> contains the formal names of the gate connectors. Order is relevant as it allows to determine for example what is the Reset connector of a FlipFlop.
<OptionalTiming> contains the timing information associated with each output pin.
<GateDefinition> is defined according to <ASYLGateName> :
For PGATE TG : <GateDefinition> :: =
<LibGateName> <Area> <Clock> <CType> <Reset> <Set> <ClockEnable> <3StateCtrl> <InputT> <Output> <CompOutput> <HiImpOutput>
For PGATE DG : <GateDefinition> :: =
<LibGateName> <Area> <Clock> <CType> <Reset> <Set> <ClockEnable> <3StateCtrl> <Input> <Output> <CompOutput> <HiImpOutput>
For PGATE LG : <GateDefinition> :: =
<LibGateName> <Area> <Clock> <CType> <Reset> <Set> <ClockEnable> <3StateCtrl> <Input> <Output> <CompOutput> <HiImpOutput>
For PGATE JKG : <GateDefinition> :: =
<LibGateName> <Area> <Clock> <CType> <Reset> <Set>
<ClockEnable> <3StateCtrl> <InputJ> <InputK> <Output> <CompOutput> <HiImpOutput>
For PGATE RSG : <GateDefinition> :: =
<LibGateName> <Area> <Clock> <CType> <Reset> <Set>
<ClockEnable> <3StateCtrl> <inputR> <inputS> <Output> <CompOutput> <HiImpOutput>
For PGATE DMUXG : <GateDefinition> :: =
<LibGateName> <Area> <Clock> <CType> <Reset> <Set>
<ClockEnable> <3StateCtrl> <InputSel> <InputA> <InputB> <Output> <CompOutput> <HiImpOutput>
The <InputA> is selected if <InputSel> is 0
The <InputB> is selected if <InputSel> is 1
For PGATE LMUXG : <GateDefinition> :: =
<LibGateName> <Area> <Clock> <CType> <Reset> <Set>
<ClockEnable> <3StateCtrl> <InputSel> <InputA> <InputB> <Output> <CompOutput> <HiImpOutput>
The <InputA> is selected if <InputSel> is 0
The <InputB> is selected if <InputSel> is 1
For PGATE THG : <GateDefinition> :: =
<LibGateName> <Area> <Input> <3StateCtrl> <Output>
For PGATE ITHG : <GateDefinition> :: =
<LibGateName> <Area> <Input> <3StateCtrl> <Output>
For PGATE BUF : <GateDefinition> :: =
<BufType> <LibGateName> <area> <BufInput> <BufOutput> <BufPad> <Buf3StateCtrl>
<optionaltiming>
where
<LibGateName> ::= name of the gate in the library.
For Example "JKRS" may be the name of a JK with Reset and Set.

<Clock> ::= Clock Name
<CType> ::= type of the clock or of the G signal for FlipFlops and Latches
<Input> ::= Input Name
<InputJ> ::= J input of a JK gate
<InputK> ::= K input of a JK gate
<InputSel> ::= Input Selector (DMUXG or LMUXG muxed input gates)
<InputA> ::= A input connector (DMUXG or LMUXG muxed input gates)
<InputB> ::= B input connector (DMUXG or LMUXG muxed input gates)
<Reset> ::= Reset connector
<Set> ::= Set connector
<ClockEnable> ::= Clock Enable
<3StateCtrl> ::= 3-State control (See note 1)
<InputT> ::= Input of a TG Gate
<Output> ::= Gate Output (direct output)
<CompOutput> ::= Complemented output (if available)
<HiImpOutput> ::= Hi Impedance Output (if available)
<BufType> ::= Type of the IO Buffer
(In, out, bidirectional, clock, 3-State) <BufInput> ::= OUT or BID Buffer input
<BufOutput> ::= IN or BID Buffer output
<BufPad> ::= IN, OUT or BID Buffer PAD
<Buf3StateCtrl> ::= BID Buffer Tri State (See note 1)
<LibGateName>, <Clock>, <Input>, <InputJ>, <InputK>, <InputSel>, <InputA>, <InputB>
::= <String>
<CType> ::= RECK | FECK | L0 | L1
<BufType> ::= I_BUF | O_BUF | BI_BUF | CLK_BUF | TS_BUF
<Reset>, <Set>, <ClockEnable>, <3StateCtrl>, <InputJ>, <InputK> ::= {"!"} <String> | "*"
A "*" is used if the connector is not available
A "!" indicates that the signal is active low:
if A=0, the pin has an effect on the gate
<InputT>, <Output>, <CompOutput>, <HiImpOutput>,
<BufInput>, <BufOutput>, <BufPad>, <Buf3StateCtrl> ::= <String> | "*"
<DT>, <DU> ::= 0 | 1
<Area> ::= <real>
<OptionalTiming> is defined by :
RELATED_OUTPUT_PIN <OutputName> <MaxFanout>
{PIN <PinName> <PinPhase> <InputCapacitance> <InputLoad> <RiseDelay> <RiseFanoutDelay>
<FallDelay> <FallFanoutDelay> <SetupTime> <HoldTime>}+
The * convention is supported for the PIN section associated with a given RELATED_OUTPUT_PIN.
Note 1 (control signal polarities):
All control signals (3-State, Set, Reset, ClockEnable, InputJ, InputK) are active high unless if preceded by an ! mark.
--------------------Examples ---------------------------------------
D FlipFlop Gate :

```
PGATE 0 0       DG  DF1  1.00          CLK RECK * * * * D Q * *
      RELATED_OUTPUT_PIN Q  16
                PIN  CLK UNKNOWN 1.60 1.00 7.30 0.50 7.30 0.50 0.00 0.00
                PIN    D UNKNOWN 1.60 1.00 8.10 0.00 8.10 0.00 0.00 0.00
```

JK FLIP FLOPS :

```
#
PGATE 0 0       JKG     jkbtnb  7.67       cp RECK sdn cdn * * j !k q qn *
      RELATED_OUTPUT_PIN q   3
                PIN  cp UNKNOWN 0.06 0.06 2.22 2.77 2.41 2.41 0.00 0.00
                ...
      RELATED_OUTPUT_PIN qn  3
                PIN  cp UNKNOWN    0.06    0.06    3.31    3.03    3.16    2.41
 0.00    0.00
                ...
```

```
LATCHES :
PGATE 0 0    LG    labfnb 5.00          en L0 !cdn !sdn * * d q qn *
        RELATED_OUTPUT_PIN q  3
                PIN  en UNKNOWN    0.06    0.06    2.14    2.91    2.58    2.51
  0.00    0.00
                ...
        RELATED_OUTPUT_PIN qn  3
                PIN  en UNKNOWN    0.06    0.06    2.59    2.89    2.55    2.51
  0.00    0.00
                ...
T FLIP FLOPS :
PGATE 0 0    TG    TF1B 1.00          CLK FECK !CLR * * * T Q * *
        RELATED_OUTPUT_PIN Q  16
                PIN  CLR UNKNOWN  1.60    1.00    7.30    0.50    7.30    0.50
  0.00    0.00
                ...
3-STATE GATES :
# z output will be Z if oe is 0 (low)
# z will be tight to i if oe is 1 (high)
PGATE 1 1 THG  nt01d1  1.67  i !oe  z
        RELATED_OUTPUT_PIN z  3.30
                PIN   i UNKNOWN    0.05    0.05    1.02    3.03    0.96    2.21
  0.00    0.00
                PIN   oe UNKNOWN   0.09    0.09    0.75    3.03    0.69    2.21
  0.00    0.00
# z output will be Z if oen is 1 (high)
# z will be tight to not i if oen is 0 (low)
PGATE 1 1 ITHG it02d5   3.67  i oen zn
        RELATED_OUTPUT_PIN zn 16.13
                PIN   i UNKNOWN    0.10    0.10    1.55    0.61    1.54    0.49
  0.00    0.00
                PIN  oen UNKNOWN   0.11    0.11    1.66    0.62    1.65    0.56
  0.00    0.00
BUFFERS :
# Non inverting 3-State buffer, PAD is Z when E is 0
PGATE 1 1    BUF    TS_BUF  TRIBUFF    0.00   D    *    PAD    !E
    RELATED_OUTPUT_PIN PAD  16
 PIN    E UNKNOWN   1.60 1.00 15.00 0.37 13.08 0.26
 PIN    D UNKNOWN   1.60 1.00 6.00 0.12 8.40 0.17
```

```
----------------------3. Unknown Gate Definition ------------------------
<UnknownGate Definition> ::= UGATE <Don'tTouch> <Don'tUse>
                 <LibGateName> <Area>
                 <NbIn> {<InPinName>}*(NbIn)
                 <NbOut> {<OutPinName>}*(NbOut)
                 {<OptionalTiming>}*
```

