PHELMA Apprentis 2ème année - 2015/2016

Projet informatique

SYNTHESE DE CIRCUITS SEQUENTIELS en VHDL

katell.morin-allory@imag.fr, michele.portolan@imag.fr

Table des matières

Chapitre 1

Présentation générale du projet

L'objectif de ce projet informatique est de concevoir puis implémenter en langage C++, sous Linux, un logiciel permettant de synthétiser une machin à états décrites en VHDL sous la forme d'une netlist VHDL **Le projet final doit fonctionner correctement à PHELMA, sous Linux, le jour de l'examen !**

Le rôle d'un outil de synthèse matériel est de transformer un programme écrit dans un langage informatique accessible à l'homme, **le langage VHDL**, en un ensemble de portes décrivant le même circuit mais cette fois au niveau portes, destiné à être soit implémanté sur une carte FPGA, ou placé-routé pour un circuit ASIC.

Le logiciel doit donc prendre en entrée un fichier texte contenant un programme écrit en VHDL, et produire plusieurs sorties :

— un arbre représentant le programme compilé

— un fichier VHDL au niveau porte

Pour ce projet nous considérerons en fait un sous-ensemble de VHDL. Une présentation générale du langage VHDL considéré sera introduit. Le chapitre 4 décrit ensuite les instructions à gérer dans le projet. Les chapitres 5, 6 et 7 présenterons enfin quelques notions sur la manière de programmer un outil de synthèse logique, les spécifications précises du travail demandé, puis des informations sur l'organisation générale du projet.

Les intérêts pédagogiques de ce projet informatique sont multiples. Il permet tout d'abord de travailler sur un projet de taille importante sous tout ses aspects techniques (analyse d'un problème, conception puis implémentation d'une solution, validation du résultat) mais aborde aussi les notions de gestion de projet et de respect d'un planning. Ce projet vous permettra également d'améliorer votre connaissance et maîtrise du langage C++, qui est particulièrement utilisé pour la programmation scientifique et le développement industriel, ainsi que des outils de développement associés (systèmes Unix/Linux, outil Make, débogueur, etc.). Enfin, il illustre et met en pratique les connaissances relatives à la synthèse matérielle, vues notamment dans le cours de logique et de VHDL de première année et dans certains cours d'architecture ou de micro-électronique.

Chapitre 2

Le langage VHDL : déclaration

La syntaxe présentée ici est issue de la norme 93 de VHDL. De nombreuses restrictions de syntaxe ont été apportées afin d'éviter de commettre des erreurs difficiles à détecter.

L'usage des majuscules ou minuscules est indifférencié.

Un programme VHDL se présente comme une liste d'unité de conception. Il est possible (et même fortement recommandé pour aérer le texte) de rajouter des lignes blanches. Les unités de conception sont de quatre types : bibliothèque, entité, architecture ou commentaires.

2.1 Les commentaires

Un commentaire [1] commence sur une ligne par le caractère `--` et se termine à la fin de la ligne.

Exemple :

```
-- ceci est une ligne entièrement commentée
 CLK   <= not(CLK) after 10 ns; -- ici le commentaire commence après l'instruction
```

2.2 Les déclarations de bibliothèques

Elles ont la forme générale suivante, les champs entre crochets étant optionnels :

```
LIBRARY [nom_bibliothèque];
USE [nom_bibliothèque].[nom_paquet].[all|nom_partie];
```

exemple :

```
LIBRARY lib_FILTRE;
USE lib_FILRE.all;
USE lib_FILTRE.FILTRE;
```

1. On ne le répetera jamais assez, abusez des commentaires ! Aussi bien dans vos programmes en VHDL que dans le source C++ de ce projet d'ailleurs...

