RAPPORT D'ANALYSE

PROJET C

Bastien MAHTAL et Mathieu ESCOUTELOUP
2A-CSI
Année Universitaire 2016-2017

Sommaire

I. Introduction

L'objectif de ce projet est de réaliser en binôme un synthétiseur VHDL. Cet outil, qui prend en entrée un fichier écrit en VHDL, est capable après plusieurs étapes de vérifications du code de donner une liste des différentes cellules logiques nécessaires à la conception du circuit : une netlist.

Au final, un ensemble de fichiers codés en C++, d'exécutables … doit être livré et doit réaliser correctement cette étape de synthèse. D'autres conditions sur son fonctionnement ont été imposées, comme l'environnement qui doit être un OS Linux installé sur un des ordinateurs de PHELMA.

Un tel projet représente une quantité de travail importante et nécessite une bonne organisation. C'est l'occasion de mieux comprendre comment fonctionne un outil essentiel de la conception numérique, ainsi que d'appliquer des connaissances vues en première année, dans les cours de VHDL, C++, circuits logiques …

Le but de ce rapport est de nous aider à nous poser les bonnes questions. Nous allons donc voir comment nous avons décidé de découper en plusieurs parties ce projet, mais aussi comment nous allons nous organiser sur ces plusieurs mois de travail.

II. Décomposition modulaire du programme

Le synthétiseur VHDL qu'il s'agit de réaliser doit suivre une succession d'étapes clés pour passer d'un fichier VHDL non traité à un fichier de sortie vérifié sous forme de netlist. On compte ainsi cinq blocs, à commencer par trois étapes qui permettent de gérer le parsing du fichier source :

- **L'analyse lexicale** (ou analyseur de lexèmes) permet de détecter les lexèmes écrits dans le fichier texte et de les découper dans le but de vérifier l'orthographe de chaque mot. Cette première étape permet un premier traitement du fichier, si un lexème n'est pas correctement orthographié on ne pourra pas passer à la suite.

- **L'analyse syntaxique** a pour rôle de vérifier que l'architecture du VHDL est correcte, elle va ainsi identifier les lexèmes afin d'obtenir en sortie un arbre de structure de données.

- **L'analyse contextuelle** (ou pré-synthèse) va vérifier que l'arbre de structure de données obtenu précédemment est correct, l'idée est ainsi de s'assurer que ce qui est écrit dans l'arbre est réalisable physiquement par la suite.

Une fois ces analyses effectuées, le fichier est dit « parsé » et les données sont structurées dans un arbre qui schématise le déroulement d'un fichier VHDL. Ces étapes sont donc fondamentales avant de passer à la synthèse du fichier et au traitement post-synthèse :

- **La synthèse** consiste à retranscrire chaque élément du code VHDL en portes logiques.

- **La post-synthèse** aura pour rôle de fournir le chemin critique de la netlist. Les caractéristiques de ce chemin sont très importantes, elles permettent de gérer l'optimisation de la synthèse si besoin car on va vouloir que le chemin critique soit le plus faible possible dans la plupart des cas.

Cette décomposition du projet de synthétiseur VHDL est entre autres un premier moyen de nous répartir les tâches au sein du groupe, ce que nous verrons plus tard dans le planning prévisionnel. Chaque bloc étant indépendant, nous allons pouvoir les tester un à un au fur et à mesure de l'avancement du projet, avant d'effectuer un test du synthétiseur complet.

Nous allons présenter chaque bloc plus précisément dans la suite du rapport d'analyse.

Voici un schéma descriptif des différentes étapes ordonnées avec dans chaque bloc le nom du bloc ainsi que son entrée :