Where :
<Don'tTouch>, <Don'tUse>, <LibGateName>, <Area> and <OptionalTiming> are defined as for Predefined Gates.
<InPinName>, <OutPinName> are Strings.
<NbIn>, <NbOut> are positive integers giving respectively the number of inputs and of outputs of the UGATE.
Defining an unknown gate in a library allows to migrate a design from one library to another or to optimize a given design on a same library, even if the knowledge on some gates is not available.
For example, assume a design with a gate UK1, the functionality of which is unknown. Defining UK1 as an Unknown gate will force the synthesis system to reconnect this gate in a second design resulting from the initial one.
If you forget to declare the unknown gates, the mapping step will usually lead to an error : "Unable to map <Unknown Gate Name>"

```
----------------Examples ----------------------------------------
UGATE 1 1 TA10     1.00
 3 C B A
 1 Y
      RELATED_OUTPUT_PIN Y  16
           PIN  C UNKNOWN      1.60    1.00    8.10    0.00    8.10    0.00    0.00
 0.00
           PIN  B UNKNOWN      1.60    1.00    8.10    0.00    8.10    0.00    0.00
 0.00
           PIN  A UNKNOWN      1.60    1.00    8.10    0.00    8.10    0.00    0.00
 0.00

-----------------------------------------------------------------
```

I.2. Exemples

```
# LIBCOMP, (CSI-INPG) V 1.00 - ASYLPLUS (IST) V2.5.0
# date : Tue Jul 28 12:27:48 1993
# Input capacitances set to 1.00
# Input loads set to 1.00
# Unit delays are assumed
# -----------------------------------------------------------------
# 2-INPUT AND GATE
GATE 0 0      and2 1.5 50.00      z = ( a1* a2);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 3-INPUT AND GATE
GATE 0 0      and3 2.00     50.00      z = (a1* a2* a3);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 4-INPUT AND GATE
GATE 0 0      and4 2.5 50.00      z = (a1* a2* a3* a4);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 6-INPUT AND GATE
GATE 0 0      and6 3.5 50.00      z = (a1* a2* a3* a4* a5* a6);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 8-INPUT AND GATE
GATE 0 0      and8 4.00     50.00      z = (a1* a2* a3* a4* a5* a6* a7* a8);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 12-INPUT AND GATE
GATE 0 0      and12 7.00     50.00      z = (a1*a2*a3*a4*a5*a6*a7*a8*a9*a10*a11*a12);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 2-INPUT NAND GATE
GATE 0 0      nand2 1.00     50.00      z = !(a1* a2);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 3-INPUT NAND GATE
GATE 0 0      nand3 1.50     50.00      z = !(a1* a2* a3);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 4-INPUT NAND GATE
GATE 0 0      nand4 2.00     50.00      z = !(a1* a2* a3* a4);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 6-INPUT NAND GATE
GATE 0 0      nand6 3.00     50.00      z = !(a1* a2* a3* a4* a5* a6);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 8-INPUT NAND GATE
GATE 0 0      nand8 4.00     50.00      z = !(a1* a2* a3* a4* a5* a6* a7* a8);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 2-INPUT OR GATE
GATE 0 0      or2 1.50 50.00      z = (a1+ a2);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 3-INPUT OR GATE
GATE 0 0      or3 2.00 50.00      z = (a1+ a2+ a3);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 4-INPUT OR GATE
GATE 0 0      or4 2.50 50.00      z = (a1+ a2+ a3+ a4);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 6-INPUT OR GATE
GATE 0 0      or6 3.50 50.00      z = (a1+ a2+ a3+ a4+ a5+ a6);
 PIN     * NONINV 1.00 1.00 1.00 0.00 1.00 0.00
# 8-INPUT OR GATE
GATE 0 0      or8 4.00 50.00      z = (a1+ a2+ a3+ a4+ a5+ a6+ a7+ a8);
```

```
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# 12-INPUT OR GATE
GATE  0 0      or12  7.00      50.00      z =  (a1+a2+a3+a4+a5+a6+a7+a8+a9+a10+a11+a12);
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# 2-INPUT NOR GATE
GATE  0 0      nor2  1.00      50.00      z =  !(a1+ a2);
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# 3-INPUT NOR GATE
GATE  0 0      nor3  1.50      50.00      z =  !(a1+ a2+ a3);
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# 4-INPUT NOR GATE
GATE  0 0      nor4  2.00      50.00      z =  !(a1+ a2+ a3+ a4);
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# 6-INPUT NOR GATE
GATE  0 0      nor6  3.00      50.00      z =  !(a1+ a2+ a3+ a4+ a5+ a6);
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# 8-INPUT NOR GATE
GATE  0 0      nor8  4.00      50.00      z =  !(a1+ a2+ a3+ a4+ a5+ a6+ a7+ a8);
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# INVERTER (1X)
GATE  0 0      not  0.50  50.00      z =  !a1;
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
PGATE  1 1     BUF    BI_BUF bibuf   0.00    in      y      pad    !oe
 RELATED_OUTPUT_PIN pad  16
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00
PGATE  1 1     BUF    TS_BUF tri     0.00    in      *      out    !oe
      RELATED_OUTPUT_PIN out  16
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00
# NON-INVERTING BUFFER (1X)
GATE  0 0      inbuf  0.50  50.00    z =  a1;
 PIN     * NONINV  1.00 1.00 1.00 0.00 1.00 0.00
# 2-INPUT EXCLUSIVE NOR
GATE  0 0      xnor2  1.00           z =  !(( a1+ a2)*!( a1* a2));
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00
# 2-INPUT EXCLUSIVE OR
GATE  0 0      xor2  1.00      50.00      z =  ( ( a1+ a2)*!( a1* a2));
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00
# D FLIP-FLOP WITH PRESET AND CLEAR
PGATE  0 0     DG  dff  5.00        cp RECK cdn sdn * * d q qn *
  RELATED_OUTPUT_PIN q 50.00
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
  RELATED_OUTPUT_PIN qn 50.00
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# T FLIP-FLOP WITH PRESET AND CLEAR
PGATE  0 0     TG  tff  5.00        CLK RECK CLRN PRN * * T Q * *
  RELATED_OUTPUT_PIN Q 50.00
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# D FLIP-FLOP WITH CLEAR
PGATE  0 0     DG     dffc  5.00        cp RECK cdn * * * d q qn *
 RELATED_OUTPUT_PIN q 50.00
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
 RELATED_OUTPUT_PIN qn 50.00
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
PGATE  0 0     DG  dffk  4.50        cp RECK * * * * d q qn *
 RELATED_OUTPUT_PIN q 50.00
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
 RELATED_OUTPUT_PIN qn 50.00
       PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# D FLIP-FLOP WITH PRESET
```

```
PGATE 0 0    DG  dffp  5.00           cp RECK * sdn * * d q qn *
  RELATED_OUTPUT_PIN q 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
  RELATED_OUTPUT_PIN qn 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# JK FLIP-FLOP WITH PRESET AND CLEAR
PGATE 0 0    JKG       jkf  6.00      cp RECK sdn cdn * * j k q qn *
  RELATED_OUTPUT_PIN q 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
  RELATED_OUTPUT_PIN qn 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# JK FLIP-FLOP WITH CLEAR
PGATE 0 0    JKG       jkfc  6.00          cp RECK cdn * * * j k q qn *
  RELATED_OUTPUT_PIN q 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
  RELATED_OUTPUT_PIN qn 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# JK FLIP-FLOP
PGATE 0 0    JKG       jkfk  5.50          cp RECK * * * * j k q qn *
  RELATED_OUTPUT_PIN q 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
  RELATED_OUTPUT_PIN qn 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
PGATE 0 0    LG  latg  4.00          en L0 cdn sdn * * d q * *
  RELATED_OUTPUT_PIN q 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# LATCH
PGATE 0 0    LG  lat  2.50       e L1 * * * * d q * *
  RELATED_OUTPUT_PIN q 50.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00
# lcell
UGATE 0 0     lcellu  0.00
 1 a1
 1 z
 RELATED_OUTPUT_PIN z 1.00
      PIN     * UNKNOWN 1.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00

GATE 0 0     lcell 90.00 90.00   z =  a1;
 PIN     * NONINV  10.00 10.00 10.00 0.00 10.00 0.00
# ---------------------------------------------------------------
GATE 0 0     VCC    0    z=CONST1;
GATE 0 0     GND    0    z=CONST0;
# EOF libraries
```

