

**Grenoble INP - PHELMA - 2A CSI**

2016

**Projet C++ : Synthétiseur VHDL**

**Rapport d'analyse préliminaire**

**TAGORTI Hassan, CINçon Valérian**

Table of Contents

[I. Introduction 3](#_Toc463713800)

[II. Présentation du projet 4](#_Toc463713801)

[Description modulaire du programme 5](#_Toc463713802)

[Structure de données 6](#_Toc463713803)

[III. Description des modules 8](#_Toc463713804)

[IV. Méthodologie de test 9](#_Toc463713805)

[V. Arborescence UNIX 10](#_Toc463713806)

[VI. Planning 11](#_Toc463713807)

[VII. Conclusion 12](#_Toc463713808)

# Introduction

En tant qu’étudiant en conception des circuits intégré, la métrise des langages de description matérielle est primordiale. Mais développer son propre compilateur VHDL c’est mieux ! Et cela été l’objectif de notre projet informatique de cette année.

Ce projet consiste à développer un programme en C/C ++ dont le but est de synthétiser une machine à état décrite en VHDL sous forme d’une netlist VHDL. Plus précisément notre programme doit transformer un fichier, écrit en langage informatique compréhensible par l’Homme(VHDL), en un ensemble de portes logiques. Sans modifier le comportement du circuit. Voir figure 1.

|  |
| --- |
|  |
| Figure : Le principe du fonctionnement du programme |

Ce projet s’étalera sur le premier semestre de cette année universitaire. Pour nous c’est une première de ce lancer dans un projet informatique d’une telle ampleur. Après quelques lectures du cahier de charge. Nous avons déduit qu’il est primordiale de mettre en place bonne méthode de travail vu la quantité de code que nous serons mené à écrire.

Les objectifs de ce projet sont multiples. En faisant le lien avec le cours de VHDL et circuit logique, ce projet nous permettra de mieux comprendre le concept de la synthèse matérielle. Une technique que nous avons utilisée l’année dernière, lors des TP VHDL, sans savoir comment elle fonctionne. De plus c’est une opportunité pour tester et améliorer nos connaissances en langage C/C++. A coté de tous ces objectifs techniques, nous aborderons aussi des notions de gestion de projet et du respect du planning mis en place.

Vous trouverez dans ce premier rapport la méthode de travail que nous suivrons, les différent outils et techniques que nous utiliserons afin de bien manager ce projet. Mais aussi une première solution qui vous expliquera les différentes décisions et « manière de faire » que nous avons prises de point de vu technique (architecture, arbre…).

# Présentation du projet

Comme il était mentionné précédemment, la fonction principale de ce programme est de produire une traduction plus bas niveau d’un fichier VHDL décrivant une machine à états au niveau RTL. Mais avant d’arriver à la synthèse et la création d’arbre. Le programme doit passer par deux étapes importantes :

* L’analyse lexicale
* Analyse syntaxique
* L’analyse contextuel
* Analyse de synthèse
* Pré-synthétisation
* Synthèse

La méthode que nous suivrons dans ce projet consiste à diviser le problème principale en un ensemble de sous problème facile à résoudre et fournissant des données intermédiaires vérifiables. Pour chaque solution programmée on associe un ou plusieurs tests. Ces tests peuvent aussi être utilisés pour vérifier la non régression de notre programme suite à une modification.

On pense aussi à utilisé des fichiers intermédiaire de format xml dont l’objectif est de facilité l’échange de informations entre les différents modules qui composent notre programme et nous fournir des données à utiliser lors des vérifications.

Au niveau de l’efficacité  temporelle, nous allons mettre en place des scripts CShell et pourquoi pas des variables d’environnement afin d’automatiser la compilation de nos programmes et l’écriture des résultats dans le bon répertoire.

## Description modulaire du programme

Nous avons décomposé le programme en six modules suivants, les exemples pratique sont fournis au chapitre description des modules.

