

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>I- La démarche conception abordée et choix d'implémentation</i>	<i>1</i>
<i>II- Les problèmes rencontrés et les solutions pour y remédier</i>	<i>4</i>
<i>III- Les résultats obtenus</i>	<i>5</i>
<i>IV- Les limites et améliorations possibles de notre projet.....</i>	<i>5</i>
<i>Table des annexes.....</i>	<i>7</i>

Nous allons dans 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.

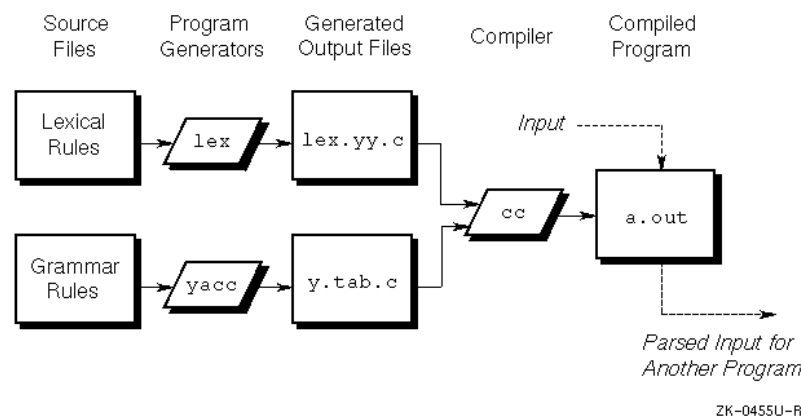
Pour finir, nous avons programmé un **microprocesseur** de type RISC avec pipeline à travers le code VHDL.

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.

Nous avons choisi de détailler notre vision du projet et l'utilité de chaque partie puisque cela nous a permis de le comprendre nous-même et d'avancer. Cela nous a donc paru intéressant de le laisser dans notre rapport même si ces notions vous paraîtront surement évidentes.

I- La démarche conception abordée et choix d'implémentation

Commençons par faire un rappel sur la partie **compilateur**. Nous rappelons que nous ne cherchons donc pas à interpréter.



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 en assembleur.

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 :

- Type de variables (int), (les autres types soulèveront une erreur et affecter une deuxième valeur à une constante ne sera pas acceptée non plus).
- Multiples opérations
- While, If et Else

Cette partie du code nous permet alors de reconnaître 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 reconnaître pour pouvoir les traduire ensuite en assembleur. Parmi celle-ci nous pouvons compter :

- Les délimitations de sections du programme
- Les paramètres

- Tout type d'expressions (déclarations, affectations...)

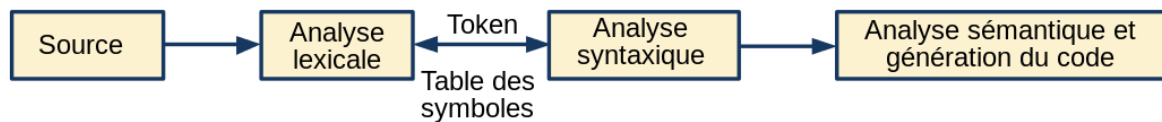
```
Instructions : Instruction Instructions
| ;

Instruction : Affectation
| tIF tPO Condition tPF {$1=add_instruc_to_tab("JMF", $3, -1, -1);}
|   Body Else {patchJMF($1, (($7)+1));}
| tWHILE tPO Condition tPF {$1 = add_instruc_to_tab("JMF", $3, -1, -1);} Body
|   {int indexJMP = add_instruc_to_tab("JMP", $1, -1, -1); patchJMF($1, (indexJMP+1));}
| tPRINT tPO Element tPF tEOI {add_instruc_to_tab("PRI", $3, -1, -1);}
```

- Les conditions
- Les opérations basiques : + - * /
- Déclarations de variables : déclaration simple, multiples ou déclaration avec affectation.

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

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 :

nom	index	type	profondeur
b	2	int	0
a	1	int	0

Après avoir associé des étiquettes aux token pour identifier les variables, on stocke dans cette table la valeur de la variable, son type, son index et sa profondeur. La profondeur correspond à son niveau d'imbrication (if, while, if dans un if...). Elle nous servira à supprimer les variables d'une même profondeur par exemple.

Dans cette table des symboles les valeurs sont mémorisées à l'exécution.

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 stocke une instruction en assembleur en stockant les paramètres qui lui sont associés. Nous lui avons donné une taille fixe et si nous dépassons cette taille un message d'erreur s'affichera.

L'étape suivante est de traduire ces fichiers C en langage assembleur. Les instructions assembleurs associées à des token ont leur comportement détaillé dans l'interpreteur.