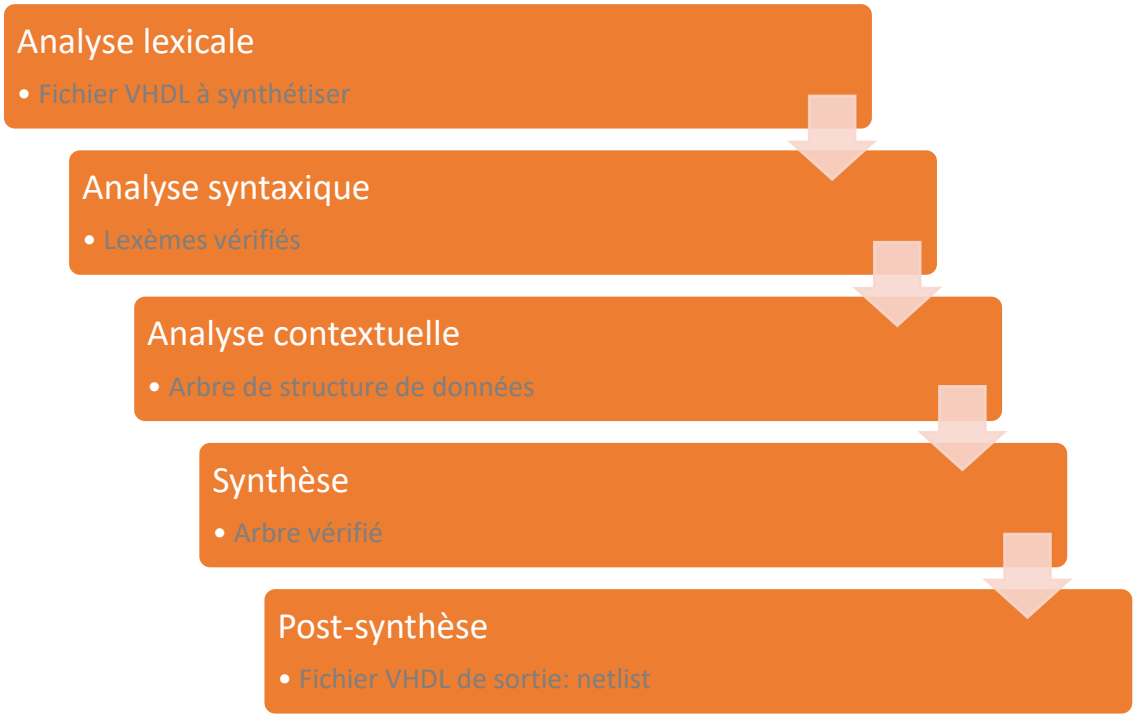


Figure 1: Décomposition modulaire du programme

III. Description des modules

1. Analyseur lexical

L'analyseur lexical constitue la première étape de ce flot, il va permettre de relever toutes les erreurs lexicales, de vérifier l'orthographe des lexèmes et de gérer leur découpage de manière ordonnée. Ce module va ainsi récupérer un fichier VHDL en entrée et va sortir une liste chaînée par lexème vérifié et découpé au travers de différentes fonctions intermédiaires.

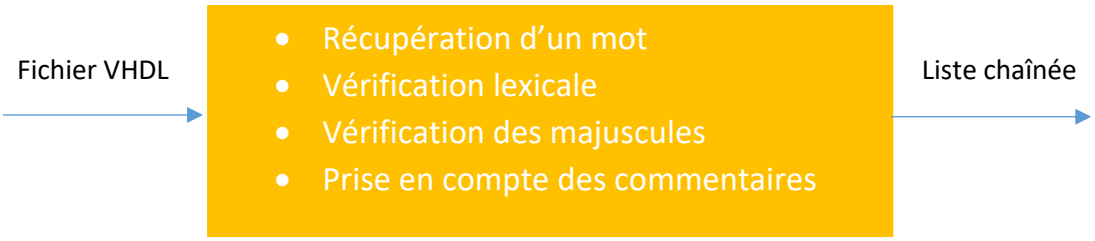


Figure 2: Module d'analyse lexicale

1. Récupération de mots

L'analyseur aura un pointeur qui va parcourir le fichier caractère par caractère, il devra ainsi s'arrêter à chaque fois qu'il aura détecter un caractère spécial ([] () ; :) ou une zone « sans caractère » telle qu'un espace ou une tabulation.

Les mots seront ainsi séparés à chaque coupure et il est important de préciser qu'un caractère spécial est également un lexème car il fait partie intégrante de la structure de données.

2. Vérification lexicale

Le VHDL suit des règles d'orthographes strictes qui régissent le bon fonctionnement d'un code ainsi que sa synthétisabilité, voici donc un relevé des différentes règles (sauf caractères spéciaux) :

- Le premier caractère doit être une lettre
- Le lexème doit uniquement être constitué des caractères alphanumériques et de l'underscore
- Il ne peut pas y avoir de caractère « underscore » en début ou en fin de lexème

Cette vérification vient logiquement juste après avoir récupérer chaque mot. En effet après avoir isoler un mot on va s'intéresser à son orthographe et si elle présente des erreurs alors le programme doit retourner et indiquer une erreur.

