

**NOM DU DOSSIER**

**Sous titre**

Date

Service

**RAPPORT DE PROJET**

**Antoine ALAVERDOV, Clémence LEMEILLEUR**

**Promo 56, Année 2021/2022 – 4IR-SI-B1**

*« Projet Systèmes Informatiques »*

S2 2022

Encadrant : E.ALATA

**RAPPORT DE PROJET**

Antoine ALAVERDOV, Clémence LEMEILLEUR

Promo 56, Année 2021/2022 – 4IR-SI-B1

*“Projet Systèmes Informartiques*”

S2 2022

Encadrant: E.ALATA

Table des matières

[I- La démarche conception abordée 1](#_Toc104813623)

[II- Les choix d’implémentation 3](#_Toc104813624)

[III- Les problèmes rencontrés et les solutions pour y remédier 3](#_Toc104813625)

[IV- Les résultats obtenus 4](#_Toc104813626)

[V- Les instructions assembleurs rajoutées et leur justification 4](#_Toc104813627)

[VI- Les limites et améliorations possibles de notre projet 4](#_Toc104813628)

[Table des annexes 6](#_Toc104813629)

Nous allons dance ce rapport vous présenter notre travail, nos réflexions ainsi que nos choix d’implémentations. Le but de ce rapport est de compléter notre code et de le rendre plus compréhensible.

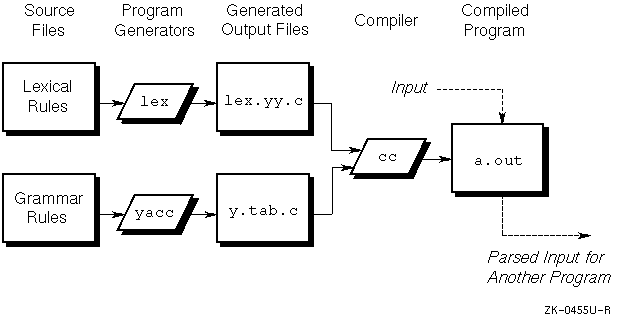
Tout au long de ce projet nous avons dans un premier temps effectuer l’analyse syntaxique et sémantique, développer un **compilateur** en utilisant le YACC et le LEX. Nous avons ensuite, toujours dans la même démarche, généré des instructions assembleurs à partir du code C.

Pour finir, nous avons programmé un **microprocesseur** de type RISC avec pipeline à travers le code VHDL.

L’objectif de ce projet était de réaliser un système informatique complet. Nous allons donc dans ce rapport vous détailler notre démarche, afin de faire un retour sur notre travail avec un certain recul.

# La démarche conception abordée

Commençons par faire un rappel sur la partie **compilateur**.



Il est important de bien comprendre que :

* Le **Lex** génère un code en C pour l’analyse lexicale (scanner). Pour cela il associe des token à des expressions spécifiques
* Le **Yacc** génère un code C pour l’analyse syntaxique (parser). Il associe un ensemble de token à ce qu’il signifie.

Pour la partie **Lex**, nous avons implémenté les tokens permettant d’effectuer les différentes fonctionnalités du langage C. Nous pouvons citer parmi celles-ci la reconnaissance de mots clés et opérateurs :

* Constantes
* Type de variables
* Multiples opérations
* Expressions régulières
* While et If
* Erreur (pour continuer à interpréter malgré la rencontre d’une erreur)
* Etc

Cette partie du code nous permet alors de reconnaitre des éléments dans le langage C afin d’y associer un token et de pouvoir par la suite les traduire en instructions assembleurs.

Pour la partie **Yacc**, nous avons donc défini toutes les règles qui nous semblaient essentielles à reconnaitre pour pouvoir les traduire ensuite en assembleur. Parmi celle-ci nous pouvons compter :

* Les délimitations de sections du programme
* Les fonctions, procédures et leurs paramètres
* Tout type d’expressions (déclarations, affectations…)
* Les conditions
* Les opérations basiques
* L’affichage à l’écran (print)
* Etc

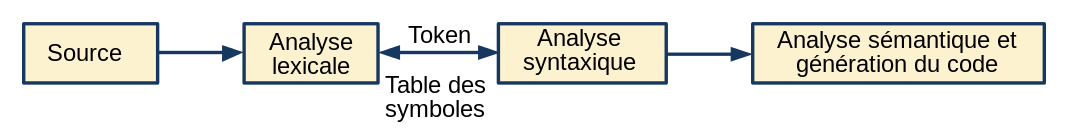
Il nous a paru important de nous attarder sur les différentes possibilités de déclarations et affectations des variables.

Une image contenant texte

Description générée automatiquement

*Remarque :* tID désigne le nom de la variable et tNB désigne un nombre en exponentielle ou non.

En effet, il peut y avoir plusieurs déclarations à la suite, tout comme une déclaration couplée à une affectation ou l’un après l’autre. Il ne faut pas oublier de cas au risque de passer à côté d’une instruction qui ne sera pas reconnue dans le fichier.

De plus, c’est à cet endroit là qu’intervient la **table des symboles** qui va contenir toutes les variables déclarées. 

Avant d’utiliser une variable on va vérifier si elle existe dans cette table des symboles. SI ce n’est pas le cas, il y aura une erreur levée.

Notre table des symboles a été construite de la façon suivante :

**METTRE SCHEMA TABLE DES SYMBOLES**

**La profondeur représente…**

Nous avons également la **table d’instructions** qui intervient à cet endroit-là. Elle regroupe toutes les instructions assembleur au fur et à mesure de l’analyse de code C avant de toutes les mettre dans un fichier. Elle associe un token de l’analyse faite à une instruction en assembleur en stockant les paramètres associés.

L’étape suivante est de traduire ces fichiers C en langage assembleur.

Pour cela nous avons suivis le sujet afin de traiter toutes les fonctionnalités essentielles dans un premier temps à savoir :

* Printf(var) : Affiche le contenu du registre qui aura été chargé de « var » au préalable.
* Déclarations de variables : déclaration simple, multiples ou déclaration avec affectation.
* Types de variables : int ou const int (les autres types soulèveront une erreur et affecter une deuxième valeur à une constante ne sera pas accepté non plus).
* Les expressions :

Une image contenant texte

Description générée automatiquement

Elle se présente de cette manière dans le Yacc et chacune des sous expressions est également détaillée. On trouve dans la table des symboles la traduction de chacun en instruction assembleur.

* Les conditions (if/else) :
* Le while :

Pour la partie **microprocesseur en VHDL**, **A ECRIRE**

# Les choix d’implémentation

# Les problèmes rencontrés et les solutions pour y remédier

# Les résultats obtenus

Après avoir implémenté toutes ces parties du code nous l’avons testé.

Afin de vérifier que notre langage assembleur se comportait de la même manière nous avons pris un code en C que nous avons traité afin de vérifier que ce qui nous était renvoyé par notre compilateur, après être passé par les instructions assembleurs générés était correct.

Nous avons fait cela grâce à l’interpréteur. Il se comporte comme une fausse machine et permet d’exécuter notre code assembleur qui sort du compilateur. C’est une sorte de machine « test » qui nous permettra de tester que le programme assembleur renvoyé par notre compilateur fonctionne bien.

**Après avoir fait ce test nous pouvons dire que ….**

# Les instructions assembleurs rajoutées et leur justification

# Les limites et améliorations possibles de notre projet

Au niveau du **Yacc et de Lex**, nous aurions pu traiter plusieurs détails supplémentaires afin de couvrir plus de cas comme :

* Les priorités au niveau des tokens utilisés, notamment pour les opérations mathématiques afin d’éviter les conflits de certaines règles du Yacc. (Décalage/réduction)
* La reconnaissance des pointeurs et des adresses à l’aide des instructions LOAD et STORE.
* La reconnaissance de la boucle for.
* La reconnaissance des warnings de compilation de code.
* La reconnaissance de fonctions dans le code.
* La reconnaissance des commentaires dans le code.

Il y a surement d’autres fonctionnalités que nous pouvons rajouter afin d’avoir une analyse lexicale et syntaxique la plus précise et détaillée possible.

***Conclusion :***

Tout au long de ce projet, nous avons réussi à répondre aux problèmes posés en codant au fur et à mesure le Lex, le Yacc.

Nous avons réussi à gérer l’approche des différentes problématiques de ces étapes et ainsi réaliser un compilateur et vérifier son fonctionnement grâce à l’interpréteur. Nous pouvons donc désormais comprendre ce qu’il se passe derrière gcc par exemple et les problématiques qui ont été rencontrées lors de sa mise en place.

La conception d’un microprocesseur de type RISC avec pipeline nous a également permis d’expérimenter et de mieux comprendre les notions d’architectures matérielles et d’automate et langage abordés en cours.

Cela nous permet donc de nous mettre une fois de plus dans le rôle de l’ingénieur qui est d’utiliser ses connaissances et de les appliquer à des cas réels.

# Table des annexes

1. Annexe 1 : Lien GitHubA

**Annexe 1 :** Lien GitHub : https://github.com/Piazo/ProjetSystemeInfo