Annexe II

-- ◊ - ◊◊◊ - ◊ --

Format de description en bibliothèque des éléments utilisés par la synthèse comportementale

II.1. Grammaire du format

Metasyntaxe:

```
<...>     : Terminal name
->        : Is derived in
|         : Alternatives
{...}     : Parenthesage
[...]     : Optionnal element
... *     : 0, 1 or several occurence(s)
... +     : 1 or several occurence(s)
```

Grammar:

```
<LibraryDesc>     -> ( Library <LibName> <ResList> )
<LibName>         -> <Idf>
<ResList>         -> ( ResourceList <ResDesc>+ )
<ResDesc>         -> <FixDesc> | <GenDesc>
<FixDesc>         -> ( Resource <ResName> <ResType> <ResArea> <ResHeigh> <ResWidth>
                  <FixPortList> <OpList> )
<GenDesc>         -> ( Generic <ResName> <ResType> <ResArea> <ResHeight> <ResWidth>
                  <ParamList> <GenPortList> <OpList> )
<ResName>         -> <Idf>
<ResType>         -> ( ResType <RType> )
<RType>           -> MEM | MEM_D | LATCH | BUFFER | OPE | MUX | TRISTATE |
                  CONNECTOR | LIB_CONST | VSS | VDD | BUS
<ResArea>         -> ( Area <ResOrOpAttribute> )
<ResHeight>       -> ( Height <ResOrOpAttribute> )
<ResWidth>        -> ( Width <ResOrOpAtribute> )
<ParamList>       -> ( ParamList { <IntParamDesc> | <StrParamDesc> }+ )
<IntParamDesc>    -> ( ParamDesc <ParamName> ( ParamType INT) <IntParValues> )
<StrParamDesc>    -> ( ParamDesc <ParamName> ( ParamType STR) <StrParValues> )
<ParamName>       -> <Idf>
<IntParValues>    -> ( ParamValues { <Number> | { <Number> Inter <Number> } }+ )
<StrParValues>    -> ( ParamValues <Idf>+ )
<FixPortList>     -> ( PortList <FixPortDesc>+ )
<GenPortList>     -> ( PortList <GenPortDesc>+ )
<FixPortDesc>     -> ( PortDesc <PortName> <PortType> <FixPortWidth> )
<GenPortDesc>     -> ( PortDesc <PortName> <PortType> <GenPortWidth> )
<PortName>        -> <Idf>
<PortType>        -  > ( PortType <PType> )
<PType>           -> INPUT | OUTPUT | INOUT | COMMAND | CLOCK | RESET_SYNC |
                  RESET_ASYNC | SET_SYNC | SET_ASYNC | CONTROL_SYNC |
                  CONTROL_ASYNC | ATTRIBUTE
<FixPortWidth>    -> ( PortWidth <Number> )
<GenPortWidth>    -> ( PortWidth { <Number> | <ParamUtil> } )
<OpList>          -> ( OpList <OpDesc>+ )
<OpDesc>          -> ( OpDesc <OpName> <OpType> <OpDelay> <OpPipeline> <OpLatency>
                  [ <OpPortComList> ] [ <OpPortDatList> ] )
<OpName>          -> <Idf>
<OpType>          -> ( Optype <OType> )
<OType>           -> ADD | ADDC2 | SUB ... |
                  LOAD | CLEAR | ... |
                  CONNEC_IN | ... |
```

```
                        Z | ... |
                        OTHER |
                        <PortName>
<OpDelay>                  -> ( Delay <ResOrOpAttribute> )
<OpPipeline>    -> ( Pipeline <Boolean> )
<OpLatency>     -> ( Latency <ResOrOpAttribute> )
<OpPortComList>    -> ( OpPortCommandList <PortComDesc>+ )
<PortComDesc>      -> ( PortCommandDesc <PortName> <PortDefValue> )
<PortDefValue>     -> ( PortDefaultValue <BinNumber> )
<OpPortDatList>    -> ( OpPortDataList <PortDatDesc>+ )
<PortDatDesc>  -> ( PortDataDesc <PortName> [ <PortDefValue> ] [ <CommutList> ] )
<CommutList> -> ( PortCommutList <PortName>+ )
<ResOrOpAttribute> -> < FunctionCall > | <Expression>
<FunctionCall>     -> ( FunctionCall  " < Path > " )
<Path>          -> <Idf>  [ / <Path>  ]
<Expression>    -> <Constant>
            -> ( <Operator>  <Expression>  <Expression>+ )
            -> ( <Operator>  <ParamUtil>  <Expression>+ )
            -> ( <Operator>  <Expression>+  <ParamUtil> )
            -> ( <Operator>  < ParamUtil>+ )
<ParamUtil>            -> ( ParamUsed <Idf> )
<Operator>             -> + | - | * | /
<Constant>             -> <Number>  | <Float>
<Float>         -> <Number> { e [ - | + ] <Number> }
            ->  [ <Number> ] . <Number>  [ e [ - | + ] <Number> ]
<Idf>           -> <Letter> { <Letter> | <Digit> }*
<Number>               -> <Digit>+
<Letter>        -> A | ... | Z | a | ... | z | _
<Digit>         -> 0 | ... | 9
<BinNumber>     -> { 0 | 1 | - }+
<Boolean>              -> 0 | 1
```