3. Vérification des majuscules

En VHDL les majuscules ont le même sens que les minuscules, on ne fait aucune distinction entre ces deux formats. En revanche le code ASCII des minuscules n'est pas le même que celui des majuscules,

on aurait donc une erreur par la suite si aucun traitement n'était effectué, c'est pourquoi on va s'arranger pour que toutes les majuscules soient converties en minuscules.

On ne s'intéresse uniquement à ce point si les deux précédentes étapes se sont bien déroulées.

4. Prise en compte des commentaires

Dans tout code VHDL il y a des commentaires pour décrire ce qui a été fait, ce qui permet une meilleure visibilité mais qui n'est pas lu lors de la synthèse. Nous avons décidé de prendre en compte ces commentaires dans l'analyse lexicale pour qu'il n'y ait pas d'ambiguïté lors des étapes suivantes.

Un commentaire en VHDL débute forcément par les caractères « -- » et se finit par un retour à la ligne. Nous avons donc décidé qu'à la lecture successive de deux caractères « - », le pointeur devait passer à la ligne suivante et au premier caractère.

5. Exemple de fichier vérifié

Si l'on prend pour exemple une simple création d'entité dans un code VHDL, voici le résultat que l'on souhaiterait obtenir après être passé dans la boucle de vérifications :

```
entity and_2 is
    port (
        a : in bit;
        b : in bit;
        c : out bit);
end and_2;
```

Figure 3: Extrait de code VHDL

entity		in
and_2		bit
is		;
port		c
(		:
a		out
:		bit
in		)
bit		;
;		end
b		and_2
:		;

Figure 4: Liste chaînée résultant de l'analyse lexicale

Ici on peut donc voir la décomposition du fichier sous forme de liste chaînée avec un lexème par élément de la liste.

2. Analyseur syntaxique

L'analyse de la syntaxe est la seconde étape majeure pour vérifier un fichier VHDL, elle vient à la suite de l'analyse lexicale et a pour rôle de vérifier le bon enchaînement des instructions écrites avant de fournir un arbre de structure de données.

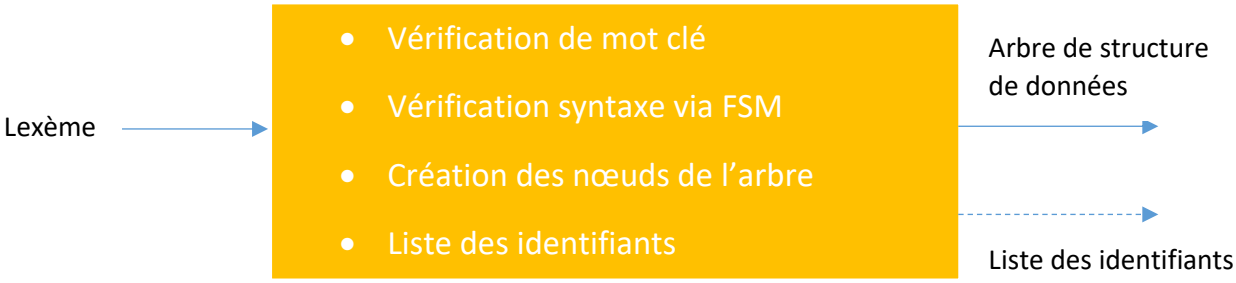


Figure 5: Module d'analyse syntaxique

1. La vérification de mot clé

En VHDL comme dans d'autres langages il est fondamental que le nom donné à un élément par l'utilisateur soit différent des mots clés du langage, auquel cas il y aurait des soucis de répétition et des erreurs apparaîtraient.

Le rôle de cette fonction va donc être de vérifier que les identifiants d'entités, d'architecture etc… ne soient pas des mots clés en comparant un lexème et une liste des mots clés en entrée.

Il serait par exemple interdit et le fait d'écrire le code suivant retournerait une erreur :

ENTITY case IS … -> Erreur liée à l'identifiant appelé « case » qui est dans la liste des mots clés.

2. La vérification de syntaxe via FSM

Cette fonction aura pour but de vérifier la syntaxe, c'est-à-dire l'enchainement des lexèmes. En effet en VHDL il y a une succession logique des lexèmes et on ne peut pas écrire une instruction dans l'ordre qu'on le souhaite, des règles strictes imposent un ordre précis.