2.3 Entité

Les entités définissent l'interface d'un composant. Elles sont caractérisées par leur nom, le nom de leur port d'entrées/sorties et le nom de leur paramètres génériques. Pour simplifier le travail, nous supporerons que nos entités n'ont pas de paramètres génériques.

```
ENTITY IS [nom_entité]
PORT( [définition ports])
END [nom_entité];
```

2.3.1 Le champ port

Le champ port permet de déclarer les signaux d'entrées/sorties de votre design. Chaque signal doit être déclaré comme étant une entrée ou une sortie.

```
[nom_signal]: IN|OUT [nom_type];
```

exemple :

```
enable: IN bit;
data_bus: OUT bit_vector(7 downto 0);
```

Exemple de déclaration d'entité :

```
ENTITY or_entity IS
PORT(
   input_1: IN std_logic;
   input_2: IN std_logic;
   output: OUT std_logic
);
END or_entity;
```

2.4 Déclaration d'architecture

Les architectures sont des vues internes d'un circuit. Elles servent à décrire le comportement ou la structure d'un circuit. A chaque architecture est associé un nom et une entité. Sa syntaxe est la suivante :

```
ARCHITECTURE [nom_architecture] OF [nom_entité] IS
[partie declaration]
BEGIN
 [instructions concurrentes]
END [nom_architecture];
```

Exemple de déclaration d'architecture :

```
ARCHITECTURE or_archi of or_entity IS

BEGIN
   output<= input_1 OR input_2;%
END or_entity;
```

2.5 Déclaration de type

Le VHDL permet de traiter une grande variété de types. Ici nous nous limiterons à quelques types prédéfinis synthétisables (bit, bit_vector), et quelques déclaration de types (énumérés et tableaux).

2.5.1 type scalaire

Ce sont des types énumérés. Ils sont ordonnés, on peut donc comparer leurs valeurs à l'aide d'opérateurs relationnels : $<, \leq, =, \geq, \setminus =, >$.

La déclaration d'un type se fait à l'aide du mot clef `type`

```
TYPE couleur IS (bleu, blanc, rouge);
TYPE std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

Pour simplifier le travail, nous supposerons que le type bit est prédéfini.

2.5.2 Types composites : les tableaux

Les types composites décrivent des collections d'objets de même type (les tableaux) ou de type différents (les articles). Dans le cadre de ce projet nous nous restreignons aux tableaux.

Les éléments d'un type tableau sont accessibles au moyen d'un ou plusieurs indices. Un tableau d'une dimension est appelé un vecteur.

Les mots clefs `TO` et `DOWNTO` permettent de préciser le sens de variation des indices.

```
TYPE mot IS (0 TO 31) OF STD_LOGIC;
TYPE drapeau IS (0 TO 2) OF couleur;
```

2.6 Déclaration de composant

Instancier un composant, est le fait de prendre une copie d'un composant décléré et de le personnaliser.

La syntaxe est la suivante

```
COMPONENT nom_du_composant
   PORT (déclaration_des_ports)
END COMPONENT;
```

Chapitre 3

Instructions et Déclarations

Ce chapitre présente le sous-ensemble des instructions de VHDL retenu pour le projet.

3.1 Instructions séquentielles

3.1.1 L'instruction affectation de signal

L'affectation de signal ne change pas la valeur présente d'un signal, mais sa valeur future. Le symbole est `<=`. Nous nous restreignons ici a l'affectation sans delai d'une seule valeur future (pas de forme d'ondes) ;

Exemple :

```
A<= B;
```

3.1.2 L'affectation de variable

L'affectation de variable se note `:='` elle change la valeur actuelle d'une variable. Les variables ne pourront être déclarées que dans des process.

Exemple :

```
x:= y;
```

3.1.3 Structure conditionnelle

Cette structure permet d'exécuter sous condition des instructions sequentielles

Sa syntaxe est la suivante

```
IF cond_bool_1 THEN
   sequence_d_instructions_1
ELSIF cond_bool_22  THEN
   sequence_d_instructions_2
...

