INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université Fédérale Toulouse Midi-Pyrénées

LAAS

Département de Génie Électrique & Informatique

Rapport de TP

Projet Systèmes Informatiques

26 mai 2020

Etudiants :

LAUR	Marie	m_laur@etud.insa-toulouse.fr
PERAIRE	Laetitia	peraire@etud.insa-toulouse.fr

Encadrants :

ALATA	Eric
DRAGOMIRESCU	Daniela
LEFEBVRE	Cédric
ROUX	Jonathan

Promo 54 : 2019/2020

4IR

Table des matières

Introduction

Ce rapport présente le travail réalisé au cours des séances de TP dans l'objectif de créer un compilateur d'un langage proche du C (très simplifié), et de concevoir et implémenter un microprocesseur correspondant à quelques instructions assembleur basiques.

Afin d'atteindre ces objectifs, nous avons respecté les horaires de TP et profité de ces moments pour poser des questions, mais nous avons aussi travaillé en parallèle de ces séances. Nous avons codé tout le projet ensemble, afin de progresser plus vite et d'avoir toutes les deux une bonne compréhension de l'ensemble.

Notre travail est disponible au lien git suivant : `https://github.com/Marie1009/compil`

1 Développement d'un compilateur en utilisant LEX et YACC

1.1 Fonctionnement général

Le compilateur que nous avons développé permet de reconnaître des tokens préalablement définis, puis de parcourir un programme simple grâce à des règles de syntaxe. On utilise Lex pour l'analyse lexicale et yacc pour l'analyse syntaxique.

Le fichier Lex qui contient toutes les instructions pour l'analyse lexicale génère un programme *lex.yy.c* qui contient une fonction *yylex()*. Cette fonction sera par la suite utilisée dans l'analyseur syntaxique afin de repérer les tokens qui composent le flux d'entrée.

Pour l'analyse syntaxique, nous avons aussi créé plusieurs structures (tableaux) et méthodes qui permettent de gérer les variables et leur unicité, l'initialisation des constantes mais aussi pour générer un code assembleur contenant toutes les instructions nécessaires au fonctionnement du programme reconnu.

1.2 LEX

L'outil Lex permet de générer un analyseur syntaxique à partir d'une description spécifique des terminaux (tokens) que l'on souhaite reconnaître. Donc la première étape de notre travail a été d'identifier les tokens nécéssaire pour notre langage (similaire au langage c mais simplifié).

Certains tokens correspondent à des mots ou des caractères bien précis et donc sont peu complexes à identifier. Par exemple, si on souhaite déclarer un entier, on devra utiliser le terme *int*, et il sera immédiatement transformé en token *tInt* par l'analyseur lexical. De la même façon, on peut reconnaître la fin d'une ligne avec le caractère *;* qui sera retourné avec un token *tEndL*.

Néanmoins, pour que le langage reconnu offre des possibilités de programmes assez variées, nous ne pouvont pas nous contenter de ce lexique trop basique. Nous avons donc utilisé plusieurs expressions régulières qui permettent d'agrandir les possibilités du langage.

Par exemple, lorsque on voudra déclarer une variable, celle-ci devra avoir un nom, et ce nom attribué pourra donc être reconnu grâce à l'expression suivante

```
Name     {Let}({Let}|{Digit}|_)*
```

De plus, afin de pouvoir récupérer le contenu de certaines variables dans l'analyseur syntaxique, nous avons aussi utilisé la variable externe *yylval*.

```
{Name}   {
          yylval.str = strdup(yytext);
          return tNom ;
        }
```

Cette portion de code représente le passage de la chaîne de caractères détectée par le regex *Name* vers l'analyseur syntaxique. En effet, la valeur est récupérée par *yytext* puis affectée au champ *str* de la structure de *yylval*. Il est important de préciser que le type de *str* correspond en fait à un type *char ** du langage c, et il est défini par l'analyseur syntaxique par la suite.

1.3 YACC

Après avoir généré l'analyseur lexical de notre compilateur, nous avons créé l'analyseur syntaxique permettant de combiner les tokens en un programme, et de traiter les différentes instructions de celui-ci.

