Projet Systèmes Informatiques : Du compilateur vers le microprocesseur

Poulin Lancelot , Aldebert Lucie

4ème année Informatique Année 2019 – 2020

1 Introduction

Ce projet s'articule autour de deux parties. Dans un premier temps le développement d'un compilateur avec LEX et YACC permettant de compiler du code en C et dans un second temps la conception d'un microprocesseur de type RISC avec pipeline pour l'exécuter.
Dans ce rapport se trouve notre démarche de conception ainsi que nos choix d'implémentation. On fera également un point sur nos résultats et les instructions assembleur utilisé.

2 Le compilateur

2.1 Le parseur lexical: LEX

La première étape de conception consiste en la réalisation d'un parseur lexical du langage C. Les mots reconnus du langage C sont retournés sous la forme de token, par exemple : "main" return(tMAIN), quand le mot main est reconnue le token tMAIN est retourné;. Il faut également retourner les valeurs: voici un exemple quand un nombre est reconnue : [0-9]+ yylval.nb=atof(yytext); return(tNB); , on retourne le token tNB et la valeur yytext converti en float avec la fonction atof. Nous n'avons pas rencontré de difficulté particulière pour la réalisation du parseur lexical.
Le fichier comporte seulement 24 lignes. Les espace les tabulation et les retour à la ligne ne retourne aucun token mais sont quand même reconnue pour pas que des erreurs soit retourné.

2.2 La reconnaissance syntaxique: YACC

La seconde étape de conception consiste mettre en place la reconnaissance syntaxique dans un fichier YACC. Le fichier YACC va utiliser les tokens et les valeurs retournés par le parseur lexical et va s'assurer que la syntaxe utilisée est bien celle du langage C.

2.2.1 Déclaration

Dans la première partie du fichier on déclare les différents tokens et leur type si besoin. On déclare également le type des variables que l'on utilise pour la reconnaissance syntaxique.

2.2.2 Corps

Dans la seconde partie du fichier c'est là où la syntaxe peut être reconnue.
On utilise les tokens pour définir la syntaxe des différentes parties qui compose un programme en C. Un programme doit être de la forme: "main () ", le corps du programme, "" en yacc on écris: Main: tMAIN tPO tPF tAO Body tAF.
Ou Body sert de variable au même titre que Main, il faut le définir avec des tokens. On fait donc la même chose pour Body, Body: Definition Body Et ainsi de suite jusqu'à définir toute la syntaxe du programme.

2.2.3 Les conflits de réduction

Une fois les expression arithmétique définit "+,-,*. . . ", on se heurte à un problème de conflit des réductions. Par exemple si on fait l'opération 4+3*6, on veut d'abord réaliser la multiplication et après l'addition. Pour cela il faut préciser dans le programme quelle opération doit être réaliser en premier. Pour cela dans la partie déclaration on va rajouter les le type de réduction %left %right et les déclarer du plus au moins prioritaire:

%left tPLUS tMOINS
%left tFOIS tDIV
%right tEQ

2.2.4 La mémoire (symboleTab.c)

Pour la gestion de la mémoire nous utilisons une table de symbole. Celle-ci est déclaré dans le fichier symboleTab.c inclus dans le fichier YACC. Quand une variable est déclarée on l'ajoute à la table des symboles, si on veut l'affecter on utilise une fonction pour trouver son adresse dans la table qui n'est d'autre que son indice. La fin de la table sert au stockage des variable temporaire. Celle-ci sont utilisé lors de calcul arithmétique les résultats intermédiaires sont stocké dans le tableau comme des variables temporaire.
La table contient également une valeur permettant de savoir si une variable est constante pour empêcher le cas venu de la modifier en cas d'affectation.

2.2.5 La profondeur

On peut déclarer des variables locales qui n'existe qu'intrinsèquement au if au for ou au while. Il faut donc bien tenir en compte la profondeur des déclarations. Elles ne sont valables qu'au profondeur supérieur à la leur. La profondeur globale doit donc être une variable globale, elle est présente dans le fichier symboleTab.c, nous n'avons pas finit d'implémenter cette fonctionalité, il est donc impossible de déclarer une variable local à un if, la variable sera global.