ELSE
   sequence_d_instructions_3

ENDIF
```

3.1.4 L'instruction CASE : bonus

Cette instruction permet de sélectionner, en fonction d'une valeur d'expression, une séquence d'instructions parmi plusieurs.

Sa syntaxe est la suivante :

```
CASE expression IS
   WHEN Valeur_1 => ...
   WHEN Valeur_2 => ...
   ...
   WHEN OTHERS => ...
END CASE
```

Les champs expressions et Valeur doivent être d'un même type discret.

3.2 Instructions concurrentes

3.2.1 Processus

Le processus regroupe des expressions séquentielles, i est sensible à une liste de signaux. Dans ce projet, nous ne traiterons que les processus de la forme suivante :

```
[label:]PROCESS {liste des signaux surveillés}
        déclarations
        BEGIN
           instructions sequentielles
        END PROCESS label;
```

3.2.2 Affectation Concurrente de signal :bonus

A chaque affectation concurrente correspond une forme séquentielle équivalente contenue dans un process.

sa syntaxe est la suivante :

```
nom_signal <= forme_onde1 when cond_booleene_1 else
              forme_onde2 when cond_booleene_2 else
              ...
              forme_onde_3
```

Ce qui est strictement équivalent au processus suivant :

```
process (signaux_presents dans forme_onde et cond_booleene)
begin
  if  cond_booleene_1 then
      nom_signal <= forme_onde1
  elsif cond_booleene_2 then
      nom_signal <= forme_onde2
  elsif
```

```
      ...
   else
      forme_onde_3
   endif;
  end process;
```

3.2.3 Instantiation explicite de composant

La syntaxe de cette instruction est :

```
label: nom_du_composant_modele
   PORT MAP(
   {correspondances ports_effectifs/ ports_locaux})
```

Exemple : Ici le composant inverseur est instancié 2 fois.