II.2. Exemples

```
(Library XBLOX
 (ResourceList

#===========================================================================  #
# This generator ands all the bits of the given generic input                 #
#===========================================================================  #

  (Generic ANDBUS
   (ResType OPE)
   (Area 0) (Height 0) (Width 0)
   (ParamList
    (ParamDesc N (ParamType INT) (ParamValues 1 Inter 64)
                 (ParamDefaultValue 16)))
   (PortList
    (PortDesc A  (PortType INPUT)  (PortWidth (ParamUsed N)))
    (PortDesc O  (PortType OUTPUT) (PortWidth 1)))
   (OpList
    (OpDesc ANDBUS (OpType OTHER)
     (Delay 0) (Pipeline 0) (Latency 0)
     (OpPortDataList
      (PortDataDesc A)
      (PortDataDesc O)))
   )
  )

#===========================================================================  #
# 2-to-1 bus Multiplexeur                                                     #
#===========================================================================  #

  (Generic MUXBUS2 (ResType OPE)
   (Area 0) (Height 0) (Width 0)
   (ParamList
    (ParamDesc N (ParamType INT) (ParamValues 1 Inter 64)
                 (ParamDefaultValue 16))
    (ParamDesc M (ParamType INT) (ParamValues 1 Inter 64)
                 (ParamDefaultValue 1)))
   (PortList
    (PortDesc M1         (PortType INPUT)   (PortWidth (ParamUsed N)))
    (PortDesc M0         (PortType INPUT)   (PortWidth (ParamUsed N)))
    (PortDesc SEL        (PortType INPUT)   (PortWidth (ParamUsed M)))
    (PortDesc MUX_OUT    (PortType OUTPUT)  (PortWidth (ParamUsed N)))
    (PortDesc SEL_ERROR  (PortType OUTPUT)  (PortWidth 1)))
   (OpList
    (OpDesc MUXBUS2 (OpType OTHER)
     (Delay 0) (Pipeline 0) (Latency 0)
     (OpPortDataList
      (PortDataDesc M1)
      (PortDataDesc M0)
      (PortDataDesc SEL)
      (PortDataDesc MUX_OUT)
      (PortDataDesc SEL_ERROR)))
   )
  )
```

```
#===================================================================#
# Universal Accumulator.                                            #
#===================================================================#

(Generic ACCUM (ResType MEM)
 (Area 0) (Height 0) (Width 0)
 (ParamList
  (ParamDesc N (ParamType INT) (ParamValues 1 Inter 64)
               (ParamDefaultValue 16)))
 (PortList
  (PortDesc B              (PortType INPUT)         (PortWidth (ParamUsed N)))
  (PortDesc C_IN           (PortType INPUT)         (PortWidth 1))
  (PortDesc ADD_SUB        (PortType COMMAND)       (PortWidth 1))
  (PortDesc LOAD           (PortType COMMAND)       (PortWidth 1))
  (PortDesc CLK_EN         (PortType COMMAND)       (PortWidth 1))
  (PortDesc CLOCK          (PortType CLOCK)         (PortWidth 1))
  (PortDesc ASYNC_CTRL     (PortType RESET_ASYNC)   (PortWidth 1))
  (PortDesc SYNC_CTRL      (PortType RESET_SYNC)    (PortWidth 1))
  (PortDesc Q_OUT          (PortType OUTPUT)        (PortWidth (ParamUsed N)))
  (PortDesc C_OUT          (PortType OUTPUT)        (PortWidth 1))
  (PortDesc OVFL           (PortType OUTPUT)        (PortWidth 1)))
 (OpList
  # Q_OUT <= ASYNC_VAL #
  (OpDesc RESET_ASYN (OpType CLEAR_ASYNC)
   (Delay 0) (Pipeline 0) (Latency 0)
   (OpPortCommandList
    (PortCommandDesc SYNC_CTRL  (PortDefaultValue -))
    (PortCommandDesc ASYNC_CTRL (PortDefaultValue 1)))
   (OpPortDataList
    (PortdataDesc Q_OUT)))
  # Q_OUT (hold) #
  (OpDesc KEEP (OpType KEEP_VALUE)
   (Delay 0) (Pipeline 0) (Latency 0)
   (OpPortCommandList
    (PortCommandDesc LOAD       (PortDefaultValue -))
    (PortCommandDesc SYNC_CTRL  (PortDefaultValue -))
    (PortCommandDesc CLK_EN     (PortDefaultValue 0))
    (PortCommandDesc CLOCK      (PortDefaultValue -))
    (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0))
    (PortCommandDesc ADD_SUB    (PortDefaultValue -)))
   (OpPortDataList
    (PortdataDesc Q_OUT)))
  # Q_OUT <= SYNC_VAL #
  (OpDesc RESET_SYN (OpType CLEAR_SYNC)
   (Delay 0) (Pipeline 0) (Latency 0)
   (OpPortCommandList
    (PortCommandDesc SYNC_CTRL  (PortDefaultValue 1))
    (PortCommandDesc CLK_EN     (PortDefaultValue 1))
    (PortCommandDesc CLOCK      (PortDefaultValue 1))
    (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0))
    (PortCommandDesc ADD_SUB    (PortDefaultValue -)))
   (OpPortDataList
    (PortdataDesc Q_OUT)))
  # Q_OUT <= B #
  (OpDesc LOAD (OpType LOAD)
   (Delay 0) (Pipeline 0) (Latency 0)
   (OpPortCommandList
```

```
          (PortCommandDesc LOAD       (PortDefaultValue 1))
          (PortCommandDesc SYNC_CTRL  (PortDefaultValue 0))
          (PortCommandDesc CLK_EN     (PortDefaultValue 1))
          (PortCommandDesc CLOCK      (PortDefaultValue 1))
          (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0))
          (PortCommandDesc ADD_SUB    (PortDefaultValue -)))
         (OpPortDataList
          (PortdataDesc B)
          (PortdataDesc Q_OUT)))
       # Q_OUT <= Q_OUT + B #
       (OpDesc ADD (OpType ADD)
        (Delay 0) (Pipeline 0) (Latency 0)
         (OpPortCommandList
          (PortCommandDesc LOAD       (PortDefaultValue 0))
          (PortCommandDesc SYNC_CTRL  (PortDefaultValue 0))
          (PortCommandDesc CLK_EN     (PortDefaultValue 1))
          (PortCommandDesc CLOCK      (PortDefaultValue 1))
          (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0))
          (PortCommandDesc ADD_SUB    (PortDefaultValue 1)))
         (OpPortDataList
          (PortdataDesc B)
          (PortdataDesc C_IN (PortDefaultValue 0))
          (PortdataDesc C_OUT)
          (PortdataDesc OVFL)
          (PortdataDesc Q_OUT)))
       # Q_OUT <= Q_OUT - B #
       (OpDesc SUB (OpType SUB)
        (Delay 0) (Pipeline 0) (Latency 0)
         (OpPortCommandList
          (PortCommandDesc LOAD       (PortDefaultValue 0))
          (PortCommandDesc SYNC_CTRL  (PortDefaultValue 0))
          (PortCommandDesc CLK_EN     (PortDefaultValue 1))
          (PortCommandDesc CLOCK      (PortDefaultValue 1))
          (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0))
          (PortCommandDesc ADD_SUB    (PortDefaultValue 0)))
         (OpPortDataList
          (PortdataDesc B)
          (PortdataDesc C_IN (PortDefaultValue 1))
          (PortdataDesc C_OUT)
          (PortdataDesc OVFL)
          (PortdataDesc Q_OUT)))
    )
   )

#==========================================================================#
# Incrementer/Decrementer.                                                 #
#==========================================================================#

   (Generic INC_DEC
    (ResType OPE)
    (Area 0) (Height 0) (Width 0)
    (ParamList
     (ParamDesc N (ParamType INT) (ParamValues 1 Inter 64)
                  (ParamDefaultValue 16)))
    (PortList
     (PortDesc A        (PortType INPUT)     (PortWidth (ParamUsed N)))
     (PortDesc INC_DEC  (PortType COMMAND)   (PortWidth 1))
     (PortDesc FUNC     (PortType OUTPUT)    (PortWidth (ParamUsed N)))
     (PortDesc C_OUT    (PortType OUTPUT)    (PortWidth 1))
```

```
      (PortDesc OVFL      (PortType OUTPUT)      (PortWidth 1)))
     (OpList
      # FUNC <= A + <const> #
      (OpDesc INCR (OpType INCR_OP)
       (Delay 0) (Pipeline 0) (Latency 0)
        (OpPortCommandList
         (PortCommandDesc INC_DEC (PortDefaultValue 1)))
        (OpPortDataList
         (PortDataDesc A)
         (PortDataDesc FUNC)
         (PortDataDesc C_OUT)
         (PortDataDesc OVFL)))
      # FUNC <= A - <const> #
      (OpDesc DECR (OpType DECR_OP)
       (Delay 0) (Pipeline 0) (Latency 0)
        (OpPortCommandList
         (PortCommandDesc INC_DEC (PortDefaultValue 0)))
        (OpPortDataList
         (PortDataDesc A)
         (PortDataDesc FUNC)
         (PortDataDesc C_OUT)
         (PortDataDesc OVFL)))
   )
  )


#=====================================================================#
# n-bit Register                                                      #
#=====================================================================#

 (Generic DATA_REG (ResType MEM_D)
  (Area 0) (Height 0) (Width 0)
  (ParamList
   (ParamDesc N (ParamType INT) (ParamValues 1 Inter 64)
                (ParamDefaultValue 16)))
  (PortList
   (PortDesc D_IN          (PortType INPUT)          (PortWidth (ParamUsed N)))
   (PortDesc CLK_EN        (PortType COMMAND)        (PortWidth 1))
   (PortDesc CLOCK         (PortType CLOCK)          (PortWidth 1))
   (PortDesc ASYNC_CTRL    (PortType RESET_ASYNC)    (PortWidth 1))
   (PortDesc SYNC_CTRL     (PortType RESET_SYNC)     (PortWidth 1))
   (PortDesc Q_OUT         (PortType OUTPUT)         (PortWidth (ParamUsed N))))
  (OpList
   # Q_OUT <= ASYNC_VAL #
   (OpDesc RESET_ASYN (OpType CLEAR_ASYNC)
 (Delay 0) (Pipeline 0) (Latency 0)
    (OpPortCommandList
     (PortCommandDesc SYNC_CTRL  (PortDefaultValue -))
     (PortCommandDesc ASYNC_CTRL (PortDefaultValue 1)))
    (OpPortDataList
     (PortdataDesc Q_OUT)))
   # Q_OUT <= (hold) #
   (OpDesc KEEP (OpType KEEP_VALUE)
    (Delay 0) (Pipeline 0) (Latency 0)
    (OpPortCommandList
     (PortCommandDesc SYNC_CTRL  (PortDefaultValue -))
     (PortCommandDesc CLK_EN     (PortDefaultValue 0))
     (PortCommandDesc CLOCK      (PortDefaultValue 1))
     (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0)))
```

```
        (OpPortDataList
         (PortdataDesc Q_OUT)))
      # Q_OUT <= SYNC_VAL #
      (OpDesc RESET_SYN (OpType CLEAR_SYNC)
       (Delay 0) (Pipeline 0) (Latency 0)
        (OpPortCommandList
         (PortCommandDesc SYNC_CTRL  (PortDefaultValue 1))
         (PortCommandDesc CLK_EN     (PortDefaultValue 1))
         (PortCommandDesc CLOCK      (PortDefaultValue 1))
         (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0)))
        (OpPortDataList
         (PortdataDesc Q_OUT)))
      # Q_OUT <= D_IN #
      (OpDesc LOAD (OpType LOAD) (Delay 0) (Pipeline 0) (Latency 0)
        (OpPortCommandList
         (PortCommandDesc SYNC_CTRL  (PortDefaultValue 0))
         (PortCommandDesc CLK_EN     (PortDefaultValue 1))
         (PortCommandDesc CLOCK      (PortDefaultValue 1))
         (PortCommandDesc ASYNC_CTRL (PortDefaultValue 0)))
        (OpPortDataList
         (PortdataDesc D_IN)
         (PortdataDesc Q_OUT)))
    )
   )


  )
 )
```