La FSM va donc être utile car elle va imposer un ordre, elle va décrire toutes les possibilités pour passer d'un point A à un point B et à chaque état elle va tester le lexème d'après pour passer à un état suivant.

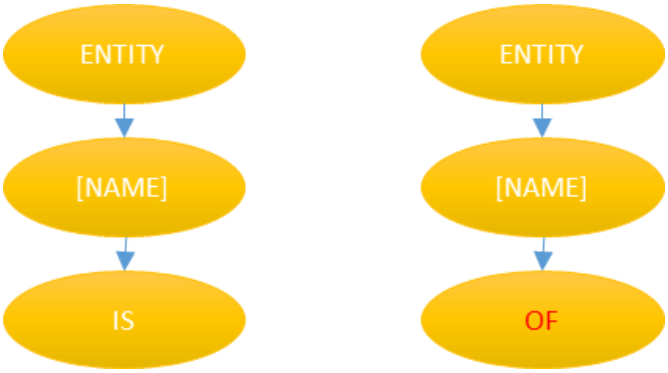


Figure 6: Exemple sans puis avec erreur de syntaxe

Dans le premier cas on n'a pas d'erreur car la suite logique à une entité puis à son identifiant est le mot clé « if » mais dans le deuxième cas il y a bien une erreur car « of » ne fait pas partie de la structure de l'entité.

3. Création des nœuds de l'arbre

Cette fonction est la fonction clé pour la création de l'arbre de structure de données, pour chaque lexème on va associer une balise afin de générer un fichier XML. L'avantage du XML est que les instructions y sont bien rangées et elles sont toutes reliées par des balises qui permettent de se rapprocher parfaitement du déroulement d'un code VHDL.

Par exemple dès que le lexème « architecture » sera détecté il s'agira de créer un nœud et donc une balise XML sous laquelle sera présent son identifiant et ainsi de suite.

4. Liste des identifiants

Le rôle premier de cette fonction sera de stocker chaque identifiant détecté pour simplifier la recherche de ces identifiants durant la phase de pré-synthèse.

Ces identifiants seront ainsi classés dans un conteneur avec à la fois l'identifiant et son adresse pour pouvoir le retrouver et l'exploiter.

IV. Génération de la Netlist

Après avoir généré un arbre sous la forme d'un fichier XML dans la première partie du projet, l'objectif est de transformer cet arbre en une netlist. Comme dit plus tôt, cette netlist décrit l'ensemble des portes logiques, des nets ou des ports nécessaires pour réaliser la fonction décrite dans le fichier RTL de base.

Nous allons avoir comment générer cette netlist en trois étapes :

- Premièrement, il est nécessaire de savoir que toutes les descriptions RTL ne sont pas synthétisables en une netlist. En effet, des règles différentes de la simple syntaxe du code sont à respecter. Ceci implique une étape de pré-synthèse où on testera ces règles une à une.
- Ensuite, une fois que nous sommes sûrs que toutes les règles sont respectées, l'étape de synthèse pure peut être réalisée. A la fin, une netlist est obtenue.
- Enfin, la dernière étape consiste à calculer/ étudier les différents timings au sein du circuit. A la fin, l'outil doit être capable de fournir un rapport sur le chemin critique.

Ces trois étapes vont être plus détaillées ci-dessous.

1. Pré-synthèse du circuit

Nous allons voir ici qu'elles sont les règles à vérifier et comment s'y prendre pour qu'à la fin, nous soyons sûrs que la synthèse est bien réalisable.

1. Les règles à respecter

Ces différentes règles, énoncées dans le sujet du projet, sont des règles contextuelles : au-delà de la syntaxe, on vérifie dans quel cadre sont utilisées les différentes instructions (affectations, initialisations …).

Voici la liste des différentes règles à respecter :

- 1 : Un signal ne peut être affecté que dans un seul et unique process. En effet, il est impossible d'avoir un signal à deux entrées.
- 2 : Les logiques combinatoires et séquentielles doivent être bien séparées dans des process différents. Autrement, la synthèse s'effectuerait mais le fonctionnement du circuit pourrait être incorrect avec certaines fonctions asynchrones réalisées de manière synchrone par exemple.
- 3 : Les différents signaux lus dans un process doivent être listés dans la liste de sensibilité. Bien que pouvant fonctionner correctement dans certains cas, ne pas respecter cette règle pourrait entraîner involontairement des lectures de signaux de manière conditionnelle.
- 4 : Un signal ne peut être lu et affecté dans un process.
- 5 : Toutes les sorties d'un process doivent être affectées, afin de ne pas générer de mémorisation et donc des latchs lors de la synthèse.