```
C1: inverseur PORT MAP (E=>A, S=>B);
C2: inverseur PORT MAP (E=>C, S=>D);
```

3.3 Fonctions et opérateurs

Pour tous les opérateurs présentés ci dessous, nous supposerons qu'il existe des portes de bases associées pour la synthèse. Les parantheses seront utilisées pour marquer la priorité des opérations.

3.3.1 Les opérateurs logiques : and , or, not, xor

Les opérateurs logiques sont prédéfinis pour réaliser les opérations logiques "et", "ou", "non" et "ou exclusif" . Ils fonctionnent aussi sur des tableaux à une dimension (bit_vector) . Le résultat de ces opérations est de type bit.

3.3.2 Les opérateurs relationels : $=, \backslash =, <, <=, >, >=$

Le résulat de ces opérations est de type bit. Ces opérateurs sont définis sur tous les types étudiés dans le cadre de ce projet.

3.3.3 Les opérateurs arithmétiques : +,-,*

Ces opérateurs portent sur des types arithmétiques. Ils peuvent être soit unaires et représentés le signe, soit binaires et implémentés une opération.

3.3.4 La concaténation : & et la sélection partielle

Elle se définit sur des tableaux à une dimension.

Chapitre 4

Synthèse

Le VHDL est un “langage de description matérielle” : son but n’est donc pas de décrire des algorithmes à exécuter par un processeur comme les langages de programmation classique, mais plutôt de fournir un outil pour décrire (et donc implémenter) des circuits logiques. La prise en main n’est donc souvent pas aisée parce que le langage présente des caractéristiques pas forcement intuitives.

4.1 Parallélisme

Les langages de programmations son essentiellement séquentiels, alors que le VHDL est intrinsèquement parallèle : dans une architecture, tous les affectations et les process s’exécutent de façon concurrente. Cela ajoute une certaine complexité pour la compréhension et la simulation, mais par contre simplifie de façon importante la synthèse : chaque élément va être directement traduit en porte logiques. Cette opération n’est pas évidente ni toujours possible : seulement un sous-ensemble du VHDL est synthétisable, et une certaine rigueur d’écriture est nécessaire pour assurer des bons résultats.

4.2 VHDL synthétisable

Pour être synthétisable, un composant en VHDL doit suivre plusieurs règles :

— Logique séquentielle et logique combinatoire doivent être implémentées dans des process séparés. Si cette règle n’est pas respectée, la synthèse va produire des résultats pas corrects

— un signal ne peut être affecté que par un seul process, sinon cela serait équivalent à connecter deux entrées à un même fil, dont la sortie serait donc indéfinie

— La liste de sensibilité d’un process doit contenir tous les signaux qu’il lit. Ne pas respecter cette règle impliquerait un filtrage conditionnel des entrées, ce qui n’est pas souhaitable.

— Seulement les éléments séquentiels ont le droit de tester un front sur un signal (leur horloge).

— On ne peut pas lire et affecter un signal dans le même process. Cela serait équivalent à lire et écrire le même “ fil”, donc à une boucle combinatoire.

— Un process doit toujours affecter une valeur à tous ses signaux de sortie. Si dans un embranchement un signal n’est pas affecté, cela est interprété comme une mémorisation (i.e. “ garde l’ancienne valeur”), ce qui résulte dans l’instanciation d’un latch.

— On n'a pas le droit d'utiliser des instructions d'attente ("wait") ou des assignations inertielles ("after") parce qu'elles demandent une maitrise direct du temps, ce qui est impossible.
— Il faut toujours séparer la description en Partie Contrôle et Partie Opérative pour aider la synthèse
— Les variables doivent toujours être initialisées avant d'être lues.

4.3 Synthèse dirigé par la syntaxe

L'algorithme de synthèse que vous développerez ici est dirigé par la syntaxe. Il ne correspond pas réellement à ce qui est implémenté de manière traditionnelle dans les outils de synthèse et donne des nets lists beaucoup plus grosses. Aucune optimisation n'est effectuée sur le résultat. Les optimisations sont laissées à votre convenance, et serviront de bonus.

A chaque stucture VHDL correspond une porte logique.

Les operateurs arithmétiques et logiques sont directement synthétisés sous forme de portes. Nous vous fournirons un fichier contenant l'interface de toutes ces portes. Les process sequentiels sont synthétisés sous forme de registre, avec ou sans "enable". Les instructions de type "if" sont synthétisées sous la forme d'un multiplexeur. Les process combinatoires n'affectant pas de valeurs a tous les signaux dans toutes les branches génèrent des latchs.

4.4 Chemin critique

Tout outil de synthèse est capable de générer le chemin critique du circuit qu'il synthétise. Pour chaque porte nous vous fournirons ses caractéristiques, et vous calculerez le temps de traverser entre 2 registres. L'outil sera capable de fournir un rapport sur le chemin critique du circuit synthétisé.

Chapitre 5

Comment créer un outil de synthèse ?

Cette section donne quelques règles simples pour développer proprement un assembleur et quelques éléments de réfléxion sur la façon de procéder.

5.1 Analyse lexicale

L'analyse lexicographique peut se définir comme l'analyse des mots (ou "lexèmes") contenus dans un langage. Dans notre cas, il s'agit d'étudier les lexèmes contenus dans le programme source en langage assembleur. Les lexèmes lus peuvent être de nature différentes : des déclarations (d'entité, d'architecture, de types), des instructions, des valeurs, des symboles, des étiquettes, etc. A titre d'exemple, on montre le découpage du petit programme VHDL suivant en lexème :

```
 -- Un commentaire...
ENTITY inv is
PORT ( A: IN bit; B: OUT bit);
END ENTIY inv;
```

Le résultat de l'analyse peut se présenter sous la forme suivante (NL signifie nouvelle ligne / New Line codée par le caractère '$\backslash n$' en langage C++) :

```
[COMMENT      ]   Un commentaire...
[NL           ]   \n
[UNITE        ]   ENTITY
[ETIQUETTE    ]   inv
[IS           ]   is
[PORT         ]   port
[PAR_OUVR ]   (
[ETIQUETTE    ]   A
[DEUX_PTS     ]   :
[DIRECTION    ]   IN
```