Annexe III

-- ◊ - ◊◊◊ - ◊ --

Nouveau format de description en bibliothèque des éléments utilisés par tout type de synthèse

III.1. Grammaire du format

III.1.1. Grammaire du fichier décrivant les informations structurelles et fonctionnelles des éléments de bibliothèque

Grammaire créé par Bison 1.22. Les terminaux sont en majuscules.

```
begin -> LIBRARY string_or_idf busstyle unit_length unit_area lparam0 lunit EndOfFile
begin -> LIBRARY error
begin -> error
busstyle -> BUSSTYLE string_or_idf string_or_idf
busstyle -> empty
unit_length -> UNIT_LENGTH string_or_idf
unit_length -> empty
unit_area -> UNIT_AREA string_or_idf
unit_area -> empty
lunit -> lunit unit
lunit -> unit
unit -> NAME string_or_idf restype generic lparam0 area heigth width touch_use lports fonctionnality
restype -> GATETOK
restype -> UGATETOK
restype -> TGATETOK T_GATE
generic -> GENERIC
generic -> NON_GENERIC
generic -> empty
area -> AREA exp
area -> empty
heigth -> HEIGTH exp
heigth -> empty
width -> WIDTH exp
width -> empty
touch_use -> TOUCH boolean USE boolean
touch_use -> empty
lports -> PORT '(' lport2 ')'
lport2 -> lport2 SEP port
lport2 -> port
port -> lidf ptype function cmd pinphase invtype synchrotype encoding optionnal exclude include portwidth
ptype -> IN
ptype -> OUT
ptype -> INOUT
function -> PORT_TYPE
function -> empty
cmd -> CMD
cmd -> NOT_CMD
cmd -> empty
pinphase -> UNKNOWN
pinphase -> INV
pinphase -> NOT_INV
pinphase -> empty
invtype -> PORT_NOT_INV
invtype -> PORT_INV
invtype -> PORT_INVMASK BINARY_MASK
invtype -> PORT_INVMASK string_or_idf
invtype -> empty
synchrotype -> ASYNC INTEGERtok
synchrotype -> SYNC
```

synchrotype -> empty
encoding -> UNSIGNED
encoding -> SIGNED
encoding -> C2
encoding -> ONE_HOT
encoding -> empty
optionnal -> OPT
optionnal -> NEC
optionnal -> empty
exclude -> EXCLUDE '(' lidf ')'
exclude -> empty
include -> INCLUDE '(' lidf ')'
include -> empty
portwidth -> WIDTH intervall_int msb_lsb
portwidth -> WIDTH exp msb_lsb
portwidth -> empty msb_lsb
msb_lsb -> MSB
msb_lsb -> LSB
msb_lsb -> empty
fonctionnality -> lequations build
fonctionnality -> loperations build
fonctionnality -> build
lequations -> EQ '(' leq ')'
leq -> leq SEP eq
leq -> eq
eq -> string_or_idf EQUALtok expbin
eq -> expbin
expbin -> boolean
expbin -> string_or_idf
expbin -> string_or_idf '(' laffect ')'
expbin -> string_or_idf '(' ')'
expbin -> '(' expbin ')'
expbin -> expbin PLUStok expbin
expbin -> expbin MULTtok expbin
expbin -> expbin XORtok expbin
expbin -> NOTtok expbin
laffect -> laffect SEP affect
laffect -> affect
affect -> string_or_idf EQUALtok string_or_idf
affect -> string_or_idf
loperations -> loperations op
loperations -> op
op -> OP string_or_idf op_type lequations2 op_port_comm conditions portlist consequences pipeline latency
op_type -> OP_TYPE
lequations2 -> lequations
lequations2 -> empty
op_port_comm -> COMM '(' lidf ')'
op_port_comm -> empty
conditions -> COND '(' lcond ')'
conditions -> empty
lcond -> lcond SEP cond
lcond -> cond
cond -> string_or_idf exp
cond -> string_or_idf RE
cond -> string_or_idf FE
cond -> string_or_idf NC
cond -> string_or_idf
portlist -> DATA '(' ldata ')'
portlist -> empty

```
ldata -> ldata SEP data
ldata -> data
data -> string_or_idf
data -> string_or_idf exp
consequences -> PUT '(' lput ')'
consequences -> empty
lput -> lput SEP put
lput -> put
put -> string_or_idf exp
put -> string_or_idf NC
pipeline -> PIPELINE exp
pipeline -> empty
latency -> LATENCY exp
latency -> empty
build -> DECLARATION '(' ldec ')' CONSTRUCTION '(' lcons ')'
build -> empty
ldec -> ldec SEP dec
ldec -> dec
dec -> string_or_idf exp
lcons -> lcons SEP cons
lcons -> cons
cons -> eq
lparam0 -> PARAM '(' lparam ')'
lparam0 -> empty
lparam -> lparam param
lparam -> param
param -> INT lidf intervall_int default_int
param -> STR lidf lstring default_str
lidf -> lidf SEP string_or_idf
lidf -> string_or_idf
intervall_int -> exp INTER exp
intervall_int -> empty
lstring -> lstring1
lstring -> empty
lstring1 -> lstring1 SEP string_or_idf
lstring1 -> string_or_idf
default_int -> DEFAULT exp
default_int -> empty
default_str -> DEFAULT string_or_idf
default_str -> empty
exp -> INTEGERtok
exp -> REALtok
exp -> string_or_idf
exp -> '(' exp ')'
exp -> exp PLUStok exp
exp -> exp MINUStok exp
exp -> exp MULTtok exp
exp -> exp DIVtok exp
exp -> exp POWERtok exp
exp -> LOG2tok exp
exp -> MINUStok exp
string_or_idf -> STRING
string_or_idf -> IDFtok
boolean -> TRUE
boolean -> FALSE
empty ->/* empty */
```