- 6 : Certaines instructions ne peuvent pas être utilisées, comme celles gérant directement le temps.
- 7 : Vérifier la séparation entre partie commande/ partie opérative pour aider la synthèse.
- 8 : Les variables doivent être initialisées avant d'être lues.

2. Implémentation des différentes règles

Pour que la synthèse soit réalisable, toutes les règles vues précédemment doivent être respectées. Si ce n'est pas le cas, le « bloc de pré-synthèse » qui aura effectué les tests devra renvoyer une erreur et ne pas lancer la synthèse.

Voici quelques pistes sur la méthode de test qui sera utilisée pour chaque règle :

- 1 : Pour chaque signal, on parcourra tout le fichier XML de base et on stockera ses différentes affectations. Si deux affectations sont faites dans deux process différents, alors on renverra une erreur.
- 3 : Pour chaque process, les signaux de la liste de sensibilité seront stockés. De même pour les signaux lus durant le process. Une comparaison sera ensuite effectuée entre ces deux listes, et un warning sera renvoyé si elles sont différentes. Un warning sera utilisé ici car la synthèse pourra être effectuée, mais avec des risques que le fonctionnement ne soit pas celui voulu.
- 4 : Pour chaque process, une liste des signaux lus sera établie, ainsi qu'une des signaux affectés. Une erreur sera renvoyée si un ou plusieurs signaux sont dans les deux listes.
- 5 : Dans le fichier XML, on vérifiera que chaque signal ait une valeur affectée. Si ce n'est pas le cas, on renverra un warning.
- 6 : Les instructions non-autorisées seront indiquées au préalable, lors de la vérification syntaxique.
- 8 : Pour chaque variable, on vérifiera que le moment de l'initialisation précède le moment de la première lecture. On renverra sinon un warning.

Une fois les différents tests effectués, si aucune erreur empêchant la synthèse n'a été détectée, on passe à l'étape de synthèse.

2. Synthèse du circuit

Après que de nombreuses vérifications aient été effectuées au préalable, la fonction cœur du synthétiseur doit être réalisée.

Dans le cadre de ce projet, il a été choisi de réaliser une synthèse dirigée par la syntaxe : concrètement, cela signifie que pour un morceau de code défini, un ensemble de portes logiques sera associé. Les portes logiques disponibles sont décrites dans un fichier fourni dans le cadre de ce projet.

Prenons quelques exemples très simples :

- A <= B XOR C; -- En admettant A, B et C sur 1 bit, on associerait à un bout de code comme celui-ci une porte XOR à deux entrées (B et C) et avec A comme sortie.
- If Y = 1 then
  Z = 0 ;

```
Else
        Z = 1 ;
End if ;
```

-- Ici, la structure conditionnelle if – else peut être traduite par un simple multiplexeur à deux entrées 1 bit (0 et 1), un sélectionneur 1 bit (Y) et une sortie 1 bit (Z).

Cette démarche de remplacement par des cellules logiques ne peut fonctionner correctement que si le code est « découpé » en petits morceaux, qui peuvent ensuite être reconnus. Voici donc au final les différentes fonctions qui devront être codées pour obtenir une synthèse la plus juste possible :

- Identification des différents blocs élémentaires du code ainsi que de leurs entrées et sorties,
- Remplacement de ces blocs par leurs équivalents en cellules logiques,
- Connection des différents blocs entre eux pour former le circuit global

Tout ceci devra être écrit directement en VHDL dans un fichier pour constituer la netlist. Une quatrième fonction d'optimisation peut être envisagée selon le déroulement du projet, par exemple dans le cas où deux blocs connectés peuvent être remplacés par un seul plus optimisés.

Une fois la synthèse réalisée, nous avons donc une netlist disponible, sur laquelle certains calculs de timing peuvent être effectués.

3. Calcul du chemin critique

Pour chaque cellule logique fournie, ses caractéristiques sont également données. A partir de là, il est possible de déterminer le temps de traversée de chacune, et donc le temps sur les différents chemins (en négligeant ici le temps de traversée des nets, les effets dus à la physique…).