* Le lexeur découpe le fichier texte en tableau de lexèmes, vérifiant au passage leur validités et leur nature remplissant un deuxième tableau qui identifie si un lexème est un nombre, un mot réservé ou un identifiant.
* Le synthaxeur fournis un arbre primaire utilisé pour les vérifications, il utilise pour sa construction des protocoles encapsulé déclenché par les lexèmes étant des mots réservés ainsi que la librairie tree.hh qui contient toute les méthode nécessaire.
* L'analyse contextuel vérifie chaque lexème par rapport à ses voisins dans l'arbre (parent, enfants, frères) et avec l'aide du protocole encapsulé en cours d'utilisation et du tableau de tag "nature" définit si ses voisins sont valide. Il produit aussi un nouveau tableau de tag "rôle" qui identifie en autre si un identifiant est utilisé comme signal assigné ou assignant. Pour cela on ce réfère au différent règles d’écriture imposer par la norme 93 de VHDL.
* L'analyse de synthèse vérifie si le code VHDL rentrée par l'utilisateur est synthétisable selon les neufs règles énuméré au chapitre 4.2
* Le pré-synthétiseur convertie l'arbre ayant servie à effectuer facilement des vérifications vers un arbre plus adapter à sa transposition vers une description structurel
* Le synthétiseur génère le fichier VHDL structurel, instanciant les composants nécessaire et créant, les signaux et les ports map. Dans un second temps il calculera le chemin critique du system.

-- Arbre vérifié

-- Tableau de tag "rôle"

-- Arbre primaire

-- Arbre sous format XML

Fichier VHDL Comportementale

-- Tableau de lexème

-- Tableau de tag "nature"

-- Arbre vérifié synthétisable

-- Arbre synthétisable

-- Arbre sous format XML

-- Fichier VHDL Structurel

--Rapport chemin critique

Figure : Description modulaire du programme

## Structure de données

Pour la réalisation de nos arbres nous comptons utiliser le librairie tree.hh et qui fournis clé en main une classe de conteneur correspondant à un arbre, ainsi que toute les fonctions nécessaire pour le créer, l'éditer et naviguer à l'intérieur de celui-ci.

|  |
| --- |
| arbre.png |
| Figure : Le principe d'organisation du conteneur tree.hh |

Nous comptons structuré les données sous la forme de deux arbres, chacun spécialisé dans leur organisation afin de faciliter leur exploitation, aux étapes de vérification et à la synthèse.

Le premier arbre sera très proche de l'aspect d'un code VHDL correctement indenté et servira pour vérifier que celui ci est correctement écrit ainsi que pour identifier le rôle des lexèmes suivant leurs placements. Il est fournis par l'analyse synthaxique.

Par exemple le même identifiant ne signifie pas la même chose s'il correspond à un signal assigné ou au contraire à un signal assignant (à gauche ou à droite du symbole "<=").

Le deuxième arbre servira purement à la synthèse pour l'instanciation des éléments logiques et surtout pour le câblage des signaux. il sera mis en forme à partir du premier arbre. Il est fournis par le pré-synthétiseur.

Nous avons fais ce choix de deux structure spécialisé plutôt qu'une seule pour faciliter notre implémentation, en effet il est à prévoir qu'une structure particulière est plus ergonomique à une utilisation qu'une autre et nous ne somme pas sure de trouver une structure ergonomique dans tous ses usages.