III.1.2. Liste des types des éléments, des ports et des opérations

Types des éléments

TG_GATE, TG_UGATE, TG_T, TG_D, TG_L, TG_JK, TG_RS, TG_DMUX, TG_LMUX, TG_COUNTER, TG_ACCUM, TG_RAM, TG_ROM, TG_SEQ, TG_TH, TG_ITH, TG_TRI, TG_IBUF, TG_OBUF, TG_BIBUF, TG_CLKBUF, TG_TSBUF, TG_INPAD, TG_OUTPAD, TG_BIPAD, TG_MUX, TG_DECOD, TG_OPE, TG_VSS, TG_VDD, TG_LIB_CONST, TG_CONNECTOR, TG_BUS

Types des ports

PT_CLK, PT_RST, PT_SET, PT_CLKEN, PT_INJ, PT_INK, PT_INR, PT_INS, PT_3STATEN, PT_SEL, PT_HIMPOUT, PT_PAD, PT_CIN, PT_COUT, PT_OVFL, PT_ADDSUB, PT_ROP, PT_LOP, PT_STD, PT_RIGHT_LEFT, PT_UPPER_LOWER

Types des opérations

OP_ADD, OP_ADDC2, OP_SUB, OP_SUBC2, OP_MULT, OP_MULTC2, OP_DIV, OP_DIVC2, OP_COMP_LE, OP_COMP_LT, OP_COMP_GE, OP_COMP_GT, OP_COMP_EQ, OP_COMP_NE, OP_COMPC2_LE, OP_COMPC2_LT, OP_COMPC2_GE, OP_COMPC2_GT, OP_COMP_ZERO, OP_COMP_ONE, OP_LOG_AND, OP_LOG_OR, OP_LOG_NAND, OP_LOG_NOR, OP_LOG_XOR, OP_LOG_XNOR, OP_LOG_NOT, OP_LOG_AO, OP_LOG_AOI, OP_LOG_OA, OP_LOG_OAI, OP_SHIFTL, OP_SHIFTR, OP_INCR, OP_DECR, OP_Z_TRI, OP_Q_TRI, OP_CONNEC_IN, OP_CONNEC_OUT, OP_BUFFER, OP_BUFFER_INV, OP_CONST, OP_LOAD, OP_CLEAR_SYNC, OP_CLEAR_ASYNC, OP_PRESET_SYNC, OP_PRESET_ASYNC, OP_LOAD_ATTRIBUTE_SYNC, OP_LOAD_ATTRIBUTE_ASYNC, OP_ROTATEL, OP_ROTATER, OP_KEEP_VALUE, OP_MUX, OP_DECOD, OP_OTHER

III.1.3. Grammaire du fichier décrivant les informations temporelles des éléments de bibliothèque

Grammaire créé par Bison 1.22. Les terminaux sont en majuscules.

```
Libraries -> Libraries Library
Libraries -> Library
Library -> LIBtok LibName LibHeader UnitDef
LibName -> StringOrIdf
LibHeader -> BaseUnitDef DefaultOpCond OperatingConditionDef
BaseUnitDef ->          /* empty */
BaseUnitDef -> BASEUNITtok BaseUnits
BaseUnits -> BaseUnits BaseUnit
BaseUnits -> BaseUnit
BaseUnit -> TemperatureUnit
BaseUnit -> VoltageUnit
BaseUnit -> ProcessUnit
BaseUnit -> TimeUnit
BaseUnit -> RoutingCapacitanceUnit
BaseUnit -> CapacitanceUnit
TemperatureUnit -> TEMPtok UnitStr
VoltageUnit -> VOLTtok UnitStr
ProcessUnit -> PROCtok UnitStr
TimeUnit -> TIMEtok UnitStr
RoutingCapacitanceUnit -> RCAPtok UnitStr
CapacitanceUnit -> CAPAtok UnitStr
UnitStr -> STRtok
DefaultOpCond ->          /* empty */
DefaultOpCond -> DEFOPCONDtok StringOrIdf
OperatingConditionDef ->          /* empty */
OperatingConditionDef -> OperatingConditions
OperatingConditions -> OperatingConditions OperatingCondition
OperatingConditions -> OperatingCondition
OperatingCondition -> OPCONDtok StringOrIdf OpConditions
OpConditions -> OpConditions OpCondition
OpConditions -> OpCondition
OpCondition -> Temperature
OpCondition -> Voltage
OpCondition -> Process
OpCondition -> DerKt
OpCondition -> DerKv
OpCondition -> DerKp
OpCondition -> RoutingCapacitance
Temperature -> TEMPtok SpecialReal
Voltage -> VOLTtok SpecialReal
Process -> PROCtok SpecialReal
DerKt -> DerKttok SpecialReal
DerKv -> DerKvtok SpecialReal
DerKp -> DerKptok SpecialReal
RoutingCapacitance -> RCAPtok SpecialReal
UnitDef ->          /* empty */
UnitDef -> Units
Units -> Units Unit
Units -> Unit
Unit -> UNITtok UnitName UnitSpec
UnitName -> StringOrIdf
UnitSpec -> Specifications
```

```
Specifications -> Specifications Specification
Specifications -> Specification
Specification -> OutputCapaSpec
Specification -> InputCapaSpec
Specification -> TimingSpec
OutputCapaSpec -> POUTCtok PortList Capacitance MaxFanout
InputCapaSpec -> PINCtok PortList Capacitance InputLoad
TimingSpec -> PTIMtok OutPortList InPortList TimingInfo
OutPortList -> PortList
InPortList -> PortList
TimingInfo -> InputTiming1
TimingInfo -> EQtok IDFtok InputTiming1
InputTiming1 ->            /* empty */
InputTiming1 -> RiseDelay InputTiming2
InputTiming2 ->            /* empty */
InputTiming2 -> RiseFanoutDelay InputTiming3
InputTiming3 ->            /* empty */
InputTiming3 -> FallDelay InputTiming4
InputTiming4 ->            /* empty */
InputTiming4 -> FallFanoutDelay InputTiming5
InputTiming5 ->            /* empty */
InputTiming5 -> SetupTime InputTiming6
InputTiming6 ->            /* empty */
InputTiming6 -> HoldTime
Capacitance -> SpecialExpr
MaxFanout -> SpecialExpr
InputLoad -> SpecialExpr
RiseDelay -> SpecialExpr
RiseFanoutDelay -> SpecialExpr
FallDelay -> SpecialExpr
FallFanoutDelay -> SpecialExpr
SetupTime -> SpecialExpr
HoldTime -> SpecialExpr
SpecialReal -> REALtok
SpecialReal -> '*'
SpecialReal -> error
SpecialExpr -> expr
SpecialExpr -> '*'
SpecialExpr -> error
PortList -> PortList2
PortList -> '*'
PortList2 -> PortList2 ',' PortName
PortList2 -> PortName
PortList2 -> error
PortName -> BusName
PortName -> BusIndex
PortName -> BusPart
BusName -> StringOrIdf
BusIndex -> BusName Index
Index -> '[' INTtok ']'
BusPart -> BusName '[' INTtok ':' INTtok ']'
StringOrIdf -> IDFtok
StringOrIdf -> STRtok
expr -> '(' exp ')'
expr -> '+' expr
expr -> '-' expr
expr -> INTtok
expr -> REALtok
expr -> IDFtok
```

```
exp -> expr
exp -> exp '+' exp
exp -> exp '-' exp
exp -> exp '*' exp
exp -> exp '/' exp
exp -> exp DIVtok exp
exp -> exp EQtok exp
exp -> exp NEtok exp
exp -> exp LEtok exp
exp -> exp GEtok exp
exp -> exp LTtok exp
exp -> exp GTtok exp
```