2.2.6 Traduction en langage assembleur

L'affectation, est traduit par l'instruction COP addrId valeur.
Les différentes expressions sont traduites part :
Un nombre est traité par l'instruction: AFC addrTemporaire valeur
L'addition: ADD addrTemporaire valeurExpression1 valeurExpression2
La soustraction: SOU addrTemporaire valeurExpression1 valeurExpression2
La division: DIV addrTemporaire valeurExpression1 valeurExpression2
La multiplication: MUL addrTemporaire valeurExpression1 valeurExpression2
L'égalité: EQU addrTemporaire valeureEpression1 valeurExpression2
Inferieur: INF addrTemporaire valeurExpression1 valeurExpression2
Superieur: SUP addrTemporaire valeurExpression1 valeurExpression2

Fonction "print" quant à elle part PRI addrId
Pour traduire un if ou if else ont à rencontrer un problème. La complexité repose sur le fait que l'on ne connaît pas en avance la taille du corps du "if" ni celui du else. Au moment où le "if (condition)" est reconnue on écrit l'instruction JMF suivit de la valeur de la condition. Mais on ne connaît pas encore la ligne à laquelle il faut sauter. C'est seulement arrivant à la fin du corps du "if" que l'on fait un patch sur l'instruction du jump à laquelle on rajoute le nombre de ligne +2. Pour l'instruction du else "JMP", on attend la fin du corps du else pour y coller le numéro de ligne d'instruction courante plus 1.

2.3 Fonctionnalités

Le compilateur permet la définition et l'affectation de variable de type Integer. Il reconnaît aussi le mot clé "const" qui permet qu'une variable reste constante, l'affectation doit se faire à la déclaration.
Il permet de réaliser des calculs arithmétiques, d'évaluer l'égalité, supérieur et inférieure. Il reconnaît et compile les structures if else. Il compile la fonction print.

2.4 Améliorations possibles

Le compilateur pourrait être amélioré en lui rajoutant des fonctionnalités.
Pour l'améliorer il faudrait rajouter les autres types: string, float... du langage C, les boucles tant que et les boucles for. On pourrait également gérer les déclarations de fonctions.
Pour coller au mieux au langage il faudrait également qu'une constante puisse être déclarer sans être instancié et enfin il faudrait gérer la profondeur des déclarations.

3 Le microprocesseur

Dans un premier temps, nous avons dû concevoir et implémenter en VHDL un microprocesseur avec pipe-line. Permettant le traitement des instructions assembleur suivante : addition, soustraction, multiplication, division, copie et affectation. Dans un second temps on a cherché à synthétiser et optimiser ce microprocesseur du point de vue de la fréquence de fonctionnement.

3.1 Conception

La conception du microprocesseur à était pensé bloc part bloc. Diffèrent module qui compose le processeur ont d'abord été codé, c'est à partir de ces briques que l'on a réalisé le processeur. Le processeur est composé de 5 pipelines générer à partir du même module de pipeline, d'un alu, d'un banc de registre, d'un banc de mémoire d'instruction et un de donnée. Il contient également trois multiplexeurs et de trois convertisseurs logiques.

3.2 Pipeline

Les 5 pipelines, prenne en entrer une instruction d'assembleur et mettent en sortie cette même instruction en fonction des tics de l'horloge.

3.3 Unité arithmétique logique

Elle prend en entrer les deux opérateurs et opérande. Opérande est appeler contrôle alu c'est une traduction opération assembleur par l'unité de conversion logique. En sortie on retrouve le résultat de l'opération est les différents flags, zéro, débordement, négatif et la retenue. Au niveau du traitement si contrôle alu est a 001 on réalise une addition s'il est à 011 une soustraction et 010 une multiplication.