Dans un premier temps, il a été nécessaire de construire la grammaire de notre langage. Il s'agit donc d'un ensemble de règles de syntaxes que nous avons choisies en essayant de rester fidèles à la syntaxe du c, mais en la simplifiant largement.

Voici donc les caractéristiques principales de notre langage :

- Le *main* ne peut pas recevoir d'arguments.
- Les types de variables connus sont les *int* et *const*, on peut déclarer ces variables en leur affectant directement une valeur mais ce n'est pas obligatoire, ils désignent tout deux des entiers, seule la propriété de modification les différencie.
- On peut réaliser les opérations d'addition, de soustraction, de division et de multiplication, ainsi que des comparaisons ($==$, $<$, $>$).
- Les blocks connus sont les blocks *if() {} else {}* et les blocks *while {}*.

L'implémentation de ces différentes fonctionnalités a été réalisée en deux temps. Premièrement, on a regroupé dans un programme en c les différentes structures et méthodes pour créer une table des symboles (*symboltable.c*). Cette table des symboles est un tableau qui sert à garder en mémoire les variables connues et des précisions sur ces variables (si ce sont des constantes, si elles sont initialisées, la profondeur à laquelle on les a déclaré).

Nous avons codé plusieurs méthodes utilisées dans l'outil Yacc pour parcourir et utiliser cette table des symboles. Tout d'abord pour gérer la profondeur des variables, nous avons deux méthodes simples qui incrémentent ou décrémentent une variable globale *global_ depth* en fonction de l'avancement dans les différents blocks.

Les méthodes *pop()* et *push()*, quand à elles, servent à créer une pile virtuelle qui est indispensable lorsque on veut effectuer au moins une opération. Elles donnent en fait des entiers correspondant à des adresses comme si c'était un pointeur de pile. Grâce à cela, on peut par la suite utiliser ces "emplacements mémoire" pour les valeurs intermédiaires des calculs.

De plus, afin de réutiliser la valeur d'une variable déclarée avant, nous avons créé une méthode *get_ address* qui permet de récupérer l'addresse d'un symbole de la table des symbole. Lors de l'ajout dans cette table, il est aussi nécessaire d'utiliser une méthode *add_ symbol* qui vérifie si une variable au nom identique à déjà été déclarée avant et l'ajoute à la table lorsque ce n'est pas le cas.

Dans un deuxième temps, on a associé les instructions reconnues dans le programme à des instructions assembleur. Les méthodes et les structures nécessaires sont dans le fichier *codeasm.c*. Ces instructions sont écrites au fur et à mesure dans un tableau de structs (avec toutes leurs opérandes), puis copiées dans un fichier externe. Nous avons considéré les instructions présentées dans le sujet, c'est à dire *ADD, SOU, MUL, DIV, INF, SUP, AFC, COP, PRI, JMP et JMF*.

Afin d'écrire les instructions dans un tableau, nous avons une méthode *add_ op3(opération, opérande1, opérande2, opérande3)* qui se décline en *add_ op2(opération, opérande1, opérande2)* et *add_ op1(opération, opérande1)* pour inscrire les différentes instructions en fonction du nombre d'opérandes nécessaires à celles-ci.

Pour le fonctionnement des blocks *if() {} else {}* et *while {}*, on a ajouté des méthodes *modifyjmf* et *modifyjmp*. Elles permettent de modifier les instructions assembleur de saut (conditionnel ou inconditionnel) après leur enregistrement dans la table car lors de la compilation, on ne peut pas anticiper la taille du block à sauter.

1.4 Tests et discussions

Pour vérifier le bon fonctionnement de notre compilateur, nous avons créé un petit programme qui doit être reconnu par le *compiler*. Tout d'abord, pour générer cet exécutable *compiler* capable d'analyser lexicalement et syntaxiquement le programme, nous avons utilisé les commandes suivantes :

```
bison -d inter.y
flex comp.l
gcc lex.yy.c inter.tab.c -o compiler symboltable.c codeasm.c
```

La première permet de compiler le fichier Yacc, la deuxième le fichier Lex, et la troisième fait le lien entre les deux. On a expliqué auparavant l'utilité de *lex.yy.c*, et on rappelle maintenant que le fichier *inter.tab.c* permet à Yacc et Lex de s'entendre sur les numéros des tokens reconnus et retournés.