II.2. Exemples

III.2.1. Exemple de fichier décrivant les informations structurelles et fonctionnelles

```
Library LPM
##    GATES       ##
Name LPM_CONSTANT
TGate TG_Lib_Const
Generic
Param ( Int "WIDTH", CVALUE )
# WIDTH : Any positive integer
# CVALUE : Any intger including negative integer
## If the value specified cannot be represented in a vector N bits wide, the RESULT vector contains the N
least significant bits of the value's two's-complement representation ##
Port ( RESULT Out Width "WIDTH" LSB )
Op CVALUE OP_Const
  Eq(RESULT = CVALUE)
  Data ( RESULT )
  Put ( RESULT CVALUE)
######################################################################
Name LPM_INV
TGate TG_Ope
Generic
Param ( Int "WIDTH" )
Port (  "DATA" In Width "WIDTH"  LSB ,
  RESULT Out Width "WIDTH"  LSB )
Op "INV" OP_Log_Not
  Eq(RESULT = ! "DATA")
  Data ( "DATA" , RESULT )
######################################################################
Name LPM_AND
 TGate TG_Ope
Generic
Param ( Int SIZE
  Str LPM_POLARITY INVERT   )
Port (  DATA0_ In Port_Invmask LPM_POLARITY Width SIZE LSB ,
  RESULT0 Out Port_Invmask LPM_POLARITY )
Op AND OP_Log_And
  Data ( DATA0_, RESULT0 )
######################################################################
Name LPM_OR
 TGate TG_Ope
Generic
Param ( Int SIZE
  Str LPM_POLARITY INVERT   )
Port (  DATA0_ In Port_Invmask LPM_POLARITY Width SIZE LSB ,
  RESULT0 Out Port_Invmask LPM_POLARITY )
Op OR OP_Log_Or
  Data ( DATA0_, RESULT0 )
######################################################################
Name LPM_XOR
 TGate TG_Ope
Generic
Param ( Int SIZE
  Str LPM_POLARITY INVERT   )
Port (  DATA0_ In Port_Invmask LPM_POLARITY Width SIZE LSB ,
```

```
RESULT0 Out Port_Invmask LPM_POLARITY )
Op XOR OP_Log_Xor
  Data ( DATA0_, RESULT0 )
##########################################################################
Name LPM_BUSTRI
 TGate TG_Th
Generic
Param ( Int "WIDTH"
  Str LPM_POLARITY INVERT   )
Port ( TRIDATA Inout PT_Himpout Port_Invmask LPM_POLARITY Width "WIDTH" LSB ,
"DATA" In Port_Invmask LPM_POLARITY Opt Width "WIDTH" LSB , ENABLETR In Cmd
Port_Invmask LPM_POLARITY Opt,
ENABLEDT In PT_3staten Cmd Port_Invmask LPM_POLARITY Opt, RESULT Out Port_Invmask
LPM_POLARITY Opt Width "WIDTH" LSB )
Op Z_ON_TRI_ON_RESULT OP_Z_tri
  Cond ( ENABLEDT 0 , ENABLETR 0)
  Data ( TRIDATA, RESULT )
Op TRI_ON_RESULT OP_Q_tri
  Cond ( ENABLEDT 0 , ENABLETR 1 )
  Data ( TRIDATA, RESULT )
Op DATA_ON_TRI OP_Q_tri
  Cond ( ENABLEDT 1 , ENABLETR 0 )
Data ( "DATA", TRIDATA, RESULT) Put ( RESULT Z )
Op DATA_ON_TRI_ON_RESULT OP_Q_tri
Cond ( ENABLEDT 1 , ENABLETR 1 ) Data ( "DATA", TRIDATA, RESULT) Put ( RESULT "DATA" )
##########################################################################
Name LPM_MUX
 TGate TG_Mux
Generic
Param ( Int SIZE 2 Inter 12
# SIZE : any positive integer > 1.
  Int WIDTHS default 4
# WIDTHS : any positive integer >= Log2(SIZE).
  Str LPM_POLARITY INVERT   )
Port ( DATA0_ In Port_Invmask LPM_POLARITY Width SIZE LSB ,
  RESULT0 Out Port_Invmask LPM_POLARITY ,
  SELECT In PT_Sel Cmd Port_Invmask LPM_POLARITY Width WIDTHS LSB)
Op MUX OP_BUFFER
# For I in ( 0 to SIZE MINUS 1 ) loop
#       DATA0_[I]2RESULT0
#       Cond ( SELECT Bin(I,WIDTHS) )
#       Data ( DATA0_[I], RESULT0 )
  Data ( DATA0_, RESULT0 )
##########################################################################
Name LPM_DECODE
 TGate TG_Decod
Generic
Param ( Int "WIDTH"
  Int DECODES Default 2
# DECODES : any positive integer <= 2Power("WIDTH").
  Str LPM_POLARITY INVERT   )
Port ( "DATA" In Port_Invmask LPM_POLARITY Width "WIDTH" LSB ,
"EQ" Out Port_Invmask LPM_POLARITY Width DECODES LSB , "ENABLE" In PT_Sel Cmd
Port_Invmask LPM_POLARITY Opt)
Op DECODER OP_LOAD
  Cond ( ENABLE 1 )
  Data ( "DATA", "EQ" )
Op NOTDECOD OP_KEEP_VALUE
  Cond ( ENABLE 0 )
```

```
  Data ( "EQ" )
  Put ( "EQ" Bin )
 Op LPM_DECODE OP_KEEP_VALUE
  Cond ( ENABLE FE )
  Data ( "DATA", "EQ" )
 ########################################################################
Name LPM_CLSHIFT
 TGate TG_Ope
Generic
Param ( Int "WIDTH"
  Int DISTWIDTH 1 Inter 4 default 6
# DISTWIDTH: optional
  Str SHIFTTYPE LOGICAL,ROTATE,ARITHMETIC
# The sign bit is only extended for ARITHMETIC. Zeros are shifted in the MS bit for LOGICAL shifts.
  Str LPM_POLARITY INVERT )
Port ( "DATA" In Port_Invmask LPM_POLARITY Width "WIDTH" LSB , DISTANCE In Port_Invmask
LPM_POLARITY Width DISTWIDTH LSB ,
# DISTANCE: Number of position to shift in direction specified by DIRECTION port. DIRECTION In
Cmd Port_Invmask LPM_POLARITY Opt ,
# DIRECTION: Low=Left, High=Right; Default is Left.
RESULT Out Port_Invmask LPM_POLARITY Width "WIDTH" LSB , OVERFLOW Out PT_Ovfl
Port_Invmask LPM_POLARITY Opt ,
# OVERFLOW: Not available for ROTATE.
  UNDERFLOW Out Pt_Ovfl Port_Invmask LPM_POLARITY Opt ) # UNDERFLOW: Not available for
ROTATE.
Op SHRIGHT OP_SHIFTR
  Cond ( DIRECTION 1 )
  Data ( "DATA", DISTANCE, RESULT, OVERFLOW, UNDERFLOW) Op SHLEFT OP_SHIFTL
  Cond ( DIRECTION 0 )
  Data ( "DATA", DISTANCE, RESULT, OVERFLOW, UNDERFLOW) Op ROTRIGHT OP_ROTATER
  Cond ( DIRECTION 1 )
  Data ( "DATA", DISTANCE, RESULT)
  Put ( SHIFTTYPE ROTATE)
Op ROTLEFT OP_ROTATEL
  Cond ( DIRECTION 0 )
  Data ( "DATA", DISTANCE, RESULT)
  Put ( SHIFTTYPE ROTATE)
Op LPM_CLSHIFT OP_KEEP_VALUE
  Cond ( DIRECTION RE )
  Data ( "DATA", DISTANCE, RESULT, OVERFLOW, UNDERFLOW)
########################################################################
##                    ARITHMETIC COMPONENTS                    ##
########################################################################
Name LPM_ADD_SUB
 TGate TG_Ope
Generic
Param ( Int "WIDTH"
  Str REPRESENTATION "UNSIGNED", "SIGNED" Default "SIGNED"
  Str LPM_POLARITY INVERT )
Port ( DATAA In Port_Invmask LPM_POLARITY Width "WIDTH" LSB ,
DATAB In Port_Invmask LPM_POLARITY Width "WIDTH" LSB , SUM Out Port_Invmask
LPM_POLARITY Width "WIDTH" LSB , CIN In PT_Cin Port_Invmask LPM_POLARITY Opt,
COUT Out PT_Cout Port_Invmask LPM_POLARITY Opt Exclude( OVERFLOW), OVERFLOW Out
PT_Ovfl Port_Invmask LPM_POLARITY Opt Exclude (COUT), ADD_SUB In PT_addSub Cmd
Port_Invmask LPM_POLARITY Opt)
# If ADD_SUB is not present, Op = ADD
Op ADDITION OP_ADD
  Comm ( DATAA, DATAB )
  Cond ( ADD_SUB 1 )
```

```
Data ( DATAA, DATAB, SUM, CIN 0, COUT ) Put (REPRESENTATION "UNSIGNED")
Op ADDITIONC2 OP_ADDC2
 Comm ( DATAA, DATAB )
 Cond ( ADD_SUB 1 )
Data ( DATAA, DATAB, SUM, CIN 0, OVERFLOW ) Put (REPRESENTATION "SIGNED")
Op SUBTRACTION OP_SUB
 Cond ( ADD_SUB 0 )
Data ( DATAA, DATAB, SUM, CIN 1, COUT ) Put (REPRESENTATION "UNSIGNED")
Op SUBTRACTIONC2 OP_SUBC2
 Cond ( ADD_SUB 0 )
Data ( DATAA, DATAB, SUM, CIN 1, OVERFLOW ) Put (REPRESENTATION "SIGNED")
Op LPM_ADD_SUB OP_KEEP_VALUE
 Cond ( ADD_SUB FE )
 Data ( DATAA, DATAB, SUM, CIN, COUT, OVERFLOW)
##    STORAGE COMPONENTS  ##
##    TABLE PRIMITIVES    ##
##    PADS                ##
```