Ci dessous sont présentés des exemples de nos structure de données

|  |
| --- |
| exemple.png |
| Figure : code VHDL d'exemple |

|  |
| --- |
| D:\Document Adm\Phelma\2ieme annee\Projet C\arbre primaire.png |
| Figure : Organisation proposé de l'arbre primaire pour la vérification |

|  |
| --- |
| D:\Document Adm\Phelma\2ieme annee\Projet C\arbre secondaire.png |
| Figure : Organisation proposé de l'arbre secondaire pour la synthèse |

## Arborescence UNIX

Ce projet étant notre première expérience nous nous somme inspirés de conseils sur des forums tout en gardant en tête que cette arborescence doit nous faciliter l'implémentation, et la modification du programmes nous permettant un accès rapide et logique au données cherchées.

|  |
| --- |
| D:\Document Adm\Phelma\2ieme annee\Projet C\arborescence UNIX.png |
| Figure : Arborescence UNIX proposé |

## Planning

Ci-dessous est présenté notre proposition de planning dans un format Gantt, nous avons répartie les modules en nombre égaux et de façons à ce que chacun ai à définir à arbre (analyse contextuel et synthèse) et à l'implémenter (analyse synthaxique et pré-synthétisation).

|  |
| --- |
| D:\Document Adm\Phelma\2ieme annee\Projet C\Gantt prévisionnel.png |
| Figure : Arborescence UNIX proposé |

# Description des modules

## Analyse lexicale : Décoder le fichier source

Cette étape consiste à décoder le fichier source. Ce dernier est de format VHDL niveau RTL. Il peut être considérer donc comme un fichier texte. Il est donc composé d’une suite de lexèmes séparé dans la plus part de temps par le caractère espace. Mais ça arrive dans le langage VHDL que le séparateur ne soit pas un espace. Par exemple on peut ne pas utiliser les espaces lors de la précision de la taille d’une variable, un port ou un signal de type « **std\_logic\_vector** ».

|  |
| --- |
|  |
| Figure : parenthèse utilisé comme séparateur |

Pour cela nous avons fait une recherche sur les différents caractères spéciaux qui peuvent être considérer comme séparateur.

La deuxième chose à prendre en compte c’est les commentaires. En fait en VHDL les commentaires commence toujours par le caractère « -- » et se termine à la fin de la ligne. Un commentaire peut commencer soit au début de la ligne soit après une instruction.

Pour parcourir le fichier source on peut se baser sur une suite de fonction qui existe dans le langage C. Par exemple on peut utiliser la fonction « fgets » pour lire notre fichier source mot par mot. Vous trouvez dans le tableau si dessous la liste des fonctions que nous utiliserons peut être pour parser le fichier source.

|  |  |
| --- | --- |
| Nom de la fonction | Rôle |
| fopen | Ouvrir un fichier |
| fclose | Fermer un fichier |
| fgetc | Lire caractère par caractère |
| fgets | Lire mot par mot |
| ftell | Donner votre position dans le fichier |
| fseek | Positionner le curseur dans un endroit précis |
| rewind | Remet le curseur au début du fichier |
| Tableau : Différentes fonctions de traitement de fichier texte en C | |

On utilisera les fonctions de lecture en boucle « while » jusqu'à sa condition n‘est plus vrai. C'est-à-dire la fonction rencontre le « eof ».

Le résultat de cette analyse peut être présenté sous deux manières. Soit on fait un tableau de correspondance dans le quel on associe à chaque lexèmes sa nature (Voir tableau ci-dessous), soit par la génération d’un fichier texte basé sur le même concept.

|  |  |
| --- | --- |
| Type | Std\_logic |
| Entité | Entity |
| Identifiant | Toto |
| Is | Is |
| Port | Port |
| Parenthèse | ( |
| etc |  |
| Tableau : tableau de correspondance nature-lexème | |

Gérer les commentaires :

Comme nous l’avons indiqué précédemment, un code VHDL peut contenir des commentaires. Ces derniers sont utilisés pour faciliter la compréhension du code pour d’autre personne ou même pour le développeur lui-même dans le cas où il reprend son code après un certain moment. Ce que nous voulons dire c’est que les commentaires ne sont pas pris en compte par le compilateur.

Nous allons donc les négliger. Pour cela ci on rencontre le symbole « -- »lors de la lecture caractère par caractère de notre fichier on saute à la ligne suivante.