Au final, voici la commande qui lance notre compilateur, et un exemple de *test.c* utilisé :

```
compiler < test.c
```

```
int main()
{
int toto = 5;
 while(toto > 1){
 toto = toto - 1;
 }
}
```

Notre compilateur écrit dans le fichier *instructions.txt* les résultats suivants :

```
1: AFC 1001 5
2: COP 0 1001
3: COP 1001 0
4: AFC 1002 1
5: SUP 1001 1002 1001
6: JMF 1001 12
7: COP 1001 0
8: AFC 1002 1
9: SUB 1001 1002 1001
10: COP 0 1001
11: JMP 3
```

On remarque que chaque affectation de valeur à une variable engendre bien une instruction AFC. Par exemple, ligne 1, on retient la valeur 5 à l'adresse de la pile 1001 avant de ranger cette variable dans la table des symboles à l'adresse 0 (instruction COP ligne 2). L'adresse 1001 et l'adresse 0 ne sont pas directement associées à de réels emplacements (par exemple, l'adresse 0 est en fait l'index du symbole dans la table). Mais elles ont pour objectif de donner des valeurs qui seront ensuite interprétées avant d'appliquer réellement les instructions dans un processeur (voir partie 2). 1001 sera transformée en une adresse de registre et 0 sera transformée en une adresse de donnée de la mémoire.

On peut voir aussi que les instructions JMP et JMF sont correctement écrites, en effet à la ligne 6, on lit une instruction JMF qui fera un saut à la ligne 12 (fin du programme) si la valeur contenue à l'adresse 1001 est *False*. La ligne 5 représente la comparaison *toto > 1* et son résultat est bien inscrit à l'adresse 1001 de la pile.

A la ligne 11, on remarque l'instruction inconditionnelle JMP qui renvoie l'exécution à la ligne 3, c'est à dire lors de la récupération de la valeur de *toto* à l'adresse 1001 de la pile afin ensuite de

comparer sa valeur au *1* affecté à l'adresse 1002 (ligne 4). C'est bien le comportement attendu pour la boucle while.

Pour arriver à ce résultat qui montre le bon fonctionnement de notre compilateur, il a fallu faire face à de nombreux obstacles. Par exemple, le fait que nous n'avions pas bien compris le lien entre les fichiers dès le début. Au cours de notre avancement, nous avons été de plus en plus efficaces et avons donc progressé sur ce point là.

Aussi, les langages utilisés nous étaient méconnus au début du semestre et lors des tests du *compiler*, les erreurs qui apparaissaient étaient parfois compliquées à localiser. Nous avons donc adopté une démarche d'exploration pas à pas pour les corriger.

2 Conception d'un microprocesseur de type RISC avec pipe-line

2.1 Objectifs et démarche

Dans un second temps, nous avons implémenté un processeur qui a pour rôle d'exécuter le code compilé auparavant. C'est à dire qu'il prend "en arguments" les instructions assembleur interprétées et enregistre et manipule les données de la mémoire.

Dans notre cas simplifié, ses principales fonctions permettent de faire des calculs mathématiques simples et de lire et modifier des données.

La démarche consiste à identifier les différents composants qui agissent (avec leurs entrées et leurs sorties), et à les assembler sur un chemin de données, c'est à dire tous reliés par des pipelines, pour que les instructions s'exécutent en parallèle. Ce chemin va former le processeur et va fonctionner au rythme d'un signal de *CLOCK*. L'objectif est d'optimiser leur temps de traitement.

Pour cela, nous avons utilisé le logiciel de Xilinx et écrit nos codes en VHDL. Des tests ont été possible grâce à un simulateur de VHDL.

2.2 Les différents composant

Les composants créés dépendent des fonctionnalités nécessaires aux instructions que notre processeur doit traiter.