Il existe dans un circuit quatre types de chemins différents :

- Reg to reg, c'est-à-dire un chemin partant d'un registre jusqu'à un autre registre,
- Reg In, d'une entrée du circuit (input)jusqu'à un registre,
- Reg Out, d'un registre jusqu'à une sortie (output) du circuit,
- Et enfin In Out, entre une entrée et une sortie seulement reliées par de la logique combinatoire.

Le calcul de ces chemins s'effectuera de manière récursive :

- On fera tous les chemins possibles en partant d'une entrée du circuit (Reg In ou In Out), et ceci pour chaque entrée,
- Puis on fera tous les chemins possibles en partant d'un registre (Reg Out ou Reg to Reg).

Au final, une liste des temps des différents chemins sera disponible et il sera facile d'en déterminer le(s) chemin(s) critique(s), avec le détail dans un rapport des différentes cellules parcourues, des temps pour chaque cellule …

A la fin de cette étape, la netlist n'a donc pas changé mais on a à présent plus d'informations concernant le timing du circuit. Il est donc possible de réaliser les étapes suivantes de la conception du circuit, qui ne sont pas traitées dans le cadre de ce projet.

V. Organisation

1. Planning Prévisionnel

Voici ci-dessus la répartition des tâches au cours du temps pour l'ensemble du projet. Ceci nous permet d'avoir une vision rapide de l'avancée du travail à n'importe quel moment du projet.

	S38	S39	S40	S41	S42	S43	S44	S45	S46	S47	S48	S49	S50	S51	S52	S1	S2	S3	S4	S5*	S6*
Compréhension du sujet																					
Organisation																					
Parser : Analyse Lexcical																					
Parser: Analyse Grammatical																					
Synth: Analyse Contextuelle																					
Synth: Synthèse																					
Synth: Chemin Critique																					
Assemblage Parser / Synth																					
Validation Projet Complet																					
Optimisation																					

	Temps Consacré
S38	Semaine à PHELMA
S42	Semaine en Entreprise
S5*	A rendre/ Examen

Figure 7: Planning prévisionnel

Nous avons choisi de développer en parallèle le parser et le synthétiseur. Ceci implique une période d'organisation plus longue, pour que les transitions entre les deux parties soient bien définies (le choix de l'utilisation d'un fichier XML à la fin du parser par exemple).

Si globalement, chaque personne sera affectée au synthétiseur ou au parser, l'objectif est tout de même d'avoir un maximum de communication au sein du binôme, pour apporter des points de vue différents et ainsi être le plus efficace possible. Il est donc possible que ce planning varie légèrement selon l'avancée du projet et les problèmes rencontrés.

2. Stockage et Organisation des fichiers

Pour la gestion des fichiers, il a été choisi d'utiliser d'outil GitHub. Il nous permettra de stocker en ligne tous nos fichiers, d'y avoir accès n'importe où, de travailler à plusieurs sur un même fichier mais surtout, il garde en mémoire les différentes versions des fichiers. En cas de modifications non satisfaisantes, il est donc possible de récupérer un ancien fichier.

Notre choix s'est également porté sur cet outil car étant de plus en plus utilisé, ceci nous sera sûrement utile plus tard en entreprise ou pour d'autres projets.