Gérer les majuscules :

Dans le langage VHDL on ne fait pas la différence entre les lettres majuscule et les minuscules. Et pour éviter tout problème de conflit ou de détection de lexème lors de la génération d’arbre. Nous allons transformer toute lettre majuscule en minuscule. Une lettre est en majuscule si son code ASCII est entre 65 et 90 en décimale.

Les règles à vérifier :

A coté des divers mots clés, une description en VHDL nécessite un nommage de différentes entités, architectures, port. Dans notre projet ces nom son associé au type Identifiant. L’écriture de ces identifiants doit respecter certaines règles :

* Un identifiant ne commence/ se termine jamais par un Under score
* Le premier caractère doit être une lettre
* Un identifiant ne peut pas contenir des caractères spéciaux « ( ) [ ] / \ etc »

Gérer les Delay.

Les fonctions « wait » et « after » ne sont pas synthétisable. On va donc chercher si ces lexèmes existent dans le code source. Si on en trouve on arrête tout et un message d’erreur s’affichera pour informer l’utilisateur que son code n’est pas synthétisable.

Comme vous l’avez remarqué, un code VDHL peut contenir des informations qui ne seront pas utilisé par le compilateur, les lexèmes ne sont pas toujours séparer par des espaces. De plus les identifiants doivent respecter certaines règles d’écriture et le plus important le code doit être synthétisable.

🡺 Beaucoup de critère à vérifier !!!

Nous avons donc décidé de réaliser cette étape de classement de lexèmes sur deux étapes. L’idée est de générer à partir un fichier source, un fichier dont il y a plus de commentaire, où tout les lexèmes sont séparer par des espaces « une parenthèse est un lexème», tous ces caractères sont en minuscule. Ce fichier sera générer si le code source ne contient pas les fonctions non synthétisable « wait et after » et si on respecte les régles d’écriture mentionné auparavant.  Cette idée est asé sur la politique que nous avons choisi de suivre pour ce projet. Diviser un gros problème en un ensemble de sous problèmes dont leurs solutions résout le notre problème de départ.

|  |
| --- |
|  |
| Figure 10: première étape du lexeur |

Si cela s’effectue correctement, on peut maintenant parcourir notre fichier standardisé avec les fonctions existantes de C/C++ sans les modifier. Je telle méthode nous permettra de d’optimiser le temps de travail. On est conscient que durant le développement de cette fonction il faut bien gérer l’aspect mémoire. Même si les fichiers texte ne sont pas aussi volumineux.

Au moment où on parcourt notre fichier lexèmes par lexèmes. Nous stockerons ces derniers dans un tableau. à la fin de cette étapes on aura un tableau de lexèmes classés dans le même ordre comme il ont été écrit.

L’étape de tag : l’étape de tag consiste à parcourir tous les cases de ce tableau. Et comparer le contenu avec une base donnée. Notre base de données doit contenir tout les mots clés du langage VHDL. Suite a cette comparaison on associe un tag à chaque lexèmes.

## Analyse syntaxique : Constituer l'arbre primaire

Le synthaxeur fournis un arbre primaire utilisé pour les vérifications, il utilise pour sa construction des protocoles encapsulé déclenché par les lexèmes étant des mots réservés ainsi que la librairie tree.hh qui contient toute les méthode nécessaire.

Les données de l'analyse syntaxique sont un tableau de lexème ordonnée par ordre de présence dans le fichier .vhd ainsi que son tableau de tag nature, pour décider du placement de chaque lexème dans l'arbre nous allons plutôt considérer sa nature comme présenté dans l'arbre ci dessous.

|  |
| --- |
| D:\Document Adm\Phelma\2ieme annee\Projet C\arbre primaire_tag_nature.png |
| Figure : Arbre sous l'aspect des Tags nature |

Par protocole j'entend tout simplement une fonction qui placera le lexème considéré soit en frère (même niveau que le lexème précédent) soit en enfant et tout processus décisionnel spécifique au protocole lui-même activé par un mot clé.