Il s'agit donc de :
- *ADD, MUL* et *SOU* pour les opérations arithmétiques (le processeur à été simplifié en omettant *DIV*)
- *COP* pour copier la valeur d'un registre dans un autre registre, les registres servent à manipuler des valeurs intermédiaires aux calculs
- *AFC* pour affecter une valeur à un registre
- *LOAD* pour copier la valeur provenant d'une adresse de la mémoire de données dans un registre, la mémoire de données sert à manipuler les valeurs des symboles déjà gérés dans le compilateur
- *STORE* pour copier la valeur d'un registre dans un emplacement de la mémoire de données

Pour commencer, il faut une mémoire d'instruction pour lire toutes les instructions assembleur données en entrée. Le fichier associé est nommé *instructions_ memory.vhd*. Nous avons pris le parti de simplifier son implémentation en créant un tableau de taille fixe, dans lequel on écrit directement les instructions que l'on veut exécuter. Cela nous a aussi permis par la suite de faire des tests très rapidement.

Ensuite, on voit que les instructions d'opérations mathématiques et d'enregistrement des données ont besoin d'accéder à des registres. Il y a donc un composant Banc de registres, c'est le fichier *banc_ registre.vhd*, qui s'occupe en parallèle de la lecture et de l'écriture des registres. Ici aussi, on utilise un tableau comme structure fixe pour les registres.

Une fois que l'on a un les structures pour manipuler des données, on peut effectuer des calculs. C'est l'UAL (Unité Arithmétique et Logique), que nous avons implémenté dans le fichier *alu.vhd*, qui s'en charge. Son architecture a la particularité d'utiliser un signal intermédiaire pour calculer le résultat puis le retourner dans un deuxième temps . En effet, il faut d'abord prendre en compte les débordements éventuels et donc limiter cette valeur de retour.

Enfin, pour affecter les valeurs aux symboles, on utilise une Mémoire des données, *data_ memory.vhd*, contenant aussi un tableau fixe d'emplacements mémoire. Pour ce composant, la lecture et l'écriture ne peuvent pas s'exécuter en parallèle.

Chaque architecture de chaque composant peut être composée de plusieurs process et d'instructions extérieures à ces derniers. Toutes les instructions sont exécutées en parallèle entre elles et aussi vis-

à-vis des process. Un process exécute toutes ses instructions les unes après les autres mais n'attribue toutes les valeurs qu'a la fin de son exécution, seule la dernière affectation d'un signal est considérée pour son changement de valeur. Les instructions s'exécutent de façon asynchrone, mais le process peut être synchroniser sur un clock d'horloge en front montant ou descendant. Dans tous les cas, un process est cyclique et se lance pour chaque changement de valeur des signaux concernés.

Pour résumer les propriétés des différents composants, seul l'UAL n'est pas synchrone et ne contient pas de process. Tous les autres composants manipulent un tableau de données (registres, instructions ou valeurs de mémoire).

2.3 Le processeur

Le processeur ne peut fonctionner que si les informations extraites des instructions assembleur sont correctement transmises aux différents composants. Si on veut que les instructions avancent en parallèle sur les composants, il faut donc synchroniser l'avancement et conditionner l'utilisation de chaque composant en fonction de l'instruction assembleur a exécuter.

Nous avons donc implémenté un dernier composant, le pipeline. En plaçant un pipeline comme intermédiaire entre les composants de base, il va pouvoir transmettre des signaux en synchronisant sont process sur le signal *CLOCK*. A chaque front montant, une instruction avance d'un pipeline. Il en faut 4 en tout.

Pour l'utilisation du pipeline au sein du processeur, nous avons choisi de n'implémenter qu'un seul type de pipeline même si comme on peut le voir sur l'image, les deux derniers n'utilisent pas de 4ème signal d'entrée, ni de sortie.