```
[TYPE         ]   bit
[PTS_VIRGULE  ]   ;
[ETIQUETTE    ]   B
[DEUX_PTS     ]   :
[DIRECTION    ]   OUT
[TYPE         ]   bit
[PAR_FERM     ]   )
[PTS_VIRGULE  ]   ;
[FIN          ]   END
[UNITE        ]   ENTITY
[ETIQUETTE    ]   inv
[PTS_VIRGULE  ]   ;
[NL           ]   \n
```

Cette opération se réalise simplement en implémentant un automate à états finis.

5.2 Analyse grammaticale

L'analyse grammaticale sert à vérifier que les lexèmes extraits lors de l'analyse lexicale sont employés dans un ordre correct, c'est-à-dire que le programme source analysé constitue un programme correct en VHDL. En général, on explicite la syntaxe d'un langage de programmation par une grammaire (d'où le nom d'analyse grammaticale). Cette grammaire peut être efficacement définie par un listing de dépendances entre objets composant le programme : chaque ligne du listing donne un objet `A` de la grammaire suivi des objets `B`, `C`, `D`... qui peuvent composer cet objet `A`. Le tout est organisé de manière à la fois récursive et hiérarchique : par exemple la ligne suivante peut décrire les objets `E`, `F`, ... qui peuvent composer `B`. On effectue ainsi une décomposition hiérarchique de l'objet global qu'est le programme vers les objets les plus élémentaires possibles qui sont les lexèmes qui composent le texte (on parle de symbole terminal), en passant par toutes les relations régissant les objets intermédiaires.

La syntaxe des différentes instructions, avec leurs opérandes et modes d'adressages, n'est pas formalisée ici. Voir le corps du texte et vos cours et TPs de VHDL pour ces spécifications.

5.3 Génération d'un arbre et de la net list

Après les phases d'analyse lexicale et grammaticale, il est nécessaire de stocker l'information dans une structure afin de l'exploiter ultérieurement pour la génération de la netlist. En effet, à partir de cette structure, il devient possible de vérifier la correction grammaticale pour la synthese. Vous definirez clairement votre format.

En résumé, l'outil de synthèse pourra effectuer son travail en quatre phases :

1. première phase : création des bibliothèques
2. deuxième phase : lecture et analyse syntaxique du texte source, production d'un arbre ;
3. troisième phase : analyse sémantique de l'arbre : est il synthétisable.
4. quatrième phase : génération de la net list à partir des différents composants.

Chapitre 6

Spécifications, travail à réaliser

Avant de pouvoir compiler il faut créér les librairies et définir les liens entre le nom logique de la librairie (nom utilisé dans le vhdl) et nom physique (nom du répertoire où les fichiers seront compilés).

Le répertoire est créé via la commande :

```
vhdllib  physical_library_name
```

Vous pourrez une commande pour supprimer la librairie :

```
vhdldel  physical_library_name
```

Le lien entre le nom logique et le nom physique est créé par la commande :

```
vhdmap  logical_library name physical_library_name
```

Cette commande ajoute dans un fichier .ini le lien entre les noms logiques et physiques. Elle supprime les liens préexistants.

Le compilateur sera appellé en tapant sous unix la commande :

```
vhdlcomp -l library_name source_filename
```

où `source_filename` est le nom du fichier texte contenant le programme à compiler. L'option `-l library_name` sert à indiquer le nom de la bibliothèque de compilation. Pour vérifier que l'arbre est synthétisable on appelera sous unix la commande :

```
vhdlsynth -l library_name source_filename
```

Pour synthétiser le circuit, la commande

```
vhdlelaborate -l library_name[top_file_name]
```

Cette commande est capable si necessaire de trouver le fichier top a synthétiser.

6.1 fichier intermédiaire

Ce fichier intermédiaire contiendra la structure de l'arbre VHDL. C'est à vous de définir le format. Vous pouvez si vous le souhaitez utiliser les formats fournis par xml.

6.2 Fichier netlist

Le fichier netlist sera codé en VHDL et contiendra une description structurelle du circuit à partir d'éléments d'une bibliothèque prédéfinie.

Pour valider votre netlist, vous pouvez la visualiser en utilisant les outils de quartus disponibles au cime, ou sous windows à l'ecole.

Chapitre 7

Agenda et organisation du projet

7.1 Aspects généraux

7.1.1 Binômes

Le projet sera mené en binôme. Un étudiant sera chargé de concevoir la génération du fichier xml, l'autre de la netlist depuis un fichier xml.

Les binômes devront être formés dès que possible au plus tard le 16 septembre. Merci de nous informer au plus tôt de la constitution des binômes par email à l'adresse suivante :