Exemple:

On commence par le protocole "default" qui insère le premier lexème dans l'arbre puis le test pour savoir s'il déclenche un autre protocole ou reste en default, en default chaque nouveau lexème est placé en tant que frère du précédent.

les mots clés "library" et "use" vont déclencher un protocole qui place tout les lexème suivant en enfant du précédent jusqu'à ce que l'on rencontre le lexème ";" l'on revient alors à la racine de l'arbre.

A chaque lexème on test si celui ci doit lancer un protocole, ainsi ceux-ci s'imbrique, le protocole actuel étant 'pausé' tant que l'on est pas retourné du nouveau protocole (une fonction) lancé.

Cette façons de faire a été pensé pour gérer l'imbrication architecture - process -if -if -if...

Le protocole if est prévue pour gérer son test puis ses instructions et se termine à son "end if;" il peut tout à fait contenir dans ses instruction un autre if qui doit relancer à zéro le protocole tout en gardant mémorisé l'exécution du protocole précédent afin de le reprendre à l'endroit laissé.

## Analyse contextuel : Vérifier l'écriture VHDL

L'analyse contextuel vérifie chaque lexème par rapport à ses voisins dans l'arbre (parent, enfants, frères) et avec l'aide du protocole encapsulé en cours d'utilisation et du tableau de tag "nature" définit si ses voisins sont valide. Il produit aussi un nouveau tableau de tag "rôle" qui identifie en autre si un identifiant est utilisé comme signal assigné ou assignant. Pour cela on ce réfère au différent règles d’écriture imposer par la norme 93 de VHDL.

De manière très similaire à l'étape précédente nous itérerons notre vérification sur l'indice du tableau de lexème et déclenchons des protocoles de vérification spécifique sur mots clé.

Ainsi le premier protocole default vérifie que le premier étage (les frères) de l'arbre n'est composé que de mots clé tels que "library", "use", "entity", "architecture".

Lors que l'on déclenche de protocole 'port' nous vérifions qu'il n'a pas de frère et un seul enfant "(", que cette parenthèse n'a que des enfants "Identifiant" excepté son dernier enfant qui doit obligatoire être un ")".

Le protocole définira les conditions à tester pour une nature de lexème, par exemple dans le protocole port lors que le lexème 'scruté' est un identifiant on vérifie que son parent est "(" ou "," et son enfant unique est ":" ou ",". En testant ainsi chaque lexème par rapport à ses voisins immédiat nous pensons atteindre une facilité d'implémentation de l'étape de vérification.

En dernier exemple lors du protocole 'signal' les identifiants seront vérifié selon des conditions différentes le lexème "(" ne sera plus accepté comme un parent possible, mais "signal" le sera.

Enfin après avoir vérifié l'arbre nous referons un passage pour précisé les rôles de certain lexème, déduit de leur position dans l'arbre, par exemple si celui-ci correspond à un signal assigné ou un signal assignant.

Ce nouveau tableau de rôle (ou arbre sous l'aspect rôle) est prévue pour nous faciliter la vérification de synthétasibilité.

|  |
| --- |
| D:\Document Adm\Phelma\2ieme annee\Projet C\arbre primaire_tag_role.png |
| Figure : Arbre sous l'aspect des Tags rôle |

## Analyse de synthèse : Vérifier la synthétasibilité

L'analyse de synthèse vérifie si le code VHDL rentrée par l'utilisateur est synthétisable selon les neufs règles énuméré au chapitre 4.2

Pour chaque règle sera crée une fonction destiné à la vérifié et renvoyer un bit à un si elle est vérifié sinon zéro et envoie de message d'erreur.

* Logique séquentielle et logique combinatoire doivent être implémentées dans des process séparés.

On vérifie dans l'arbre que tout lexème correspondant à une instruction séquentielle a comme l'un de ses ancêtres (parent et leurs parents) un process.