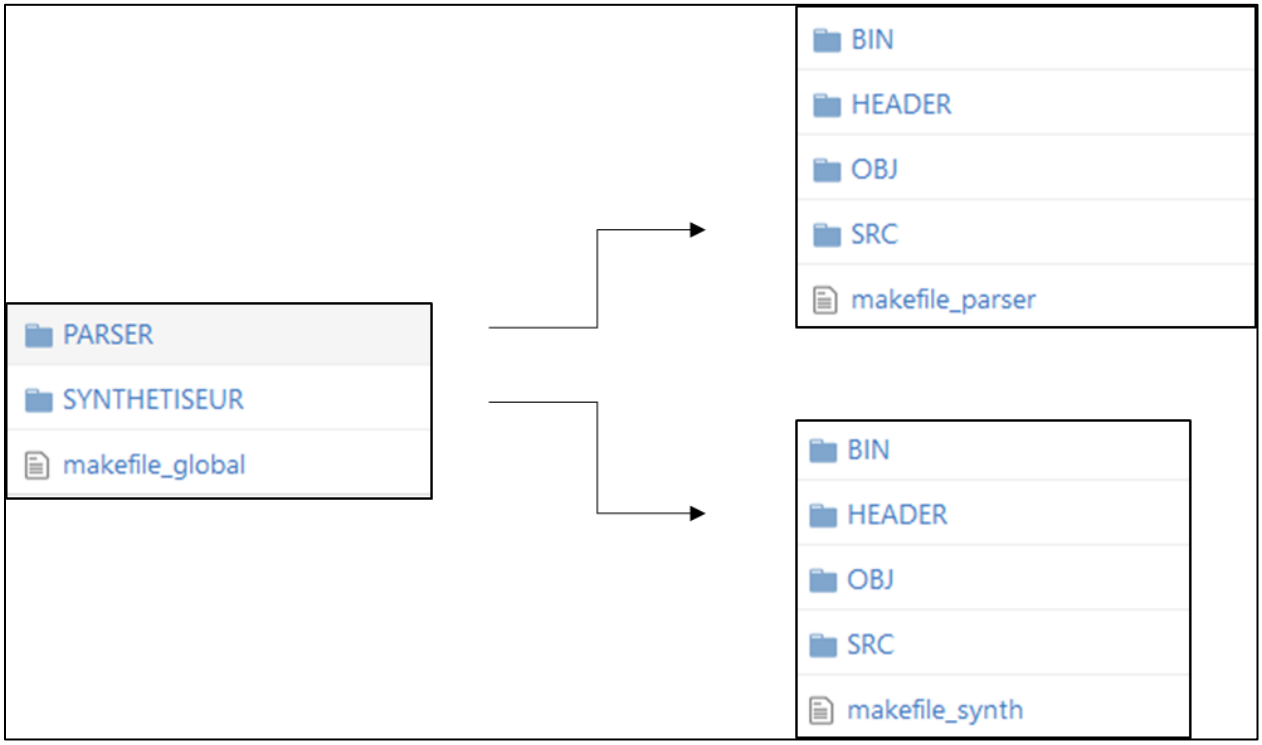


Figure 8: Arborescence des fichiers

Voici ci-dessus l'arborescence des fichiers utilisée. Les parties PARSER et SYNTHETISEUR sont bien distinctes, pour que les fichiers soient le mieux rangés possible. Différents dossiers ont été créés :

- BIN pour les fichiers binaires,
- HEADER pour les fichiers hearder, les .h,
- OBJ pour les fichiers objet, les .o,
- Et SRC pour les fichiers sources, les .cpp.

Il a également été choisi d'utiliser des makefiles pour réaliser les compilations : un au sein de chaque partie et un global pour la fin du projet. Cet outil permet de réaliser des commandes avec à chaque fois les dernières versions des différents fichiers.