III.2.1. Exemple de fichier décrivant les informations temporelles

```
Library LPM

Units Temp     "degreC"
  Volt     "Volt"
  Time     "Ns"
  RCap     "nF/um"
  Capa     "nF"

DefOpCond OC1

OpCond OC1
  Temp     25.0
  Volt     5.0
  Proc     1.0
  Kt  1.0
  Kv  1.0
  Kp  1.0
  RCap     0.6

OpCond OC2
  Temp     50.0
  Volt     5.0
  Proc     1.0
  Kt  1.6
  Kv  *
  Kp  1.8
  RCap     0.5

OpCond OC3
  Temp     50.0
  Volt     7.0
  Proc     1.5
  Kt  1.6
  Kv  *
  Kp  1.8
  RCap     0.4

Name LPM_CONSTANT
Poutc RESULT 10 10
Ptim RESULT * 3 3 3 3

Name LPM_INV
Poutc RESULT 10 *
Pinc DATA * 14
Ptim RESULT DATA 6 6 6 6

Name LPM_AND
Poutc RESULT0 * *
Pinc DATA0_ * *
Ptim * * * * * * * *

Name LPM_OR
Poutc * 4 4
Pinc * 4 4
Ptim * * 4 4 4 4

Name LPM_XOR
```

```
Poutc RESULT0 (3 * SIZE) (2 + SIZE)
Pinc DATA0_ ((5 - 10.4)/(4.8 * SIZE)) 67
Ptim RESULT0 DATA0_ * * 67 (-10.6 - 9 + (SIZE * 8))

Name LPM_BUSTRI
Poutc TRIDATA 4 5
Poutc RESULT 8 9
Pinc TRIDATA 5 5
Pinc DATA 9 0
Pinc ENABLETR,ENABLEDT 8 *
#Ptim TRIDATA,* TRIDATA,* (7 * WIDTH) (7 * WIDTH) * (7 * WIDTH)
Ptim TRIDATA TRIDATA (7 * WIDTH) (7 * WIDTH) * (7 * WIDTH)

Name LPM_MUX
Poutc * 6 *
Pinc * 7 9
Ptim RESULT0 SELECT 8 8 8 8
Ptim RESULT0 DATA0_ 9 9 * * * *

Name LPM_DECODE
Poutc EQ (9.0 * WIDTH + DECODES) *
Pinc DATA,ENABLE (8+DECODES) (5/WIDTH)
Ptim EQ DATA 7 * * 9
Ptim * ENABLE 5 * 9 9

Name LPM_CLSHIFT
Poutc UNDERFLOW,OVERFLOW,RESULT 8 8
Pinc DATA,DISTANCE[0] 9 9
Pinc DIRECTION,DISTANCE[1:31] 8 7
#Ptim SHRIGHT * * * * * * * *
Ptim Op SHLEFT RESULT * 6 6 6 6
Ptim Op SHLEFT OVERFLOW,UNDERFLOW DATA[0:9]
        ((6 * WIDTH)+DISTWIDTH+18.7) 9 9 9 9
Ptim Op SHLEFT OVERFLOW,UNDERFLOW DATA[10:31]
        ((7 * WIDTH)+DISTWIDTH) 9 9 9 9
Ptim Op SHLEFT OVERFLOW,UNDERFLOW DISTANCE, DIRECTION 8 8 * 8 *
Ptim Op ROTRIGHT RESULT [0:30] * 9 9 9
Ptim Op ROTRIGHT RESULT [31] * 4 4 4 4
Ptim Op ROTRIGHT OVERFLOW,UNDERFLOW DATA[0:32],DISTANCE,DIRECTION 8 8 * 8 *
Ptim Op ROTLEFT * DATA 5 5 5 5
Ptim Op ROTLEFT * DISTANCE,DIRECTION (12 * WIDTH) * 6 7 *
Ptim Op LPM_CLSHIFT RESULT,OVERFLOW,UNDERFLOW DATA,DISTANCE,DIRECTION
        7.5.1. * WIDTH) * ((6 * WIDTH)+DISTWIDTH+18.7) *
```