```
katell.morin-allory@imag.fr
michele.portolan@imag.fr
```

7.1.2 Tuteurs

Les étudiants bénéficient de l'aide de leur tuteur notamment pour :
— l'organisation du projet
— l'analyse du problème
— la conception du programme
— la synthese du VHDL
— la programmation en langage C++
— l'environnement de développement (make, autres outils gnu, etc)
— toute autre question liée au projet...

Les enseignants ne sont pas là pour concevoir le programme, programmer ou corriger les bugs à la place des étudiants. Si les enseignants l'estiment nécessaire, ils peuvent néanmoins débloquer les groupes en difficulté. Les questions posées par les étudiants doivent être précises et réfléchies.

7.1.3 Aspects matériel

Pour ce projet, le travail s'effectuera sur des PC avec un système d'exploitation Linux. Tous les outils nécessaire (compilateur C++, débogueur, etc.) seront installés sur les stations de travail de PHELMA. Vous pouvez naturellement travailler sur votre ordinateur personnel si vous en possédez un, mais attention **le projet final doit fonctionner correctement à PHELMA, sous Linux, le jour de l'examen !** Soyez prudent si vous développez sous Windows, le portage n'est pas toujours automatique...

7.1.4 Site web du projet

Il est disponible sur chamilo. Vous y trouverez le sujet et toutes informations que nous jugerons necessaires. Vous pourrez aussi utiliser le forum pour poser des questions.

7.2 Déroulement du projet dans le temps et évaluation

Le projet comporte deux phases : une phase d'analyse sous la responsabilité du tuteur et une phase de développement. Il sera suivi d'une démonstration sur machine.

La phase d'analyse est prévue pour durer 3 semaines (semaines 38 et 40). Des créneaux réguliers seront réservés avec le tuteur pour toutes les questions relatives au projet. La phase d'analyse se terminera par la remise au tuteur d'un rapport d'analyse au plus tard le **mercredi 07 octobre 2015**. Ce rapport donnera lieu à une note qui interviendra pour 20 % dans la note finale du projet.

La phase de développement dure 4 mois. Elle est destinée à l'implémentation en langage C++ de votre assembleur, suivant les spécifications définies dans la phase d'analyse. Pendant cette période, 6 séances en salle machine seront organisées, encadrées par les enseignants du projet, pour vous aider sur les aspects de programmation. La phase de développement se terminera à la fin de la semaine 05.

Le projet se terminera le **jeudi 04 février à 23h59**, date à laquelle *l'ensemble de votre code* devra être remis aux enseignants. Tous vos fichiers source (`*.h`, `*.cc`, `Makefile`, ...) et fichiers tests seront rendus. Lors de l'examen final, les binômes utiliseront obligatoirement le code fourni à cette date. Un espace de dépôt sera alloué sur les serveurs de l'école à chaque étudiants pour rendre ses codes, avec blocage automatique le soir du jour de remise. Aucun code ne sera accepté ultérieurement. Vous rendrez aussi un rapport qui comptera pour 10% de la note finale.

L'examen final sera organisé **semaine 06**, au créneau horaire habituel. Il comptera pour 60% de la note.

A la fin du projet, le tuteur rendra une appréciation qui comptera pour les 10% restant de la note finale du projet informatique.

7.3 Description des rapports et de l'examen

7.3.1 Rapport d'analyse

La finalité de ce rapport est de décrire la solution que vous proposez pour mener à bien votre projet. Il doit permettre au tuteur de comprendre votre analyse du problème et la conception de votre solution. En particulier, vous devrez présenter l'architecture et la décomposition modulaire de votre programme, afin de ramener le problème à un ensemble de sous-problèmes dont on possède déjà une solution, ou dont la solution est facile à programmer. Dans le cas qui nous préoccupe, la décomposition modulaire a un second but qui est de permettre à chacun des membres du binôme de définir quelle sera sa participation pendant la phase de programmation.

Rappellons que cette phase d'analyse *précède* la phase d'implémentation. Ne raisonnez pas encore par rapport à d'éventuels problèmes techniques de programmation, mais bien par rapport à la conception générale de votre assembleur, son découpages, les algorithmes retenus, etc.

Le rapport d'analyse devra comporter :

- Une introduction présentant l'analyse du sujet (ce que vous en avez compris) et les objectifs de votre projet.
- Une description de la décomposition modulaire du programme expliquant le découpage choisi.
- Une description pour chacun des modules de son interface et des modules qu'il utilise pour sa mise en oeuvre. Pour chaque procédure ou fonction, indiquez les entrées, les sorties et une description d'une ou deux phrases de ce qu'elle fait.
- Une description des éventuels problèmes techniques rencontrés dans la phase d'analyse, ainsi que des solutions techniques que l'on pense leur apporter.
- Une méthodologie de test et d'évaluation systématique de chacun des modules et du programme.