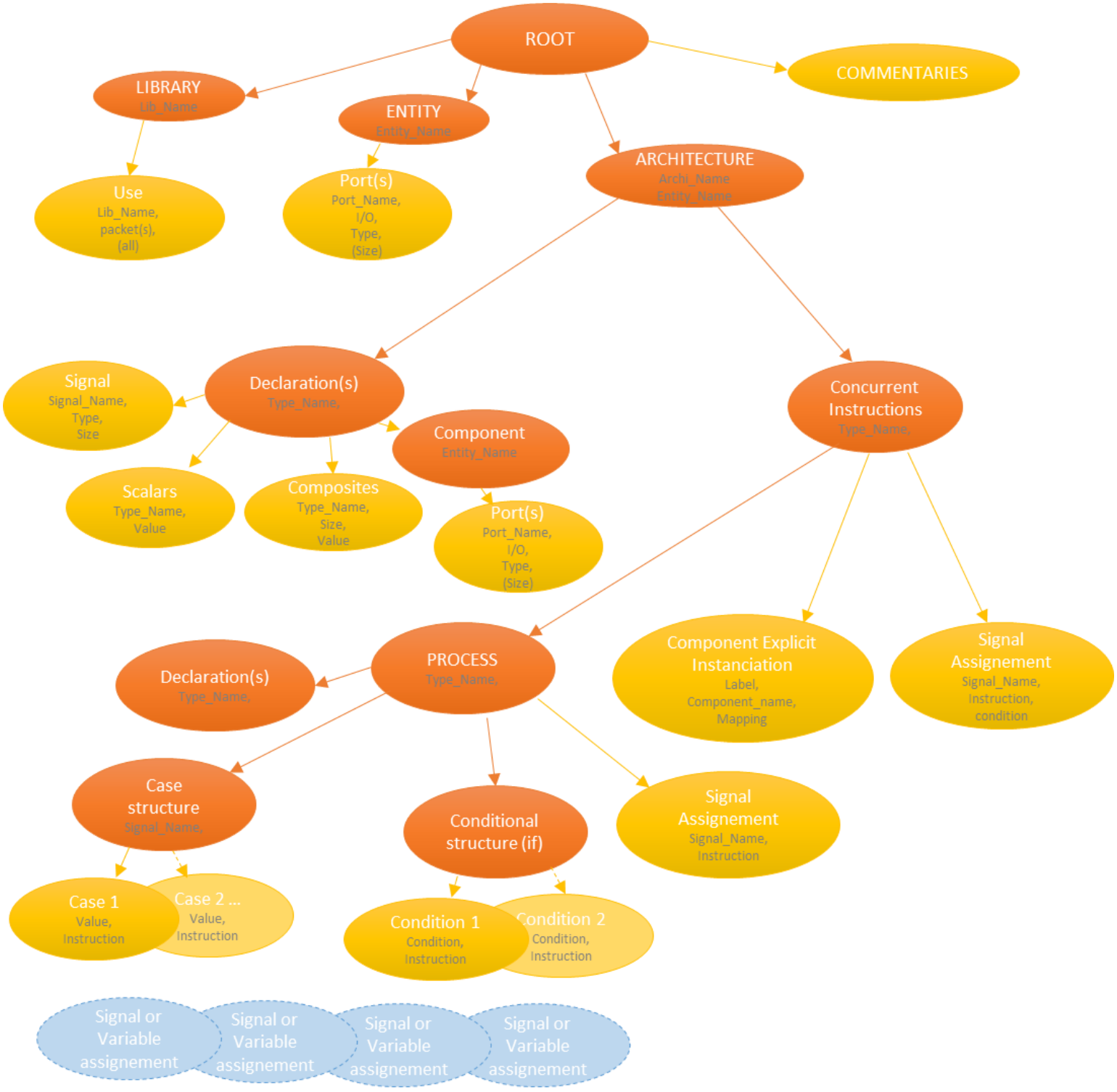
VI. Conclusion

Ce rapport avait pour objectif de nous aider à mieux comprendre le projet dans sa globalité, que ça soit d'un point de vue technique ou au niveau du travail demandé (temps, organisation …). Ainsi, il est plus facile de le découper en plusieurs parties et de s'organiser au sein du binôme.

C'est également l'occasion de commencer à mettre le doigt sur certains problèmes que nous rencontrerons, que ça soit sur la compréhension de certaines parties du sujet, ou alors sur la liaison entre le travail réalisé par les deux personnes. Par exemple, il est essentiel de définir dès le départ le format de l'arbre à la fin du parser, car il est réutilisé dès le début de la partie synthétiseur.

Plus globalement, ce rapport fût l'occasion de réfléchir à la gestion d'un projet d'une envergure bien plus importante que ce que nous avons pu faire jusqu'à présent au sein de notre formation.

VII. Annexes


ROOT
COMMENTARIES
LIBRARY
Lib_Name
ENTITY
Entity_Name
ARCHITECTURE
Archi_Name
Entity_Name
Use
Lib_Name,
packet(s),
(all)
Port(s)
Port_Name,
I/O,
Type,
(Size)
Signal
Signal_Name,
Type,
Size
Declaration(s)
Type_Name,
Component
Entity_Name
Concurrent Instructions
Type_Name,
Scalars
Type_Name,
Value
Composites
Type_Name,
Size,
Value
Port(s)
Port_Name,
I/O,
Type,
(Size)
Declaration(s)
Type_Name,
PROCESS
Type_Name,
Component Explicit Instanciation
Label,
Component_name,
Mapping
Signal Assignement
Signal_Name,
Instruction,
condition
Case structure
Signal_Name,
Conditional structure (if)
Signal Assignement
Signal_Name,
Instruction
Case 1
Value,
Instruction
Case 2 ...
Value,
Instruction
Condition 1
Condition,
Instruction
Condition 2
Condition,
Instruction
Signal or Variable assignement
Signal or Variable assignement
Signal or Variable assignement
Signal or Variable assignement