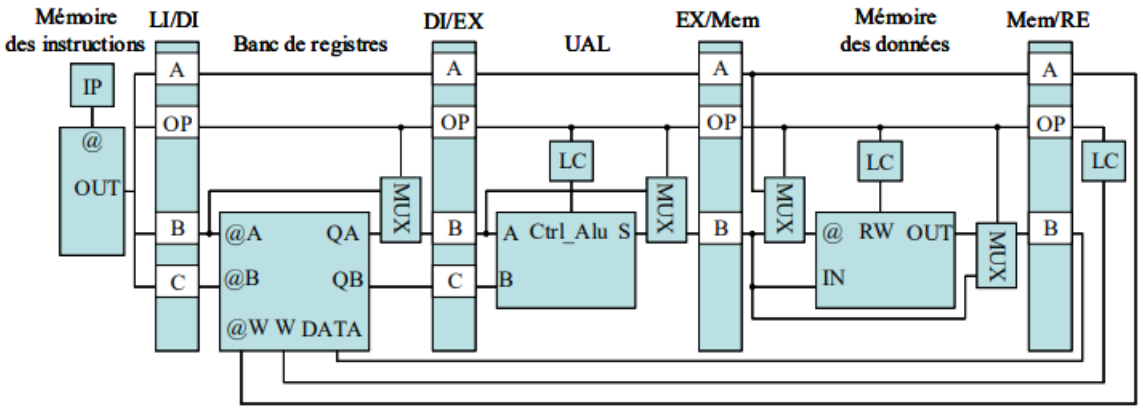

FIGURE 1 – Chemin de données

De plus, l'architecture du processeur, qui se trouve dans le fichier *processor.vhd*, contient un process qui effectue toutes les conditions associées aux MUX et aux LC pour différencier les traitements en fonction de OP, soit le code de l'instruction assembleur.

2.4 Tests et discussions

Pour vérifier le bon fonctionnement de nos différents composants, nous avons réalisé des tests de chacun individuellement, grâce à l'outil de simulation de Xilinx (ISim). Ensuite, nous avons créé un test pour le processeur, et vérifié la propagation des signaux dans les différents composants.

Ces procédures de tests ont soulevé des questions sur la synchronisation de nos composants, notamment un décalage dans la transmission entre le banc de registres et le deuxième pipeline. Néanmoins

il n'a pas été facile de les interpréter notamment car nous avons utilisé le bureau à distance MonTP au lieu d'installer Xilinx sur nos machines, et l'interface était assez réduite, complexe et ralentie. Nous avons manqué de temps pour nous adapter à cet outil et être efficaces.

Enfin, nous avons rencontré régulièrement des difficultés pour interpréter les différentes explications des sujets fournis. Nous avons remarqué que dans ces conditions, le travail en autonomie est très compliqué car nous débutons dans l'écriture d'architectures matérielles. Même si des cours et des TDs précèdent les TPs, nous sommes encore dans une phase de compréhension et d'assimilation des nouveaux concepts. C'est au cours des TPs que nous avons pu de les assimiler (en implémentant les composants par nous-même). Chaque point avec les professeurs était alors nécessaire et apprécié pour nous amener à trouver une direction dans les étapes suivantes.

Conclusion

Ce projet nous a fait découvrir les raisonnements qui lient un code informatique à l'architecture d'un processeur, et les interactions profondes qui permettent les étapes de compilation et d'exécution, si ordinaires au sein des apprentissages et des outils de notre formation. Cette initiation s'est rendue tout autant intéressante par l'exemple très concret d'un langage C simplifié, que formatrice. Nous avons appris à créer des règles de syntaxes et à manipuler les langages Lex, Yacc et VHDL.

Ce projet compte plusieurs améliorations en perspective. Le code du compilateur simplifié pourrait reconnaître un langage plus large, avec par exemple des pointeurs, la possibilité de retourner une valeur, ou encore celle de créer des fonctions intermédiaires. Il serait aussi possible d'étoffer la liste des instructions assembleur gérées par le processeur, avec par exemple *JMF* et *JMP*. Enfin, l'implémentation d'un interpréteur donnerait une dimension globale à ce projet car la partie compilateur et la partie processeur seraient reliées.

Du à la configuration à distance, il n'était pas simple de suivre le rythme des séances de TPs et d'interagir régulièrement avec les professeurs comme avec les autres étudiants. Notre organisation et l'accompagnement mis en place nous ont tout de même permis de rendre un projet assez abouti.