* Un signal ne peut être affecté que par un seul process.

On vérifie qu'un seul sous-arbre process possède une version d'un lexème identifiant d'un signal avec le tag rôle "signal\_assigné"

* La liste de sensibilité d'un process doit contenir tous les signaux qu'il possède en entrées.

Dans un sous-arbre process on vérifie que dans la branche de la liste de senssibilité sont présent en frère tous les lexemes identifiants qui ont dans ce sous-arbre au moins une fois le rôle signal assignant.

* Seuls les process séquentielles ont le droit d'être évalué sur le front d'horloge.

A FAIRE

* On ne peut pas lire et assigné un signal dans le même process.

A FAIRE

* Un process doit toujours assigné une valeur à tous ses signaux de sorties, sinon cela correspond à la mémorisation du signal par l'instanciation d'une latch.

Dans cette condition la génération d'une latch sera généré à l'étape de synthèse.

* On n'a pas le droit d'utiliser d'instruction temporelle tel que "wait" ou "after".

La détection de ces lexème est effectué à l'étape du lexeur.

* Séparation systématique de la Partie Contrôle et la Partie Opérative.

A FAIRE

* Les variables doivent toujours être initialisé avant d'être lut.

On vérifie si une variable assignant a préalablement été une variable assigné avant le "begin" du process.

## Pré-synthétiseur : Constituer l'arbre secondaire

Le pré-synthétiseur convertie l'arbre ayant servie à effectuer facilement des vérifications vers un arbre plus adapter à sa transposition vers une description structurel.

Pour ce faire nous utiliserons les fonctions décrite au chapitre 3.2 Mutating algorithms de la documentation de la librairie tree.hh

## Synthétiseur : Ecrire en VHDL structurel

A FAIRE

Description de chaque module, ses entrées/sortie, et des fonctions ou protocole qu'il utilise pour sa mise en œuvre

Le synthétiseur génère le fichier VHDL structurel, instanciant les composants nécessaire et créant, les signaux et les ports map. Dans un second temps il calculera le chemin critique du system.

Détermination du chemin critique

# Méthodologie de test

Le test et la validation du code est une partie majeur de tous projets, l'écriture même du code étant généralement bien moins chronophage que ses corrections.

Notre découpage en module et fonction nous autorise une stratégie de vérification très simple, chaque module ou fonction prend une entrée que l'on connais et nous fournis un livrable que nous connaissons aussi, du mois à l'échelle d'échantillons.

Ainsi lors de la phase d'élaboration chaque module ou fonction sera livrée avec ses tests. Ces tests corresponde à l'exécution du module ou fonction avec des données échantillons et le retour sera comparé aux résultats attendus du module ou fonction.

Il est douteux de pouvoir fournir des échantillons exhaustifs de toute les entrées possibles et leurs sorties attendus, lorsque nous pourrons évaluer la quantité de travail que cela représente nous déciderons à quels diversifications nous devons poussez la génération et ces échantillons et leurs résolutions manuelles pour déterminer les livrables correspondants.

En parallèle nous coderons en utilisant les exceptions et de ses trois fonctions try, throw et catch nous pouvons dans chaque fonction gérer l'apparition d'erreur, arrêter proprement l'exécution et fournir un log d'erreur instructif.

|  |
| --- |
| * try{ ...} (en français essaye) signale une portion de code où une erreur peut survenir ; * throw (en français lance) signale l'erreur en lançant un objet ; * catch(...){...} (en français attrape) introduit la portion de code qui récupère l'objet et gère l'erreur. |
| Source : https://openclassrooms.com/courses/programmez-avec-le-langage-c/la-gestion-des-erreurs-avec-les-exceptions |

Cela nous permettra dans le cadre de l'exécution du programme de détecter des erreurs, notamment qu'une fonction "saine" reçois en argument des valeurs incohérentes.

# Conclusion

A FAIRE