- Une description de l'organisation des fichiers (arborescence Unix) choisie ainsi que la méthode utilisée pour traiter les erreurs.
- Une proposition de planning [1] et de répartition du travail au sein du binôme pour la phase de développement (on n'oubliera pas dans l'élaboration de ce planning que chacun des modules, avant d'être intégré au programme, doit faire l'objet d'un test individuel afin de "prouver" qu'il est sans erreur et qu'il fait bien ce qu'il est censé faire).

Si nécessaire, le rapport d'analyse donnera lieu à une discussion de "mise au point" avec le tuteur. Le rapport sera noté et la note interviendra pour 20 % dans la note finale du projet.

7.3.2 Examen

L'examen final se deroule en deux phases. Lors de la premiere phase de 2 heures, les etudiants derouleront une serie de test, et corrigerons au mieux leur programme. Lors de la deuxieme phase, chaque binome sera seul avec un des enseignants du projet. Une phase de démonstration et de test aura lieu. Ces deux parties compteront pour respectivement 60% dans la note finale attribuée au projet. Il sera deplus demande un rapport final qui comptera pour 10% de la note

Tests sur machine

Selon l'état d'avancement de votre projet, il vous sera demandé de modifier votre programme (en 10-15 minutes, pas plus) par exemple en ajoutant la gestion d'une nouvelle instruction au microprocesseur. La suite sera consacrée aux tests proprement dits : l'examinateur vérifiera le bon fonctionnement du programme, à l'aide de ses propres fichiers tests et des vôtres.

Très important : les sources définitifs seront rendus le 4 fevrier à 23H59. Seule cette version sera utilisée pour la démonstration. Le projet devra donc être complètement achevé à cette date.

7.4 Quelques conseils supplémentaires

Expérimentations

Réalisez des expériences pour mieux appréhender les différents aspects du projet et préparer des fichiers de tests que vous utiliserez pour corriger et valider votre programme, ou pour la

1. Il est très important, et très délicat, de bien planifier et évaluer les temps des tâches du projet. A faire absolument !

démonstration le jour J.

- Ecrivez des programmes en VHDL. Vous pouvez commencer avec seulement quelques instructions puis compliquer la chose en gérant des données, des conditionnelles...
- Déterminez manuellement, à partir des spécifications, la synthese des différentes instructions.
- Vous pouvez ensuite créer le fichier `xml` correspondant.
- Ces fichiers pourront servir de tests pendant le projet pour déboguer vos programmes ou pour le jour de la démo.
- Constituez-vous une base de programmes tests en langage VHDL couvrant les différentes instructions.

Conception et développement

Après ces petites (ou pas si petites) expériences, vous pouvez vous lancer dans la conception du programme : identification des différentes tâches, choix des structures de données intermédiaires, découpage modulaire du code (quels fichiers ? quelles fonctions ?), etc.

Soyez bien conscient que **la phase de conception est extrêmement importante dans un tel projet**. Si elle est bien faite, la programmation se fera sans soucis, ou uniquement techniques dûs à votre manque d'expérience sur un projet de cette taille. Sinon, vous vous rendrez compte très tard des problèmes de conception, ce qui obligera à modifier et à recommencer des parties parfois très importantes de votre travail.

Il faut prévoir une décomposition du développement de manière à pouvoir tester et corriger le programme au fur et à mesure que vous l'écrivez. Sinon, vous risquez d'avoir un programme très difficile à corriger, ou vous risquez de devoir réécrire de grandes portions de code. De manière générale, on code d'abord les cas les plus généraux et/ou les plus simples, avant de coder les cas particuliers et/ou compliqués. Certains modules constituent le noyau de l'assembleur et seront implémentés en priorité, alors que d'autres modules additionnels pourront être implémentés dans un second temps.

Pensez à concevoir le programme de manière à pouvoir ajouter facilement une instruction le jour de la démonstration (voir **??**). La programmation modulaire permet en outre d'avoir un code plus concis et donc plus facile à déboguer. Programmez de manière **défensive** : pour les cas que votre programme ne devrait jamais rencontrer si vous avez programmé correctement, mettez-un message compréhensible du type `Erreur interne, fonction bidule` et arrêtez proprement le programme, afin de pouvoir déboguer plus facilement. Placez des traces d'exécutions dans vos programmes de manière à pouvoir suivre le déroulement du programme. Utilisez les macros C++ pour faire afficher ou supprimer ces traces facilement (cf. cours de C++ de première année). Il est fortement recommandé de se familiariser avec l'utilisation d'un débogueur (`gdb` ou `ddd`).

Gardez-comme ligne directrice d'avoir le plus tôt possible **un programme qui fonctionne**, même s'il ne gère pas tout. Ensuite, améliorez-le au fur et à mesure.

Voilà tout y est. Maintenant, bon travail à vous tous !

Annexe A

Portes fondamentales