3.4 Mémoire d'instruction

Pour simplifier le processeur, les instructions sont directement stockées dans le code de la mémoire d'instruction. Le tableau contenant les instructions à une taille limite de 255 instructions. Celle-ci sont écrit directement sous forme hexadécimal chaque opération d'assembleur est codé sous forme hexadécimal. L'instruction correspondante à l'adresse donnée en entré est mise en sortie de manière synchrone avec l'horloge.

3.5 Mémoire de donnée

La mémoire des données permet un accès en lecture ou en écriture. L'adresse de la zone mémoire concerner est fourni en entrée. Pour réaliser une lecture, RW doit être positionné à 1 et pour réaliser une écriture, RW doit être positionné à 0. Dans le cas d'une écriture, le contenu de l'entrée IN est copié dans la

mémoire à l'adresse donné. Le reset, RST, permettra d'initialiser le contenu de la mémoire à 0x00. La lecture, l'écriture et le reset sont réalisé de manière synchrone avec l'horloge.

3.6 Banc de registre

On réalise un banc de 196 registres de 8 bits. Les opérations d'écriture et de reset sont réalisées de manière synchrone avec l'horloge. Si le signal reset est à zéro le contenue est à 0x00. En entrée on prend deux adresses @A et @B elles servent à propager les valeurs correspondantes en sortie (QA et QB). Pour écrire dans un registre l'entrée W doit être à 1. Si c'est le cas l'entrée DATA est écris dans le registre d'adresse @W.

3.7 Chemin de données

Le chemin de donnée est la partie que l'on a trouvé le plus difficile. La réalisation des modules nous a paru évidente avec l'aide des schéma fournis et des précision fournis dans le sujet. Mais de rassembler les différentes pièces pour la création du processeur demande plus de rigueur. On a voulu tout relier directement sans tester au fur et à mesure ce qui a rendu la tâche de recherche d'erreur plus complexe.

Finalement, on n'a pas rencontré plus de difficulté que ça sauf pour le LOAD. Le problème que l'on a rencontré est un retard de la donnée sortante au niveau de la mémoire de donnée. De ce fait en entrée du dernier pipeline on avait d'abord OP et A correcte mais pas la donnée dans B et ensuite, OP et A de l'instruction suivante se trouvant être un NOP comme on ne gère pas les aléas de donnée et le B contenant la bonne information. Pour faire fonctionner le LOAD on a donc pris la décision de donnée en entrée du banc de registre sur DATA la sortie du Mux4 sans passer par le pipeline pour rattraper le retard du B.

3.8 Fonctionnement et fréquence de fonctionnement

Le processeur pour fonctionner nécessite de rentrer dans le tableau de la mémoire d'instruction, les instructions. Les opérations d'assembleur sont codées en hexadécimal comme suit : 0x01 pour l'addition 0x02 pour la soustraction 0x03 pour la multiplication 0x05 pour la copie et 0x06 pour l'affectation 0x07 pour le chargement et enfin 0x08 pour la sauvegarde. Pour l'instant seul c'est instruction d'assembleur sont permise. Il faudrait pouvoir rajouter les jumps. Notre processeur ne géré pas les aléas de donnée donc il faut impérativement laisser des instruction "vide" entre chaque instructions. La période minimale de fonctionnement est de 8,394 nano seconde.

4 Conclusion

Le projet nous a vraiment permis de voir la complexité et l'étendue du code se trouvant derrière un compilateur d'un langage informatique. Notre compilateur bien que permettant que peu de chose, nous a permis de bien faire le lien avec le cours d'automates et langage. Il nous laisse entrevoir la complexité de retranscrire la syntaxe d'un langage de programmation mais encore d'une langue parlée. Les langages de programmation bien que très codifié reste complexe à reconnaître bien que leur syntaxe soit très simplifié comparé à la langue française, le nombre de token à parser est également très limité en comparaison. On a pu également expérimenter avec le langage VHDL pour la réalisation du processeur. On à constater la aussi également la complexité des chemins de données à expérimenter.