Pour la partie **microprocesseur en VHDL**, nous avons implémenter un microprocesseur avec pipe-line qui correspond aux instructions assembleurs suivantes :

- Addition

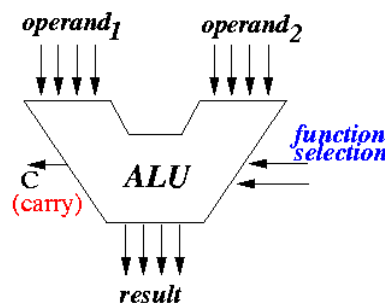
- Soustraction
- Multiplication
- Division
- Copie
- Affectation

Cette partie a pour but de recréer les différents composants (une ALU, les mémoires d'instructions et de données et le banc de registres) d'un processeur sur une carte FPGA. On les met en suite en séquence avec des pipe-line (5 dans notre cas) pour recréer le comportement d'un processeur.

Au niveau de la conception générale nous avons choisi de partir sur 8 bits pour avoir 2^8 valeurs. Par exemple pour l'addition, le max du résultat sera de 2×8 valeurs soit 16 bits.

Le processeur 8 bits comporte donc 16 bits d'opération mais prend des variables sur 8 bits et renvoie un résultat sur 8 bits. Nous renvoyons les bits de poids faible.

Au niveau de l'ALU son implémentation peut être visualisée de la façon suivante :



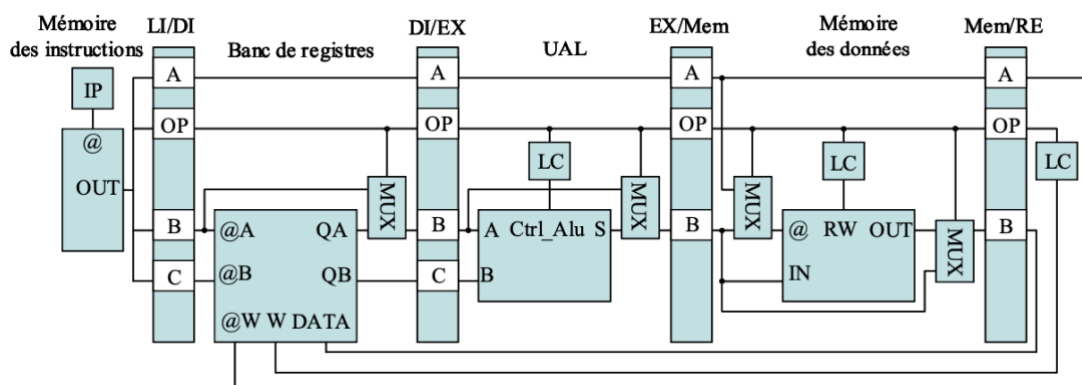
Nous avons donc dû prendre en compte des retenues dans certaines opérations par exemple. De plus, nous avons dû prendre en compte certaines contraintes comme :

- Z pour savoir si cela vaut 0
- C pour la retenue de l'addition
- O si le résultat de la multiplication est overflow
- N si le résultat est négatif

Ces flags valant 0 ou 1 nous ont servi pour nos tests.

Pour ce qui est des **mémoires d'instructions et de données et du banc de registre** nous avons repris les informations et les schémas du sujet pour les mettre en place. Nous rappelons que RST est actif à 0 et que w est actif à 1 et que s'il est actif, DATA va à @w.

Nous avons ensuite mis en séquence tous ces composants grâce aux pipe-line selon le schéma suivant afin d'obtenir le chemin de données :



Cette partie du projet nous a donc permis de comprendre de manière détaillée les interactions entre les composants d'un processeur de manière détaillée.

Une des grosses parties du microprocesseur a été la **gestion des aléas**. Il faut être sûr de la conséquence de la suite de plusieurs instructions assembleur.

II- Les problèmes rencontrés et les solutions pour y remédier

Lors de l'implémentation de ces différentes fonctionnalités nous avons rencontrés plusieurs soucis auxquels nous avons dû remédier.

- 1- Le premier a été pour nous la compréhension du projet en général et ses contraintes implicites. Nous avons eu du mal à visualiser l'ensemble et le rôle de chaque élément. C'est la raison pour laquelle ce rapport est autant détaillé puisque cela a été notre outil de travail pour poser les concepts et mieux les aborder.

Nous avons dû nous y prendre à plusieurs reprises avant de réussir à avancer vers quelques choses de satisfaisant. Il était difficile de prendre en compte tous les cas possibles sans en oublier et d'arriver à quelques choses de fonctionnel.