```
library portes;
use portes.all;



package cell is

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

  end and_2;


  entity or_2 is
    port (
      a : in  bit;
      b : in  bit;
      c : out bit);
  end or_2;

  entity not_1 is
    port (
      a : in  bit;
      c : out bit);
  end not_1;

  entity xor_2 is
    port (
      a : in  bit;
      b : in  bit;
      c : out bit);
  end xor_2;


 entity latch is

   port (
     D     : in  bit;
```

```
    clk   : in  bit;
    reset : in  bit;
    Q     : out bit);

end latch;

entity flip_flop is

  port (
    D     : in  bit;
    clk   : in  bit;
    reset : in  bit;
    Q     : out bit);

end flip_flop;

entity flip_flop_e is

  port (
    D     : in  bit;
    e     : in  bit;
    clk   : in  bit;
    reset : in  bit;
    Q     : out bit);

end flip_flop_e;


entity registre is

  generic (
    S : integer);

  port (
    D          : in  bit_vector (S-1 downto 0);
    clk, reset : in  bit;
    Q          : out bit_vector (S-1 downto 0));

end registre;

entity registre_e is

  generic (
    S : integer);

  port (
    D          : in  bit_vector (S-1 downto 0);
    e          : in  bit;
    clk, reset : in  bit;
    Q          : out bit_vector (S-1 downto 0));

end registre_e;


entity add is

  generic (
```

```
    S : integer);

  port (
    A, B  : in  bit_vector (S-1 downto 0);
    R     : out bit_vector (S-1 downto 0);
    c_out : out bit);

end add;

entity sub is

  generic (
    S : integer);

  port (
    A, B  : in  bit_vector (S-1 downto 0);
    R     : out bit_vector (S-1 downto 0);
    c_out : out bit);

end sub;

entity mult is

  generic (
    S : integer);

  port (
    A, B  : in  bit_vector (S-1 downto 0);
    R     : out bit_vector (S-1 downto 0);
    c_out : out bit);

end mult;

end cell;


package body cell is



end cell;
```