Un exemple pour illustrer ce problème peut être le suivant. Nous nous sommes beaucoup attardés sur les différentes possibilités de déclarations et affectations des variables. Voici le Yacc que nous utilisions dans une première version :

```
Declarations : TypeVar Declaration Findeclaration;

Declaration : tID {addsymbol($1, type);}
            | tID {addsymbol($1, type);} tEGAL Expr;

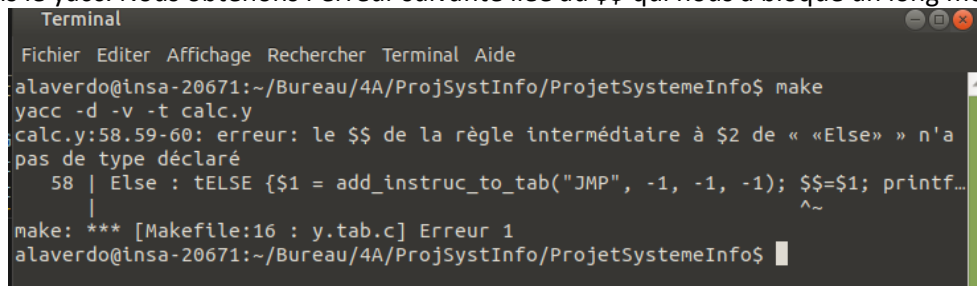
Findeclaration : tEOI
              | tVIRG Declaration Findeclaration;

DeclaAffec : TypeVar tID {addsymbol($1, type);} tEGAL TypeNb tEOI
           | TypeVar tID {addsymbol($1, type);} tEGAL Operation tEOI;

Affectation : tID tEGAL TypeNb tEOI
            | tID tEGAL Operation tEOI;
```

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. Nous avons ensuite trouvé comment l'aborder de manière plus simple.

- 2- Un de nos soucis majeur a été cette erreur lors du lancement de notre code au niveau du else dans le yacc. Nous obtenons l'erreur suivante liée au \$\$ qui nous a bloqué un long moment :



```
Terminal
Fichier Editor Affichage Rechercher Terminal Aide
alaverdo@insa-20671:~/Bureau/4A/ProjSystInfo/ProjetSystemeInfo$ make
yacc -d -v -t calc.y
calc.y:58.59-60: erreur: le $$ de la règle intermédiaire à $2 de « Else » n'a
pas de type déclaré
58 | Else : tELSE {$1 = add_instruc_to_tab("JMP", -1, -1, -1); $$=$1; printf...
   | ^~
make: *** [Makefile:16 : y.tab.c] Erreur 1
alaverdo@insa-20671:~/Bureau/4A/ProjSystInfo/ProjetSystemeInfo$
```

Nous l'avons solutionné en plaçant le \$\$=\$1 après le patchJMP.

- 3- Nous avons également des difficultés au niveau de la libération de l'espace mémoire associé à certaines variables temporaires

III- 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. Il détaille ce qu'il faut faire pour chaque instruction assembleur.

Après avoir fait ce test avec un programme C basique suivant :

```
main(){
    int test;
    test = 2+5*2+100;
    printf(test);
    int a = 1;
    int b=5;
    while(a<10){
        printf(a);
        a=a+2;
    }
    a=a+50;
    if(a<15){
        printf(b);
    }else{
        int c=5555;
        if(c>5){
            printf(c);
        }else{
            int d=44;
            printf(d);
        }
    }
    printf(a);
}
```

Nous pouvons dire que notre compilateur fonctionne et donne un résultat satisfaisant bien qu'il ne soit pas optimal et ne couvre pas tous les cas particuliers et exceptions. Nous détaillerons cela dans les limites de notre code.

IV- 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 et ainsi créer une table comme la table des symboles mais pour les fonctions (sans profondeur puisqu'on évite les fonctions imbriquées).
- La reconnaissance des commentaires dans le code.
- Gestion des aléas de données et de branchement.
- Supprimer l'espace mémoire occupé par les variables temporaires
- La reconnaissance de constante
- La reconnaissance des erreurs (pour continuer à interpréter malgré la rencontre d'une erreur)

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

Comme évoquer précédemment nous avons donc un code qui est fonctionnel et répond à sa fonction principale. En revanche il comporte des limites dans ces applications et serait à approfondir et développer davantage pour le rendre encore plus polyvalent.

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

I. <u>Annexe 1</u> : Lien GitHub	A
--	---

Annexe 1 : Lien GitHub : <https://github.com/Piazo/ProjetSystemeInfo> (la branche « yo » est la plus à jour)

INSA Toulouse

135, avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr



MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE